

DATOVÉ MODELOVÁNÍ

VOJTĚCH MERUNKA

Vzor citace:  
Vojtěch Meruňka, Datové modelování  
Praha: Alfa Publishing, 2006

Recenzovali:  
Prof. RNDr. Jiří Hřebíček, CSc.  
Prof. RNDr. Václav Snášel, CSc.  
Doc. Ing. František Huňka, CSc.  
RNDr. Vratislav Datel, CSc.

#### KATALOGIZACE V KNIZE – NÁRODNÍ KNIHOVNA ČR

Meruňka, Vojtěch  
Datové modelování / Vojtěch Meruňka. – 1. vyd. – Brno : Alfa Publishing,  
2006. – 180 s. – (Management studium)  
ISBN 80-86851-54-0

004.652 \* 004.422.63 \* 004.41.045

- \* datové modely
- \* datové struktury
- \* objektově orientované metody
- \* učebnice

004.4/.6 – Programové vybavení. Programové prostředky [23]

Na vydání knihy se podílela Provozně ekonomická fakulta  
Česká zemědělská univerzita v Praze



Provozně ekonomická  
fakulta

Nakladatelství: ©Alfa Publishing, s. r. o.  
Rok prvního vydání: 2006  
ISBN 80-86851-54-0  
[www.alfaknihy.cz](http://www.alfaknihy.cz)

VOJTĚCH MERUNKA

# DATOVÉ MODELOVÁNÍ

Alfa Publishing



# Obsah

O čem je tato kniha .....	11
Úvod .....	13
Co to je datové modelování .....	13

## 1. oddíl

### Datové modelování a objektově orientovaný přístup

<b>1. Teoretické základy modelování na počítačích .....</b>	<b>17</b>
1.1 Lambda-kalkul .....	17
1.1.1 Formální zápis, beta-redukce, alfa-konverze .....	18
1.1.2 Lambda-výraz jako data .....	20
1.1.3 Příklad alfa-konverze .....	21
1.1.4 Eta-redukce .....	22
1.2 Základy objektově orientovaného přístupu .....	23
1.2.1 Historie a přehled .....	23
1.2.1.1 Čistý versus smíšený přístup .....	24
1.2.2 Objekt, zpráva, metoda .....	25
1.2.2.1 Zprávy .....	26
1.2.2.2 Protokol objektu .....	26
1.2.2.3 Data a metody .....	27
1.2.2.4 Polymorfismus .....	28
1.2.3 Datové modelování s objekty .....	29
1.2.3.1 Kolekce objektů .....	30
1.2.3.2 Třídy objektů, instance tříd, extenze tříd .....	31
1.2.3.3 Třídy versus kolekce .....	32
1.2.3.4 Grafické zobrazení .....	32
1.2.3.5 Hierarchie dědění objektů .....	34
1.2.3.6 Skládání objektů .....	36
1.2.3.7 Data a operace s nimi .....	37
1.2.3.8 Změny protokolu kolekcí při operacích s daty .....	40
1.2.3.9 Jak správně použít kolekce, atributy a skládání .....	42
<b>2. Modelovací jazyk UML .....</b>	<b>44</b>
2.1 Struktura UML .....	44
2.1.1 Diagram tříd .....	45
2.1.1.1 Asociace .....	46
2.1.2 OCL .....	47
<b>3. Smalltalk .....</b>	<b>48</b>
3.1 Jazyk .....	48
3.1.1 Pojmenování .....	48

3.1.2	Zprávy	49
3.1.2.1	Unární zprávy	49
3.1.2.2	Binární zprávy	50
3.1.2.3	Slovní zprávy	50
3.1.2.4	Kaskáda zpráv	51
3.1.2.5	Bloky	51
3.1.3	Zápis metod	53
3.1.3.1	Přístupové metody	54
3.1.4	Řízení výpočtu	54
3.1.5	Architektura programů ve Smalltalku	55
3.1.6	Reflexe	56
3.2	Vývojová prostředí	56
3.2.1	VisualWorks	56
3.2.2	STX	59
3.2.3	Squeak	59
3.2.3.1	Croquet	60
3.3	Daskalos	61
3.4	LambdaTalk	65
<b>4.</b>	<b>Gemstone</b>	<b>70</b>
4.1	Historie Gemstone	70
4.2	Vlastnosti Gemstone	70
4.3	Programovací jazyk Smalltalk DB	71
4.4	Příklad objektové databáze	72
4.4.1	Popis úlohy	72
4.4.2	Implementace úlohy	73
4.4.3	Program v jazyce Smalltalk DB databázového systému Gemstone	74
4.5	Příklady dotazů	77
4.6	Shrnutí	78
<b>5.</b>	<b>Příklady datových modelů</b>	<b>80</b>
5.1	Evidence přátel	80
5.2	Obchod s pivem	83
5.2.1	Třídy a kolekce	84
5.2.2	Data	86
5.2.3	Dotazy	86
<b>6.</b>	<b>Pokročilé metody návrhu datového modelu</b>	<b>90</b>
6.1	Jak poznat správný návrh?	90
6.2	Objektová normalizace	90
6.2.1	Datový objekt, atributy objekty	91
6.2.2	Tři objektové normální formy	91
6.2.2.1	1ONF	92
6.2.2.2	2ONF	93
6.2.2.3	3ONF	94

6.3	Transformace datového modelu	95
6.3.1	Změny objektového schématu	95
6.3.2	Refaktoring	98
6.4	Návrhové vzory	99
6.4.1	Co to je návrhový vzor	99
6.4.2	Příklady návrhových vzorů	100
6.4.2.1	Adaptér	100
6.4.2.2	Skladba	101
6.4.2.3	Dekorátor	102
6.4.2.4	Stav	103
<b>7.</b>	<b>Seznam použitých symbolů</b>	<b>105</b>
7.1	Formální zápis	105
7.2	Jazyk Smalltalk	106
<b>2. oddíl</b>		
<b>Datové modelování v projektování a tvorbě informačních systémů</b>		
<b>1.</b>	<b>Objektové programování</b>	<b>111</b>
1.1	Programovací jazyky a prostředí	111
1.1.1	Objektově orientované programovací jazyky	111
1.1.2	Smíšené programovací jazyky	112
<b>2.</b>	<b>Databázové systémy</b>	<b>113</b>
2.1	Objektově orientované databáze	113
2.1.1	Objektový datový model	113
2.1.2	Jak vytvořit objektovou databázovou aplikaci	115
<b>3.</b>	<b>Metody analýzy a návrhu informačních systémů</b>	<b>117</b>
3.1	Otázka transformace zadání od uživatele do podoby objektového modelu	118
<b>4.</b>	<b>Dnešní stav objektových nástrojů a technik</b>	<b>120</b>
4.1	Přínosy OOP	120
4.2	Problémy OOP	121
4.2.1	Odklon od původního OOP	121
4.2.2	Problém s UML	121
4.2.3	Nedostatečnost metod analýzy	124
4.3	Pokrok v oblasti programovacích jazyků a prostředí	124
4.4	Objektový přístup v databázových systémech	124
4.4.1	Čisté objektové a objektově relační databáze.	125
4.4.2	Situace v České republice a ve světě	127
4.4.3	Formální techniky návrhu objektových databází	127
4.5	Metody řízení projektů informačních systémů	128
4.5.1	Iterativní a evoluční versus sekvenční model životního cyklu	129

4.5.1.1	Sekvenční model životního cyklu .....	129
4.5.1.2	Iterativní model životního cyklu .....	129
4.5.1.3	Evoluční model životního cyklu .....	130
4.5.2	Rigorózní versus agilní metodiky .....	130
4.5.2.1	Rigorózní metodiky .....	130
4.5.2.2	Agilní metodiky .....	130
4.5.2.3	Příčina sporu .....	131
4.6	Tvorba informačních systémů v kontextu podnikového managementu .....	132
4.6.1	Procesy a procesní modely - requirement engineering .....	133
4.6.2	Myšlenka konvergenčního inženýrství .....	133
4.6.3	Vztah mezi informačním a řídicím systémem uvnitř organizace .....	134
4.6.4	Vztah k OOP .....	136
<b>5.</b>	<b>Jak správně využít objektový přístup v projektech informačních systémů .....</b>	<b>137</b>
5.1	Celopodnikový pohled .....	137
5.2	Model životního cyklu projektu informačního systému .....	138
5.2.1	Iniciace .....	141
5.2.2	Konstrukce .....	142
5.2.3	Dodání .....	142
5.2.4	Provoz .....	143
5.2.5	Jednotlivé týmy v procesech .....	143
5.2.6	Provozní, testovací a vývojová platforma .....	145
5.3	Postupná transformace datového modelu při projektování .....	146
5.3.1	Metoda BORM .....	146
5.3.1.1	Použití BORMu v praxi .....	147
5.3.2	Vývoj pojmu objekt během projektování .....	147
5.3.3	Fáze expanze a konsolidace .....	150
5.3.4	Objekty reálného světa (business objekty) .....	150
5.3.4.1	Metoda OBA .....	151
5.3.4.2	Diagram ORD .....	153
5.3.4.3	Podrobná analýza procesů .....	155
5.3.4.4	Rozšíření modelu business procesů směrem nahoru ...	156
5.3.4.5	Rozšíření modelu obchodních a správních procesů směrem dolů .....	156
5.3.4.6	Simulace procesů .....	157
5.3.4.7	Změna procesů – Business Process Reengineering ...	159
5.3.5	Logické - konceptuální objekty .....	160
5.3.6	Přechod od business objektů ke konceptuálním objektům .....	160
5.3.6.1	Diagramy konceptuálních objektů .....	160
5.3.7	Softwarové - implementační objekty .....	161
5.3.8	Přínos rozdělení modelu na business, konceptuální a softwarové objekty .....	161
5.3.9	Evoluce hierarchií objektů .....	162

5.3.10	Tři dimenze objektového modelu – zjednodušení složitosti . . . .	165
5.3.11	Chyby kterých je třeba se vyvarovat při modelování . . . . .	166
5.3.12	Zkušenosti . . . . .	167
<b>6.</b>	<b>Závěr . . . . .</b>	<b>168</b>
<b>7.</b>	<b>Použitá literatura . . . . .</b>	<b>169</b>
7.1	Vlastní publikace . . . . .	169
7.2	Ostatní . . . . .	170
	Seznam obrázků . . . . .	174
	Seznam tabulek . . . . .	176

# Poděkování

Na tomto místě je mojí milou povinností poděkovat všem, bez jejich inspirujících diskusí by tato práce nemohla vzniknout. Především všem kolegům, expertům v objektových technologiích, aktivně se účastnícím odborných konferencí Tvorba Softwaru a Objekty a jmenovitě (v abecedním pořadí) Jiřímu Bergerovi, Antonínu Cardovi, Vratislavu Datlovi, Pavlu Drbalovi, Jamesi Hallovi, Rogeru Knottovi, Jiřímu Kofránkovi, Richardu von Lavante, Martinu Molhancovi, Rudolfu Pecinovskému, Jiřímu Polákovi, Tomáši Richtovi, Petru Štěpánkovi, Jaroslavu Švastovi, Stergiu Tzortziovi, Tomáši Vaňákovi, Jiřímu Vaníčkovi, Miroslavu Viriusovi, Vladimíru Vlachovskému a Janu Vranému. A za pomoc a podporu manželce Ivetě a rodičům.

děkuji Vám všem

autor

## O čem je tato kniha

O objektově orientovaném přístupu byla již napsána řada knih. Tyto knihy se ale zaměřují na to, aby se čtenář naučil programovat. Ve většině případů se opírají o smíšené programovací jazyky (například C++, C# nebo Java), ve kterých je objektový přístup obsažen jen částečně a v kombinaci se starším procedurálním přístupem.

Tato kniha je jiná. Má za cíl vysvětlit základy objektově orientovaného přístupu ve tvorbě softwaru pomocí modelování dat bez zbytečného programování, které čtenář nemusí umět. Na druhou stranu ale text knihy předpokládá, že čtenář má základní znalosti práce s počítačem, dokáže jako uživatel pracovat s relačními databázemi a ovládá formální matematický jazyk.

Datové modelování je samostatným oborem. Jeho cílem není psaní programů ani kreslení schémat, ale umění pracovat s daty na počítačích. Obsah knihy je rozdělen do dvou částí.

➤ V první části jsou diskutovány techniky a nástroje vlastního datového modelování.

Výklad se zde opírá o speciální software, zjednodušený objektově orientovaný programovací jazyk a nevyhýbá se ani teorii a formálnímu aparátu.

➤ Druhá část se zabývá širšími souvislostmi datového modelování v projektování a tvorbě informačních systémů. V této části jsou rozebírány také různé problémy spojené s tvorbou softwaru a naznačena některá možná řešení.

Vyložené principy jsou aplikovatelné ve většině dnes používaných objektových programovacích jazyků a především objektových databázových systémů. Znalosti předložené v této knize budou čtenářům užitečné v projektech softwarového inženýrství při formulaci a verifikaci požadavků na informační systémy a při práci s nejrůznějšími aplikačním softwarem pro zpracování dat.

Tato kniha je doporučeným učebním textem pro studijní obory Provozně ekonomické fakulty České zemědělské univerzity v Praze a také pro studijní obory Elektrotechnické fakulty ČVUT v Praze.



# Úvod

V minulosti byl každý, kdo dovedl pracovat s počítačem, považován automaticky za programátora. S rozvojem hotových programů nejrůznějšího použití již nebyla znalost principů činnosti počítačů tolik potřeba, a tak se objevili takzvaní „uživatelé“. Podle tohoto historického dělení na „programátory“ a „uživatele“ lze dnes prohlásit, že pravděpodobně více než 99 % lidí pracujících s počítačem nejsou „programátoři“, ale jen „uživatelé“. Dokonce i většina z těch, kteří se programování učili ve škole, jej v praxi nikdy nepoužije.

Proto nepovažuji toto dělení za rozumné. Uživatelem počítače může být například malé dítě, které ještě neumí ani číst a psát, ale dobře ovládá svoji oblíbenou počítačovou hru, dokáže počítač správně zapnout i vypnout a umí spustit jakýkoliv nainstalovaný program pomocí ikonky a menu grafického uživatelského rozhraní. Uživatelem ale může být také fyzik, statistik nebo třeba architekt, který pracuje se složitým softwarem vyžadujícím hluboké znalosti v daném oboru.

Uživatelem je totiž dnes každý, kdo pracuje s počítačem. Proto tato kategorie ztrácí smysl. Znalosti konstrukce počítačů, principů činnosti počítače a programování počítačů nejsou pro využívání počítačů téměř potřeba a je to tak dobře. (Podobně jako znalosti teorie elektromagnetického pole nejsou nezbytné pro poslech rozhlasu nebo znalost principu pracovního cyklu tepelného stroje pro řízení automobilu.) Například ekonom, chce-li pracovat s počítačem, může, a dokonce musí zůstat ekonomem. Nemůžeme po něm chtít, aby se stal „programátorem pro ekonomy“.

## Co to je datové modelování

Vysvětlovat, jak se počítačové programy a to včetně například grafického uživatelského rozhraní a spolupráce s operačním systémem programují, je dnes pro většinu uživatelů počítačů zbytečné, protože pracují s hotovým softwarem. (Jako minimální rozumná úroveň práce s hotovými programy se považuje znalost práce s kancelářskými programy, prohlížečem internetu, klientem pošty a orientace v operačním systému. Takový soubor znalostí je označován jako „řidičský průkaz na počítač“.)

Být „uživatelem s řidičským průkazem“ na hotové programy však v mnoha případech odborné praxe nestačí. Není totiž pravda, že na všechno je k dostání hotový software, který předem uspokojuje požadavky uživatelů. Není to možné. Proto je značná část počítačových programů nastavitelných, lze je přizpůsobovat a měnit. A tam, kde ani toto nepomáhá, se musí nové programy vyrobit na míru. To je především vlastnost podnikových informačních systémů a vůbec všech programů na zpracování dat.

Proto by měl kvalifikovaný uživatel vědět, jak svůj hotový program přizpůsobit pro práci s údaji z nových úloh, tato nová data si vytvořit a pracovat s nimi. A když to nestačí, tak by měl umět tvůrcům softwaru dobře sdělit svoje požadavky na novou funkčnost tak, že popíše i prakticky otestuje, jak a s jakou informací potřebuje na počítači pracovat. Právě tento soubor znalostí patří do oboru datového modelování.

Scott Ambler ve své knize „The Object Primer“ (Ambler 2004) definuje datové modelování jako:

*„Data modeling is the act of exploring data-oriented structures. Like other modeling artifacts data models can be used for a variety of purposes, from high-level conceptual models to physical data models.“*

Jinou, mnohem stručnější, definici přináší Wikipedia:

*„In computer science, data modeling is the process of structuring and organizing data.“*

Datové modelování je specifická část softwarového inženýrství. Datové modelování nemá za cíl tvorbu programů ani obsluhu databázových systémů. Naivním uživatelům se sice nástroje datového modelování, jako jsou například diagramy nebo formální zápis, které obsahuje tato kniha, mohou zdát totožné s programováním, ale ve skutečnosti tomu tak není. Datové modelování ale na druhou stranu není ani pouhé kreslení diagramů a psaní manažerské dokumentace, jak si mnoho lidí od IT myslí. Proto si pro větší názornost datové modelování porovnáme s programováním následující tabulkou.

	programování	datové modelování
Používání programovacích jazyků.	Všechny prostředky jazyka včetně např. překladače a prostředků pro ladění a testování.	Jen vybraná část programovacích jazyků jako nástroj pro zápis dat a pro manipulace s daty.
Používání knihoven softwarových komponent.	Pro realizaci funkčních požadavků vytvářeného softwaru.	Nepoužívá se.
Používání návrhových vzorů.	Všechny vzory především pro realizaci funkčních požadavků vytvářeného softwaru.	Jen některé strukturální vzory pro popis dat.
Používání formálního aparátu.	Nepoužívá se.	Je nástrojem pro popis dat a pro manipulaci s daty. (Výroková logika, operace s množinami...)
Používání diagramů při analýze.	Všechny druhy diagramů včetně diagramů popisujících chování a změny systému v čase.	Jen diagramy popisující vlastnosti dat a vztahy mezi nimi.

Tab. 1: Srovnání datového modelování a programování

ODDÍL



**DATOVÉ MODELOVÁNÍ  
A  
OBJEKTIVĚ ORIENTOVANÝ  
PŘÍSTUP**



## Teoretické základy modelování na počítačích

Dříve se počítače označovaly také jako „matematické stroje“. Toto pojmenování je v jistém smyslu výstižnější. Počítače jsou skutečně přístroje, které pracují na základě principů matematiky a logiky, jejich kořeny sahají až do starověku.

Významným mezníkem pro pozdější vznik dnešních počítačů bylo odvětví matematiky, které se nazývá teorie množin. (Patří sem například pojem prvku, množiny, existence prvku v množině, sjednocení množin, průnik množin atd.) Další práci moderní doby již přímo související se dnešními počítači jsou práce italského matematika Giuseppe Peana z počátku 20. století, který položil základy dodnes používaného zápisu či jazyka sloužícího k zaznamenávání logických a matematických formulí. Patří sem například používání znaků  $\forall$ ,  $\exists$ ,  $\in$ ,  $\notin$  nebo používání hranatých, složených a lomených závorek atd.

Dalším významným krokem je rok 1936, kdy se objevily hned tři různé formální nástroje (Biermann 1990, Davis 1986), které měly již bezprostřední vliv na vznik prvních samočinných počítačů ve 40. a 50. letech:

- Prvním nástrojem je takzvaný Turingův stroj britského matematika Alana Turinga. (Hankin 1994) se poprvé objevil popis zařízení, které manipuluje s buňkami paměti a řídí se instrukcemi. Právě tuto teorii později prakticky využil von Neumann při konstrukci hardwaru prvních počítačů.
- Druhým nástrojem byly práce Kleena a Hilberta zabývající se rekurzivními funkcemi. Toto bylo později použito při konstrukci mnoha algoritmů a samozřejmě pro tvorbu programovacích jazyků.
- Třetím nástrojem je lambda-kalkul amerického matematika Alonza Churcha. Lambda-kalkul je nástroj k zápisu a manipulaci s počítačovým kódem, tedy s příkazy, které říkají, co a jakým způsobem má počítač dělat. Lambda-kalkul s nimi nakládá stejnou formou, jako pracujeme s obvyčejnými matematickými výrazy (například krácení, vytýkání před závoreku, dosazování, rozklady apod.). Při určité míře zjednodušení lze prohlásit, že lambda-kalkul je standard, pomocí kterého můžeme přesně, úsporně napsat počítačový program a popsat činnost počítače bez nutnosti jej psát pomocí příkazů nějakého konkrétního programovacího jazyka.

Je důležité vědět, že všechny tyto tři nástroje byly nejen „vynalezeny“ a popsány, ale že tvrzení v nich obsažená jsou náležitě podepřená důkazním aparátem. Nejde tedy jen o soubory myšlenek typu „bylo by dobré, kdyby...“.

### 1.1

## Lambda-kalkul

Lambda-kalkul je založen na pojmech abstrakce, funkce a aplikace funkce. Jde o prostředek, kterým lze úplně popsat, jak má počítač pracovat. Každý dnes používaný programovací

jazyk nějakým způsobem souvisí s lambda-kalkulem, ale pouze několik málo programovacích jazyků má všechny vlastnosti, které jsou v něm obsaženy. Bohužel, dnes nejpoužívanější programovací jazyky jako je například Java, C, C#, SQL a jazyky pro Internet a Web mezi ně nepatří.

Princip lambda-kalkulu si ukážeme na jednoduchém příkladě. Představme si, že budeme v obchodě kupovat 5 koláčů s cenou 8 korun za jeden. Celková cena samozřejmě bude

$$5 \cdot 8,$$

což je celkem 40 korun. Zde jsme museli spolu vynásobit dvě konkrétní hodnoty 5 a 8. Když provedeme nákup jiného počtu koláčů, například 7, tak celková cena bude

$$7 \cdot 8,$$

což je celkem 56 korun. Opět jsme násobili nějaké dvě konkrétní hodnoty. Nyní si představme, že budeme chtít naučit náš počítač, aby nám uměl počítat celkovou cenu nákupu pro libovolný počet koláčů. Musíme tedy provést abstrakci. Proměnlivý počet koláčů budeme muset nějak pojmenovat, například jako „koláče“. Výsledná formule pro celkovou cenu potom bude

$$\textit{koláče} \cdot 8.$$

Tato formule ale počítači ještě nestačí, protože mu ještě musíme sdělit, že když budeme znát konkrétní počet koláčů, tak má jméno „koláče“ nahradit tímto počtem, aby se mohla spočítat celková cena nákupu. To, co tedy chceme, aby počítač provedl, lze napsat například takto:

$$\text{NAHRAĎ } \textit{koláče} \text{ DO } \textit{koláče} \cdot 8.$$

Tento zápis je již pro počítač srozumitelný, protože pokud budeme chtít spočítat cenu konkrétního nákupu například pro 12 koláčů, tak tuto hodnotu můžeme aplikovat na náš zápis například takto:

$$(\text{NAHRAĎ } \textit{koláče} \text{ DO } \textit{koláče} \cdot 8) \text{ HODNOTOU: } 12.$$

Počítač si vezme hodnotu 12 a nahradí touto hodnotou jméno „koláče“, vznikne mu výraz  $12 \cdot 8$  a ten dává hodnotu 96, což je cena, kterou jsme chtěli spočítat.

### 1.1.1

## Formální zápis, beta-redukce, alfa-konverze

Lambda-kalkul slouží k tomu, abychom mohli formule a manipulace s nimi zapisovat a aplikovat stejným způsobem, jako se v matematice standardně pracuje s obyčejným matematickým a logickým zápisem, a to včetně běžných operací, jako je například dosazování, vytýkání před závorku atd. Existuje dokonce hned několik způsobů takového

zápisu. Lambda-kalkul se také stal základem složitějších teorií, pomocí kterých se odborníci snaží popisovat chování speciálních algoritmů, dat v databázích, výpočetních modelů atd. V této knize vystačíme s vybranými částmi originálního kalkulu, které budou později doplněny o možnost práce s některými pojmy objektivě orientovaného přístupu.

Syntaxi (= způsob zápisu) lambda-kalkulu si ukážeme na příkladě z předchozí kapitoly. Výraz

NAHRAĎ *koláče* DO *koláče* · 8

pomocí lambda-kalkulu zapisujeme takto:

$(\lambda \text{ koláče} \mid \text{koláče} \cdot 8)$ .

Lambda-výraz se skládá ze dvou částí oddělených nějakým znakem. Nejčastěji se používá znak „,“ nebo „|“, který jsme zvolili i v této knize.

➤ První část lambda-výrazu je takzvaná hlavička, ve které je seznam proměnných uvozených řeckým písmenem lambda, které dalo název celému aparátu. Tyto proměnné se jmenují „lambda-proměnné“. Když na nějaký lambda-výraz aplikujeme nějaké hodnoty, tak se na takovéto lambda-proměnné tyto hodnoty navazují.

➤ Druhá část lambda-výrazu je takzvané tělo, ve kterém se nachází vlastní formule. Tato část lambda-výrazu se zapisuje i chová stejně jako běžný matematický zápis.

Pokud budeme chtít aplikovat nějakou hodnotu na nějaký lambda-výraz, jako například v příkladu předchozí kapitoly,

(NAHRAĎ *koláče* DO *koláče* · 8) HODNOTOU: 12,

tak tuto „aplikaci lambda-funkce“ zapíšeme takto:

$(\lambda \text{ koláče} \mid \text{koláče} \cdot 8) \triangleleft: 12$

Znakem  $\triangleleft:$  zde vyznačujeme aplikaci lambda-výrazu na hodnotu. (Originální lambda-kalkul dokonce žádné znaky nepoužívá a hodnotu od lambda-výrazu odděluje jenom mezerou v textu. Zde však budeme lambda-výraz používat i v kombinaci s objekty, a tak by tento původní úsporný zápis mohl vést k horší čitelnosti).

Způsob vyhodnocování lambda-výrazu, což také znamená popis chování počítače během výpočtu, si rozevíšeme podrobněji:

$(\lambda \text{ koláče} \mid \text{koláče} \cdot 8) \triangleleft: 12$	Toto je výchozí stav. Hodnota 12 se bude aplikovat na lambda-proměnnou koláče.
$\Rightarrow (\lambda \text{ koláče}=12 \mid \text{koláče} \cdot 8)$	Hodnota 12 se svázala s proměnnou koláče.
$\Rightarrow 12 \cdot 8$	Došlo k redukci celého výrazu na pravou část, protože hodnota proměnné byla dosazena na příslušné místo na pravé straně.
$\Rightarrow 96$	Výraz byl běžným matematickým způsobem zpracován na výslednou hodnotu.

Výměna lambda-proměnných v pravé části výrazu za konkrétní hodnoty po aplikaci lambda-funkce je operace, kterou Alonzo Church nazval  $\beta$ -redukce (čte se beta-redukce).

Lambda-výraz může obsahovat libovolně pojmenované lambda-proměnné. Proto také náš výraz  $(\lambda \text{ koláče} \mid \text{koláče} \cdot 8)$  můžeme napsat také třeba jako

$$(\lambda x \mid x \cdot 8)$$

beze změny jeho významu. Tento nový výraz je obsahově totožný s předchozím výrazem, což znamená, že je-li použit stejným způsobem jako předchozí, musí dát také stejný výsledek. To si můžeme předvést na stejném příkladu:

$$\begin{aligned} & (\lambda x \mid x \cdot 8) \triangleleft 12. \\ \Rightarrow & (\lambda x=12 \mid x \cdot 8). \\ \Rightarrow & 12 \cdot 8. \\ \Rightarrow & 96. \end{aligned}$$

Pokud tedy v lambda-výrazu přejmenujeme proměnné shodně v levé i pravé části, tak se daný lambda-výraz nemění. Takovéto operaci „přejmenování“ se říká  $\alpha$ -konverze (čte se alfa-konverze).  $\alpha$ -konverze se nejčastěji používá tam, kde se kombinuje víc lambda-výrazů mezi sebou a hrozí záměna stejně pojmenovaných proměnných v různých výrazech.

## 1.1.2

### Lambda-výraz jako data

Lambda-kalkul má jednu velmi zajímavou a důležitou vlastnost. Právě pro tuto vlastnost se stal nejpoužívanějším formálním nástrojem pro popis a práci s vyčíslitelnými funkcemi a tedy i s počítačovým kódem. Jde o to, že samotný lambda-výraz může být považován za hodnotu a může být proto aplikován do jiného lambda-výrazu úplně stejným způsobem, jako jsme v našem příkladě aplikovali obyčejné číslo.

Tuto možnost si ukážeme na příkladu. Mějme náš známý lambda-výraz pro výpočet ceny nákupu koláčů:

$$(\lambda x \mid x \cdot 8).$$

Na tento výraz jsme aplikovali hodnotu 12, abychom vypočítali, kolik stojí 12 koláčů. Představme si teď, že počet koláčů je také výsledkem nějakého výpočtu. Například můžeme koláče nakupovat ke snídani pro dětský tábor. Počet koláčů je potom dán počtem dětí, kde budeme počítat dva koláče na jedno dítě, a počtem koláčů, které sní vedoucí tábora, kterému dva koláče nemusí stačit. Celkový počet koláčů je potom dán lambda-výrazem

$$(\lambda a \lambda b \mid 2a + b),$$

kde  $a$  je počet dětí a  $b$  je množství koláčů na vedoucího. Budeme-li mít například 14 dětí a vedoucí sní 3 koláče, tak nám náš počítač zjistí, že musíme koupit 31 koláčů:

$$\begin{aligned}
 & (\lambda a \lambda b \mid 2a + b) \triangleleft: 14 \triangleleft: 3. \\
 \Rightarrow & (\lambda a=14 \lambda b=3 \mid 2a + b). \\
 \Rightarrow & 2 \cdot 14 + 3. \\
 \Rightarrow & 31.
 \end{aligned}$$

Nyní máme dva lambda-výrazy. Jeden slouží k výpočtu ceny nákupu ze známého počtu koláčů a druhý slouží ke zjištění počtu koláčů. Můžeme je používat odděleně tak, že si nejprve vypočítáme počet koláčů a potom z této hodnoty spočítat cenu nákupu. Lambda-kalkul ale dovoluje s lambda-výrazy pracovat jako s hodnotami. Takže v tomto příkladu můžeme rovnou použít výraz pro výpočet počtu koláčů jako „hodnotu“ pro výpočet ceny nákupu, což si počítač upraví jako:

$$\begin{aligned}
 & (\lambda x \mid x \cdot 8) \triangleleft: (\lambda a \lambda b \mid 2a + b). \\
 \Rightarrow & (\lambda x=(\lambda a \lambda b \mid 2a + b) \mid x \cdot 8). \\
 \Rightarrow & ((\lambda a \lambda b \mid 2a + b) \cdot 8).
 \end{aligned}$$

Výsledkem této operace je lambda-výraz  $((\lambda a \lambda b \mid 2a + b) \cdot 8)$  hodnoty ceny nákupu. Není to ale konkrétní číslo. To je samozřejmé, protože jsme ještě neuvedli, kolik máme dětí a kolik koláčů sní vedoucí. Proto je výsledkem našeho výpočtu ceny nákupu výraz, který pracuje s počtem dětí a počtem koláčů na vedoucího. Budeme-li mít například 14 dětí a vedoucí sní 3 koláče, tak počítač z naší „lambda ceny nákupu“ vyrobí konkrétní cenu nákupu v korunách:

$$\begin{aligned}
 & ((\lambda a \lambda b \mid 2a + b) \cdot 8) \triangleleft: 14 \triangleleft: 3. \\
 \Rightarrow & ((\lambda a=14 \lambda b=3 \mid 2a + b) \cdot 8). \\
 \Rightarrow & (2 \cdot 14 + 3) \cdot 8. \\
 \Rightarrow & 248.
 \end{aligned}$$

Tento příklad nám ukázal, že lambda-výrazy se chovají jako data a lze s nimi jako s daty nakládat. Náš příklad s nákupem koláčů může být třeba součástí nějakého většího informačního systému, například pro management dětského tábora. Lambda-výraz představující cenu nákupu potom může být přímo uložen mezi ostatní data do databáze. V případě, že počítač bude znát počet dětí a hladovost vedoucího tábora, tak se nám tento lambda-výraz bude mezi ostatními daty chovat stejně, jako bychom místo něj pracovali s obyčejným číslem.

### 1.1.3

#### Příklad alfa-konverze

Vraťme se ještě jednou k  $\alpha$ -konverzi a našemu příkladu. Pro výpočet celkové ceny z počtu koláčů nám sloužil výraz  $(\lambda x \mid x \cdot 8)$ . Už víme, že výraz pro výpočet počtu koláčů  $(\lambda a \lambda b \mid 2a + b)$  můžeme také napsat jako  $(\lambda x \lambda y \mid 2x + y)$ . Nastane tu však problém s lambda-proměnnou  $x$ , protože jednou je v prvním výrazu a podruhé se vyskytuje jako jiné  $x$  v druhém výrazu. Bez  $\alpha$ -konverze bychom při aplikaci jednoho výrazu na druhý narazili na problém s různými  $x$ :

$$\begin{aligned}
& (\lambda x \mid x \cdot 8) \triangleleft: (\lambda x \lambda y \mid 2x + y). \\
\Rightarrow & (\lambda x = (\lambda x \lambda y \mid 2x + y) \mid x \cdot 8). \\
\Rightarrow & ???
\end{aligned}$$

Pomocí  $\alpha$ -konverze je proto třeba jeden z výrazů upravit, například takto:

$$(\lambda x \mid x \cdot 8) \Rightarrow (\lambda z \mid z \cdot 8).$$

Nyní je již možné provést aplikaci jednoho výrazu na druhý shodně, jak to bylo ukázáno v předchozí kapitole:

$$\begin{aligned}
& (\lambda z \mid z \cdot 8) \triangleleft: (\lambda x \lambda y \mid 2x + y). \\
\Rightarrow & (\lambda z = (\lambda x \lambda y \mid 2x + y) \mid z \cdot 8). \\
\Rightarrow & ((\lambda x \lambda y \mid 2x + y) \cdot 8).
\end{aligned}$$

Podobně jako  $\beta$ -redukcí, tak i  $\alpha$ -konverze je operace, kterou „lambda-počítač“ vykonává automaticky během manipulací s výrazy.

### 1.1.4 Eta-redukce

Mějme formuli

$$\lambda x \mid (\text{výraz} \triangleleft: x),$$

kde výraz označuje libovolnou jinou lambda-formuli. Protože aplikace jakékoliv hodnoty do tohoto výrazu jako

$$(\lambda x \mid (\text{výraz} \triangleleft: x)) \triangleleft: \text{hodnota}$$

se upravuje na

$$(\lambda x = \text{hodnota} \mid (\text{výraz} \triangleleft: x)) \text{ a dále na } \text{výraz} \triangleleft: \text{hodnota},$$

tak můžeme prohlásit, že

$$\lambda x \mid (\text{výraz} \triangleleft: x) = \text{výraz}.$$

Toto zjednodušení formule  $\lambda x \mid (\text{výraz} \triangleleft: x)$  na výraz se v lambda-kalkulu nazývá  $\eta$ -redukce a je užitečná při úpravách výrazů.

## 1.2

### Základy objektově orientovaného přístupu

V této kapitole si vysvětlíme základy objektově orientovaného přístupu (ve zkratce OOP). Obecné povědomí o OOP je takové, že se jedná o moderní, ale složitý programovací styl. Často je možné zaslechnout od zájemců o tuto problematiku větu: „Tohle já nikdy nepochopím, protože jsem neuměl ani to normální programování.“ Názor, kdy se OOP vykládá jako další stupeň programování, považujeme za škodlivý omyl, a proto se pokusíme vyložit jeho principy jinak, než z perspektivy dnes nejrozšířenějších programovacích jazyků.

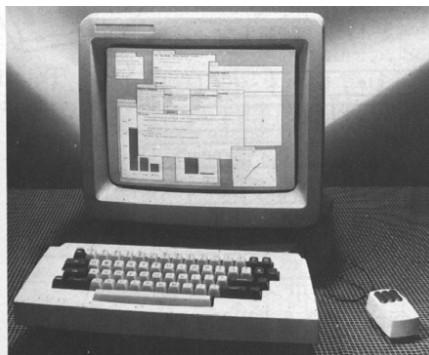
### 1.2.1

#### Historie a přehled

Počátky OOP sahají až do 60. let 20. století. Prvním systémem, který měl jeho rysy, byl již programovací jazyk Simula, který pochází z roku 1967. V té době šlo ale jen o jednotlivé vlastnosti a pojmy, které byly využity a rozvinuty až později.

Vlastní OOP se zrodil v průběhu 70. let v USA. V té době bylo aktivní výzkumné středisko Palo Alto Research Center (PARC) v Kalifornii, kde se řešila řada projektů, které výrazně ovlivnily následný rozvoj počítačů. (Kromě OOP zde také vyvinuli například Ethernet, laserové tiskárny, tablet, myš nebo dotykovou obrazovku.) Pro OOP byly důležité dva týmy: Learning Research Group vedený Alanem Kayem a System Concepts Laboratory vedený Adelou Goldbergovou. Předmětem jejich výzkumu, který byl financován v největší míře firmou Xerox, byl projekt Dynabook pro vývoj osobního počítače budoucnosti (HOPL 1988).

Počítač z projektu Dynabook se měl skládat z grafického displeje formátu listu papíru o velikosti přibližně A4 s jemnou bitovou grafikou, klávesnicí, perem na dotykové obrazovce (později nahrazeným jednodušeji pracující myší a tabletem), a jeho součástí mělo být i síťové rozhraní. Pro vzhled systému byla poprvé na světě použita překryvná okna, vnořovací menu a ikony. V průběhu 70. let bylo dokonce vyrobeno několik úspěšně fungujících prototypů takových počítačů (viz fotografie).



Obr. 1: Xerox Dorado, 1976

Předpokládalo se, že počítač bude obsahovat jednotné softwarové prostředí, které bude současně plnit úlohu operačního systému i programovacího jazyka s vývojovým prostředím. Tento software, který měl být zároveň jazykem i operačním systémem, dostal název Smalltalk.



Obr. 2: Tektronix 4404, 1982

Ve Smalltalku se nejvíce odrazily prvky z jazyka LISP, který byl v té době nejčistší implementací lambda-kalkulu, a z již připomenutého jazyka Simula.

Po ukončení projektu na počátku 80. let část týmu v PARC zůstala pod vedením Adely Goldbergové a zakládala různé firmy, které spolu s ostatními rozvíjejí Smalltalk dodnes. Jiní spolu s Alanem Kayem odešli do firmy Apple Computer, kde poté uvedli na trh první dostupný komerční osobní počítač Lisa (předchůdce řady Macintosh) s grafickým uživatelským rozhraním inspirovaným Dynabookem. Myšlenka Dynabooku dala během 80. let vznik nejen systémům Apple, ale později také MS Windows a X-Window.

Od poloviny 80. let vznikla řada dalších objektových programovacích jazyků. I samotný Smalltalk se dále vyvíjel a vyvíjí se dodnes. Kromě programovacích jazyků má OOP podstatný vliv na rozvoj operačních systémů a v neposlední řadě i databázových systémů, které by jinak pravděpodobně ustrnuly na bohužel dodnes stále ještě nepoužívanějším relačním datovém modelu.

### 1.2.1.1

#### Čistý versus smíšený přístup

Od vzniku OOP dodnes dochází ke vzájemnému soutěžení dvou koncepcí. Jde o čistý a o smíšený přístup k realizaci myšlenek OOP.

Čisté OOP je po teoretické stránce ideálním nástrojem pro modelování dat i tvorbu softwaru. Jeho praktická implementace na dnešních počítačích však není jednoduchá. Na rozdíl od předchozích softwarových paradigmat je totiž objektové paradigma na značně vyšší úrovni abstrakce, která neodpovídá tomu, jak počítače uvnitř pracují. V počítačové

technologii bohužel platí cosi, co bychom mohli nazvat jako „zákon zachování složitosti“. Jde o to, že pokud budeme chtít, aby na počítači pracoval systém, který je snadno srozumitelný pro svého uživatele (například systém řízení výroby v nějakém podniku nebo geografický informační systém), jeho vlastní implementace je velmi složitá. Naopak to, co je pro běžného smrtelníka velmi složité vzhledem k nízké úrovni abstrakce (například strojový kód), počítače zvládají bez problémů.

Smíšený přístup je kompromisem, kde za cenu ztráty některých abstraktních vlastností OOP docílíme zjednodušení počítačové implementace a tím ulehčíme práci tvůrcům takových systémů. Na druhou stranu ale přitížíme uživatelům a jejich programátorům, kteří v těchto systémech musejí sestavovat aplikace. Touto cestou jde většina programovacích jazyků – jazyky C++, Java a C# nevyjímaje. Pokud bychom neměli problémy s konstrukcí počítačů, netrápila nás zátěž zpětné kompatibility se starším softwarem a řada dalších nepříjemných faktorů, smíšený přístup bychom nepotřebovali. Dnes již řada těchto faktorů díky rozvoji technologií neplatí, ale v 80. letech šlo o téměř nepřekonatelné potíže.

Před čtvrt stoletím bránila nedostatečnost hardwaru většímu rozvoji čistých objektových technologií. Dnes to již neplatí. Ale v průběhu uplynulých let smíšené technologie zdomácněly a staly se uživatelským standardem. Takže v dnešní době jsou uživatelé nuceni užívat primitivnější, a tím pádem i pro ně složitější software, než by mohli mít. Samozřejmě trh to kompenzuje cenou produktů, nabídkou dokonalých grafických nadstaveb diskutovaného softwaru, příručkami, školeními, nabídkou poradenských služeb atd., což tuto neblahou situaci jistě zpřjemňuje. Typickým příkladem tohoto stavu je nabídka databázových systémů. Pokud se ale čtenář této knihy stal obětí školení datového modelování například v nějakém relačně-objektovém databázovém systému, jistě ví, o čem se zde píše.

Na závěr ale chceme poznamenat, že smyslem této kapitoly i celé knihy není naplivat na smíšené jazyky oheň a síru, ale pokusit se o netradiční pohled na problematiku OOP. Naopak, velmi si vážíme odborníků, kteří se zabývají touto technologií. Bez jejich znalostí by na světě nefungovala většina dnešního softwaru. Je také pravda, že poslední vývoj některých smíšených jazyků (například C# nebo i Java) naznačuje posun k čistému OOP.

## 1.2.2

### Objekt, zpráva, metoda

Základem OOP je pojem objekt. Na objekt lze nahlížet jako na stavební jednotku, která modeluje nějakou část reálného světa. V objektu jsou pohromadě uzavřena data i jejich vlastnosti a chování. Objekt může popisovat například živou bytost (člověka, zvíře...), nějaké zařízení (například tiskárna, detektor pohybu, kameru...), neživý předmět (například faktura, hotelový pokoj...) nebo i nějaký abstraktní pojem (například stav počasí, manželství, diagnózu u lékaře...). Objekty jednoho systému jsou mezi sebou propojené různými vazbami a vzájemně na sebe působí.

### 1.2.2.1

## Zprávy

Objekty mají tu vlastnost, že dokážou reagovat na požadavky, které jsou jim posílány. Požadavkům se v terminologii OOP říká zprávy. O objektech se potom hovoří, že jsou příjemci zpráv. Máme-li například objekt, který reprezentuje nějakou osobu, je rozumné předpokládat, že takové osobě budeme klást požadavky například na její jméno, příjmení a adresu. Bude-li takový objekt pojmenován například „*zákazník*“, můžeme tyto operace posílání zprávy zapsat takto:

*zákazník*◁*jméno*.  
*zákazník*◁*příjmení*.  
*zákazník*◁*adresa*.

Takovéto zprávy samozřejmě posíláme objektům proto, abychom se dozvěděli příslušné údaje. Proto i zde uvedené posílání zpráv dává výsledky, jako například:

*zákazník*◁*jméno* ⇒ Jan.  
*zákazník*◁*příjmení* ⇒ Novák.  
*zákazník*◁*adresa* ⇒ Pardubice.

Na uvedené zprávy se proto můžeme dívat i tak, že slouží k „vybírání“ údajů z objektů. To ale není jediná funkce, kterou zprávy v objektovém systému plní. Druhou důležitou funkcí zpráv je „*zapisování*“ údajů do objektů. Toto dělají zprávy, které obsahují data, která se nazývají parametry zpráv. Přestěhuje-li se například náš zákazník na jinou adresu, musíme jeho objektu poslat zprávu s parametrem-údajem nové adresy, která tuto novou adresu do tohoto objektu zapíše:

*zákazník*◁*adresa*: Kolín.

Kromě vybírání a zapisování údajů v objektech zprávy slouží ještě ke spouštění mnoha dalších operací, z nich některé mohou být složité jako celé samostatné programy a právě proto se tak využívají. Toto je však již doména objektového programování a pro vysvětlení základů datového modelování se tím zabývat nemusíme.

### 1.2.2.2

## Protokol objektu

Protokolem objektu rozumíme množinu všech zpráv, které je možné příslušnému objektu poslat. Bez znalosti protokolu uživatelé s objekty nemohou pracovat, protože nevědí, co objekty umějí udělat. Formálně se na protokol můžeme dívat jako na funkci, která nám pro příslušný objekt určuje množinu vlastností tohoto objektu. Pro náš objekt modelující zákazníka to je například:

$\Pi(\text{zákazník}) = [\text{jméno}, \text{příjmení}, \text{adresa}]$ .

Řecké písmeno velké „ $\Pi$ “ použijeme jako zkratku pro slovo „protokol“ a lomené závorky se v matematickém zápisu používají pro označení výčtu prvků, u kterých nezáleží na jejich pořadí (tj. neuspořádaná množina).

### 1.2.2.3

## Data a metody

Z dosavadního výkladu lze vytušit, že objekty v sobě uchovávají údaje. Jinak by si například objekt v našem příkladu nedokázal zapamatovat novou adresu, kterou se dozvěděl díky zprávě ( $\text{zákazník} \langle \text{adresa: Kolín}$ ). Tato data můžeme zapsat jako

$\Delta(\text{zákazník}) = [\text{jméno: Jan}, \text{příjmení: Novák}, \text{adresa: Kolín}, \dots]$ .

Kromě těchto vnitřních dat ale objekty uchovávají ještě operace, které lze pomocí zpráv spouštět. Takové operace se v terminologii OOP nazývají metody. Jak taková metoda může vypadat, si ukážeme na našem příkladu. Předpokládejme, že náš zákazník má protokol

$\Pi(\text{zákazník}) = [\text{jméno}, \text{příjmení}, \text{adresa}, \text{datum narození}, \text{věk}]$ .

To znamená, že pošleme-li například zprávu

$\text{zákazník} \langle \text{věk}$ ,

tak dostaneme číselný údaj, jak je zákazník starý. Můžeme samozřejmě předpokládat, že tento číselný údaj je v našem objektu uchován stejně jako jeho jméno, příjmení, adresa i datum narození. To je sice možné, ale pravděpodobně nepraktické, protože každý den bychom museli objekty v našem systému aktualizovat a pokud by měly narozeniny, tak nastavit údaj věku o jedničku větší. Proto je výhodnější věk počítat z datumu narození, protože tento údaj se nemění. Tento výpočet nám bude provádět metoda. Každá metoda se skládá ze dvou částí, které zapisujeme takto:

$\langle \text{hlavička}, \text{tělo} \rangle$ .

Hlavička, která plní úlohu jména metody, nám označuje, jakou zprávou se tato metoda vyvolává. Tělo je lambda-výraz, který popisuje, co příslušná metoda umí udělat. Lomené závorky se v matematickém zápisu používají pro uspořádané n-tice prvků – zde to je uspořádaná dvojice. Na metodu lze proto nahlížet jako na pojmenovaný lambda-výraz. Pokud náš počítač má zabudované hodiny a aktualizuje si sám datum, tak metoda, která umí spočítat věk našeho objektu, může být

$\langle \text{věk}, (\text{dnešní datum} - \sigma \text{datum narození}) / 365.2422 \rangle$ ,

Symbolem  $\sigma$  (řecké písmeno „sigma“) označujeme objekt, kterému tato metoda patří. Tento symbol je potřeba proto, že musíme vyznačit, odkud se bere údaj, který je součástí výrazu. Díky tomu počítač pozná, že *datum narození* je údaj ze stejného objektu jako celá metoda.

Pro potřebu formálního zápisu je ještě potřeba zavést novou funkci *Meth*, která označuje množinu všech metod daného objektu. Nyní již můžeme napsat, že výše uvedená metoda patří objektu *zákazník*:

$$\langle \text{věk}, (\text{dnešní datum} - \sigma \text{datum narození}) / 365.2422 \rangle \in \text{Meth}(\text{zákazník}).$$

Lambda-výraz v tomto příkladu neměl žádnou lambda-proměnnou. To proto, že zpráva  $\langle \text{věk} \rangle$  neposílá do objektu žádná data a jen chce z objektu „vybírat“. Proto se žádná hodnota do lambda-výrazu neaplikuje. Pokud bychom ale například měli metodu pro zprávu  $\langle o \text{ kolik starší než:} \rangle$ , která zjišťuje věkový rozdíl mezi dvěma objekty, tak její zápis lambda-proměnnou mít bude:

$$\langle o \text{ kolik starší než:} \rangle, (\lambda x \mid \sigma \langle \text{věk} - x \langle \text{věk} \rangle \rangle).$$

Použijeme-li poté tuto zprávu ve výrazu

Jan Novák  $\langle o \text{ kolik starší než:} \rangle$  Ivana Horáková

Tak dostaneme jako výsledek věkový rozdíl dvou objektů, z nichž první objekt je příjemce zprávy a druhý objekt vystupuje jako data ve výše uvedené zprávě. Tento druhý objekt se při vyhodnocování naváže na lambda-proměnnou v těle metody. Zápis  $\sigma \langle \text{věk} \rangle$  říká, že zpráva *věk* je posílána stejnému objektu, kterému patří i celá metoda  $\langle o \text{ kolik starší než:} \rangle, \dots$ . Jinými slovy to znamená, že metoda  $\langle o \text{ kolik starší než:} \rangle, \dots$  ve svém těle dvakrát využívá služeb metody  $\langle \text{věk} \rangle, \dots$ ; jednou u sama sebe (příjemce zprávy Jan Novák) a podruhé v parametru zprávy (Ivana Horáková, které se naváže na lambda-proměnnou  $x$ ).

#### 1.2.2.4

### Polymorfismus

Polymorfismus v OOP znamená, že stejná zpráva může vyvolávat různé operace, které se z pohledu toho, kdo zprávu poslal, jeví jako stejné, i když samy o sobě stejné nejsou. Jinými slovy to znamená, že pokud mají dva různé objekty shodné protokoly, nemusí to ještě znamenat, že mají stejnou datovou strukturu a stejné metody. Představme si, že v systému máme tři objekty znázorněné tabulkou:

jméno	příjmení	adresa	datum narození	věk
Jan	Novák	Kolín	10. 3. 1970	37
Ivana	Horáková	Brno	26. 1. 1982	25
Petr	Horák	Brno	16. 4. 2006	1

Tab. 2: Objekty v tabulce

Všechny tři objekty mají protokoly, do kterých patří prvky

[*jméno, příjmení, adresa, datum narození, věk*].

Z tohoto důvodu mohly být zobrazeny pohromadě v jedné tabulce. Sloupce této tabulky reprezentují to, co příslušné protokoly mají společné. To, že v tabulce vidíme údaje dané protokolem, ale ještě neznamená, že po poslání zpráv  $\langle$ jméno,  $\langle$ příjmení,  $\langle$ adresa,  $\langle$ datum narození a  $\langle$ věk musí všechny objekty reagovat stejně. Věk třeba počítají stejnou metodou, ale s adresou to již tak jednoznačné není. Třetí objekt totiž představuje malé dítě, které asi nebude bydlet jinde než jeho matka. Představme si, že do dat tohoto objektu patří i údaj *matka*, například jako

[*matka: Ivana Horáková*]  $\subset$   $\Delta$ (Petr Horák).

Potom k tomuto objektu můžeme vytvořit metodu, která říká, že adresa této osoby je dána adresou jeho matky:

$\langle$ adresa,  $\sigma$ matka $\langle$ adresa $\rangle \in$  Meth(Petr Horák).

V naší tabulce tedy můžeme mít společně objekty s různou strukturou a různými metodami. Dospělé osoby mají adresu danou svými daty, nezletilé děti mají adresu počítanou z adresy jejich matky. Tento strukturální rozdíl mezi objekty ale na tabulce není vidět, protože všechna data se v ní zobrazují stejně.

Na našem příkladu jsme si tak ukázali, že lze společně pracovat s objekty, které mají různou strukturu i chování. Jedinou postačující podmínkou, aby takové objekty mohly být například pohromadě jako prvky nějaké množiny, je neprázdný průnik jejich protokolů (tj. protokoly musejí mít něco společného).

Tento příklad nám také ukázal, že tabulka je jen výhodný způsob zobrazení dat objektů, jejichž implementace může být různá. To je právě typická vlastnost softwarových prostředí založených na čistém OOP. U smíšených systémů to bohužel tak není. Například v relačních databázích by náš příklad s více „druhy“ objektů v jedné tabulce nebylo možné realizovat. Proto jsou uživatelé smíšených systémů často přesvědčováni, že se musí datový model „doplňovat“ třeba o další tabulky nebo druhy objektů nebo vazby, které sice nevyplývají z čistě abstraktního pohledu na problém, ale jsou nezbytné pro jeho počítačovou implementaci.

### 1.2.3

## Datové modelování s objekty

Při tvorbě datového modelu klasickým způsobem se snažíme prvky reálného světa zobrazit do předem připravených struktur pevně daného druhu. U objektů je tomu obráceně; pro prvky reálného světa si vytváříme nové objekty, které se jim podobají. Do těchto objektů potom vkládáme nejen údaje (například jméno, datum narození...), ale i chování, které potřebujeme (například jak spočítat věk, jak určit adresu...).

Datové modelování s pomocí objektů není pouhá tvorba seznamů nebo tabulek s jednotlivými údaji typu adresa, jméno, cena, název... Při tvorbě systému s objekty se snažíme vytvořit umělý obraz části reálného světa.

### 1.2.3.1 Kolekce objektů

Kolekce objektů jsme si již ukázali na příkladě s osobami. Tabulka, kterou jsme viděli, byla ve skutečnosti zobrazením nějaké množiny tří objektů. Pro větší přesnost v OOP raději nehovoříme o množinách objektů, ale o kolekcích objektů. Je to proto, že jen jeden z více možných druhů kolekcí má vlastnost matematických množin. V praktických systémech opírajících se o čisté OOP můžeme najít velké množství druhů kolekcí. (Například ve Smalltalku jich je více než 120.) Pro základní potřebu však vystačíme se třemi druhy:

- ↗ Množina (anglicky „Set“) je kolekce, ve které prvky nemají žádné uspořádání. Přidává-li se do této kolekce prvek, který v kolekci již je, zůstává v této kolekci jen jednou. Tento druh kolekcí jako jediný přesně odpovídá matematickému pojetí množin.
- ↗ Ranec (anglicky „Bag“) je kolekce, ve které prvky také nemají žádné uspořádání. Přidává-li se ale do této kolekce prvek, který v kolekci již je, objeví se v této kolekci po takovém přidání více kopií tohoto prvku. Jiný v českém jazyce používaný název je „batoh“. (Pecinovský 2006)
- ↗ Seznam (anglicky „List“) je kolekce, která se chová jako ranec a navíc zachovává vnitřní uspořádání prvků, které obsahuje. Na rozdíl od rance i množiny proto v této kolekci můžeme najít první, druhý... a poslední prvek.

Druh kolekce je možné měnit. To znamená, že máme-li například ranec s nějakými objekty, tak můžeme tento ranec přeměnit třeba na seznam obsahující stejné prvky jako ranec. Tyto přeměny (cizím slovem konverze) lze zapsat jako funkce:

$Set(A)$	přemění kolekci $A$ na množinu.
$Bag(A)$	přemění kolekci $A$ na ranec.
$List(A)$	přemění kolekci $A$ na seznam.
$List(A, \Lambda)$	přemění kolekci $A$ na seznam s prvky seřazenými podle hodnoty dané lambda-výrazem $\Lambda$ .

Takové konverze jsou velmi užitečné pro praktickou práci s daty. Mějme například ranec objektů  $X$ , o kterém víme, že obsahuje více kopií stejných hodnot. Formulace výpočtu, kolik různých hodnot se v něm vyskytuje, je poměrně obtížná algoritmická záležitost, protože musíme vymyslet, jak vyloučit všechny možné nadbytečné kopie prvků. S použitím konverze to je ale velmi jednoduché:

$|Set(X)|$ .

Stačí totiž jen konvertovat ranec na množinu. Počet prvků této množiny již určuje počet různých hodnot v původní kolekci.

Jiný příklad je setřídění množiny podle nějakého kritéria. Následující zápis říká, že kolekce objektů *Zákazníci* se setřídí podle věku:

$List(Zákazníci, (\lambda z \mid z.<věk))$ .

Lambda-výraz  $(\lambda z \mid z.<věk)$  slouží k určení hodnoty, podle které jsou prvky v seznamu uspořádány.

### 1.2.3.2

## Třídy objektů, instance tříd, extenze třídy

V předchozím příkladě s dospělými osobami a s nezletilými dětmi jsme se setkali s různými druhy objektů, které se lišily strukturou dat i metodami. Takové druhy objektů se v OOP nazývají třídy objektů. Třída objektů slouží tomu, abychom mohli strukturu dat a množinu metod patřících jednomu druhu objektů jednoduše popsat společně pro všechny objekty, které do příslušného druhu patří. Třída sama o sobě je v čistých objektových prostředích také objektem, který může mít vlastní data i metody a může patřit jiné třídě. Tuto abstrakci, která nás posunuje do oblasti metamodelování, však pro výklad základních vlastností objektového modelování nepotřebujeme.

Třidu každého objektu můžeme zapisovat jako funkci

$\xi(x)$ ,

kde  $x$  je objekt, pro který třídu zjišťujeme. Z definice třídy vyplývá, že pokud dva objekty (například  $a$  a  $b$ ) náleží stejné třídě, mají i stejné metody a také i protokoly:

$\forall a, b \in \Omega \quad \xi(a) = \xi(b) \rightarrow Meth(a) = Meth(b)$ .

$\forall a, b \in \Omega \quad \xi(a) = \xi(b) \rightarrow \Pi(a) = \Pi(b)$ .

Symbolem  $\Omega$  budeme označovat množinu všech objektů v systému. Podobně jako v matematice platí dohoda, že například  $N$  značí množinu všech přirozených čísel,  $R$  množinu všech reálných čísel apod.

Je třeba si dobře uvědomit, že výše uvedená tvrzení obráceně neplatí. Můžeme tedy mít různé objekty se stejným protokolem, které nepatří do stejné třídy, i různé objekty se stejnými metodami, které nepatří do stejné třídy. Tato vlastnost, kdy

$\exists a, b \in \Omega \quad Meth(a) = Meth(b) \rightarrow \xi(a) \neq \xi(b)$  a především

$\exists a, b \in \Omega \quad \Pi(a) = \Pi(b) \rightarrow \xi(a) \neq \xi(b)$

totiž nevyjadřuje nic jiného než polymorfismus, o kterém jsme hovořili v předchozím textu.

Objekty, které přísluší nějaké třídě, jsou v terminologii OOP označovány jako instance třídy. Pokud si zavedeme funkci *Inst*, která dává množinu všech instancí dané třídy, platí vztah

$$\forall a, b \in \Omega \quad a = \xi(b) \leftrightarrow b \in \text{Inst}(a).$$

Taková množina všech instancí dané třídy se nazývá extenze třídy.

### 1.2.3.3

## Třídy versus kolekce

Je bohužel třeba říci, že pojem třídy v OOP není zcela totožný s pojmem třídy v obecné matematice. Matematikové totiž pod pojmem třídy rozumějí vymezení skupiny prvků majících nějakou vlastnost. A množiny prvků jsou v matematice takové třídy, které splňují další podmínky vyplývající z matematické definice množin. Jednoduše řečeno, třída je pro matematiky obecnějším pojmem než množina. To znamená, že každá množina je třídou, ale ne každá třída je množinou. V OOP pojmy třídy a množiny (či kolekce) tento vzájemný vztah nemají.

Na třídy v OOP je proto nejlepší nahlížet jako na popis toho, jaké druhy objektů v systému potřebujeme. Pomocí tříd v modelu vyjadřujeme, že potřebujeme například pracovat s osobami, dětmi, fakturami, platbami... Pro každou takovou třídu potom definujeme, jakou budou mít její objekty strukturu a metody.

Kolekce v OOP potom vyznačují, jak jsou objekty v systému hromadně organizovány. Kolekcemi se tedy popisuje, že potřebujeme pracovat například se seznamy osob, množinami objednávek, soupisem zboží, kartotékou pacientů atp. Zjednodušeně řečeno jde o to, že třídami modelujeme to, JAKÁ informace je potřeba, a kolekce modelujeme KDE a JAK bude tato informace hromadně uchováována a zpracováována.

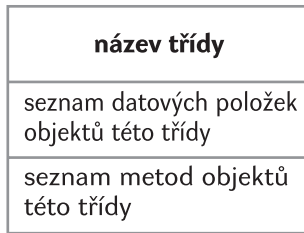
V našem příkladu s dospělými osobami a dětmi jsme měli jednu kolekci, ve které byly objekty dvou různých tříd. Rovněž tak je možné modelovat více kolekcí obsahujících objekty stejné třídy. Jde tedy o to, že pojem třídy a kolekce je na sobě zcela nezávislý. To ale platí jen v čistých objektových systémech. Ve smíšených systémech jsou kolekce závislé na třídách a není například možné mít kolekci, která obsahuje objekty různých tříd. V některých případech dokonce smíšené systémy nedovolí pracovat s jinými množinami, než jsou extenze tříd.

### 1.2.3.4

## Grafické zobrazení

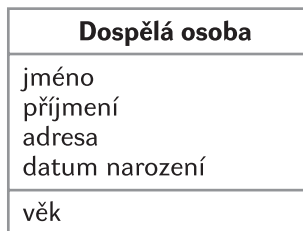
Model skládající se z tříd a kolekcí lze vyjádřit i graficky v podobě diagramů. Používaným standardem v praxi je tzv. diagram tříd, který je součástí většího standardu UML. Podle UML se však pro třídy i kolekce používá stejný symbol, přičemž sdělení, zda jde o třídu nebo o kolekci, je třeba řešit doprovodným textovým popisem. Pokud modelujeme s vědomím, že systém bude používat smíšenou technologii, tak to tolik nevádí, protože nám prostředí stejně dovolí modelovat pouze kolekce nad jednotlivými třídami.

Standard UML pro zobrazení třídy používá následující symbol obdélníka se dvěma přeplázkami:



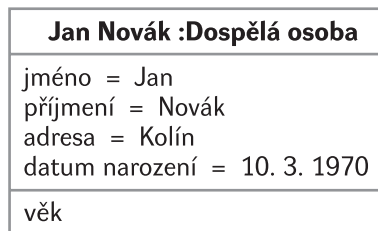
Obr. 3: Symbol třídy podle UML

Třídu dospělých osob z našeho příkladu tak můžeme zobrazit jako



Obr. 4: Konkrétní třída podle UML

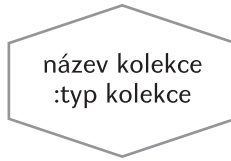
Standard UML také dovoluje graficky zobrazit i konkrétní instance příslušných tříd. Například objekt Jan Novák z našeho příkladu lze znázornit jako



Obr. 5: Instance třídy podle UML

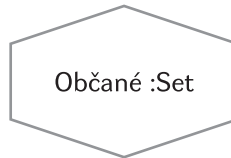
Symbole pro třídu a její instanci jsou v UML téměř shodné. Rozdíl je v tom, že u instancí má hlavička s názvem podtržený text a u datových položek se přidává konkrétní hodnota.

Pro kolekci můžeme použít stejné symboly s tím, že v hlavičce přepíšeme, že se jedná o kolekce. Každá kolekce je totiž také objektem. Lepší je však mít vlastní symbol. Pro kolekce byl dokonce takový symbol v polovině 90. let zaveden (v návrhu UML 0.8), ale v pozdějších verzích UML byl odstraněn. Analytici jej zřejmě pod tlakem smíšeného přístupu nepoužívali. Současný standard UML 2.0 je však otevřenější a dovoluje definici vlastních nových symbolů. Tak je možné používat i dříve zrušené symboly. Protože zde modelujeme podle čistého OOP, použijeme pro kolekce symbol:



Obr. 6: Symbol kolekce

Budeme-li mít například množinu občanů s objekty z našeho příkladu, tak příslušný symbol v diagramu bude vypadat takto:



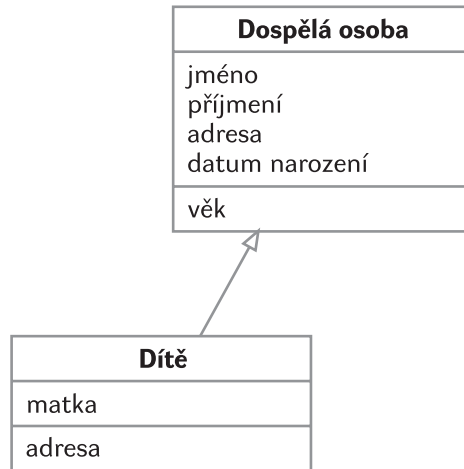
Obr. 7: Konkrétní kolekce

### 1.2.3.5

## Hierarchie dědění objektů

Dědění objektů je velmi oblíbeným nástrojem při smíšeném i čistém objektovém programování. Je to vlastnost, která může být přímo mezi objekty bez účasti tříd, ale v praxi se nejčastěji používá mezi třídami objektů. To bude i náš případ. Dědění mezi dvěma třídami znamená, že definice jedné třídy využívá definici druhé třídy. Třída, která má definici nejen pro sebe, ale i pro jiné třídy, se nazývá nadtřída. Třída, která takovou definici jiné třídy používá, se nazývá podtřída.

V datovém modelování můžeme dědění mezi třídami výhodně použít v případě, kdy při konstrukci nějaké nové třídy chceme znovu použít nějakou již dříve zavedenou třídu, protože nechceme stejnou informaci (o struktuře a metody) vytvářet znovu. V našem příkladě to může být dědění mezi třídami *Dospělá osoba* a *Dítě*. Třída *Dítě* definuje stejnou strukturu dat a metod jako třída *Dospělá osoba*, jen má navíc datovou položku *matka* a metodu *adresa*. Pomocí dědění od nadtřídy *Dospělá osoba* do podtřídy *Dítě* zařídíme, že definice třídy *Dítě* bude moci být interpretovaná „*Dítě* má všechno stejné jako *Dospělá osoba*, jen má navíc položku *matka* a novou metodu *adresa*“. V UML se dědění vyznačuje šipkou s trojúhelníkovitou hlavičkou směrem od podtřídy k nadtřídě:



Obr. 8: Dědění podle UML

Jak je vidět na obrázku, v symbolu podtřídy *Dítě* už nepíšeme informaci, která je shodná s informací, kterou popisuje nadtřída *Dospělá osoba*.

Popíšeme si nyní vlastnosti dědění mezi třídami podrobněji. Nejprve si zavedeme funkci *super*, která dává nadtřídou k třídě. V našem příkladu například platí, že

$$\text{super}(\text{Dítě}) = \text{Dospělá osoba}.$$

Dědění lze potom definovat pomocí vlastností metod jako

$$\forall a, b \in \Omega \quad a = \text{super}(b) \rightarrow \text{Meth}(a) \subseteq \text{Meth}(b) \text{ a proto také}$$

$$\forall a, b \in \Omega \quad a = \text{super}(b) \rightarrow \Pi(a) \subseteq \Pi(b).$$

Jinými slovy to znamená, že všude tam, kde můžeme použít instance nadtřídy, mohou být i instance podtřídy, protože se od těch prvních liší pouze v tom, že umějí ještě něco navíc.

Na závěr tohoto odstavce je třeba poznamenat, že analytik by měl s děděním nakládat opatrně. Dědění mezi třídami objektů je dobrý nástroj, který ale svádí k použití i tam, kde se nehodí. Není například vhodné dědit třídu *Osoba* od třídy *Město* jenom proto, že osoby mají jména stejně jako města. Tyto dvě třídy totiž reprezentují natolik odlišné objekty, že je lepší tuto jedinou společnou vlastnost definovat u každého zvlášť a spolehnout se na polymorfismus. Bohužel, ve většině smíšených systémů si takový luxus s polymorfismem bez dědění nemůžeme dovolit a tak se nadtříd implementujících „jakýkoliv objekt, který zná jméno“ nevyhneme. Každopádně je ale vždy rozumnější nejprve všechny potřebné třídy v systému nejprve samostatně rozpoznat a dobře popsat a teprve potom mezi nimi vyhledávat podobnosti a vytvářet hierarchii dědění. Situace je ještě o to složitější, že hierarchie dědění je ve své podstatě jen implementační nástroj a o implementaci při analýze nejde. Proto bude dále v textu ještě jednou rozebírána tato problematika z pohledu postupně se transformujících hierarchií objektů od abstraktního pohledu na implementační model.

### 1.2.3.6 Skládání objektů

Již víme, že objekty mají datové položky. V našem příkladu to jsou údaje *jméno*, *příjmení*, *datum narození*, *matka*... Na tyto údaje „uvnitř“ objektů můžeme nahlížet také jako na objekty, takže z ryze abstraktního pohledu je skládáním objektů každá položka včetně jednoduchých dat typu číslo, datum, text apod. V čistých objektových prostředích se to dokonce výhodně používá při programování.

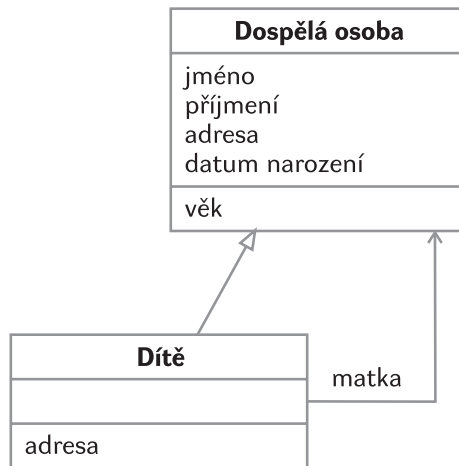
Náš příklad ale platí stejně bez ohledu na to, zda pracujeme s čistým nebo se smíšeným prostředím. Ivana Horáková je matkou dítěte Petr Horák a tato skutečnost je realizována jako

$[matka: Ivana Horáková] \subset \Delta(\text{Petr Horák})$ .

Nyní už víme, že

$\xi(\text{Ivana Horáková}) = \text{Dospělá osoba}$  a  
 $\xi(\text{Petr Horák}) = \text{Dítě}$ .

Jeden z údajů v objektu Petr Horák třídy *Dítě* obsahuje hodnotu, která je sama o sobě objektem třídy *Dospělá osoba*. Právě takový vztah se označuje jako skládání objektů a lze jej v UML vyjádřit takto:



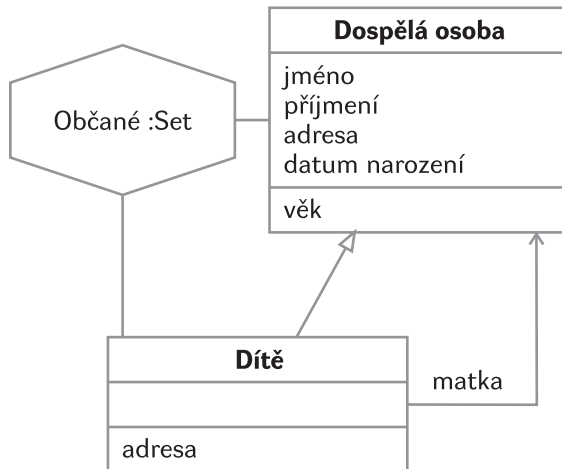
Obr. 9: Skládání podle UML

Pojmenovanou šipkou, která je na obrázku, se v UML říká, že instance třídy *Dítě* mají položku *matka*, která obsahuje instanci třídy *Dospělá osoba*. Je-li položka takto vyznačena pomocí vazby, nepíše se již do seznamu položek ve druhém oddíle symbolu třídy.

Jazyk UML má ještě jiné zobrazení skládání, kdy se na straně skládajícího objektu kreslí na konci čáry kosočtverec. To má význam pro podrobný návrh, protože se tím

zobrazuje, že skládající objekt fyzicky obsahuje skládaný objekt jako svoji složku. V modelech v této knize to nepotřebujeme.

Na závěr se ještě podívejme společně na kolekce, třídy, dědění a skládání. Pokud budeme předpokládat, že objekty z našeho příkladu jsou uloženy v kolekci *Občané*, tak si lze celý model pomocí UML znázornit takto:



Obr. 10: Diagram UML s kolekcí

Diagram nám říká, že kolekce *Občané* je typu *Set* a že obsahuje instance tříd *Dospělá osoba* i *Dítě*. Vztah mezi objekty a jejich kolekcí lze také považovat za skládání objektů mezi sebou.

### 1.2.3.7

#### Data a operace s nimi

Jak bylo řečeno v úvodu knihy, datové modelování slouží k sestavení struktury objektů, která slouží k řešení nějakého problému. Mnoho lidí si zjednodušeně myslí, že k tomu stačí vyrobit obrázek s diagramem tříd řešeného problému. To je samozřejmě nedosta- tečné, protože model se musí nějak ověřit a vyzkoušet. Do určité míry lze na dobře sesta- vený datový model nahlížet jako na funkční prototyp informačního systému, který budou později programátoři dělat.

Pro umění datově modelovat proto nestačí navrhnout potřebné třídy, kolekce a vztahy mezi nimi, ale je třeba ještě vložit do této struktury nějaká skutečná data a pracovat s nimi. Zde se datové modelování dotýká databázové technologie. Na rozdíl od databá- zové technologie ale pracujeme jen s modelovacím softwarem, takže se nemusíme trápit fyzickým uložením dat, transakcemi, komunikací mezi klientem a serverem atd. konkré- tních databázových systémů a můžeme se tím pádem plně soustředit na vlastní řešení pro- blém a zůstat na vysoké úrovni abstrakce.

Správnost datového modelu nejlépe ověříme pomocí operací s kolekcemi objektů. Takovým operacím se v databázové technologii říká dotazy nad bází dat. Zde to je

podobné. Dotazy samozřejmě nevolíme náhodně, ale formulujeme je z potřeb zadání, pro které byl diskutovaný datový model sestaven.

## selection (česky „výběr“)

První operací je selekce kolekce. Selekcí je operace, která z nějaké kolekce sestavuje její podmnožinu na základě nějaké podmínky. Výběr budeme zapisovat takto:

*kolekce // podmínka.*

Pro formulaci podmínky je výhodné použít lambda-výraz. Následující příklad ukazuje operaci, která z kolekce všech občanů vybírá občany starší 60 let:

*Občané // ( $\lambda x \mid x \text{ věk} > 60$ ).*

Ve výsledné podmnožině budou jen ty prvky z množiny *Občané*, které při aplikaci na lambda-výraz ( $\lambda x \mid x \text{ věk} > 60$ ) daly kladný výsledek.

Dalším příkladem je například výběr dospělých osob žijících v Brně:

*Inst(Dospělá osoba) // ( $\lambda x \mid x \text{ adresa} = \text{Brno}$ ).*

Množinu dospělých osob jsme získali získáním všech instancí třídy *Dospělá osoba*. Pokud bychom měli jistotu, že v kolekci *Občané* jsou kromě jiných objektů také všechny instance třídy *Dospělá osoba*, mohli bychom stejný dotaz formulovat přímo nad touto kolekcí:

*Občané // ( $\lambda x \mid \xi(x) = \text{Dospělá osoba} \wedge x \text{ adresa} = \text{Brno}$ ).*

## collection (česky „sběr“)

Další operací je sběr kolekce. Pomocí této operace můžeme prvky nějaké kolekce přeměnit na jiné. Jinými slovy jde o vytvoření jedné kolekce pomocí druhé kolekce tak, že „posbíráme“ výsledky transformací jednotlivých prvků druhé kolekce. Tuto operaci zapisujeme takto:

*kolekce >> transformace prvků.*

Transformace prvků se také výhodně zapisuje lambda-výrazem. Následující příklad ukazuje dotaz, který zjišťuje matky všech dětí:

*Inst(Dítě) >> ( $\lambda x \mid x \text{ matka}$ ).*

Pomocí lambda-výrazu ( $\lambda x \mid x \text{ matka}$ ) bude každé dítě „transformováno“ na svoji matku. Tím získáme z kolekce dětí kolekci jejich matek. Operace sběr není v běžné databázové technologii tak známá a používaná jako výběr. Je tomu tak proto, že transformace prvků je závislá na vlastnostech struktury objektů. Kdybychom totiž nepoužili skládání

objektů pro vyjádření vztahu mezi matkou a dítětem, nemohli bychom sber použít. Na druhou stranu je tato operace velmi užitečným nástrojem v rukou toho, kdo ji umí využít.

Druhý příklad je dotaz na věk nejmladšího občana. S využitím sberu převedeme množinu občanů na množinu jejich věků a pak už jen stačí z této množiny najít minimální prvek:

$$\min(\text{Občané} \gg (\lambda x \mid x \triangleleft \text{věk})).$$

## projection (česky „projekce“)

Tato operace se hojně používá v databázové technologii. Pro datové modelování, kde nepotřebujeme příliš optimalizovat rozhraní mezi uživatelem a počítačem, ji tolik nepotřebujeme. Jde o operaci, která je podobná sberu. Můžeme dokonce prohlásit, že projekce je zvláštním případem sberu, protože neprovádí transformaci prvků na jiné obecným lambda-výrazem, ale jen prvky transformuje tak, že jim zmenšuje protokol. Laicky tato operace vypadá, jako kdyby se z tabulek se zobrazenými objekty vybíraly sloupce. Operaci zapisujeme takto:

$$\text{kolekce} \gg \text{seznam atributů}.$$

Budeme-li například mít kolekci všech občanů, tak její zobrazení v tabulce bude mít tolik sloupců, kolik má společná část protokolů prvků této kolekce. Prakticky to může být velké množství, a tak se zobrazení stává nepřehledné. Pomocí této operace můžeme výslednou tabulku omezit například jen na dva sloupce – příjmení a adresa:

$$\text{Občané} \gg [\text{příjmení}, \text{adresa}].$$

Jak již bylo řečeno, pro práci s modelem není tato operace tak důležitá, protože při datovém modelování nejde o testování detailů zobrazení uživatelských výstupů. V databázové praxi je však tato operace velmi užitečná.

## další operace s daty

Dalšími operacemi s kolekcemi objektů jsou manipulace s kolekcemi objektů, které známe z běžné matematiky. Jde o sjednocení kolekcí, průnik kolekcí, rozdíl kolekcí a kartézský součin kolekcí. Pokud je potřebujeme, tak je zapisujeme standardními symboly:

$$A \cup B, A \cap B, A - B \text{ a } A \times B.$$

V relační databázové technologii tolik oblíbené spojení kolekcí můžeme popisovat jako výběr z kartézského součinu:

$$(A \times B) // A.$$

Této operaci se však raději v objektovém datovém modelování vyhýbáme a dáváme přednost skládání objektů mezi sebou.

## 1.2.3.8

## Změny protokolu kolekcí při operacích s daty

U objektové operace výběru i dalších je velmi zajímavé to, jak se vlivem této operace můžou měnit protokoly kolekcí objektů. Pro protokol kolekce  $A$  objektů  $a_1, a_2, \dots, a_n \in \Omega$  platí vztah:

$$\forall A [a_1, a_2, \dots, a_n] \Pi(A) = \Pi(a_1) \cap \Pi(a_2) \cap \Pi(a_2) \cap \dots \cap \Pi(a_n),$$

Což znamená, že protokol kolekce objektů je roven průniku protokolů objektů, které jsou prvky této kolekce. Prakticky to značí, že při zobrazení příslušné kolekce formou tabulky bude mít tato tabulka sloupce, které jsou dány společnými atributy všech objektů v kolekci obsažených.

## změna protokolu při operaci výběru

Mějme kolekci  $B$ , která je výsledkem výběru z kolekce  $A$  pomocí lambda-výrazu  $\Lambda$  jako

$$B = A // \Lambda.$$

Potom pro vztah mezi protokoly obou kolekcí platí, že

$$\Pi(A) \subseteq \Pi(B).$$

důkaz (sporem):

Nechť  $B = A // \Lambda$ . Potom lze prohlásit, že  $B = A \cup X$ , kde  $|X| \geq 0$ . Předpokládejme, že  $\Pi(A) \supset \Pi(B)$ , tedy že  $\Pi(B \cup X) \supset \Pi(B)$ . Z definice protokolu kolekce musí pro  $X = [x_1, x_2, \dots, x_n]$  platit, že

$$(\Pi(B) \cap \Pi(x_1) \cap \Pi(x_2) \cap \dots \cap \Pi(x_n)) \supset \Pi(B),$$

což odporuje vlastnostem průniku množin.

Prakticky to znamená, že budeme-li mít tabulku zobrazující nějakou kolekci objektů, tak po výběru z této kolekce může být výsledek zobrazen do tabulky, ve které pak jako by automaticky dojde k přidání dalších sloupců. To si můžeme ukázat i na našem příkladu s kolekcí *Občané*, která obsahuje instance tříd *Dospělá osoba* a *Dítě*. Tato kolekce zobrazená tabulkou vypadá takto:

jméno	příjmení	adresa	datum narození	věk
Jan	Novák	Kolín	10. 3. 1970	37
Ivana	Horáková	Brno	26. 1. 1982	25
Petr	Horák	Brno	16. 4. 2006	1

Tab. 3: Kolekce občanů

Provedeme-li například výběr osob mladších 5 let,

$Občané // (\lambda x | x.věk < 5)$

tak dostaneme novou kolekci, která obsahuje již jen instanci třídy Dítě. Dojde tak k přirozenému rozšíření protokolu o atribut matka a výsledné zobrazení v tabulce vypadá takto:

jméno	příjmení	matka	adresa	datum narození	věk
Petr	Horák	Ivana Horáková	Brno	16. 4. 2006	1

Tab. 4: Kolekce občanů po výběru

Některé transformace dat tak může dobře navržený datový model dělat sám ze své podstaty a uživatel se tím zbaví řady operací projekce, sběr a nebo kartézský součin. Tato vlastnost „automatických“ změn protokolů po operacích s kolekci objektů je v praxi málo známá. Může za to dominance relačního přístupu v databázové technologii, který tyto vlastnosti nemá. Pokud ale analytik o této možnosti ví, může ji při návrhu datového modelu dobře využít, i když se třeba potom musí při implementaci systému nějak transformovat kvůli smíšené technologii.

Podívejme se proto ještě stručně na změny protokolů kolekcí u některých dalších operací:

### změna protokolu při operaci sjednocení kolekcí

Protože na rozdíl od tradiční „relační“ představy můžeme sjednocovat i kolekce s různými protokoly, tak pro protokoly platí vztah

$$\Pi(A \cup B) \subseteq \Pi(A), \Pi(B) \text{ nebo také}$$

$$\Pi(A \cup B) = \Pi(A) \cap \Pi(B).$$

### změna protokolu při operaci průnik kolekcí

Rovněž tak můžeme vytvářet průniky kolekcí s různým protokolem. Proto pro protokoly platí, že:

$$\Pi(A \cap B) \supseteq \Pi(A), \Pi(B)$$

## změna protokolu při operaci rozdíl kolekcí

Při rozdíl kolekcí se protokoly chovají podobně jako při průniku kolekcí:

$$\Pi(A-B) \supseteq \Pi(A).$$

### 1.2.3.9

## Jak správně použít kolekce, atributy a skládání

Na pojem třídy se v OOP pod vlivem jejich implementace v programovacích jazycích nahlíží současně dvojným způsobem:

Třída je realizace objektového typu; ve třídě uchováváme popis struktury objektů tohoto typu a množinu jeho operací/metod. V čistě objektových jazycích je toto dokonce implementováno také pomocí objektů, což znamená, že třída objektů je také sama o sobě objektem, který samozřejmě není totožný s žádnou instancí třídy ani jejich množinou:

$$a \neq \text{Inst}(a) \text{ a ani}$$

$$a \notin \text{Inst}(a).$$

Úplně jiný pohled na třídu je ale chápání třídy jako množiny všech instancí, které dané třídě náleží. Tento pohled třídu nesprávně ztotožňuje s jejím extentem:

$$a = \text{Inst}(a).$$

Naneštěstí se v metodách opírajících se o UML oba způsoby nazírání na pojem třídy mísí dohromady a v různých kontextech platí různá interpretace. To vede k tomu, že většina analytiků množiny (kolekce) objektů modeluje pomocí tříd.

Zajímavou otázkou je, zda v modelu potřebujeme pro všechny skupiny objektů vytvářet kolekce. Každá třída má totiž svůj extent, a proto může být zbytečné všechny její instance ještě vkládat do zvlášť vytvořené kolekce.

Rovněž tak můžeme kolekce nahradit nějakou operací, která nám objekty najde z jiných kolekcí. V našem příkladu s občany například nemusíme vytvářet speciální kolekci „dospělých osob z Brna“, protože tuto kolekci vždy snadno získáme výrazem

$$\text{Inst}(\text{Dospělá osoba}) // (\lambda x \mid x \text{<adresa} = \text{Brno}).$$

Mohlo by se tedy zdát, že můžeme vystačit pouze s extenty tříd systému, protože pokud naše objekty obsahují příslušné atributy, můžeme pomocí výrazů získat libovolnou kolekci, která je potřeba. To je do značné míry pravda, ale datový model se konstruuje proto, aby elegantně a srozumitelně splňoval zadání, kvůli kterému byl konstruován.

Právě z tohoto důvodu nemusí být vždy rozumné všechny vlastnosti objektů implementovat jako jejich vlastní atributy. Je třeba vědět, že samotná příslušnost nějakého

objektu v nějaké množině tvoří novou vlastnost tohoto objektu. Proto nemusíme pro vyjádření nějaké vlastnosti vždy vkládat do objektů nová data.

Kromě toho může být stejný objekt prvkem ve více množinách, což dává další možnosti využití při modelování. Nezanedbatelnou roli zde také hraje moment, kdy třídy objektů jsou již vytvořeny, v novém systému je jen znovupoužíváme, uživatel jim potřebuje přidat novou vlastnost, ale pokud by byla implementována přidáním nového atributu do objektů, došlo by ke zbytečnému rozšíření diskutované třídy ve všech jejích výskytech, a proto i tam, kde nový atribut není potřeba.

Například, budeme-li mít datový model evidence žáků nějaké školy, tak u požadavku na doplnění tohoto modelu o evidenci, kteří žáci pojednou na výlet, není rozumné měnit třídu, ale raději pro výletníky vytvořit novou kolekci. To, že žák na výlet pojede, se potom pozná podle toho, že je jeho objekt prvkem této kolekce.

Jednotlivé kolekce můžeme také skládat do jiných objektů. Pojede-li se na náš školní výlet například dvěma autobusy, vytvoříme novou třídu objektů pro autobusy a jejím instancím sestrojíme datovou položku cestující, která bude příslušné žáky obsahovat. Kolekce je zde skryta v podobě hodnoty datové složky objektů reprezentujících autobus.

Tato vlastnost, kdy kolekce jsou schovány pod datovými složkami objektů, je od 70. let známá pod pojmem kontejner (anglicky container). Pod vlivem relačního datového modelu se ale v konceptuálním modelování přestala používat.

## Modelovací jazyk UML

Modelovací jazyk UML (Unified Modeling Language, česky sjednocený modelovací jazyk) se vytváří od roku 1995 jako univerzální standard pro vizuální modelování systémů. Přestože je nejčastěji spokojován s modelováním objektově orientovaných softwarových systémů, může mít mnohem širší využití, protože má zabudované rozšiřovací mechanismy.

Jazyk UML byl navržen proto, aby spojil dosavadní postupy modelovacích technik a softwarového inženýrství. Velmi brzy pro svém představení byl UML akceptován drtivou většinou výrobců modelovacích nástrojů.

Je důležité si uvědomit, že definice jazyka UML neobsahuje žádný druh metodiky modelování. Určité části metodiky můžeme najít v popisech každého z elementů, z nichž se UML skládá, ale samotný jazyk UML jako celek poskytuje pouze standard pro zobrazení a popis modelů.

Klíčové základní práce o objektové analýze a návrhu, které se staly základem UML, se objevily mezi roky 1988 až 1995. Byly to knihy a autoři:

- ↗ Sally Shlaer a Steve Mellor napsali knihu (Shlaer 1992) o analýze a návrhu objektových systémů pomocí modelování procesů s objekty. Jejich přístup se vyvinul v jejich metodu Recursive Design Approach.
- ↗ Peter Coad a Edward Yourdon vyvinuli z Coadovu na prototypování orientovaného přístupu metodu a napsali o ní knihy (Coad 1993) a (Coad et al. 1995).
- ↗ Smalltalkovská komunita v Portlandu přišla se svým odpovědností řízeným návrhem Responsibility-Driven Design a CRC kartami (Wirfs-Brock 1990).
- ↗ Grady Booch vycházel ze své práce ve firmě Rational se systémy založenými na jazyce Ada a publikoval vlastní metodu (Booch 1994).
- ↗ Jim Rumbaugh vedl tým v laboratořích General Electric, kde vyvinuli metodiku OMT – Object Modeling Technique (Rumbaugh et al. 1991).
- ↗ Jim Odell a James Martin napsali knihy na základě svých dlouholetých zkušeností s návrhem business informačních systémů a informačním inženýrstvím (viz Martin et al. 1995).
- ↗ Ivar Jacobson napsal své knihy na základě svých zkušeností s programováním telefonních ústředěn u firmy Ericsson (Jacobson 1992). Ve svých pracích jako první zavedl koncept případu užití (anglicky use case).

### 2.1

#### Struktura UML

UML lze rozdělit na čtyři části (UML 2004):

- ↗ Definice notace UML (syntaxe čili způsob zápisu).
- ↗ Metamodel UML (sémantika čili význam).
- ↗ Jazyk OCL pro popis dalších vlastností modelu.
- ↗ Specifikace převodu do výměnných formátů (CORBA, IDL, XMI).

Notace UML vyjadřuje jeho syntaxi. Při tvorbě modelů pomocí jazyka UML postupně pomocí diagramů vyjadřujeme jednotlivé stránky systému (statickou, dynamickou, funkční). Pro zachycení těchto stránek modelu je v UML devět typů diagramů.

Součástí specifikace UML jsou také definice výměnných formátů dat. Pomocí nich můžeme přenést diagramy vytvořené pomocí UML do jiných modelovacích nástrojů – pro to slouží formát XMI (XML Metadata Interchange).

V UML je definováno devět typů diagramů, rozdělených do dvou skupin podle toho, jestli zachycují dynamickou nebo statickou podstatu systému. Statický model zachycuje objekty a strukturní vztahy mezi nimi. Dynamický model naproti tomu zachycuje způsob, jakým na sebe objekty navzájem působí, aby bylo dosaženo požadovaného chování systému.

Statický model systému lze zachytit pomocí diagramů:

- ↗ diagram tříd,
- ↗ objektový diagram,
- ↗ diagram komponent a
- ↗ diagram nasazení.

Pro zachycení dynamického chování systému lze použít diagramy:

- ↗ diagram případů užití,
- ↗ sekvenční diagram,
- ↗ diagram spolupráce,
- ↗ stavový diagram a
- ↗ diagram aktivit.

## 2.1.1 Diagram tříd

Diagram tříd je nejsložitějším diagramem standardu UML. Dohromady obsahuje několik desítek různých pojmů, které mají uplatnění od analýzy až po popis na úrovni detailu softwarové implementace v nějakém konkrétním programovacím jazyce. Někteří programátoři dokonce žertem říkají, že tento diagram je ve skutečnosti do obrázků překreslený kód v jazyce C++.

Pro datové modelování je rozumné využívat jen malou část z celého repertoáru pojmů UML. Analytický model dat úlohy je totiž především komunikační nástroj pro uživatele, analytiky a vývojáře. Proto by měl vystačit s malým počtem různých pojmů a už vůbec by neměl obsahovat pojmy mající význam jen pro budoucí programovou realizaci.

S UML je proto třeba velmi opatrně zacházet a umět si správně vybrat jen rozumnou část jeho nástrojů.

### 2.1.1.1

## Asociace

Kromě vazeb skládání a dědění, se kterými jsme již pracovali, má diagram tříd ještě jeden pro modelování důležitý abstraktní pojem, který se anglicky nazývá association. Český jej doslova můžeme přeložit jako přiřazení nebo přidružení, ale používaný termín je asociace. Je to vazba mezi objekty, která říká, že na základě zadání mezi objekty v modelu existuje vztah, kdy tyto objekty musejí mít něco společného nebo kdy se vzájemně potřebují, protože si třeba vyměňují nebo sdílejí data.

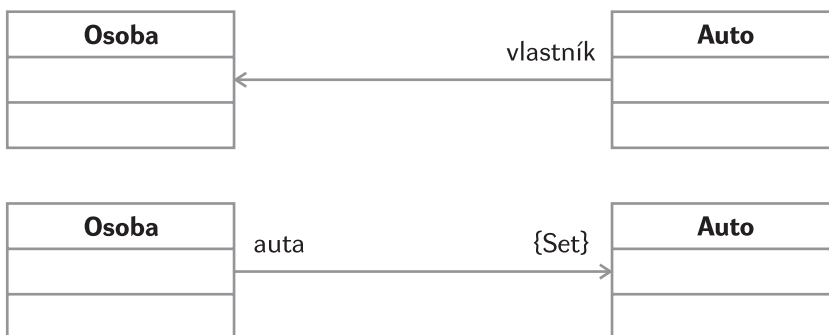
Asociaci si ukážeme na příkladu. Mějme model systému pro evidenci automobilů a jejich vlastníků. V systému proto budeme mít třídu *Osoba* a třídu *Auto*. Mezi objekty těchto tříd bude asociace, kterou vyjádříme, že auta mají vlastníky:

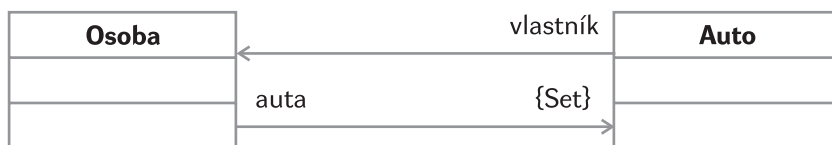


Obr. 11. Příklad asociace mezi objekty podle UML

Asociace se zakresluje jednoduchou čarou bez šipek. Na obou koncích se udává takzvaná kardinalita. Je to vyjádření četnosti vztahu. Například na našem obrázku je napsáno, že každá osoba v systému je vlastníkem jednoho či více aut (údaj 1..\* u aut) a že každá auto má jednoho nebo žádného vlastníka (údaj 0..1 u osob). Černý trojúhelníček je pouze ukazatel směru čtení popisky. Neznamená to, kterým směrem asociace platí a kterým ne – to je velmi častá chyba. Na našem obrázku tedy vidíme, že osoby jsou vlastníky automobilů, tedy že k osobám přísluší auta, a také i obráceně, že k autům patří osoby.

Asociace jsou prvotním vyjádřením vztahu, který může být při podrobnější analýze nahrazen skládáním objektů. Jinou možností, jak splnit požadavky asociace, může být posílání zpráv od jednoho objektu k druhému aniž by se oba objekty nějak přímo skládaly. Pokud ale zůstaneme u skládání objektů, máme hned tři možnosti, jak tuto asociaci mezi osobami a auty realizovat:





Obr. 12: Tři možnosti skládání

Výběr vhodné varianty skládání objektů proto záleží na konkrétním zadání, pro které systém modelujeme. Pokud budeme muset často provádět dotazy nad množinou aut, je nejlepší první varianta, protože vlastník auta je zde skládáním přístupný od jeho auta. Pokud budeme ale častěji pracovat s osobami a na auta nahlížet jen jako na „údaje“ osob, platí druhá varianta s obrácenou vazbou. Třetí varianta kombinuje výhody obou předchozích variant, ale za cenu složitější softwarové realizace a proto ji programátoři neradi vidí.

## 2.1.2

### OCL

Ne všechny vlastnosti lze vyjádřit pomocí diagramů. Ke každému prvku UML můžeme přidat omezení, které přesněji definuje vlastnosti prvků. Pro definici těchto omezení definice jazyka UML obsahuje jazyk OCL (Object Constraint Language).

Pomocí OCL se ke struktuře zobrazené v diagramech doplňují výrazy v podobě logických tvrzení, která musí nad informací z diagramu platit (UML 2004).

Budeme-li mít například objekty s osobami a budeme-li chtít vyjádřit, že žádné dvě různé instance této třídy nemůžou mít stejné jméno, což lze napsat jako

$$\forall p_1, p_2 \in \text{Inst}(\text{Osoba}) \quad p_1 \neq p_2 \rightarrow p_1.\text{jméno} \neq p_2.\text{jméno}.$$

Tento výraz se v OCL píše

```
Osoba.allInstances ->
forAll(p1, p2 | p1 <> p2 implies p1.jméno <> p2.jméno).
```

Budeme-li mít například nějaký datový model zaměstnanců nějaké firmy, můžeme pro vyjádření toho, že zaměstnanci musí být starší 18 let, nad kolekcí *Zaměstnanci* formulovat výraz:

$$\text{Zaměstnanci} // (\lambda p \mid p.\text{věk} < 18) = \emptyset,$$

který v OCL můžeme napsat jako

```
Zaměstnanci.select (p | p.věk < 18) -> isEmpty()
```

OCL zná také všechny v této knize popisované tři druhy kolekcí (Set, Bag a Sequence neboli List), operace výběru (select), sběru (collect) a další včetně konverzí mezi kolekcemi.

## Smalltalk

### 3.1 Jazyk

Syntaxe jazyka Smalltalk je jednoduchá, ale dost odlišná od většiny programovacích jazyků. Na druhou stranu je zápis Smalltalku velmi čitelný a úsporný a do značné míry i blízký matematickému zápisu, jak napoví následující ukázky kódu. Smalltalk je možné použít nejen na programování aplikací, ale i při manipulaci s datovými objekty a také k vyjádření pravidel, jako se používá OCL, který je Smalltalku podobný.

Dnes používaný jazyk je označován Smalltalk-80. To proto, že jeho syntaxe, která se vyvíjela v průběhu 70. let, byla publikována v roce 1980. Od té doby se jazyk i jeho knihovna stále doplňuje, ale označení ST-80 zůstává, protože až na malé výjimky je dodržováno pravidlo zpětné kompatibility kódu (HOPL 1988).

Smalltalk využívá třídě-instanční objektově orientovaný model, tj. skládání objektů, dědění, závislosti mezi objekty, polymorfismu a vícenásobné použitelnosti kódu. Proto je Smalltalk považován za jeden z nejzdařilejších pokusů o implementaci čistěho objektově orientovaného programovacího jazyka. Smalltalk je integrován s programovacím prostředím, které je napsané taktéž v jazyce Smalltalk a přístupné včetně zdrojových kódů. Navenek se systém chová jako jediný rozsáhlý program, nebo ještě lépe jako jeden kompaktní operační systém. Programování ve Smalltalku je založeno na manipulaci s databází objektů, tříd a kolekcí v paměti počítače. Z pohledu uživatele to vypadá, jako by programátor svoje programy postupně vytvářel a doplňoval za jejich chodu.

Kód Smalltalku se skládá pouze ze dvou příkazů. Je to pojmenování objektu a poslání zprávy. Všechny ostatní konstrukty včetně větvení výpočtu, iterací nad sadami objektů, posílání dat či ošetření výjimek ve výpočtu jsou ve skutečnosti jen různé zprávy posílané příslušným objektům. Ve Smalltalku nenajdeme žádné „neobjektově orientované“ příkazy jako ve smíšených jazycích.

#### 3.1.1 Pojmenování

Pojmenování (anglicky *assignment*) je operace, která objektu přiděluje nějaké konkrétní jméno. Budeme-li mít například objekt obsahující číslo 12, můžeme jej pojmenovat například hodnota:

```
hodnota := 12.
```

Je třeba dát pozor na to, že ve Smalltalku se na konci výrazu píše tečka – stejně jako v živých jazycích. Když je objekt pojmenován, tak například výraz

hodnota \* 2 + 1.

nám dá výsledek 25. Čtenář si již jistě uvědomil, že operace pojmenování objektu je analogická operaci přiřazení hodnoty do proměnné ve většině smíšených jazyků. Hlavní rozdíl je v tom, že ve Smalltalku jde o vytvoření abstraktního odkazu na příslušný objekt a ne o kopírování jeho obsahu z jednoho místa paměti na druhé. Proto například výraz

a := b.

nám objekt se jménem b pojmenuje ještě jménem a. V systému bude stále jen jeden objekt, ale dostane dvě jména. Proto pokud je potřeba vytvořit během pojmenování dva objekty, je třeba tuto druhou hodnotu nějak vyrobit, například „naklonováním“ první hodnoty jako:

a := b copy.

Smalltalk používá automatickou správu paměti. To znamená, že pokud nějaká část kódu ukončí svůj výpočet, všechny objekty v této části obsažené jsou smazány z paměti.

## 3.1.2 Zprávy

Zprávy se ve Smalltalku rozdělují do tří kategorií. Jsou to zprávy unární (unary messages), binární (binary messages) a slovní (keyword messages). Pokud nejsou použity kulaté závorky, mají nejvyšší prioritu zprávy unární, potom binární a nakonec slovní.

### 3.1.2.1 Unární zprávy

Unární zprávy jsou takové příkazy objektům, které do objektu neposílají žádná data. Použijeme-li příklad s osobami z předchozí kapitoly, najdeme v něm řadu unárních zpráv

*osoba* <jméno,  
*osoba* <příjmení,  
*osoba* <adresa a nebo  
*osoba* <věk.

Tyto zprávy ve Smalltalku zapíšeme takto:

osoba jméno.  
osoba příjmení.

osoba adresa.  
osoba věk.

### 3.1.2.2

## Binární zprávy

Binární zprávy jsou takové zprávy, které posílají příjemci právě jeden objekt neboli jeden parametr. Jde proto o operace nad dvěma objekty. Prvním objektem je příjemce zprávy a druhým objektem je parametr zprávy. Binární zprávy se typicky používají pro matematické operace. Například smalltalkový výraz

$a + b$ .

se interpretuje jako poslání zprávy

$a \triangleleft plus: b$ .

Tímto způsobem jsou ve Smalltalku implementovány nejen matematické operace sčítání, násobení..., ale i relační operace pro test rovnosti či velikosti. Například výraz

$a > b$ .

který dává hodnotu true či false podle obsahu proměnných  $a$ ,  $b$ , se chápe jako zpráva

$a \triangleleft je\ větší\ než: b$ .

Pro zajímavost si ještě uvedeme, že příslušné metody k těmto binárním zprávám, tedy kódy na sčítání, násobení, porovnávání..., jsou ve Smalltalku součástí databáze stejně jako jakákoliv jiná metoda. Je proto možné do těchto kódů zasahovat, například je doplňovat nebo měnit.

### 3.1.2.3

## Slovní zprávy

Slovní zprávy jsou všechny ostatní zprávy s parametry, které nejsou binární. Je to celá řada zpráv, která se objektům posílá proto, aby se na objekt aplikovala nějaká hodnota a došlo k výpočtu příslušné metody. I samotnou aplikaci hodnoty na lambda-výraz lze považovat za takovou zprávu, protože

$lambda\text{-výraz} \triangleleft: hodnota$

se ve Smalltalku píše jako

$lambdaVýraz\ value: hodnota$ .

Samozřejmě lze ve Smalltalku nalézt příslušnou metodu, která tuto zprávu vykonává. V našem příkladu jsme měli celou řadu slovních zpráv, jako například:

```
osoba<jméno: Jan,
osoba<příjmení: Novák,
osoba<adresa: Kolín
osobaA <o kolik starší než: osobaB.
```

Tyto zprávy se ve Smalltalku píší jako

```
osoba jméno: 'Jan'.
osoba příjmení: 'Novák'.
osoba adresa: 'Kolín'.
osobaA oKolikStaršíNež: osobaB.
```

### 3.1.2.4

#### Kaskáda zpráv

Jsou-li zprávy posílány stejnému objektu, potom je můžeme postupně napsat za sebou v jednu výrazu. Jednotlivé zprávy jsou ve výrazu odděleny středníkem. Předchozí příklad lze proto zapsat i jako:

```
osoba jméno: 'Jan'; příjmení: 'Novák'; adresa: 'Kolín'.
```

### 3.1.2.5

#### Bloky

Důležitou součástí Smalltalku pro práci s objekty jsou bloky výrazů. Bloky výrazů představují softwarovou implementaci lambda-výrazu. Blok výrazů je jako celek také objektem, může být pojmenován, mohou mu být posílány příslušné zprávy, může být použit v jiných výrazech (zprávách) jako příjemce nebo parametr a může být skládán a být třeba i prvkem kolekce.

Například lambda-výraz ( $\lambda x \mid x \cdot 8$ ) pro výpočet ceny nákupu se ve Smalltalku píše jako

```
[:x | x * 8].
```

Aplikace hodnoty 12 na tento lambda-výraz ve tvaru ( $\lambda x \mid x \cdot 8$ ) <: 12 se ve Smalltalku píše

```
[:x | x * 8] value: 12.
```

Bloky je samozřejmě možné pojmenovat. S využitím pojmenování lze stejný příklad napsat například jako

B := [:x | x \* 8].  
 B value: 12.

Bloky lze také používat jako parametry zpráv. Pro datové modelování to je velmi výhodné především v operacích s kolekcemi. Například výraz vybírající osoby starší 60 let

*Občané* // ( $\lambda x \mid x \text{věk} > 60$ )

lze napsat jako

Občané select: [:x | x věk > 60].

Totéž lze napsat s použitím pojmenování také jako

B := [:x | x věk > 60].  
 Občané select: B.

Dalším příkladem byl výběr dospělých osob žijících v Brně:

*Inst(Dospělá osoba)* // ( $\lambda x \mid x \text{adresa} = \text{Brno}$ ),

který se napíše

(DospěláOsoba allInstances) select: [:x | x adresa = 'Brno'].

Funkci *Inst* Smalltalk zná jako unární zprávu allInstances. Také samotné operace s kolekcemi jsou ve Smalltalku implementovány jako metody kolekcí a proto se vyvolávají zprávami. (Zde to je slovní zpráva select: pro operaci výběru) Tento příklad byl popsán i jiným výrazem:

*Občané* // ( $\lambda x \mid \xi(x) = \text{Dospělá osoba} \wedge x \text{adresa} = \text{Brno}$ ).

což se ve Smalltalku napíše jako

Občané select: [:x | (x class = DospěláOsoba) & (x adresa = 'Brno').]

Funkci  $\xi$  zná Smalltalk jako unární zprávu class a operaci logického součinu jako binární zprávu &.

Jako poslední příklad si ukážeme příklad operace sběru, která zjišťuje hodnotu věku nejmladší osoby:

*min(Občané >> ( $\lambda x \mid x \text{věk}$ )).*  
 (Občané collect: [:x | x věk])min.

S pomocí bloků lze psát i logické výrazy popisující podmínky pro kolekce dat. Například výraz omezující spodní věkovou hranici zaměstnanců

$(\text{Zaměstanci} \parallel (\lambda x \mid x.\text{věk} < 18)) = \emptyset,$

Který jsme v OCL psali jako

`Zaměstnanci.select ( x | x.věk < 18) -> isEmpty()`,

můžeme ve Smalltalku zapsat takto:

`(Zaměstanci select: [:x | x věk < 18]) isEmpty.`

### 3.1.3 Zápis metod

Hlavička metody vypadá ve Smalltalku stejně jako zápis zprávy, který tuto metody vyvolává. Proto například zprávě věk přísluší metoda s hlavičkou věk, zprávě adresa: hodnota přísluší metoda s hlavičkou adresa: hodnota atd.

Celý zápis metody vypadá tak, že na první řádek napíšeme hlavičku metody a na další řádek můžeme napsat komentář ohraničený uvozovkami. Potom následuje tělo metody. Na prvním řádku těla metody můžeme napsat seznam jmen pro pomocné objekty, pokud je potřebujeme.

Celkově zápis vypadá takto:

```
hlavička metody
“komentář metody“
  | seznam jmen pomocných objektů |
  tělo metody
```

Například metoda pro výpočet věku osoby, kterou jsme definovali jako

$\langle \text{věk} , (\text{dnešní datum} - \sigma \text{datum narození}) / 365.2422 \rangle,$

se ve Smalltalku může napsat jako

```
věk
“tady je výpočet věku osoby z datumu narození“
  | dnes |
  dnes := Date today.
  ^ ((dnes - datumNarození)/365.2422) truncated.
```

Knihovna objektů Smalltalku totiž obsahuje objekt Date, který po poslání unární zprávy today sdělí hodnotu dnešního datumu. Zpráva truncated slouží k odstranění desetinné části čísla. Díky tomu budeme dostávat stáří v podobě celého čísla.

Dalším prvkem zde vysvětlované syntaxe Smalltalku je znak ^, který se jmenuje anglicky „caret“. Je to označení toho výrazu v těle metody, který určuje návratovou hodnotu po provedení celé metody. Ukažme si ještě, jak můžeme v těle metody použít

jinou metodu stejného objektu. Budeme mít metodu, která testuje, o kolik je starší osoba přijímající zprávu a nebo osoba v parametru:

*(o kolik starší než: , ( $\lambda x \mid \sigma \langle \text{věk} - x \rangle \text{věk}$ )).*

Lambda-proměnná  $x$  se stane součástí hlavičky. Smalltalk zná i objekt, kterému metoda patří pod jménem self, kam lze posílat zprávy stejně jako jiným objektům:

oKolikStaršíNež: x

“kladný výsledek znamená, že příjemce zprávy je starší“

^ self věk - x věk.

### 3.1.3.1

## Přístupové metody

Ve Smalltalku je poslání zprávy jedinou možností, jak pracovat s objekty. Proto se i „čtení“ a „zápis“ datových položek v objektech implementuje pomocí metod. Pro jednoduchost mají tyto metody zpravidla stejné hlavičky jako jména datových položek, které čtou nebo zapisují. Například pro příjmení osoby tyto metody vypadají takto:

příjmení

“metoda pro unární zprávu, která čte hodnotu příjmení“

^ příjmení.

příjmení: text

“metoda pro zprávu, která zapisuje hodnotu příjmení“

příjmení := text.

### 3.1.4

## Řízení výpočtu

Někdy je třeba v zápisu těla metody použít podmínku, která určuje, zda se má nějaká formule provést nebo ne. Zápisem

*logická hodnota* ifTrue:  $\Lambda_1$  nebo

*logická hodnota* ifTrue:  $\Lambda_1$  iffFalse:  $\Lambda_2$

docílíme podmíněné vykonávání kódu. Samozřejmě platí, že příkazy ifTrue: a ifTrue:iffFalse: jsou zprávy, které používají bloky (lambda-výrazy) jako parametry. Například formule

$$c \leftarrow \frac{2a + b \quad b - a}{a > b}$$

podle které je-li  $a$  větší než  $b$ , tak platí  $c \Leftarrow 2a + b$ , jinak platí  $c \Leftarrow b - a$ , se ve Smalltalku napíše jako

$(a > b)$  ifTrue:  $[c := 2 * a + b]$  iffFalse:  $[c := b - a]$ .

a nebo také jako

$c := (a > b)$  ifTrue:  $[2 * a + b]$  iffFalse:  $[b - a]$ .

Pomocí bloků můžeme psát také cykly. Například formule

$$\left\{ \begin{array}{l} c \Leftarrow c + 10 \\ c < 100 \end{array} \right\}$$

podle které se výraz  $c \Leftarrow c + 10$  opakuje tak dlouho, dokud platí podmínka  $c < 100$ , se ve Smalltalku napíše

$[c < 100]$  whileTrue:  $[c := c + 10]$ .

Bloky lze použít i pro hromadné vykonávání lambda-výrazů nad prvky kolekcí. Například výraz, který do proměnné *suma* přičítá hodnotu všech prvků kolekce *A* jako

$\forall a \in A \ (\lambda x \mid \textit{suma} \Leftarrow \textit{suma} + x) \ll a$ ,

se ve Smalltalku napíše takto:

*A* do:  $[:x \mid \textit{suma} := \textit{suma} + x]$ .

### 3.1.5

## Architektura programů ve Smalltalku

Smalltalk nepracuje přímo se strojovým kódem počítače, na kterém běží. Namísto toho překládá programy do byte-kódu, který je za běhu interpretován do hostitelského strojového kódu. Tato koncepce například inspirovala 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 byte-kódu Smalltalku ukládají do vyrovnávací paměti, čímž je dosahováno dobré výkonnosti beze ztráty pružnosti systému. Byte-kód Smalltalku je navíc několikrát kompaktnější než odpovídající strojový kód a je nezávislý na použitém hardwaru počítače. Programy (hotové i rozpracované) jsou přenositelné mezi AppleOS, MS Windows, Linuxem a mnoha typům unixů (Solaris, HPUNIX, AIX, SGI...).

Smalltalk podporuje metaprogramování (třídy mohou být instancemi jiných tříd), paralelní programování, má podporu pro řízení procesů jako např. semaforey nebo vyrovnávací buffery. Z dalších vlastností například má ošetřování výjimek v programu,

sledování verzí kódu během programování s možností návratu do libovolného z přechodných stavů aj. Programátor může do objektů přiřazovat nejen data, ale i kód.

Pro práci ve Smalltalku je také důležité vědět, že třídy mohou být za chodu programu zpracovávány jako data; mohou být parametry zpráv, mohou vznikat, zanikat nebo měnit svůj datový obsah, kterým jsou například i kódy metod nebo vazby dědičnosti.

### 3.1.6 Reflexe

Reflexe je vlastnost, kdy kód umí číst a zpracovávat sám sebe stejně jako jiná data. Smalltalk má tuto vlastnost. V následující tabulce je přehled vybraných užitečných zpráv poskytujících reflexi:

a class.	Zjistí třídu objektu $a$ jako $\xi(a)$ .
c allInstances.	Zjistí extent třídy, tj. množinu všech instancí třídy $c$ jako $Inst(c)$ .
c allSelectors.	Zjistí množinu všech možných zpráv pro instance třídy $c$ jako $\Pi(c)$ .
c superclass.	Zjistí nadtřídu třídy $c$ jako $super(c)$ .
a respondsTo: m.	Zjistí, zda objekt $a$ dokáže přijmout zprávu $m$ jako test zda $m \in \Pi(a)$ .
c methodDictionary.	Zjistí množinu všech metod třídy $c$ jako $Meth(c)$ .
Smalltalk.	Pojmenovaná kolekce všech objektů v systému jako $Smalltalk \Leftarrow \Omega$ .

## 3.2 Vývojová prostředí

Jazyk Smalltalk je dnes dostupný v různých distribucích od Freewaru a GNU až po drahá komerční řešení. V současné době je na trhu cca 20 vývojových prostředí pro Smalltalk od různých dodavatelů. Smalltalk můžeme najít v miniaturní podobě na PDA a nebo také v aplikacích pro výkonné víceprocesorové počítače. Zde se krátce zmíníme o třech nejvýznamnějších implementacích tohoto jazyka.

### 3.2.1 VisualWorks

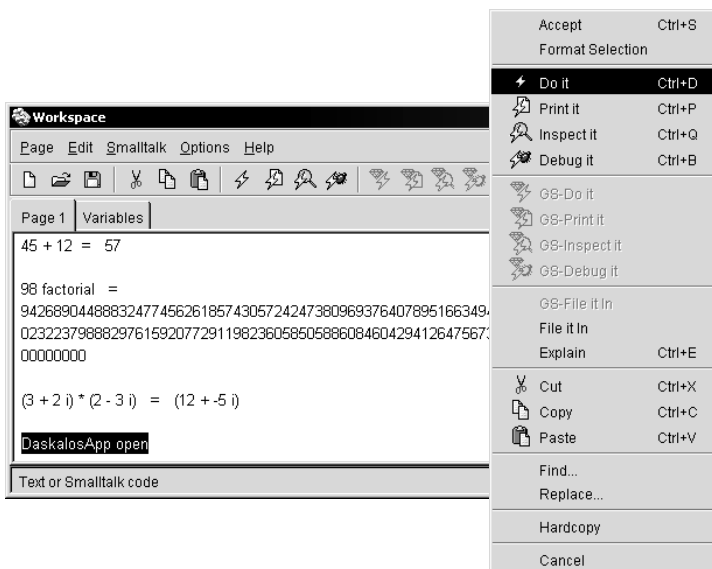
VisualWorks je „vlajkovou lodí“ systémů Smalltalk. Jde o přímého následníka původních systémů ze 70. let. Je to také nejrozšířenější systém v komerčních aplikacích. VisualWorks má řadu doplňkových nástrojů například pro práci s relačními i objektovými databázemi a pro tvorbu internetových aplikací. Podrobnější informaci lze získat na adrese [www.cincom.com/smalltalk](http://www.cincom.com/smalltalk).

Na příkladu systému VisualWorks si nyní popíšeme základní vlastnosti vývojového prostředí Smalltalku.

Jako většina systémů se Smalltalkem, tak i VisualWorks startuje ze souboru s názvem „smalltalk image“, ve kterém je uložený obsah paměti z předchozího spuštění, a to včetně

případných běžících procesů. Smalltalk se tedy chová podobně jako operační systém na notebooku, který lze uspat a znovu nastartovat do stavu, ve kterém byl při předchozím ukončení. Tuto vlastnost mohou mít i aplikační programy. Program ve VisualWorks nebo celý systém je dokonce možné takto „uspat“ například na počítači PC s operačním systémem Windows a poté jej „rozeběhnout“ na počítači Apple nebo na unixové pracovní stanici. Současné verze VisualWorks takto podporují asi 20 různých počítačových platform.

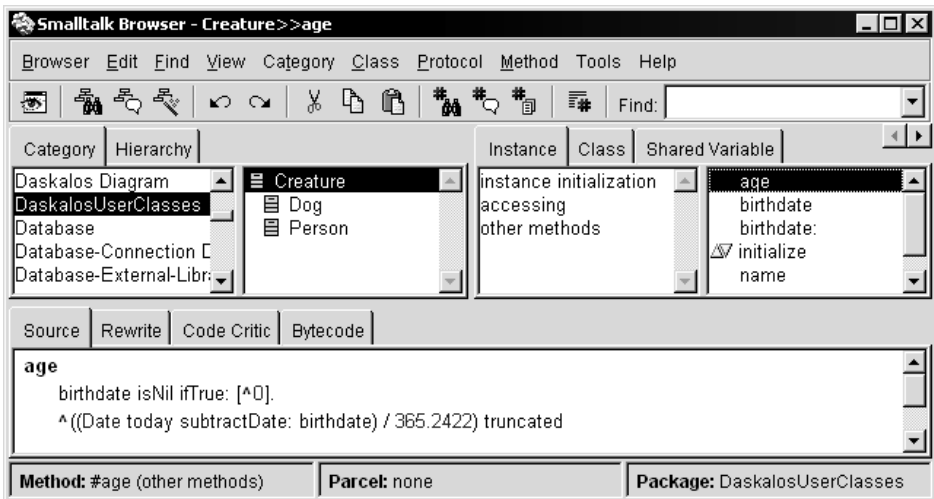
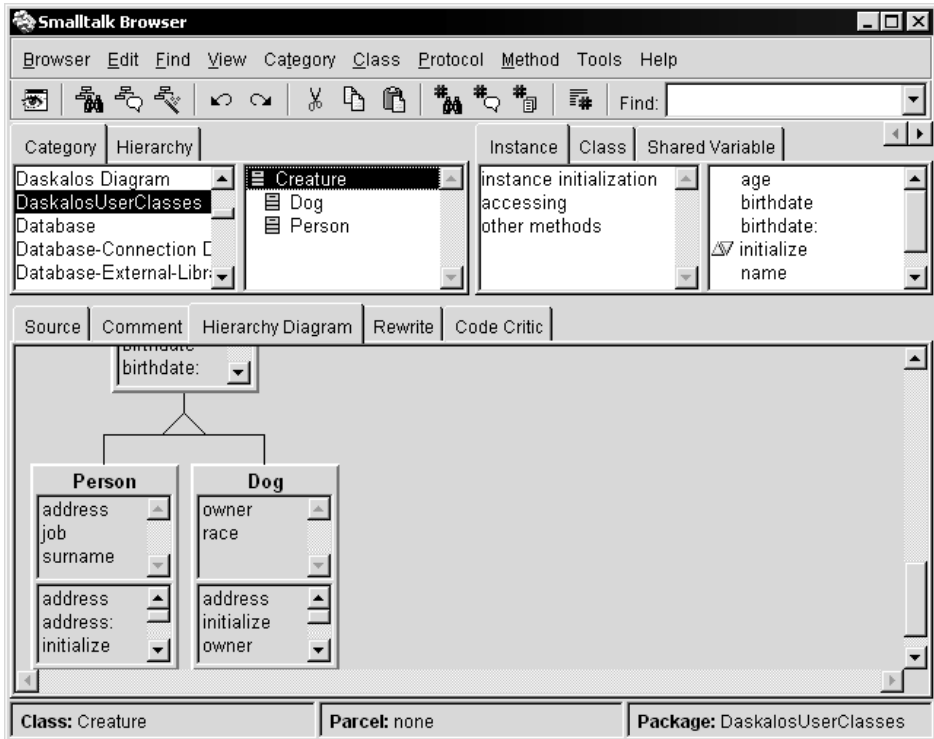
Další zajímavou vlastností VisualWorks je, že si zachovává některé rysy z původní myšlenky Smalltalku jako operačního systému. V prostředí VisualWorks je proto řada zajímavých nástrojů, z nichž jeden je tzv. Workspace.



Obr. 13: Workspace

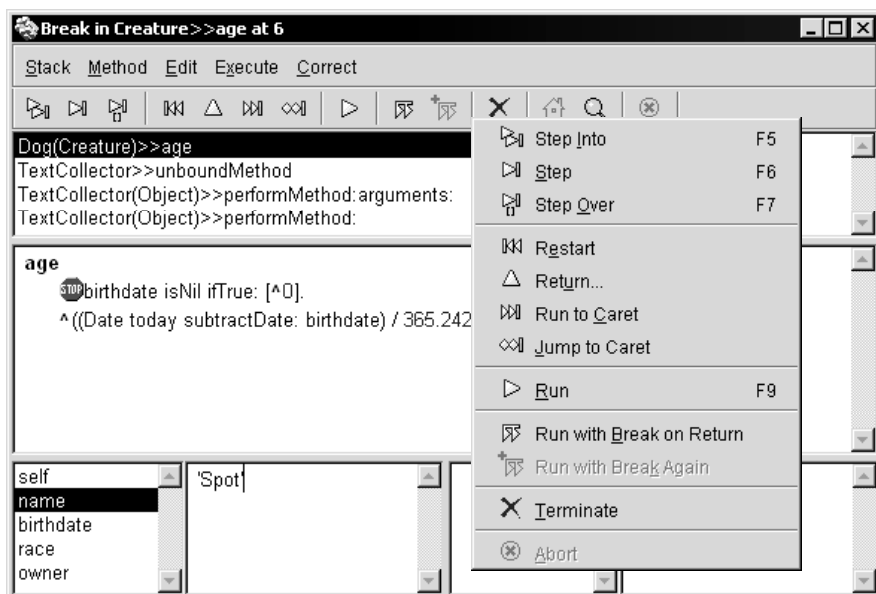
Workspace je něco jako textový editor kombinovaný s konzolí s příkazovým řádkem v operačních systémech. Na obrázku je vidět, že do Workspace lze napsat libovolný výraz v jazyce Smalltalk a ten potom pomocí volby z menu spustit, vytisknout nebo třeba krokovat.

Dalším zajímavým nástrojem je browser. Je to ve své podstatě nástroj na prohlížení i úpravy databáze objektů v systému. Na Smalltalk lze totiž nahlížet jako na množinu objektů a tříd se svými daty a metodami. Tvorba softwaru (tvorba tříd a psaní metod) ve VisualWorks je proto mnohem více podobná manipulaci s obsahem databáze objektů v grafickém prostředí než psaní zdrojových kódů programů jak je tomu u řady jiných jazyků. VisualWorks umožňuje správu verzí kódu v čase podobnou databázovým transakcím a víceuživatelský přístup. Tedy skoro všechno to, co mají databázové systémy.



Obr. 14: Browser

Jako poslední nástroj si ukážeme debugger. Tyto nástroje programátoři používají pro testování a ladění programů, kde například pomocí krokování zkoušejí, zda program dělá správně to, co jeho programátor chtěl.



Obr. 15: Debugger

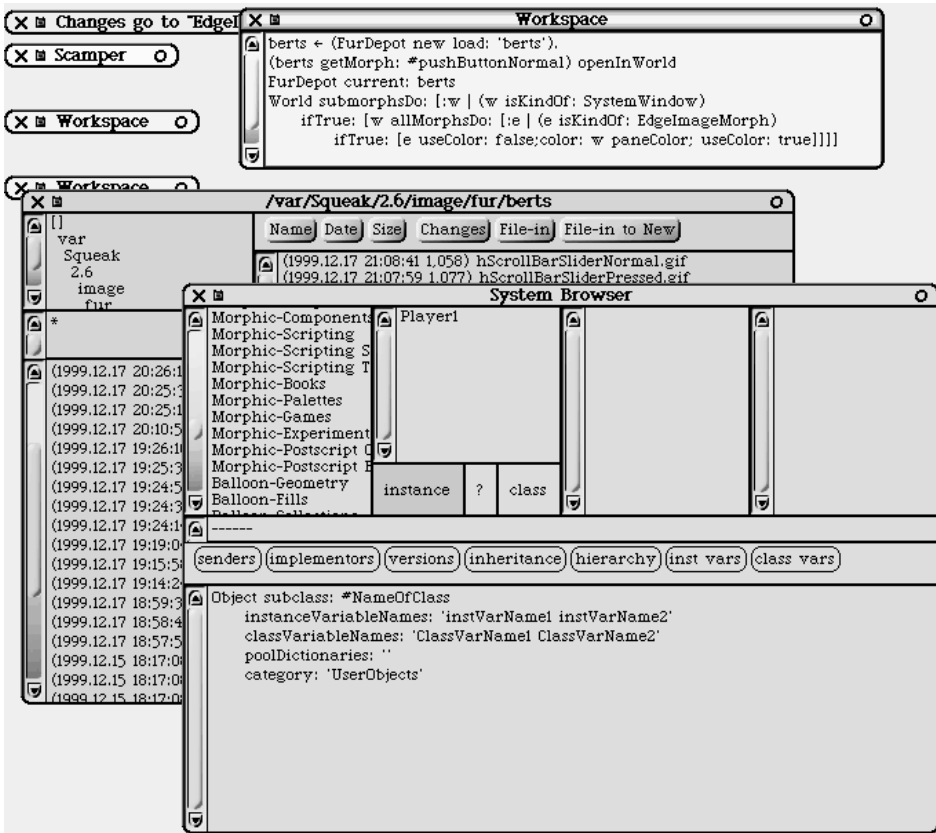
Debugger ve Smalltalku není jen obyčejný prohlížeč nástroj. Je navíc integrovaný s browserem. Prakticky to znamená, že může sloužit i k psaní nebo úpravám kódu programů a obsahu objektů za jejich chodu.

## 3.2.2 STX

Smalltalk/X je velmi kvalitní implementace Smalltalku, kterou sestavuje v Německu Claus Gittinger. Projekt STX je založen na podobné myšlence jako projekt Linuxu vzhledem ke komerčním systémům Unix. Velkou předností STX je propojení s jazykem C, rychlost programů a bezkonkurenční podpora paralelních procesů a práce v síti. Ve srovnání s VisualWorks je ale STX méně portabilní a nemá tak širokou paletu aplikačních nástrojů, především pro práci s databázemi.

## 3.2.3 Squeak

Squeak je univerzitní projekt, který si klade za cíl pokračovat v původním projektu Dynabook. Squeak je pravděpodobně nejčistší implementací Smalltalku.



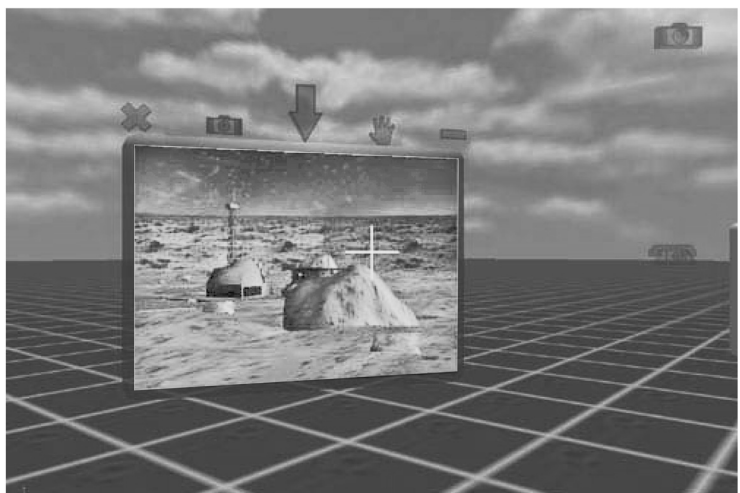
Obr. 16: Systém Squeak

### 3.2.3.1 Croquet

Croquet je systém vybudovaný ve Smalltalku Squeak, který realizuje trojrozměrné uživatelské rozhraní pro operační systém budoucnosti. Na projektu se podílí jeden z autorů Smalltalku Alan Kay. Tak jako v 70. letech původní Smalltalk přinesl světu řadu revolučních novinek (okna, ikony, menu, ovládání myši), tak se i projekt Croquet snaží o další pokračování. Croquet podporuje virtuální realitu a pro komunikaci více počítačů na počítačové síti přináší originální koncepci propojených světů, mezi kterými lze volně procházet.



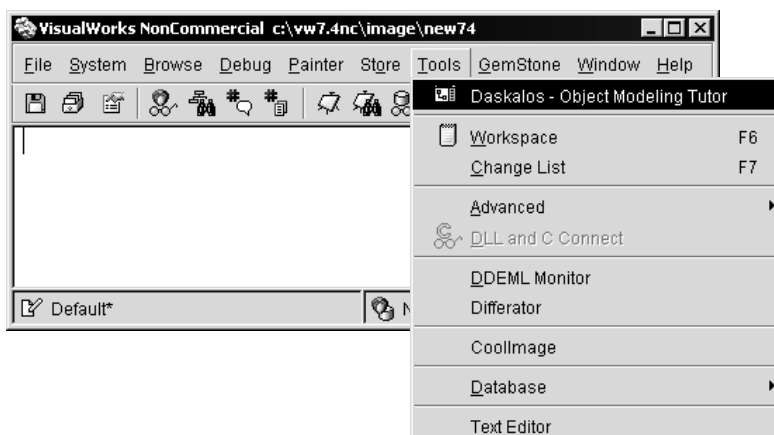
Obr. 17: Ovládání aplikací v trojrozměrném prostředí



Obr. 18: Pohled z jednoho světa do druhého

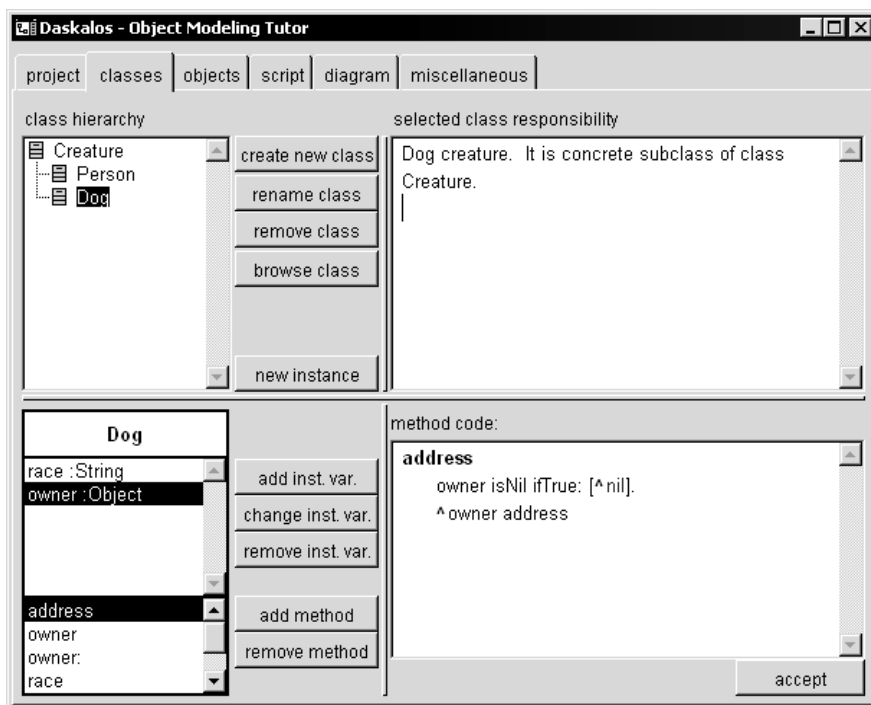
### 3.3 Daskalos

Daskalos (z řeckého slova ο Δάσκαλος – učitel) je počítačový program, který slouží k výuce objektivě orientovaného modelování podle zásad diskutovaných v této knize. Je naprogramován autorem této knihy jako samostatná parcela systému VisualWorks/Smalltalk verze 7.4, která je pro účely výuky a výzkumu zdarma.



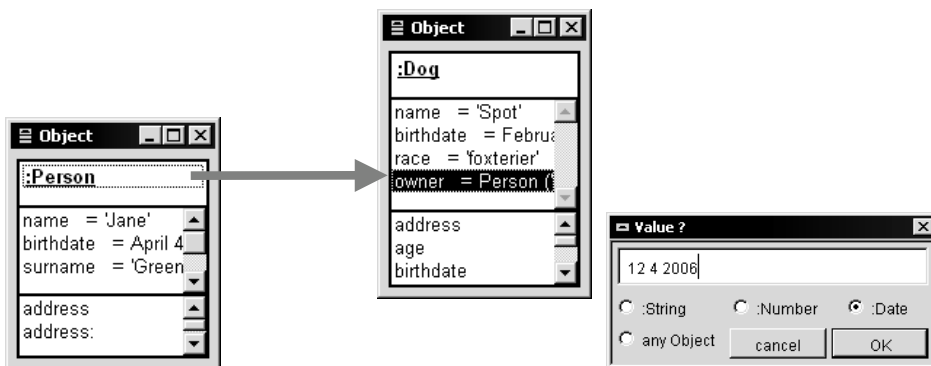
Obr. 19: Spuštění Daskalu z VisualWorks

V Daskalu je možné vytvořit třídy a množiny objektů a programovat metody. Protože vzniklý kód je součástí standardního vývojového prostředí Smalltalku, lze Daskalos použít jako vizuální nástroj pro tvorbu datových objektů určených pro běžně vyvíjené aplikace.



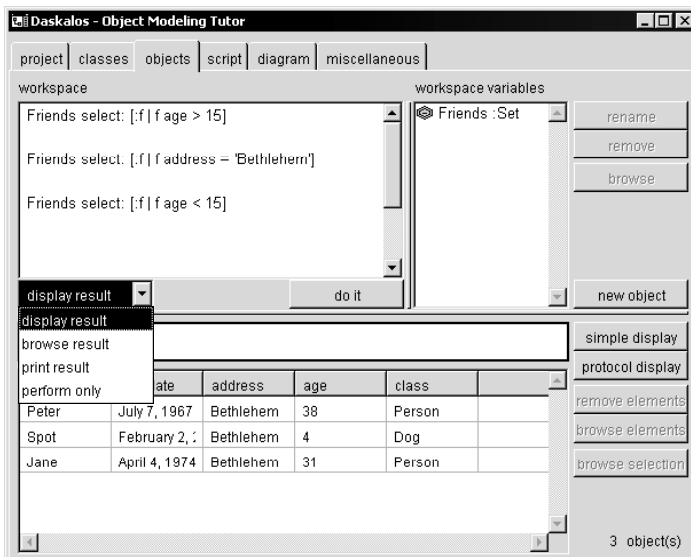
Obr. 20: Tvorba tříd a metod

Daskalos také dovoluje objekty testovat. Objekty a třídy objektů jsou zobrazovány podle standardu UML, přičemž s obsahem takto zobrazených symbolů lze přímo pracovat. Objekty lze vyčleňovat do samostatných oken a takto zobrazeným objektům lze také posílat zprávy přímo kliknutím na zobrazený objekt nebo přímo manipulovat s atributy objektů způsobem táhni-pusť:



Obr. 21: Manipulace s objekty

Pro komplikovanější operace s objekty – například kladení dotazů nad množinami objektů – je možné využít pracovní panel, ve kterém lze příslušné výrazy vyhodnocovat a pracovat s jejich výsledky:



Obr. 22: Pracovní panel

Objekty, které jsou potřeba k testování, je možné vytvářet nejen vizuálními prostředky, ale i obvyklým způsobem ze zdrojového kódu.

```

Friends := Set new.

p := Person new.
p name: 'Peter'.
p surname: 'Black'.
p birthdate: '5 2 1981' asDate.
p address: 'Easton'.
p job: 'taxi driver'.
Friends add: p.

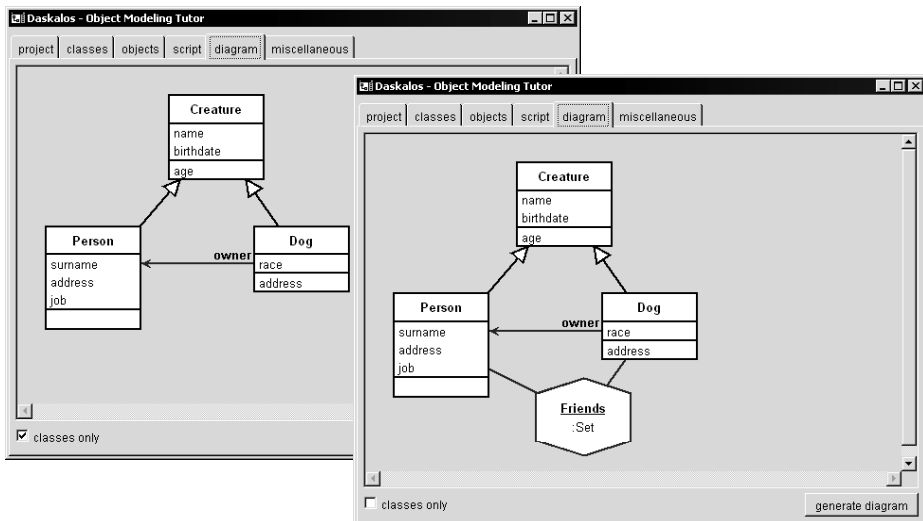
j := Person new.
j name: 'Jane'.
j surname: 'Green'.
j birthdate: '4 4 1974' asDate.
j address: 'Bethlehem'.
j job: 'pop singer'.
Friends add: j.

p2 := Person new.
p2 name: 'Peter'.

```

Obr. 23: Panel zdrojového kódu

Třídy a množiny objektů, se kterými se pracuje, jsou v Daskalu zobrazovány také v podobě diagramu tříd. Symboly tohoto diagramu, jejich obsah a vazby mezi nimi jsou synchronizovány se skutečným obsahem objektů z pracovního panelu. To znamená, že podoba diagramu se mění podle toho, jak se s objekty v pracovním panelu pracuje. Pokud například v paměti není žádný konkrétní objekt, který skládá jiný objekt, vazba skládání mezi symboly se v diagramu neobjeví.



Obr. 24: Panel s diagramem

Daskalos je primárně určen pro výuku OOP začátečníkům. Proto se projekt ukládá nejen do datového souboru ve formátu XML, ale je také generována dokumentace obsahující zdrojové kódy, data i diagramy ve formátu HTML.

The screenshot shows a web browser displaying the HTML documentation for a project titled "Creatures - Persons and Dogs". The page is organized into several sections:

- Workspace:** Contains code snippets for creating objects like `j := Person new.`, `p2 := Person new.`, and `p3 := Dog new.`
- Workspace Objects:** Shows a set of friends: `Friends := Set`.
- Script:** Contains code for creating objects and setting attributes, such as `p := Person new.` and `p := Person new.`
- Diagram:** A class diagram showing `Person` and `Dog` inheriting from `Creature`. `Person` has attributes `sumname` and `address`. `Dog` has attributes `race` and `address`. There is an `owner` relationship between `Person` and `Dog`.
- Classes:**
  - Creature:** An abstract class with instance variables `birthdate` and `name`, and methods `age`, `birthdate`, `initialize`, and `name`.
  - Person:** A concrete subclass of `Creature` with instance variables `address`, `sumname`, and `sumname`, and methods `address` and `address`.
  - Dog:** A concrete subclass of `Creature` with instance variables `address` and `address`, and methods `address` and `address`.

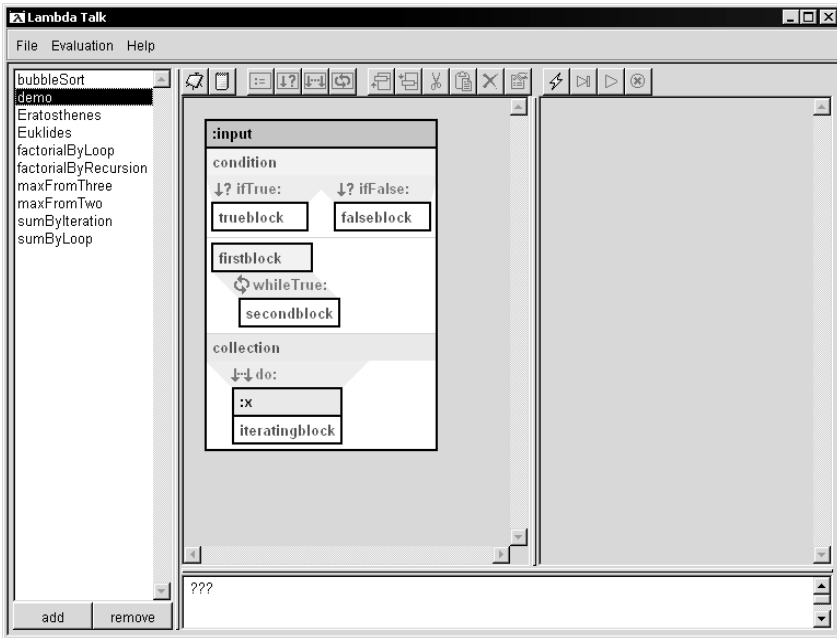
Obr. 25: Dokumentace ve formátu HTML zobrazená webovým prohlížečem

### 3.4 LambdaTalk

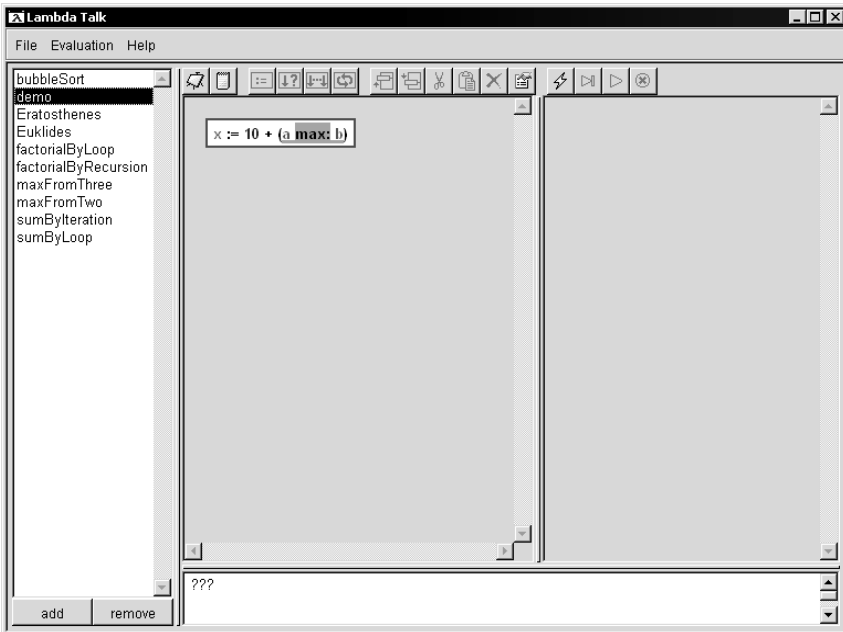
LambdaTalk je aplikace určená pro výuku algoritmicke pro začátečníky. Jde o samostatnou komponentu prostředí VisualWorks naprogramovanou autorem této knihy.

V LambdaTalku lze vizuálně kreslit lambda-výrazy jako smalltalkové bloky, které popisují příslušné algoritmy. Z této grafické reprezentace umí LambdaTalk vygenerovat zdrojový kód v textovém tvaru. Algoritmus je také možné spouštět a krokovat.

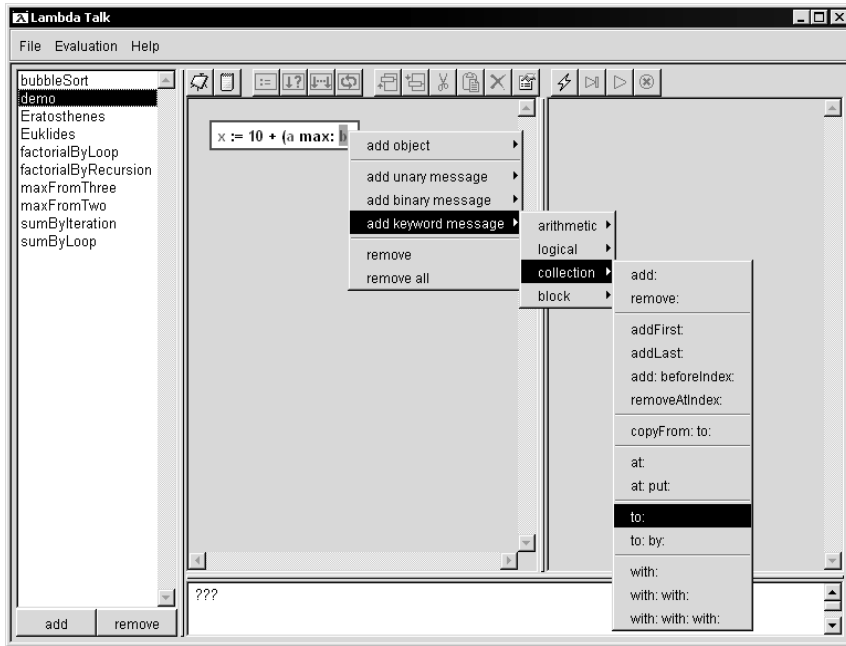
LambdaTalk používá syntaxi diagramů Nassiho a Shneidermana (Nassi, Shneiderman 1973) v úpravě pro lambda-výrazy a OOP. Silné čáry ohraničují lambda-výrazy a oddělují jejich hlavičky o těl. Pro konstrukty větvení, cyklu a iterace jsou využity různé barvy a tvary. Při sestavování kódu má LambdaTalk předdefinovanou sadu unárních, binárních a slovních zpráv. Vestavěný syntaktický editor postupně sestavuje derivační strom výrazu, automaticky označuje příjemce, parametry a selektory zpráv a podle kontextu také automaticky doplňuje závorky.



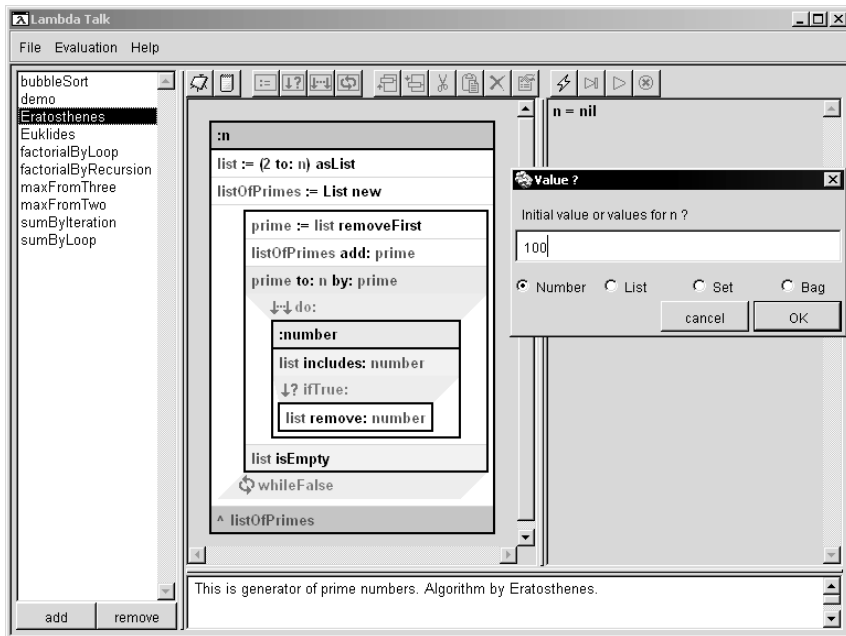
Obr. 26. Přehled symbolů pro řízení výpočtu



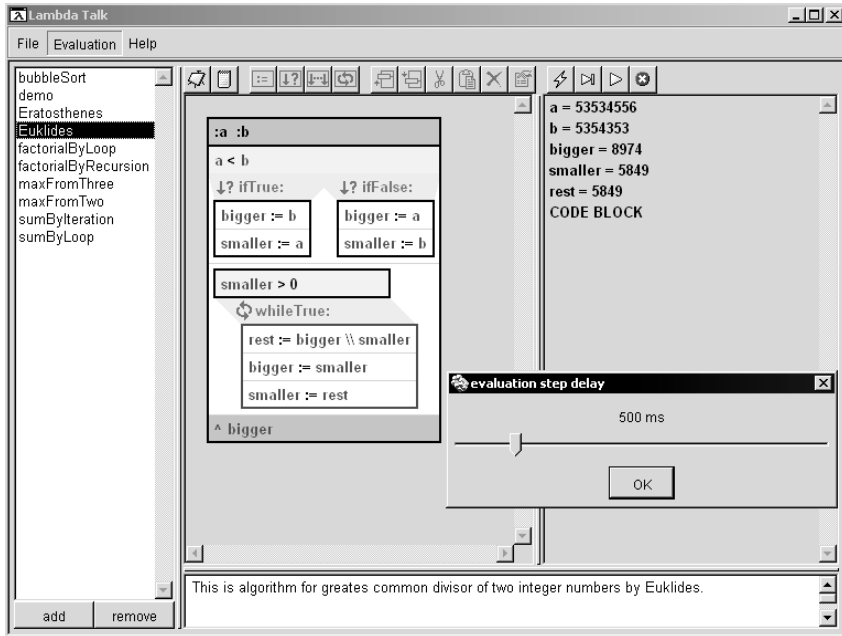
Obr. 27: Zvýrazňování syntaxe



Obr. 28: Příklad předdefinovaných zpráv



Obr. 29: Start simulace

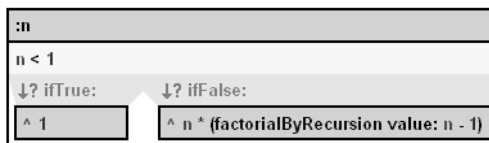


Obr. 30: Průběh simulace

Na příkladu je rekurzivní algoritmus pro výpočet faktoriálu jako

$$factorial \Leftarrow \left( \lambda n \left( \begin{array}{l} 1 \quad n \cdot (factorial \ \<: \ (n - 1)) \\ n < 1 \end{array} \right) \right)$$

Tento algoritmus je graficky znázorněn následujícím diagramem.

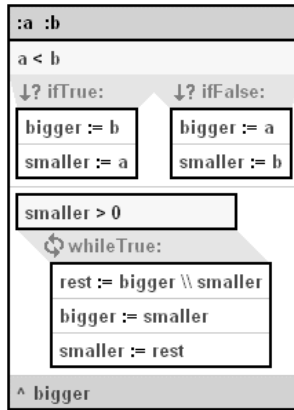


Obr. 31: Výpočet faktoriálu

Na následujícím příkladu je Euklidův algoritmus pro zjištění největšího společného dělitele dvou přirozených čísel:

$$\left( \lambda a \lambda b \left. \frac{\left( \begin{array}{l} \text{bigger} \Leftarrow b \\ \text{smaller} \Leftarrow a \end{array} \right) \left( \begin{array}{l} \text{bigger} \Leftarrow a \\ \text{smaller} \Leftarrow b \end{array} \right)}{a < b} \left( \frac{\left( \begin{array}{l} \text{rest} \Leftarrow \text{bigger} \bmod \text{smaller} \\ \text{bigger} \Leftarrow \text{smaller} \\ \text{smaller} \Leftarrow \text{rest} \end{array} \right)}{\text{smaller} > 0} \right) \right. \text{bigger} \left. \right)$$

který je v LambdaTalku graficky znázorněn jako:



Obr. 32: Euklidův algoritmus

## Gemstone

Gemstone je název databázového systému založeného na objektově orientovaném datovém modelu. Systém nemá nic společného s relačním datovým modelem. Tabulky jsou zde pouze jednou z možných forem výstupní prezentace uložených dat. Datový model Gemstone se nicméně může podobat strukturám síťových databází (Gemstone 2004).

### 4.1

#### 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.

### 4.2

#### 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 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 univerzální objektově orientovaný programovací jazyk.
3. **klient-server systém.** Gemstone obsahuje rozhraní na jazyky Smalltalk, Java, C a C++. Kromě toho je na server připojitelný 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 proto není třeba data z databáze nějak načítat, konvertovat do lokální podoby, 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, není s ním není možné pracovat z jiné transakce. Optimistický režim naopak data neuzamyká a případné kolize více transakcí se řeší až v době pokusu o uzavření každé transakce.

6. **připojitelnost na externí zdroje dat.** Gemstone má rozhraní na relační databáze podle standardu SQL, 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.

### 4.3

## Programovací jazyk Smalltalk DB

Jazyk Smalltalk DB je databázovým rozšířením programovacího jazyka Smalltalk-80. Smalltalk DB je tedy klon standardního Smalltalku. I když se jedná o databázový jazyk, zachovává si plně 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. Gemstone lze proto použít jako úložiště pro data nějaké aplikace na straně klienta (např. VisualWorks), ale i samostatně jako aplikační server.

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, data...)	ano	ano
knihovna pro práci se streamy, soubory na disku, TCP-IP komunikace	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

Tab. 5: Srovnání jazyků Smalltalk-80 a Smalltalk DB

*Selection block* je zvláštní typ klasického smalltalkového bloku, který obsahuje logický výraz odkazující na datovou hierarchii objektů, tedy na 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 jako

Companies // ( $\lambda c$  | c<director>name = Smith).

Předpokládá se, že každý prvek množiny Companies má odkaz director na svého ředitele a každý ředitel má odkaz name na svoje jméno.

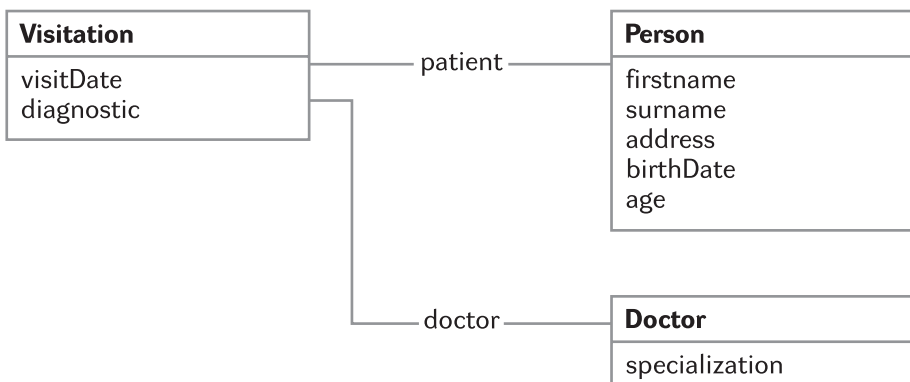
## 4.4

### Příklad objektové databáze

#### 4.4.1

#### Popis úlohy

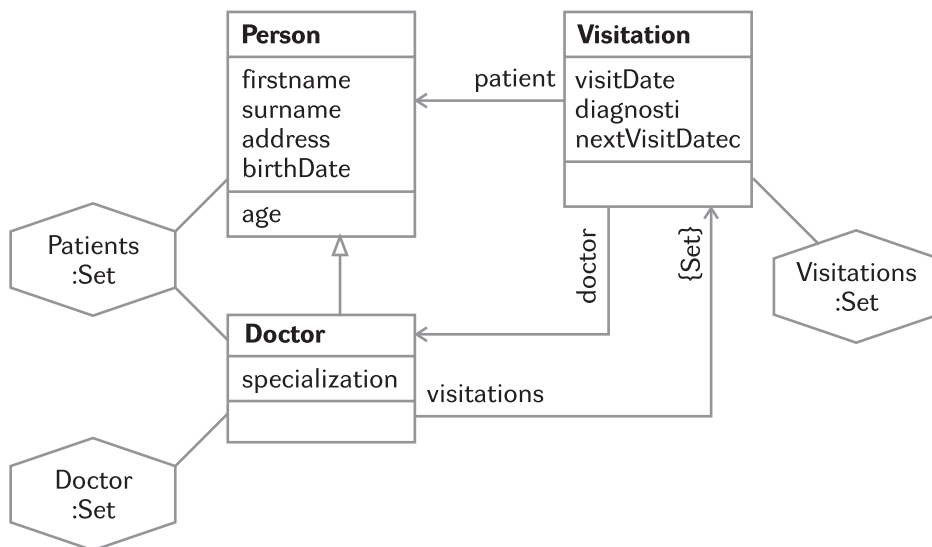
Vlastnosti objektové databáze budou prezentovány pomocí následující úlohy: Mějme jednoduchou databázi pro evidenci návštěv v ordinacích lékařů. U každého pacienta bude evidováno jeho jméno, příjmení, adresa, datum narození a věk. U každého doktora ještě navíc jeho specializace. U návštěvy pacienta u lékaře bude evidováno datum návštěvy a diagnóza. Není vyloučeno, aby se jeden lékař stal druhému lékařovi pacientem. Úlohu znázorňuje následující obrázek:



Obr. 33: Počáteční model úlohy s třídami a asociacemi

## 4.4.2 Implementace úlohy

- 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`.
- V aplikaci lze navrhovat nejen třídy objektů, ale také kolekce 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.
- V návrhu datového modelu si můžeme 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, je možné realizovat dokonce oba dva směry vazby současně. Tak tomu bude v našem příkladu u vazby mezi doktory a vyšetřeními.
- Podobně jako směr vazby, 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, není množina `Doctors` nutná, protože data o doktorech jsou dosažitelná z objektů třídy `Visitation` pomocí operace sběru. V našem příkladě jsou ale z didaktických důvodů všechny množiny ponechány.



Obr. 34: Datový model úlohy

### 4.4.3

## 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 (pokud není známo datum narození, věk bude roven nule):

$$\left\langle \text{age} , \frac{0 \text{ (Date} < \text{today} - \text{birthdate}) / 365.2422}{\text{birthdate} = \emptyset} \right\rangle \in \text{Meth}(\text{Person})$$

```
age
^birthdate isNil
ifTrue: [0]
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 odebírání vyšetření tak, aby byla zajištěna konzistence i „z druhé strany“:

$$\left\langle \text{add visitation} : , \left( \lambda x \left| \begin{array}{l} x \in \sigma \text{visitations} \\ x < \text{doctor} : \sigma \end{array} \right. \right) \right\rangle \in \text{Meth}(\text{Doctor})$$

```
addVisitation: x
  visitations add: x.
  x doctor: self.
```

$$\left\langle \text{remove visitation} : , \left( \lambda x \left| \begin{array}{l} x \notin \sigma \text{visitations} \\ x < \text{doctor} : \emptyset \end{array} \right. \right) \right\rangle \in \text{Meth}(\text{Doctor})$$

```
removeVisitation: x
  visitations remove: x.
  x 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'.
```

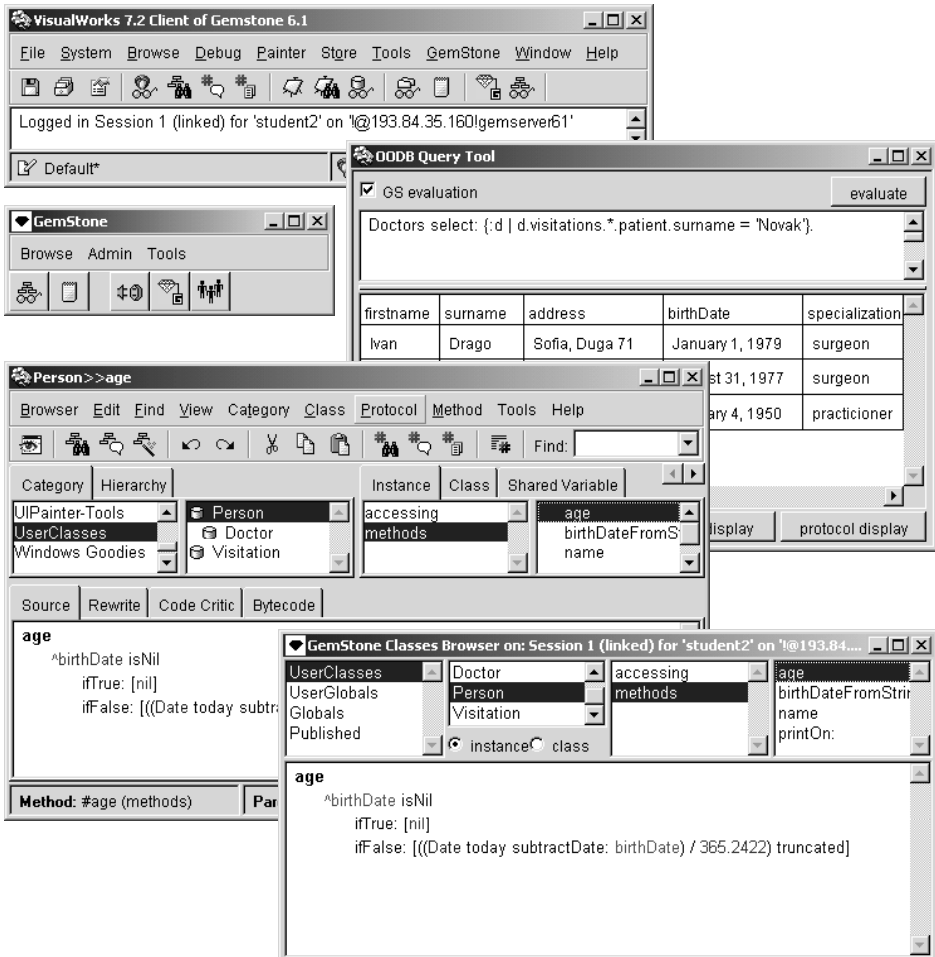
Objekty d, p, v je ještě třeba propojit (propojujeme je přímo – spojením odkazem z cizích na primární klíče nepotřebujeme):

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 prostředí Gemstone integrovaném do VisualWorks:



Obr. 35: Gemstone v prostředí VisualWorks

## 4.5 Příklady dotazů

Následujících několik ukázek dotazů předvede dotazovací možnosti jazyka Smalltalk DB. Pro srovnání je uveden také formální zápis a zápis v jazyce OQL, který je součástí standardu ODMG. (Catell 1998) OQL vychází z jazyka SQL, který je standardem většiny relačních databázových systémů:

### Najdi vyšetření, které prováděl doktor Dyba

*Visitations //* ( $\lambda v \mid v \langle \text{doctor} \langle \text{surname} = \text{Dyba} \rangle \rangle$ ).

Visitations select:  $\{ :v \mid v.\text{doctor.surname} = \text{'Dyba'} \}$ .

```
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 //* ( $\lambda d \mid \{ d \langle \text{visitations} \rangle \langle \text{patient} \langle \text{surname} = \text{Novák} \rangle \rangle \}$ ).

Doctors select:  $\{ :d \mid d.\text{visitations}.*.\text{patient.surname} = \text{'Novak'} \}$ .

```
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 //* ( $((\lambda v \mid v \langle \text{diagnosis} = \text{angína} \rangle)) \gg (\lambda v \mid v \langle \text{patient} \langle \text{address} \rangle \rangle)$ ).

```
(Visitations select: {v | v.diagnosis = 'angina'})
collect: {v | v.patient.address}.
```

```
SELECT v.patient.address
FROM v IN Visitations
WHERE v.diagnosis = 'angina';
```

## Najdi pacienty starší 60 let

*Patients // ( $\lambda p \mid p.<age > 60$ ).*

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 je na patrné, že srovnatelná relační databáze by byla komplikovanější a odpovídající dotazy složitější.

## 4.6

### Shrnutí

Je pravda, že objektově relační databáze dokáží implementovat naši úlohu také. Především poslední verze Oraclu dovolují využít hodně smíšených objektově-relačních vlastností. Cílem této kapitoly ale bylo prakticky ukázat čistý objektový datový model. Smíšená objektově relační technologie je totiž ještě méně standardizovaná než objektová, ale má obrovskou výhodu v tom, že za ní stojí velké firmy. Tvorba aplikací, která se o smíšený přístup opírá, se snadno dostane do vleku specifických funkcí posledních verzí konkrétního systému. Oproti tomu zde popsaný přístup je aplikovatelný nejen v Gemstone, ale ve většině objektově orientovaných databází (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 informatiků 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 datvé modelování a zaměňují jednu z možných softwarovým implementací za abstraktní přístup k řešení problému.

Znalost objektově orientovaného 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, budou mít dojem, že „tabulky“ a vazby mezi nimi jsou

jediný prakticky použitelný způsob analýzy, návrhu a implementace dat v informačních systémech a na objekty budou nahlížet jen jako na podivnosti, se kterými si hrají programátoři.

## Příklady datových modelů

V této kapitole si ukážeme příklady datových modelů dvou konkrétních úloh. Tyto příklady byly vypracovány pomocí programu Daskalos.

### 5.1

#### Evidence přátel

Mějme za úkol vytvořit datový model evidence či kartotéky. Budeme evidovat nejen osoby, ale i zvířata, konkrétně pejsky. Člověk a pes se ale od sebe liší, a proto nebude rozumné pracovat s pouze jedinou třídou objektů. Vytvoříme si proto hned tři třídy:

##### třída *Creature*

Tato třída bude implementovat, co mají lidé a pejsci společného. Protokol této třídy proto může být

$$\Pi(\textit{Creature}) = [\textit{name}, \textit{birthdate}, \textit{age}].$$

Atribut věk bude realizován již známou metodou

$$\langle \textit{age}, (\textit{today} - \sigma \textit{birthdate}) / 365.2422 \rangle \in \textit{Meth}(\textit{Creature}).$$

##### třída *Person*

Tato třída rozšiřuje protokol objektů na

$$\Pi(\textit{Person}) = \Pi(\textit{Creature}) \cup [\textit{surname}, \textit{address}, \textit{job}],$$

což znamená, že tato třída bude z třídy *Creature* dědit.

Tři konkrétní objekty této třídy  $p_1, p_2, p_3 \in \textit{Inst}(\textit{Person})$  potom můžou mít například:

$$\Delta(p_1) = [\textit{name}: \textit{Peter}, \textit{surname}: \textit{Black}, \textit{birthdate}: \textit{Feb 5 1981}, \\ \textit{address}: \textit{Easton}, \textit{job}: \textit{taxi driver}],$$

$$\Delta(p_2) = [\textit{name}: \textit{Jane}, \textit{surname}: \textit{Green}, \textit{birthdate}: \textit{Apr 4 1974}, \\ \textit{address}: \textit{Allentown}, \textit{job}: \textit{pop singer}],$$

$$\Delta(p_3) = [\textit{name}: \textit{Peter}, \textit{surname}: \textit{White}, \textit{birthdate}: \textit{Jul 7 1967}, \\ \textit{address}: \textit{Easton}, \textit{job}: \textit{pop singer}].$$

### třída Dog

Tato třída rozšiřuje protokol třídy *Creature* jinak. Pejskové nemají příjmení ani zaměstnání jako lidé. Na druhou stranu je ale dobré u pejsků evidovat rasu a majitele, což jsou zase atributy, které není dobré dávat lidem. Proto vytvoříme třídu *Dog* s protokolem

$$\Pi(Dog) = \Pi(Creature) \cup [race, owner]$$

a můžeme vytvořit jednoho konkrétního pejska *d* s atributy

$$\Delta(d) = [name: Spot, race: foxterier, birthdate: Feb 2 2002, owner: p_2].$$

Adresu pejska můžeme odvozovat z adresy jeho pána. Pokud je pejsek bez pána, je tak i bez adresy. Proto budeme mít metodu

$$\left\langle address, \frac{\emptyset \sigma owner < address}{\sigma owner = \emptyset} \right\rangle \in Meth(Dog)$$

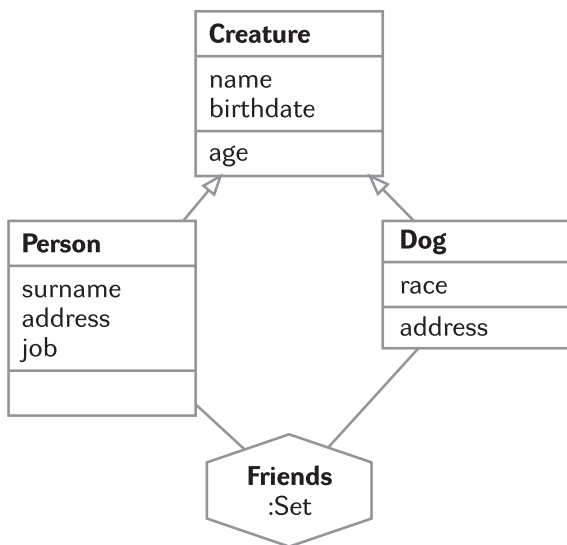
což se v jazyce Smalltalk napíše takto:

```
address
  owner isNil ifTrue: [^nil] ifFalse: [^owner address].
```

### kolekce Friends

Poslední, co je třeba vytvořit, je kolekce, kde budou naše objekty uloženy. Mějme proto kolekci *Friends* jako

$$Friends \Leftarrow [p_1, p_2, p_3, d].$$



Obr. 36: Přátelé – datový model

Tento datový model lze potom napsat v jazyce Smalltalk takto

```
Friends := Set new.
```

```
p1 := Person new.
p1 name: 'Peter'.
p1 surname: 'Black'.
p1 birthdate: '5 2 1981' asDate.
p1 address: 'Easton'.
p1 job: 'taxi driver'.
Friends add: p1.
```

```
p2 := Person new.
p2 name: 'Jane'.
p2 surname: 'Green'.
p2 birthdate: '4 4 1974' asDate.
p2 address: 'Allentown'.
p2 job: 'pop singer'.
Friends add: p2.
```

```
p3 := Person new.
p3 name: 'Peter'.
p3 surname: 'White'.
p3 birthdate: '7 7 1967' asDate.
p3 address: 'Easton'.
p3 job: 'pop singer'.
Friends add: p3.
```

```
d := Dog new.
d name: 'Spot'.
d owner: p2.
d race: 'foxterier'.
d birthdate: '2 2 2002' asDate.
Friends add: d.
```

Takto vytvořený datový model můžeme prakticky vyzkoušet pomocí následujících dotazů.

Nejprve si vybereme přátele z Eastonu:

```
Friends // (λx | x address = Easton).
Friends select: [:x | x address = 'Easton'].
```

Jako odpověď dostaneme výsledek v tabulce:

<i>name</i>	<i>surname</i>	<i>birthdate</i>	<i>address</i>	<i>job</i>	<i>age</i>
Peter	Black	Feb 5 1981	Easton	taxi driver	26
Peter	White	Jul 7 1967	Easton	pop singer	40

Tab. 6: Výsledek výběru

Dalším příkladem může být výběr přátel mladších 10 let:

*Friends* // ( $\lambda x \mid x \triangleleft \text{age} < 10$ ).

*Friends* select: [ $:x \mid x \text{ age} < 10$ ].

Výsledek lze zobrazit v tabulce:

<i>name</i>	<i>birthdate</i>	<i>owner</i>	<i>race</i>	<i>address</i>	<i>age</i>
Spot	Feb 2 2002	Jane Green	foxtier	Allentown	5

Tab. 7: Výsledek dotazu

I když tato tabulka ukazuje výběr ze stejné kolekce jako předchozí výběr, tak je vidět, že tabulky mají jiné sloupce. Jinými slovy to znamená, že výsledky obou výběrů nemají stejné protokoly, i když jde o výběry ze stejné kolekce.

Jako poslední příklad si ukážeme operaci sběr, která zjistí, na jakých adresách bydlí naši přátelé:

*Friends* >> ( $\lambda x \mid x \triangleleft \text{address}$ ).

*Friends* collect: [ $:x \mid x \text{ address}$ ].

Jako výsledek dostaneme kolekci adres:

<i>address</i>
Easton
Allentown

Tab. 8: Výsledek dotazu

## 5.2 Obchod s pivem

Druhým příkladem bude evidence obchodních smluv. Jde o dodávku piva. U každé smlouvy (kontraktu) budeme evidovat, komu se bude dodávat, jaký produkt se bude dodávat a kdo je výrobcem příslušného produktu.

## 5.2.1

### Třídy a kolekce

Provedeme-li techniku normalizace, o které se podrobně hovoří v samostatné kapitole, tak dojdeme k závěru, že v naší úloze budeme potřebovat čtyři třídy:

#### třída **Contract**

Tato třída implementuje vlastní smlouvu. Do protokolu objektů této třídy patří datum smlouvy, předmět smlouvy – tj. příslušný produkt, dohodnuté množství a celková cena:

$$\Pi(\text{Contract}) = [\text{date}, \text{amount}, \text{product}, \text{total price}].$$

Pro celkovou cenu budeme potřebovat metodu, protože celková cena je rovna násobku dohodnutého množství a ceny za jednotku:

$$\langle \text{total price}, (\text{product} \triangleleft \text{price per unit} \cdot \sigma \text{amount}) \rangle \in \text{Meth}(\text{Contract}),$$

Což ve Smalltalku napíšeme jako

```
totalPrice
    ^ product pricePerUnit * amount.
```

#### třída **Product**

Tato třída implementuje objekty, které jsou předmětem smlouvy. Do protokolu patří název výrobku, jednotková cena (kterou jsme právě použili v metodě třídy *Contract*) a výrobce:

$$\Pi(\text{Product}) = [\text{name}, \text{price per unit}, \text{producer}].$$

#### třída **Company**

Tato třída implementuje objekty, kterými jsou jednak výrobci našich výrobků, ale v některých případech i zákazníci smluv. Každá firma má protokol, do kterého patří jméno firmy a adresa firmy:

$$\Pi(\text{Company}) = [\text{name}, \text{address}].$$

#### třída **Person**

Tato třída implementuje osoby, které mohou být zákazníky smluv. Do protokolu osob zde patří jméno, příjmení a adresa:

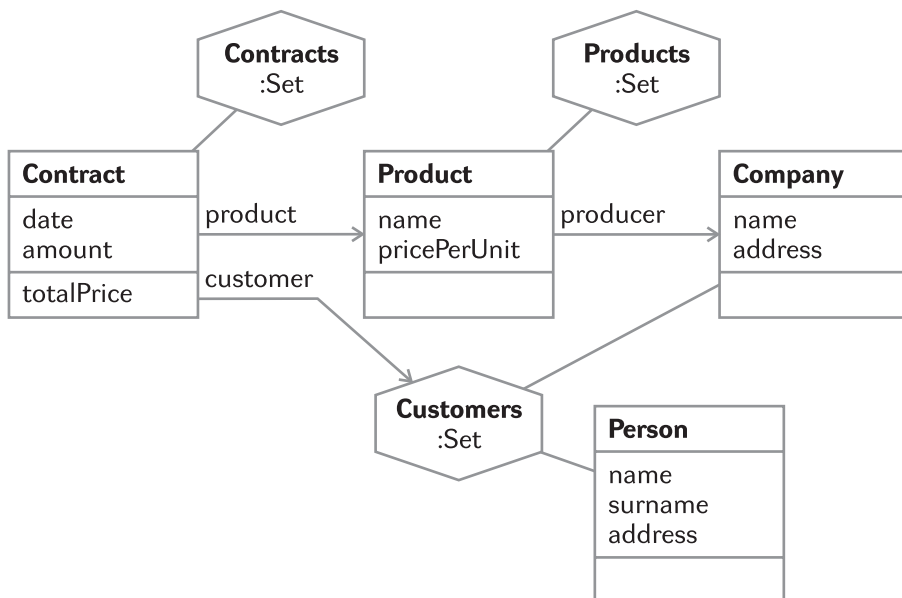
$$\Pi(\text{Person}) = [\text{name}, \text{surname}, \text{address}].$$

#### kolekce **Contracts, Products a Customers**

Datový model se z pohledu uživatele neprezentuje svými třídami, ale kolekce. To proto, že uživatel chce vidět nějaká reálná data, která musí být nějak organizována.

Nejjednodušší, ale také nevhodný způsob řešení jsou extenty tříd. V tomto případě bychom tedy měli kolekci všech osob, kolekci všech výrobků a kolekci všech firem. Tak vypadá obvyklé řešení datového modelu, které vede nakonec k implementaci v relační databázi.

Je zde ale ještě jiné řešení, které objekty organizuje ne podle jejich druhu (typu), ale podle jejich účelu, který vyplývá z příslušného zadání. Tato struktura se od extentů tříd liší. V našem zadání se hovoří o smlouvách, výrobcích a zákaznících. Budeme proto mít kolekci smluv (*Contracts*), kolekci výrobků (*Products*) a kolekci zákazníků (*Customers*). V kolekci zákazníků budeme mít pohromadě objekty tříd firma i osoba, protože oba druhy objektů mohou být zákaznicky. Celkové schéma úlohy ukazuje následující obrázek:



Obr. 37: Obchod s pivem – datový model

V tomto návrhu chybí kolekce výrobců. Lze ji ale snadno získat transformací z kolekce výrobků jako:

$Producers \Leftarrow Products \gg (x \mid x \triangleleft producer)$ .

nebo pomocí Smalltalku jako

$Producers := Products collect: [:x \mid x producer]$ .

Srovnáme-li toto řešení s řešením založeným na pouhých extentech tříd, uvidíme největší rozdíl v kolekcích *Customers* a *Producers* oproti extentům *Inst(Company)* a *Inst(Person)*.

V obou řešeních by mohly být stejné objekty. Ale rozdíl je v tom, jak by se s nimi pracovalo. Pokud máme kolekce *Customers* a *Producers*, můžeme nad těmito kolekcemi

snadno provádět operace podle potřeb zadání. Pokud ale budeme mít jen například extent  $Inst(Company)$ , budou se nám v tomto extentu míchat dohromady firmy výrobce s firmami zákazníků, přičemž zákazníci osoby budou ležet odděleně v jiném extentu  $Inst(Person)$ .

Tento rozdíl v řešeních si nejlépe ukážeme na příkladě dotazu, kdy budeme chtít najít zákazníky z Prahy. Tento dotaz, který je v námi doporučeném řešení jednoduchý:

$Customers // (\lambda x \mid x \triangleleft address = Praha)$ , nebo Smalltalkem jako

Customers collect: [:x | x address = 'Praha'].

by se pro případ extentů musel sestavit takto:

$$Inst(Company) // (\lambda x \mid (x \triangleleft address = Praha) \wedge ((\exists y Inst(Contracts) y \triangleleft producer = x)) \cup Inst(Person) // (\lambda x \mid x \triangleleft address = Praha),$$

což lze číst „Vyber firmy, které jsou výrobci a jsou z Prahy, a sjednoť výsledek s osobami z Prahy.“ Podobně komplikovanější by byla i většina příkladů z následující kapitoly.

## 5.2.2

### Data

Nyní je potřeba připravit data. Nejprve si vytvoříme kolekce.

Customers := Set new.

Contracts := Set new.

Products := Set new.

Po kolekcích následují instance tříd. Nejprve si vytvoříme osoby a firmy, které budou výrobci nebo zákazníci.

pc1 := Person new.

pc1 name: 'Jim'; surname: 'Adams'; address: 'New York'.

pc2 := Person new.

pc2 name: 'Peter'; surname: 'Green'; address: 'Easton'.

pc3 := Person new.

pc3 name: 'Tony'; surname: 'Dvorak'; address: 'Prague'.

cc1 := Company new.

cc1 name: 'Hotel Hilton'; address: 'Prague'.

cc2 := Company new.  
cc2 name: 'Comfort Suites'; address: 'Allentown'.

c1 := Company new.  
c1 name: 'Samuel Adams Brewery'; address: 'Boston'.

c2 := Company new.  
c2 name: 'Budweiser'; address: 'Budweis'.

c3 := Company new.  
c3 name: 'Pilsen Brewery'; address: 'Pilsen'.

c4 := Company new.  
c4 name: 'Anheuser-Busch'; address: 'St. Louis'.

Dále si vytvoříme produkty, které budou předmětem kontraktů.

p1 := Product new.  
p1 name: 'Sam Adams'; pricePerUnit: 5.

p2 := Product new.  
p2 name: 'Budweiser'; pricePerUnit: 4.

p3 := Product new.  
p3 name: 'Pilsner Urquell'; pricePerUnit: 3.

p4 := Product new.  
p4 name: 'Budweiser'; pricePerUnit: 5.

Objekt  $p_2$  je český Budweiser a  $p_4$  americký Budweiser. Máme zde dva různé objekty zatím se stejnými daty. Různé výrobce jim zadáme později. Nyní si vytvoříme smlouvy.

x1 := Contract new.  
x1 date: '9-JAN-2004' asDate; amount: 3.

x2 := Contract new.  
x2 date: '7-FEB-2004' asDate; amount: 4.

x3 := Contract new.  
x3 date: '3-FEB-2004' asDate; amount: 150.

x4 := Contract new.  
x4 date: '9-SEP-2004' asDate; amount: 10.

x5 := Contract new.  
x5 date: '5-SEP-2004' asDate; amount: 8.

x6 := Contract new.  
 x6 date: '9-AUG-2004' asDate; amount: 20.

x7 := Contract new.  
 x7 date: '9-SEP-2004' asDate; amount: 35.

A nakonec všechny objekty propojíme a uložíme do kolekcí.

x1 customer: pc1; product: p1.  
 x2 customer: pc1; product: p1.  
 x3 customer: cc1; product: p2.  
 x4 customer: pc3; product: p3.  
 x5 customer: pc2; product: p4.  
 x6 customer: cc2; product: p4.  
 x7 customer: cc2; product: p4.

p1 producer: c1.  
 p2 producer: c2.  
 p3 producer: c3.  
 p4 producer: c4.

Customers add: pc1; add: pc2; add: pc3; add: cc1; add: cc2.  
 Contracts add: x1; add: x2; add: x3; add: x4; add: x5; add: x6; add: x7.  
 Products add: p1; add: p2; add: p3; add: p4.

### 5.2.3

#### Dotazy

S takto vytvořeným datovým modelem již můžeme prakticky pracovat. Ukážeme si proto několik zajímavých dotazů, které využívají polymorfismus a skládání objektů v naší struktuře:

#### **Jaké výrobky jsou objednávány zákazníky z Prahy?**

Je třeba nejprve vybrat smlouvy zákazníků z Prahy a z výsledku sesbírat výrobky:

```
(Contracts // (λc | c<customer<address = Prague))
  >> (λc | c<product)
```

nebo

```
(Contracts select: [:c | c customer address = 'Prague'])
  collect: [:c | c product ]
```

Výsledek si můžeme zobrazit v následující tabulce:

<i>name</i>	<i>price per unit</i>	<i>producer</i>
Budweiser	4	Budweiser
Pilsner Urquell	3	Pilsen Brewery

Tab. 9: Výsledek dotazu

**Zobraz smlouvy seříděné podle celkové ceny objednávky.**

*List(Contracts , (λc | c<total price))*

nebo

Contracts asList sortBy: [:c | c totalPrice].

<i>date</i>	<i>amount</i>	<i>customer</i>	<i>product</i>	<i>total price</i>
Jan 9 2004	3	Jim Adams	Sam Adams	15
Feb 7 2004	4	Jim Adams	Sam Adams	20
Sep 9 2004	10	Tony Dvorak	Pilsner Urquell	30
Sep 5 2004	8	Peter Green	Budweiser	40
Aug 9 2004	20	Comfort Suites	Budweiser	100
Sep 9 2004	35	Comfort Suites	Budweiser	175
Feb 3 2004	150	Hotel Hilton	Budweiser	600

Tab. 10: Výsledek dotazu

**Z jakých firem jsou výrobky objednávané za celkovou cenu menší než 100?**

Nejprve zjistíme objednávky za méně než 100 peněz a z nich sesbíráme firmy výrobců:

*(Contracts // (λc | c<totalPrice < 100))*  
*>> (λc | c<product<producer)*

nebo

(Contracts select: [:c | c totalPrice < 100])  
 collect: [:c | c product producer].

<i>name</i>	<i>address</i>
Anheuser-Busch	St. Louis
Pilsen Brewery	Pilsen
Samuel Adams Brewery	Boston

Tab. 11: Výsledek dotazu

# Pokročilé metody návrhu datového modelu

## 6.1

### Jak poznat správný návrh?

Čtenář si jistě všiml, že v příkladech této knihy jsou několikrát různým způsobem implementované objekty modelující živé osoby. Nabízí se tu otázka, která varianta je nejlepší. Zda to je ta s nadřídou pro živé tvory, nebo řešení z příkladu lékařské ordinace nebo z úvodu knihy či z obchodu s pivem. To, že se vyjmenované příklady liší, je zvoleno záměrně. Řada analytiků se totiž domnívá, že pro každý objekt existuje jedno jediné univerzálně použitelné nejlepší řešení bez ohledu na řešený problém. Tedy že například fakturu modelujeme správně takto... osoby takto... auta takto... apod. A že datové modelování konkrétních úloh je jen o vybírání a propojování předem jednoznačně daných popisů tříd.

Domníváme se, že to není pravda. V různých úlohách totiž potřebujeme pro zdánlivě stejné objekty jiné atributy a jiné chování. Nikdy neděláme „simulační model celého světa“, ale vždy jen nějakou horizontálně i vertikálně vymezenou a zjednodušenou část z celého světa. Nejde jen o to, že spoustu atributů v daných problémech nepotřebujeme, a tak že je zbytečné je na objektech držet nepoužité. Jsou totiž i případy, kdy atributy mít nesmíme. Například v lékařské ordinaci je věk pacienta potřebný, ale v obchodním informačním systému si lze představit situace, kdy věk osob nejen nepotřebujeme, ale ze zákona ho dokonce používat nesmíme. Tento malý příklad s osobami tak podává důkaz, že konkrétní struktura objektů je velmi závislá na konkrétním požadavku na práci s těmito objekty podle aktuálního zadání.

Datové modelování je proto složitým procesem, který si lze představit jako přeměnu zadání do konkrétních struktur a dat. Naštěstí zde nejsme odkázáni jen na intuici a zkušenost analytiků, ale máme k dispozici několik pokročilých metod, které mohou tento proces usnadnit a upřesnit.

## 6.2

### Objektová normalizace

Formálními technikám návrhu datových struktur je třeba věnovat velkou pozornost. I když již byla publikována řada prací, je v této oblasti OOP stále na samotném počátku. Komunita analytiků a návrhářů od techniky objektové normalizace očekává následující vlastnosti:

- a) Musí být pokud možno jednoduchá, přesná, srozumitelná a měla by vystačit s minimem pojmů podobně jako „klasická“ normalizace, jak ji známe z relačních databází. Zavádění složitých definic výrazně přesahující rozsah klasických normálních forem, více druhů vazeb a podobně není správná cesta.
- b) Měla by být konkrétně zaměřena na návrh databází, tedy struktur objektů, které budou sloužit k ukládání a manipulaci s daty a znalostmi v databázových systémech. Není třeba sem zahrnovat i objekty, které jsou zodpovědné za „chod“ aplikací. Pro „správný“ návrh aplikačních objektů jsou vhodné návrhové vzory, které jsou také komunitou programátorů široce používány.
- c) Je možné, že časem se objektový přístup stane univerzálním přístupem k datovému modelování a entitně-relační paradigma se omezí jen na jednu z možných implementačních variant. Takže se dnešní role otočí. Bylo by proto velmi rozumné novou teorii budovat záměrně tak, aby byla analogická s konceptem entitně-relačního modelování a relační normalizace. V nejlepším případě by měla být relační normalizace (jako nástroj šitý na míru relační technologii) nějakým způsobem z nové teorie odvoditelná.

## 6.2.1

### Datový objekt, atributy objektu

Nejprve musíme definovat, co rozumíme datovým objektem. Takový objekt slouží „jen“ k ukládání a manipulaci dat. Není to objekt, který by obsluhoval pomocí svých algoritmů nějakou část aplikace. V tom analytici s programátorskou zkušeností často chybují. Nepotřebujeme popisovat všechno, co je důležité pro programování, ale vystačíme jen s pojmem atribut objektu, protože naše objekty budou „jen“ nosiči dat. Nebudeme rozlišovat, zda je příslušný atribut implementovaný do objektu vloženým údajem a nebo zda je výsledkem zpracování dat objektu pomocí metody, protože to je pro datové použití z pohledu uživatele irrelevantní.

## 6.2.2

### Tři objektové normální formy

Výše uvedené požadavky nejlépe splňuje upravená podoba Ambler-Beckova přístupu (Ambler 1997, Beck 2003). Tato verze již byla několikrát vyzkoušena například v (Bar-toška et al. 2004) a je součástí výuky. Jde o postup, který by měl při modelování předcházet všem možným následným úvahám o použití dědění, skládání a dalších vazeb mezi objekty.

## 6.2.2.1

### 1ONF

#### definice

Třída je v první objektové normální formě (1ONF), jestliže její objekty neobsahují skupinu opakujících se atributů. Takové atributy je třeba vyčlenit do objektů nové třídy a skupinu opakujících se atributů nahradit jednou vazbou na kolekci objektů této nové třídy. Schéma je v 1ONF, jestliže všechny třídy objektů v něm jsou v 1ONF.

Mějme tedy nějaký objekt  $a \in \Omega$ , kde pro  $k \geq 1$  (délka skupiny opakujících se atributů) a  $n \geq 1$  (počet opakování skupin atributů) platí, že

$\Delta(a) = [\dots, x_1^1, \dots, x_k^1, \dots, x_1^n, \dots, x_k^n, \dots]$  je takové, že  
 $\forall i \in (1, \dots, k) \xi(x_i^1) = \xi(x_i^2) = \dots = \xi(x_i^n)$ .

Potom je třeba vytvořit nové objekty  $b_j \in \Omega$  pro  $j \in (1, \dots, n)$  a upravit objekt  $a$  tak, že

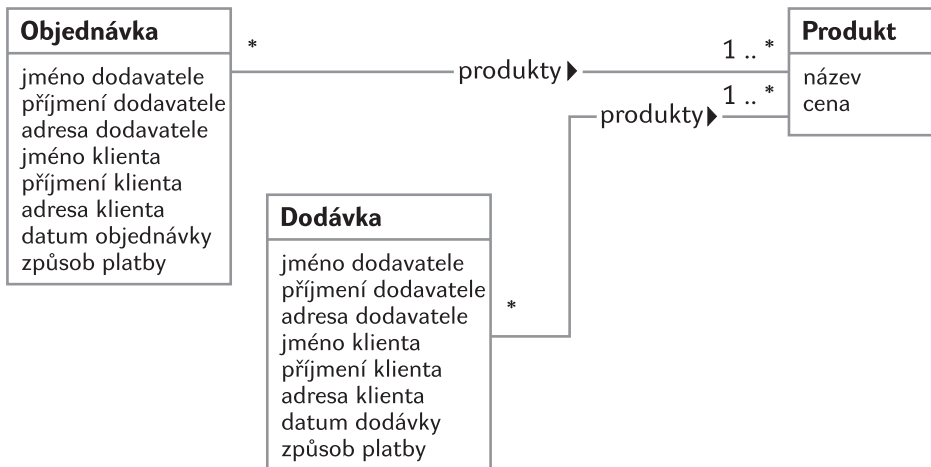
$\Delta(a) = [\dots, \{ b_j \}, \dots]$  a  
 $\Delta(b_j) = [x_1^j, \dots, x_k^j]$ .

Objednávka	Dodávka
jméno dodavatele	jméno dodavatele
příjmení dodavatele	příjmení dodavatele
adresa dodavatele	adresa dodavatele
jméno klienta	jméno klienta
příjmení klienta	příjmení klienta
adresa klienta	adresa klienta
datum objednávky	datum objednávky
způsob platby	způsob platby
název prvního produktu	název prvního produktu
cena prvního produktu	cena prvního produktu
název druhého produktu	název druhého produktu
cena druhého produktu	cena druhého produktu
název třetího produktu	název třetího produktu
cena třetího produktu	cena třetího produktu

Obr. 38: Příklad úlohy v nenormalizovaném tvaru

Na obrázku je příklad úlohy v nenormalizovaném tvaru a stejná úloha v 1ONF. Původní amblerův přístup jsme doplnili o pojistku pro případ, že návrháři sami nerozpoznají opakované skupiny atributů a nevyčlení je do samostatné třídy. Podle našich praktických zkušeností je pravidlo v tomto znění skutečně potřeba. Problém není vždy tak triviální

jako ve zde uvedeném případě. Opakované atributy se mohou ukrývat i pod různými na první pohled nenápadnými názvy.



Obr. 39: Příklad úlohy v 1ONF

## 6.2.2.2 2ONF

### definice

Třída je v druhé objektové normální formě (2ONF), jestliže její objekty neobsahují atribut nebo skupinu atributů, které by byly sdílené s nějakým jiným objektem. Sdílené atributy je třeba vyčlenit do objektu nové třídy a ve všech objektech, kde se vyskytovaly, nahradit vazbou na tento objekt nové třídy. Schéma je v 2ONF jestliže všechny třídy objektů v něm jsou v 2ONF.

Mějme tedy dva objekty  $a, b \in \Omega$  takové, že pro pro  $k \geq 1$  (délka skupiny společných atributů) je

$$\Delta(a) = [\dots, x_1, \dots, x_k, \dots] \text{ a}$$

$$\Delta(b) = [\dots, y_1, \dots, y_k, \dots] \text{ a platí, že}$$

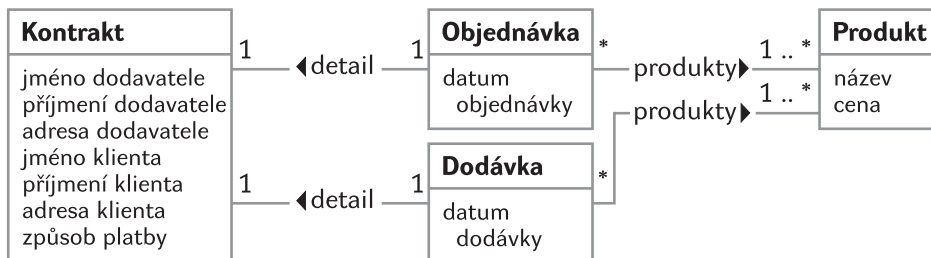
$$\forall i \in (1, \dots, k) x_i \equiv y_i.$$

Potom je třeba vytvořit nový objekt  $c \in \Omega$  a upravit objekty  $a$  a  $b$  tak, že

$$\Delta(c) = [x_1, \dots, x_k] \equiv [y_1, \dots, y_k] \text{ a}$$

$$\Delta(a) = [\dots, c, \dots] \text{ a}$$

$$\Delta(b) = [\dots, c, \dots].$$



Obr. 40: Příklad úlohy v 2ONF

Příklad demonstruje aplikaci tohoto pravidla na atributech jméno dodavatele, příjmení dodavatele a adresa dodavatele a jméno klienta, příjmení klienta, adresa klienta a způsob platby. Tyto atributy byly společné pro konkrétní objednávku i dodávku a tak bylo třeba pro ně zavést nový objekt třídy *Kontrakt*.

### 6.2.2.3 3ONF

#### definice

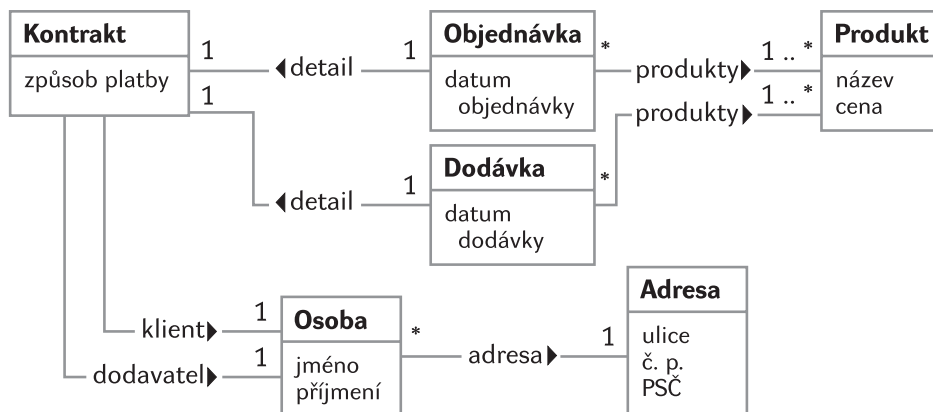
Třída je ve třetí objektové normální formě, jestliže její objekty neobsahují atribut nebo skupinu atributů, které mají samostatný význam nezávislý na objektu, ve kterém jsou obsaženy. Pokud takové atributy existují, je třeba je vyčlenit do objektu nové třídy, a v objektu, kde byly obsaženy, nahradit vazbou na tento objekt nové třídy. Schéma je v 3ONF, jestliže všechny třídy objektů v něm jsou v 3ONF.

Mějme tedy objekt  $a \in \Omega$  takový, že pro  $k \geq 1$  (délka skupiny atributů samostatného významu) je

$$\Delta(a) = [\dots, x_1, \dots, x_k, \dots],$$

kde  $[x_1, \dots, x_k]$  je samostatná množina atributů. Potom můžeme vytvořit nový objekt  $b$  a upravit objekt  $a$  tak, že

$$\begin{aligned} \Delta(b) &= [x_1, \dots, x_k] \text{ a} \\ \Delta(a) &= [\dots, b, \dots]. \end{aligned}$$



Obr. 41: Příklad úlohy v 3ONF

V našem příkladu to byly údaje o dodavateli a klientovi v objektech třídy *Kontrakt*. Tyto atributy totiž reprezentují osoby a jsou na kontraktech nezávislé. Budeme-li s nimi totiž pracovat samostatně, budou stále obsahovat stejnou informaci o osobách. A vyhovuje-li to zadání, můžeme totéž prohlásit i o adresách.

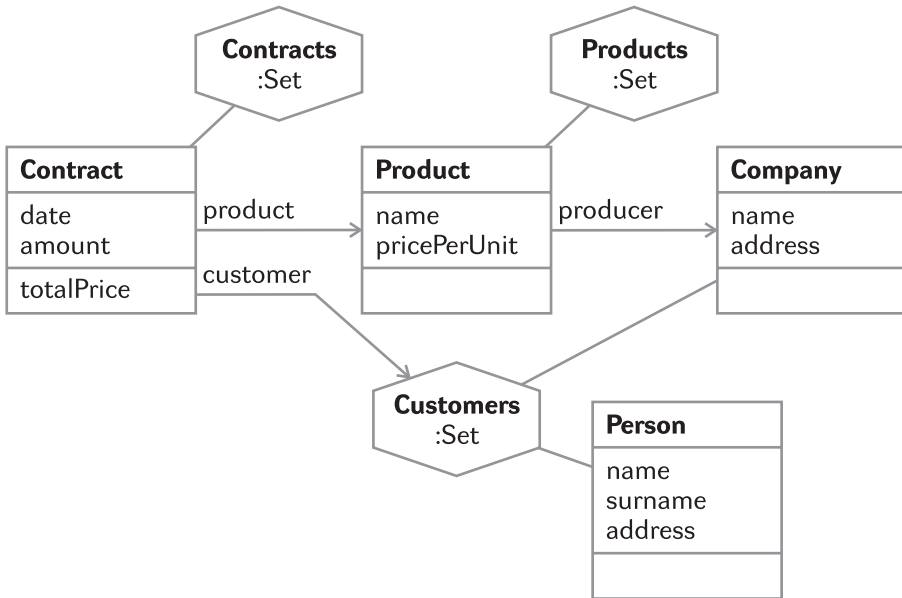
Zde předložený přístup s sebou přináší řadu otázek a námětů k přemýšlení, jak jej rozšířit. Jedním z nich je otázka, jak do tohoto přístupu zahrnout vztah dědění i další používané vazby mezi objekty, jejichž použití by mělo následovat po transformaci do 3ONF. Nabízí se například úvaha, že správným využitím dědění by se mohla zabývat následná čtvrtá normální forma.

## 6.3 Transformace datového modelu

Analytikovi se málokdy podaří navrhnout datový model napoprvé optimálním způsobem. Proto je velmi důležité, aby modelovací nástroj dokázal měnit proměnné a metody instancí a strukturu a vazby mezi třídami i v případě, kdy model již obsahuje konkrétní data. Tento požadavek se může zdát analytikům zvyklým pracovat v „klasických“ prostředích přemrštěný. Ale vývojáři znají práce s čistými objektově orientovanými prostředím tuto vlastnost běžně využívají.

### 6.3.1 Změny objektového schématu

Změnu objektového schématu si ukážeme na příkladu datového modelu.



Obr. 42: Obchod s pivem – datový model

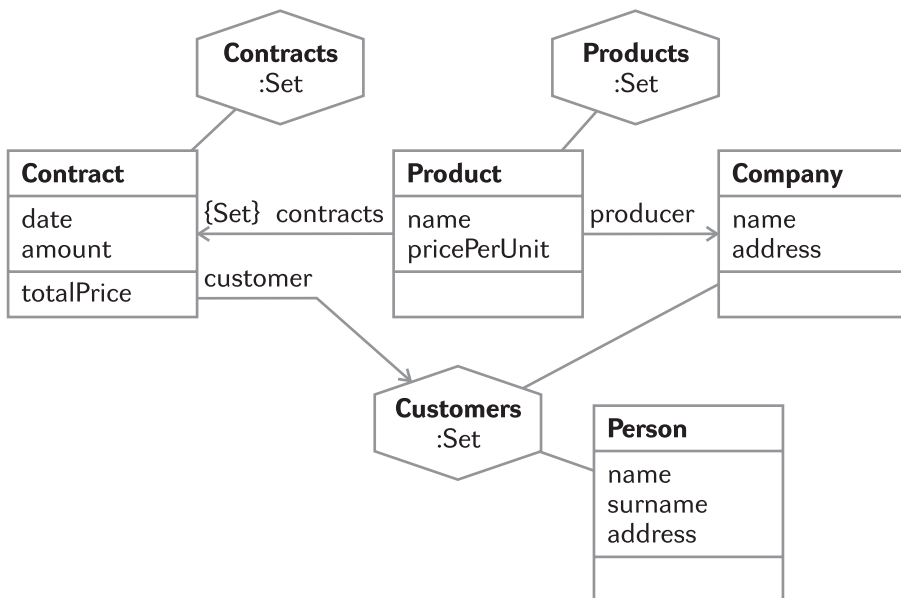
Nad tímto datovým modelem jsme vykonali dotaz na výrobky, které si objednávají zákazníci z Prahy jako

```
(Contracts // (λc | c.customer.address = Prague)) >> (λc | c.product)
```

což lze v Gemstone zapsat jako

```
(Contracts select: {c | c.customer.address = 'Prague'})
  collect: {c | c.product }.
```

Jak je tedy vidět, výrobky objednané zákazníky z Prahy získáme tak, že nejprve vybereme smlouvy zákazníků z Prahy a potom z výsledku sesbíráme výrobky.



Obr. 43: Obchod s pivem – jiný datový model

Kdyby ale měla vazba mezi třídou *Contract* a *Product* opačný směr, jak ukazuje obrázek 43, dotaz mohl by náš být jednodušší, protože namísto dvou operací budeme mít jen jeden výběr:

$Products // (\lambda p \mid \{p \triangleleft contracts\} \triangleleft customer \triangleleft address = Prague),$

což lze například v Gemstone zapsat jako

Products select:  $\{p \mid p.contracts.*.customer.address = 'Prague'\}.$

Abychom ale mohli uvedený dotaz vykonat, musíme již existující datový model přeměnit na jiný. Tuto přeměnu si popíšeme v postupných krocích:

1. Do třídy *Product* doplníme atribut *contracts* jako

$\Pi(Product) \cup [contracts] \Rightarrow \Pi(Product),$

což lze ve Smalltalku napsat jako

Product addInstVarName: 'contracts'.

2. Instancím třídy *Products* tuto novou složku naplníme daty. Každá instance třídy *Product* dostane vybranou podmnožinu svých instancí třídy *Contracts*:

$\forall p \in Products \ p \triangleleft contracts: (Contracts // (\lambda x \mid x \triangleleft product = p)),$

což se ve Smalltalku napíše jako

```
Products do: [:p | p contracts: (Contracts select: [:x | x product = p])].
```

3. Nakonec odstraníme z třídy *Contract* atribut *product*, protože již máme vazbu z druhé strany hotovou.

$$\Pi(\text{Contract}) - [\text{product}] \Rightarrow \Pi(\text{Contract}),$$

což lze ve Smalltalku napsat jako

```
Contract removeInstVarName: 'product'.
```

Takto transformovaný datový model bude stále obsahovat stejná data. Rozdíl je jen v tom, že smlouvy a výrobky budou propojeny opačným směrem.

Tento příklad transformace nám ukázal, že datový model lze řešit různými způsoby, o kterých nelze jednoznačně prohlásit, že jeden je horší a druhý lepší. Vždy záleží na konkrétních potřebách zadání. A právě možnost pracovat s konkrétními daty nebo transformovat model či vykonávat dotazy jsou užitečným nástrojem k rozpoznání té nejvhodnější podoby datového modelu.

## 6.3.2 Refaktoring

Refaktoringem se rozumí takové přepracování datového modelu, které není navenek rozpoznatelné. Jde tedy o zvláštní případ transformace, kdy dochází ke změně vnitřní struktury, ale vnější vlastnosti datového modelu zůstávají beze změny. Refaktoring se velmi často používá za účelem vylepšení kódu a nalezení znovupoužitelných komponent, podpory lepší údržby nebo možných budoucích rozšíření modelu. Tento soubor činností se v projektovém managementu nazývá také „generalizace modelu“. (Beck 2003)

Mezi refaktoring patří například následující transformace:

1. Přesun metody do nadtřídy. Protože se metoda z nadtřídy dědí, zůstává funkčnost instancí beze změny.
2. Přesun deklarace datové položky do nadtřídy. Protože se deklarace z nadtřídy dědí, zůstává funkčnost instancí beze změny.
3. Rozdělení kódu jedné metody na dvě metody, kde kód jedné metody obsahuje volání druhé metody.
4. Operace inverzní k operacím 1., 2. a 3.
5. Přejmenování jména třídy nebo metody nebo datové složky včetně přejmenování všech volání této třídy, metody nebo datové složky.
6. Vykonání transformace podle pravidla nějaké normální formy.

## 6.4 Návrhové vzory

Návrhové vzory jsou nedílnou součástí znalostní vybavy každého profesionála v oblasti tvorby softwaru. Umění aplikovat návrhové vzory je dnes stejně důležité jako znát knihovny a syntaxi příslušného programovacího jazyka. O návrhových vzorech vychází řada knih, z nichž lze doporučit především (Gamma et al. 2003) a (Beck 1997).

Návrhové vzory popisují mnohokrát vyzkoušená jednoduchá a elegantní řešení různých dílčích problémů při tvorbě softwaru. V návrhových vzorech je uchována dlouhodobá praktická zkušenost tvorců softwaru, kteří museli mnohokrát řešit a přepracovávat své modely, až nakonec došli k elegantním řešením.

Právě aplikace nějakého návrhového vzoru je velmi často cílem transformačních kroků nad postupně vytvářeným datovým modelem.

### 6.4.1 Co to je návrhový vzor

Návrhový vzor vysvětluje a popisuje nějaký dílčí problém se strukturou objektů. Každý návrhový vzor má následující strukturu:

- ↗ Název vzoru. Název je krátké nejlépe jednoslovné označení.
- ↗ Popis problému, který návrhový vzor řeší.
- ↗ Řešení, které návrhový vzor přináší. Zde se nejlépe uplatňuje kombinace diagramu s textem.
- ↗ Důsledky použití vzoru.

V profesionálních publikacích lze najít celé katalogy návrhových vzorů. Dnes známe více než 50 různých vzorů. Vynikající českou publikací, která obsahuje katalog návrhových vzorů, je (Pecinovský 2006). Většina publikací návrhové vzory člení do následujících kategorií (Gamma et al. 2003):

- ↗ Tvořivé vzory, které řeší problémy s tvorbou objektů za chodu systému. Tyto vzory jsou velmi důležité pro pokročilý návrh softwarového řešení, ale pro potřeby datového modelování se jimi nemusíme tolik zabývat jako programátoři.
- ↗ Strukturální vzory, které popisují jak objekty strukturovat, aby měly požadované vlastnosti. Tato skupina vzorů je velmi dobře použitelná i v datovém modelování.
- ↗ Behaviorální vzory, které popisují řešení problémů s implementací chování objektů za chodu systému a jejich proměn v čase. Některé tyto behaviorální vzory jsou také použitelné v datovém modelování.

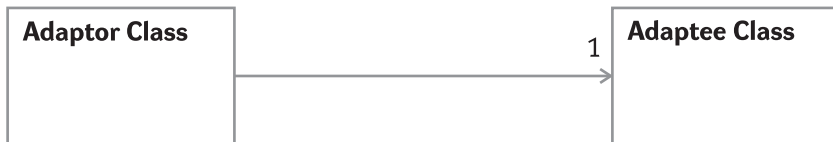
## 6.4.2 Příklady návrhových vzorů

### 6.4.2.1 Adaptér

Adaptér (anglicky Adaptor nebo jiným názvem Wrapper) je strukturální vzor. Tento vzor převádí protokol nějakého objektu na jiný protokol. Tím umožňuje využití objektů, které by jinak v datovém modelu nemohly pracovat.

#### řešený problém

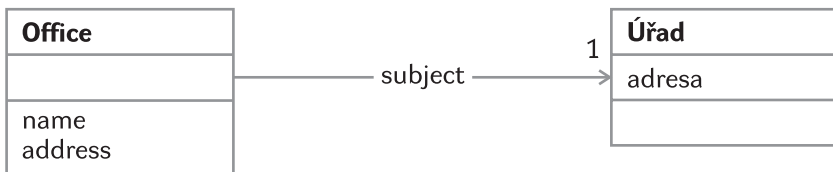
Někdy se stane, že bychom rádi použili nějaký objekt, o kterém víme, že má vlastnosti, které potřebujeme, ale náš systém od toho objektu požaduje jiný protokol, než ten, co už daný objekt má.



Obr. 44: Vzor Adaptér

#### příklad použití

Adaptér lze výhodně použít jako „tlumočníka“. Mějme náš příklad datového modelu obchodu s pivem, který potřebujeme rozšířit o další zákazníky z institucí – například z obecních úřadů. Problém ale může být v tom, že třída Úřad je již vytvořena v českém jazyce a navíc je název úřadu dán jeho adresou, takže datovou složku pro název nemají. (Například obecní úřad obce Dlouhá Lhota má oficiální název „Obecní Úřad Dlouhá Lhota“).



Obr. 45: Příklad použití Adaptéru

V kolekcích potom budeme mít objekty třídy *Office*, které dokáží adaptovat původně nekompatibilní objekty třídy *Úřad*. Metody třídy *Office* budou následující:

Pro „překlad adresy z češtiny do angličtiny“ to bude

$$\langle address, (\sigma_{subject \leftarrow adresa}) \rangle \in Meth(Office)$$

a pro sestavení oficiálního názvu úřadu to bude

$\langle name, (Obecní\ Úřad\ \sigma subject \langle adresa \rangle) \rangle \in Meth(Office),$

protože budeme skládat konstantní text „Obecní úřad“ s adresou.  
 Ve Smalltalku tyto metody vypadají takto:

```
address
    ^ subject adresa.
```

```
name
    ^ 'Obecní Úřad ', subject adresa.
```

**rozbór**

Adaptér se používá, když potřebujeme použít již hotový, ale poněkud odlišný objekt nebo když potřebujeme nějaký objekt využít v různých situacích různými způsoby a proto ho musíme „obalit“ různými adaptéry.

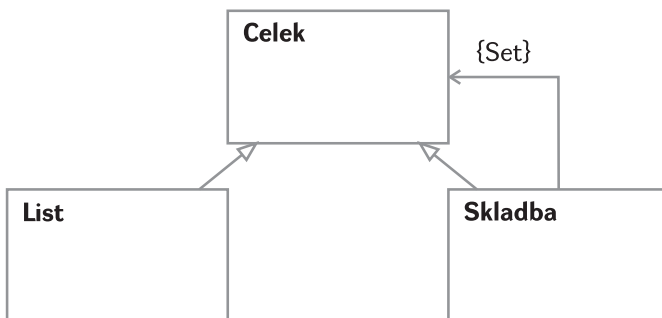
Tento vzor je podobný dekorátoru, ale na rozdíl od něj může dát objektu úplně jiný protokol. Dekorátor totiž protokol jen doplňuje o nové atributy.

### 6.4.2.2 Skladba

Skladba (anglicky Composite) je strukturální vzor. Tento vzor je návodem k implementaci hierarchických datových struktur.

**řešený problém**

Někdy se stane, že pracujeme s objekty, které se skládají z dalších objektů a ty opět z dalších objektů a tak dále. Zároveň nemůžeme předem odhadnout hloubku takového hierarchického skládání. Tento problém řeší vzor Skladba. Třída *Celek* je abstraktní třída bez instancí, ale dědí z ní třídy, které mají v datovém modelu instance. *List* je element, který dále nic neobsahuje, *Skladba* je element, který obsahuje další elementy (listy i jiné skladby). Jde o hierarchickou strukturu analogickou se strukturou souborů a adresářů na disku počítače.



Obr. 46: Vzor Skladba

**příklad použití**

Vzor skladba s úspěchem použijeme například při implementaci skladu součástek (tzv. kusovník). Každá součástka má své katalogové číslo, ale většina součástek se skládá z dalších součástek, které dále mají další součástky. (Například nějaký stroj obsahuje motor, motor obsahuje alternátor, alternátor obsahuje další součástky atd.). Jiný příklad použití může být datový model pro evidenci pracovníků a pracovních skupin. (Pracovní skupiny se skládají z jiných pracovních skupin atd.)

**rozbor**

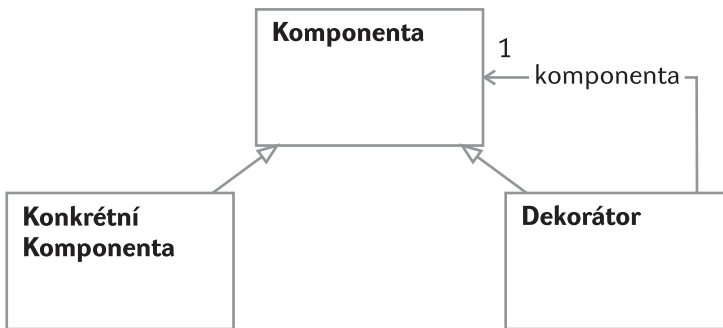
Tento vzor se často používá v kombinaci s dekorátorem. Pomocí dekorátoru se totiž výhodně implementují specifické vlastnosti listů ve skladbě.

### 6.4.2.3 Dekorátor

Dekorátor (anglicky Decorator) je strukturální vzor. Pomocí dekorátoru můžeme přidávat objektům vlastnosti aniž bychom rušili jejich stávající vlastnosti.

**řešený problém**

Někdy potřebujeme přidat vlastnosti jen některým objektům dané třídy přičemž ostatní instance dané třídy zůstávají beze změny. Dekorátorem je podtřída, jejíž instance obsahují jako složku konkrétní komponentu. Toto skládání je podobné adaptéru, ale vlivem dědění přebíráme také protokol nadtřídy.



Obr. 47: Vzor Dekorátor

**příklad použití**

Dekorátorem lze zařídit, že nějaký konkrétní datový objekt dostane více funkčnosti aniž bychom ho museli měnit. Budeme-li mít například kolekci faktur, se kterou potřebujeme pořád pracovat stejným způsobem, ale u některých faktur budeme muset navíc řešit jejich pozdní úhrady. Konkrétní komponentou je zde faktura. Dekorátorem je zde objekt, který umí řešit pozdní platby. Objekty s fakturami jsou beze změny, jen některé z nich jsou obsaženy v dekorátorech. Pokud nějakou fakturu v kolekci nahradíme dekorátorem obsahujícím tuto fakturu, kolekce faktur se z pohledu vlastností faktur nemění.

**rozbor**

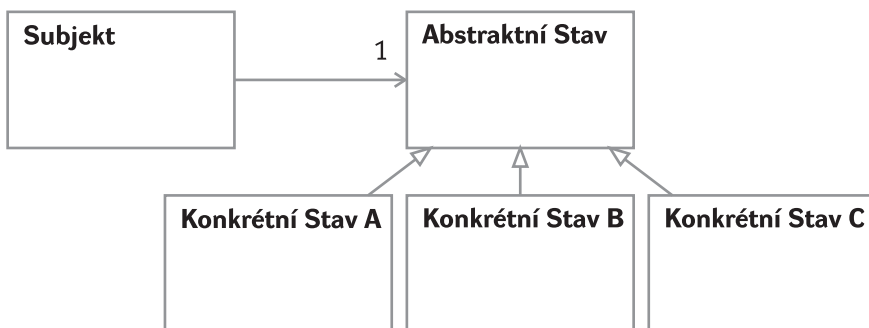
Dekorátor je svou funkcí velmi podobný adaptéru, ale nepřekrývá původní protokol ošetřovaného objektu. Na rozdíl od prostého dědění má tu vlastnost, že toto doplnění funkčnosti můžeme uskutečnit jen u některých instancí původní třídy. Kdybychom použili dědění, vyrobili bychom vytvořením nové třídy nové chování pro všechny instance této třídy, a tak i pro ty objekty, které to nepotřebují.

**6.4.2.4****Stav**

Stav (anglicky State Pattern) je behaviorální vzor. Umožňuje za chodu systému měnit vlastnosti objektu tak, že se z pohledu uživatele zdá, že objekt mění svoji třídu.

**řešený problém**

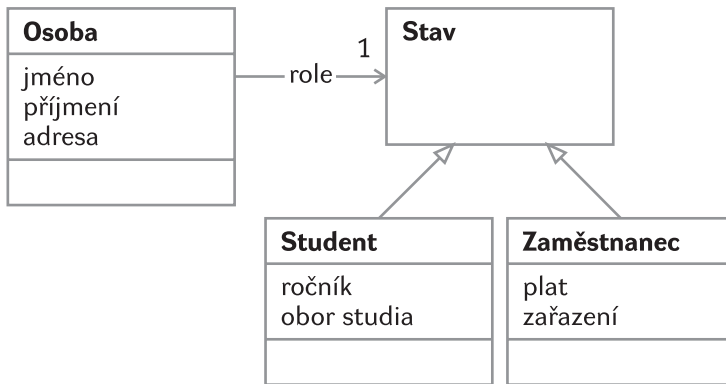
Často se stává, že nějaké objekty v datovém modelu zobrazují proměnlivé entity reálného světa. Bez použití tohoto vzoru bychom při každé takové změně museli původní objekt smazat a namísto něj vytvořit nový jiný objekt. Vzor Stav ale tyto proměny zapouzdřuje do vloženého objektu. Atributy, které změně nepodléhají, zůstávají v původním objektu, ale atributy, které jsou proměnlivé, jsou uloženy ve „vnitřním“ objektu. Navenek se celá struktura chová jako jediný objekt.



Obr. 48: Vzor Stav

**příklad použití**

Mějme například model, kde budou studenti i zaměstnanci vysoké školy. Často se však stává, že se studenti po úspěšném absolvování studia stávají zaměstnanci. Bez tohoto vzoru bychom museli v tomto případě studenta z dat vymazat a namísto něj vytvořit nového zaměstnance. Byl by to tedy jiný objekt, jen by se „náhodou“ jmenoval a měl adresu stejnou, jako odstraněný student. To je obvyklý způsob řešení. Problém je ale v tom, že pokud naše objekty jsou obsaženy v nějakých jiných kolekcích – například jako oprávněné osoby k parkování na parkovišti – museli bychom dávat do pořádku i obsah takových kolekcí. Pokud však použijeme vzor Stav, zůstávají nám osoby osobami a jen „vyměňují“ svoji roli. Identita objektů a tím pádem i konzistence obsahu kolekcí a další možný kontext našich objektů zůstává zachován.



Obr. 49: Příklad použití Stavů

### rozběr

Stav je podobný adaptéru v tom, že i zde je dvojice objektů (vnější a vnitřní), která se navenek jeví jako jediný objekt. Rozdíl je ale v tom, že u adaptéru je hlavním nosičem dat vložený objekt, ale u stavu je vložený objekt jen v roli „přídavných“ atributů, které se mohou měnit v čase.

## Seznam použitých symbolů

### 7.1

#### Formální zápis

$a, b, c, \dots$	Objekty nebo také lambda-proměnné.
$A, B, C, \dots$	Kolekce objektů.
$(\lambda x \mid \dots x \dots)$	Lambda-výraz s proměnnou $x$ v hlavičce.
$(\lambda x \mid \dots x \dots) \triangleleft$ : hodnota	Aplikace hodnoty do lambda-výrazu
výraz $A \Rightarrow$ výraz $B$	Přepis výrazu $A$ na výraz $B$ .
$jméno \Leftarrow$ výraz	Pojmenování výrazu.
$objekt \triangleleft$ zpráva	Poslání zprávy.
$objekt \triangleleft$ zpráva: hodnota	Poslání zprávy s parametrem hodnota.
$\Pi(objekt)$	Protokol objektu.
$\Delta(objekt)$	Množina datových položek objektu.
$\{a\}$	Množina prvků s vlastností $a$ .
$[x, y, z]$	Neuspořádaná $n$ -tice obsahující prvky $x, y, z$ .
$[jméno_a: x, jméno_b: y]$	Neuspořádaná $n$ -tice obsahující pojmenované prvky $x, y$ .
$\langle x, y, z \rangle$	Uspořádaná $n$ -tice obsahující prvky $x, y, z$ .
hlavička, $(x \mid \dots x \dots)$	Metoda objektu s hlavičkou a tělem.
$\sigma$	Objekt, kterému výraz (například metoda) patří.
$\emptyset$	Prázdný prvek.
$a \in A$	Objekt $a$ je prvkem množiny $A$ .
$a = b$	Objekt $a$ má stejnou hodnotu jako objekt $b$ .
$a \neq b$	Objekt $a$ nemá stejnou hodnotu jako objekt $b$ .
$a \equiv b$	Objekt $a$ je totožný s objektem $b$ .
$A \subset B$ nebo $B \supset A$	Množina $A$ je podmnožinou množiny $B$ .
$A \subseteq B$ nebo $B \supseteq A$	Množina $A$ je podmnožinou nebo rovna množině $B$ .
$A \cup B$	Sjednocení množin $A$ a $B$ .
$A \cap B$	Průnik množin $A$ a $B$ .
$A - B$	Rozdíl množin $A$ a $B$ .
$A \times B$	Kartézský součin množin $A$ a $B$ .
$Meth(a)$	Množina všech metod objektu $a$ .
$Inst(c)$	Množina všech instancí třídy $c$ . Extenze třídy $c$ .
$\zeta(a)$	Třída objektu $a$ .
$super(c)$	Nadtřída třídy $c$ .
$Set(A)$	Přeměna kolekci $A$ na množinu.
$Bag(A)$	Přeměna kolekci $A$ na ranec.
$List(A)$	Přeměna kolekci $A$ na seznam.

$List(A, (\lambda x \mid \dots x \dots))$	Přeměna kolekci $A$ na seznam s prvky seříděnými podle hodnoty dané lambda-výrazem.
$  A  $	Velikost množiny $A$ (tj. počet prvků množiny $A$ ).
$\Omega$	Množina všech objektů v systému.
$\forall a$ výraz	Výraz platí pro všechny prvky $a$ .
$\exists a$ výraz	Výraz platí pro alespoň jeden prvek $a$ (tj. existuje alespoň jeden prvek s touto vlastností).
výraz $A \rightarrow$ výraz $B$	Jestliže platí výraz $A$ , platí i výraz $B$ (implikace výrazů).
výraz $A \leftrightarrow$ výraz $B$	Vzájemná rovnost (ekvivalence) výrazů.
výraz $A \wedge$ výraz $B$	Platí oba dva výrazy (logický součin).
výraz $A \vee$ výraz $B$	Platí alespoň jeden výraz (logický součet).
$\neg$ výraz	Platí opak výrazu (logická negace).
$a \bmod b$	Zbytek po celočíselném dělení čísla $a$ číslem $b$ .
$A // (\lambda x \mid \dots x \dots)$	Výběr (selection) z množiny $A$ podle pravidla daného lambda-výrazem.
$A \gg (\lambda x \mid \dots x \dots)$	Sběr (collection) z množiny $A$ podle pravidla daného lambda-výrazem.
$A \gg [jméno\_a, jméno\_b]$	Zobrazení (projection) množiny $A$ .
$\frac{výrazA \quad výrazB}{podmínka}$	Je-li podmínka pravdivá, potom platí výraz $A$ , jinak platí výraz $B$ (větvení výrazů).
$\left( \frac{výrazA}{výrazB} \right)_{\text{nebo}}$	Vyhodnotí se výraz $A$ i výraz $B$ (sekvence výrazů).
$(výrazA \quad výrazB)$	
$\left\{ \frac{výraz}{podmínka} \right\}$	Výraz se opakuje, dokud platí podmínka (iterace výrazů).

## 7.2

### Jazyk Smalltalk

$(\lambda x \mid \dots x \dots)$	$[:x \mid \dots x \dots]$ .
$(\lambda x \mid \dots x \dots) \triangleleft$ : hodnota	$[:x \mid \dots x \dots]$ value: hodnota
jméno $\Leftarrow$ výraz	jméno := výraz.
objekt $\triangleleft$ zpráva	objekt zpráva.
objekt $\triangleleft$ zpráva: hodnota	objekt zpráva: hodnota.
$\Pi(\text{objekt})$	objekt allSelectors.
$\Delta(\text{objekt})$	objekt instanceVariables.
$\{a\}$	Set withAll: a.
$[x, y, z]$	Set with: x with: y with: z.
$\langle x, y, z \rangle$	List with: x with: y with: z.
$\sigma$	self.
$a = b$	$a = b$ .
$a \neq b$	$a \sim = b$ .

$a \equiv b$	<code>a == b.</code>
$\emptyset$	<code>nil.</code>
$a = \emptyset$	<code>a isNil.</code> nebo <code>a isEmpty.</code>
$a \in A$	<code>A includes: a.</code> nebo <code>A add: b.</code>
$A \cup B$	<code>A union: B.</code>
$A \cap B$	<code>A intersection: B.</code>
$A - B$	<code>A difference: B.</code>
$Meth(a)$	<code>a methodDictionary.</code>
$Inst(c)$	<code>a allInstances.</code>
$\xi(a)$	<code>a class.</code>
$super(c)$	<code>c superclass.</code>
$Set(A)$	<code>A asSet.</code>
$Bag(A)$	<code>A asBag.</code>
$List(A)$	<code>A asList.</code>
$List(A, (\lambda x   \dots x \dots))$	<code>A asList sortBy: [:x   ... x ...].</code>
$ A $	<code>A size.</code>
$\Omega$	<code>Smalltalk.</code>
$\forall a$ výraz	<code>a do: výraz.</code>
$výrazA \wedge výrazB$	<code>výrazA &amp; výrazB.</code>
$výrazA \vee výrazB$	<code>výrazA   výrazB.</code>
$\neg$ výraz	<code>výraz not.</code>
$a \bmod b$	<code>a \\\ b.</code>
$A // (\lambda x   \dots x \dots)$	<code>A select: [:x   ... x ...].</code>
$A \gg (\lambda x   \dots x \dots)$	<code>A collect: [:x   ... x ...].</code>
$\frac{výrazA \quad výrazB}{podmínka}$	podmínka <code>ifTrue: výrazA</code> <code>iffalse: výrazB.</code>
$\left( \frac{výrazA}{výrazB} \right)$ nebo $(výrazA \quad výrazB)$	<code>výrazA. výrazB.</code>
$\left\{ \frac{výraz}{podmínka} \right\}$	<code>výrazA whileTrue: výrazB.</code>



ODDÍL

2

**DATOVÉ MODELOVÁNÍ  
V PROJEKTOVÁNÍ A TVORBĚ  
INFORMAČNÍCH SYSTÉMŮ**



## Objektové programování

Základem objektového programování je myšlenka vyšší abstrakce při tvorbě softwaru, důraz na modularitu, znovupoužitelnost a standardizaci.

Následující kapitoly se zabývají objektovou orientací podrobněji, jak se její aplikace postupně projevila od 70. let do dnešní doby v oblasti programovacích jazyků, databázových systémů a metod analýzy a návrhu informačních systémů.

### 1.1

#### Programovací jazyky a prostředí

Pro programátorskou obec je příznačné, že si pod pojmem objektově orientovaného programování představují především jeho implementaci v konkrétním programovacím jazyce, nejčastěji v Javě, Delphi, C# 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, Dragon, Mainsail, ESP, Perl, Java, Beta, ABCL, Actalk, Plasma, Ruby, Oberon atd.

Pro naivní uživatele výpočetní techniky je příznačné, že pojem „objektově orientovaný“ ztotožňují s výhodami, které přináší grafická uživatelská rozhraní současných softwarových systémů. Podrobný popis vlastností objektově orientovaných systémů lze nalézt například v (Abadi 1996, Lacko 1996, Biermann 1990).

Objektově-orientované principy totiž neslouží jen k lepšímu návrhu grafického uživatelského rozhraní aplikací. Dobrý software 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.

#### 1.1.1

##### Objektově orientované programovací jazyky

Vznik objektově orientovaného přístupu je spojen s takzvanými „čistě objektově orientovanými“ programovacími jazyky. Tyto programovací jazyky jsou nazývány jako jazyky založené na čistých objektově orientovaných prostředích (EPOL – environment-based pure object languages). Nejznámějším jazykem této kategorie je Smalltalk. Mezi další jazyky se uvádí například CLOS a Eiffel.

Mezi společné vlastnosti EPOL patří skutečnost, že se nejedná pouze o kompilátory příslušných jazyků, které by pracovaly pod nějakým klasickým operačním systémem. EPOL 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, X-Window nebo Apple OS. EPOL je proto možné považovat za objektově orientované operační systémy (či nadstavby operačních systémů) s možností nejen vytvářet, ale i spouštět programy.

Druhou zvláštností jsou vlastní programovací jazyky. EPOL byly a jsou od začátku navrhovány jako výhradně objektově orientované. Jejich čistá syntaxe i model výpočtu jsou proto zpočátku pro klasický založeného programátora zvláštní. Z určitého pohledu jsou při osvojování si nějakého EPOL jazyka zvýhodnění úplní začátečníci oproti ostříleným programátorům například v jazycích C, FORTRAN či COBOL. V EPOL jazycích chybí některé procedurální konstrukce, jako jsou například podprogramy, příkazy skoku a podobně, protože jsou pokryty jinými konstrukcemi nad objekty. Naopak tyto jazyky dovolují elegantně využít většinu výhod objektově orientovaného přístupu ve srovnání se smíšenými jazyky, v nichž lze (nebo je programátor nucen) objektově orientovanou technologii různými způsoby šidit. Uvedené odlišnosti EPOL jazyků od smíšených objektově orientovaných (object-oriented) jazyků vedou některé autory k definici EPOL jazyků jako jazyků přímo „objektově založených“ (object-based).

## 1.1.2

### Smíšené programovací jazyky

Největší skupinou dnes používaných programovacích jazyků jsou smíšené jazyky, které vznikly obohacením klasických jazyků o vybrané objektově orientované rysy. Mezi tyto jazyky patří především Object-Pascal (Delphi), Objective C, C++, Java, C# a Visual Basic. Právě smíšeným programovacím jazykům a především jejich přímé návaznosti na klasické programovací jazyky dnes vděčíme za stále rostoucí široký zájem o využívání objektově orientovaných systémů.

I když tyto jazyky umožňují využívat většinu objektových vlastností, je pro ně charakteristická ta vlastnost, že jejich kompilátory jsou proto často i schopny přeložit „neobjektově orientované“ programy jakoby psané pro jejich „předka“, ze kterého byly vytvořeny.

Bohužel pojem OOP je u nás i ve světě spojován téměř výhradně s jeho využitím v těchto smíšených jazycích. Dochází potom k situaci, kdy je komunitou analytiků a tvůrců softwaru dokonce i v teoretické rovině nahlíženo na OOP jen skrze syntaktické konstrukce smíšených jazyků. Vlastnosti, které jsou pro OOP přirozené, ale nejsou podporovány například v Javě či C++, jsou z tohoto důvodu téměř neznámé. Je proto třeba pamatovat, proč smíšené jazyky vnikly a proč se používají. Nešlo nikdy o žádné výrazné zdokonalení OOP. Hlavním důvodem vzniku smíšených jazyků v 80. letech byl technologický kompromis, který umožnil prakticky využívat OOP na z dnešního hlediska málo výkonném hardwaru počítačů, což například dokladuje historie vzniku jazyka C++ (HOPL 1988).

## Databázové systémy

Databázové systémy jsou založené na různých datových modelech. Jde o datový model síťový (a jeho variantu hierarchický datový model), relační, objektivě relační a objektivě orientovaný. (Někteří autoři také považují za databázové datové modely ještě modely fulltextové, hypertextové a modely založené na sémantických sítích.) (Burlison 1999, Barry 1998, Silberhatz 2004)

### 2.1

#### Objektivě orientované databáze

Hlavním motivem pro vznik objektivě orientovaných databází (OODB) byly problémy s ukládáním a zpracováním objektů v relačních databázích. Relační datový model (RDM) objektivě orientovanému programování nevyhovuje, protože je příliš jednoduchý. Z tohoto důvodu vznikl tlak na konstrukci nových databázových systémů, které by lépe dokázaly pracovat s objekty. Pod obecným označením „objektivě orientované databáze“ se však skrývají dva vzájemně odlišné datové modely:

- a) Objektivě relační datový model představuje evoluční trend vývoje. Jde o doplnění relačního datového modelu o možnost práce s některými datovými strukturami, které známe z oblasti objektivě orientovaných programovacích jazyků. Většina výrobců velkých relačních databázových systémů (např. Oracle) zvolila tuto variantu. Objektivě relační datový model ale ve svých principech zůstává původním relačním datovým modelem.
- b) Objektivě 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 objektivě orientovaného programování (Loomis 1994).

#### 2.1.1

##### Objektivě orientovaný datový model

Objektivě orientovaný datový model (ODM) se výrazně liší od relačního datového modelu. Podrobný popis lze nalézt například v (Bertino et al. 1995, Silberhatz 2004, Loomis et al. 1994). Tabulky jsou v ODM pouze jedna z možných forem výstupní prezentace uložených dat. ODM se nicméně může podobat strukturám síťových databází, jak byl implementován v systémech IDMS. Na ODM je možno nahlížet jako na renesanci síťového datového modelu. Při určité míře zjednodušení lze připustit vztah:

*síťový datový model + objektové typy dat + polymorfismus = objektový datový model.*

Objektové databáze se také liší ve filosofii práce s daty vzhledem k uživatelům databáze. Relační i síťové databáze svým uživatelům poskytují prostředky, jak lze pomocí programů běžících na klientském počítači pracovat s daty na discích na serveru, takže na logické úrovni se všechna data databáze prezentují svým uživatelům jako uložená na disku vzdáleného počítače a odtamtud přístupná. Objektové databáze ale vytvářejí svým uživatelům na svých rozhraních zdání toho, že všechna data jsou přítomna na klientu, jen jsou navíc sdílená s ostatními. S daty se tedy pracuje podobným způsobem jako s běžnými proměnnými z vyšších programovacích jazyků nebo aplikačních programů.

Vyjmenujme si nyní základní charakteristiky objektového datového modelu v databázích:

1. Objektová databáze podporuje více typů kolekcí objektů na rozdíl od relačního datového modelu, kde je relační tabulka jediným „druhem“ kolekce. V konkrétních databázových systémech to může být až několik desítek různých typů tak, jak je známe z knihoven objektových programovacích jazyků. (Je to například *Set*, *MultiSet*, *Bag*, *Dictionary*, *Array*, *List*, *OrderedCollection*, *SortedCollection*...)
2. Objektová databáze rozlišuje mezi pojmem třída objektů a kolekce (kolekce) objektů. Třída je jen realizace datového typu objektů a kolekce je jen úložiště pro objekty. Můžeme mít například více kolekcí 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 kolekci provádět například selekci. Relační databáze tuto nezávislost mezi druhem dat a úložištěm dat nezná, protože tabulky mají zároveň roli třídy i kolekce.
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 OODB je ale důležité si uvědomit, že mezi atributy objektů patří nejen jejich datové složky (jako v RDM), ale i metody poskytující další data. (Například atribut „věk“ osoby z příkladů v této knize.)
4. Polymorfismus objektů nevzniká pouze děděním tříd. Pokud mají objekty společné atributy, 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 změň 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 objektové databázi 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 SŘBD. V takové databázi potom její uživatel vidí objekty, které se přímo propojují a skládají mezi sebou. Objekty tak nemusejí obsahovat „cizí klíče“, aby se mohly skládat. Tak tomu bylo i v příkladu datového modelu v systému Gemstone v této knize.

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 objektové databázi migrovat mezi různými třídami a 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 kolekce)
tabulka	1. třída objektů (jako datový typ) 2. kolekce objektů (i z různých tříd)
atribut (položka řádku tabulky)	1. datová složka objektu 2. metoda objektu, která vypočítává data
primární klíč (nemá vztah k umístění v paměti)	OID (je ukazatelem do virtuální paměti)
Propojení dvou záznamů dvou tabulek tak, že hodnota cizího klíče u jednoho záznamu je shodná s hodnotou primárního klíče u druhého záznamu.	Skládání objektů. Datovou položkou objektu je celý objekt a ne jen hodnota jeho „klíče“.

Tab. 12: Porovnání relačního a objektového datového modelu

## 2.1.2

### 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,

kteřé ODM má. Jinou možností je převzít schéma objektů a tříd tak, jak jsou navrženy v softwarové aplikaci, která má s databází pracovat. Právě proto byly objektové databáze vynalezeny. Jenomže struktura objektů výhodná pro algoritmy v aplikaci může komplikovat jejich efektivní databázové zpracování nad datovými objekty. Jde totiž o to, že řada objektových struktur v programech je založena na dynamickém chování, které si pro kolekce s velkým počtem datových objektů uložené na externích pamětech nemůžeme dovolit.

Při tvorbě objektové databáze jsme bohužel odkázáni na zkušenost expertů – programátorů objektových databází. Znalost pokročilých technik návrhu jako například objektové normalizace nebo návrhové vzory je mezi analytiky téměř neznámá. Proto si zde popíšeme praxí vyzkoušený postup, jak objektovou databází vytvořit:

1. Sestavit konceptuální model úlohy. Zde je možné použít diagram tříd UML. Zatím ale modelujeme jen kolekce (množiny) a nezabýváme se nadbytečnými detaily softwarové implementace. Atributy objektů zde rozpoznané budou později implementovány nejen jako datové složky, ale také jako metody. V tomto kroku se to ještě nerozlišuje.
2. K prvkům množin najít potřebné třídy a provést normalizaci tříd.
3. Rozhodnout, které atributy budou implementovány datovými složkami a které metodami. Tyto metody pak napsat.
4. Naplnit databázi daty. Tedy vytvářet konkrétní objekty (instance) tříd a ukládat je do vybraných kolekcí a otestovat pomocí nich správnost modelu.

## Metody analýzy a návrhu informačních systémů

Neustálé zkracování vývojových, realizačních i produkčních cyklů je důsledkem prorůstání softwarových aplikací do všech složek lidských aktivit. V oblasti tvorby softwaru jsou zřetelné usilovné snahy po dokonalejším, spolehlivějším, efektivněji vytvořeném programovém vybavení a jeho zajištění v nebyvalých rozměrech, vytvářeným s průměrnými náklady. Existuje zde nepochybně paralela s výrobními odvětvími a nabízí se tu srovnání s průmyslovou revolucí z přelomu 18. a 19. století. Ta měla za důsledek přesun objemu výroby od drobných řemeslníků k prvním velkovýrobci, což bylo doprovázeno novou technologií i organizací práce.

Proto bylo jen logické, že společně s rozvojem praktického používání OOP došlo k rozvoji metod analýzy a návrhu informačních systémů využívajících objektovou technologii. Zpočátku se používaly klasické metody založené na strukturovaném přístupu a role nástrojů OOP byla omezena jen do oblasti implementace. Tento přístup se však neosvědčil. Přibližně od konce 80. let bylo vyvinuto několik vzájemně odlišných metodologií pro objektově orientovanou analýzu a návrh (Drbal 1996, Wilkie 1993) Patří mezi ně především:

1. OMT – Object Modelling Technique, jejímž autory jsou J. Rumbaugh, M. Bláha, W. Premierlani, 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 vybrané objektové, tak i klasické nástroje. Nejčastěji se používala pro návrh databázově orientovaných aplikací s objektovým klientem a relačním serverem (Blaha et al. 1995, Rumbaugh et al. 1991).
2. Coad-Yourdonova metoda, jejímiž autory jsou P. Coad a E. Yourdon (autor velmi používané metody klasické strukturované analýzy a návrhu). Metoda byla poprvé publikována v časopise American Programmer v roce 1989 a posléze v knihách obou autorů nakladatelství Yourdon Press. Z uvedených technik je nejsnazší co do počtu používaných pojmů (Yourdon 1994).
3. 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 byla určena pro týmový vývoj rozsáhlých aplikací v jazyce C++ a Smalltalk, a ze všech zde vyjmenovaných metod pokrývá nejvíce objektových vlastností (Booch 1994).
4. 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 a je dodnes používána (Jacobson 1992).

5. 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.
6. 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, rozšířené o objektové vlastnosti. Je výhodná pro použití v objektově orientovaných databázích.
7. Unifikovaný modelovací jazyk UML podporují od roku 1996 autoři G. Booch, J. Rumbaugh a I. Jacobson pod záštitou firmy Rational Inc. Metoda začala být publikována průběžně na Internetu a v sérii knih, vydávaných firmou Rational Inc, která dodává také vlastní CASE nástroj Rational Rose. UML je dnes doporučovaným průmyslovým standardem pro notaci (způsob kreslení) diagramů a je stále ve vývoji. Představuje sjednocení myšlenek původních metod svých autorů na platformě OMT. Ve srovnání s původní OMT je patrný posun směrem k větší míře podpory objektových vlastností a určitý odklon od původních „hybridních“ vlastností OMT – například zrušení datového modelování pomocí DFD. Je třeba mít na paměti, že UML je ve skutečnosti jen jazyk, tedy návrh standardu k zakreslování nejrůznějších objektových diagramů, který se navíc stále vyvíjí, mění a doplňuje (UML 2004).
8. Metoda J. Martina (jako E. Yourdon známý i z dřívějšího období) a J. Odella. Metoda byla publikována v sérii knih nakladatelství Martin Books, a představuje jako UML pokus o sjednocení objektových zkušeností předchozích metod. Používá velké množství nejrůznějších diagramů a pojmů (Martin et al. 1995).

Kromě výše uvedených metodologií vzniklo ještě několik vesměs velmi kvalitních technik nebo i jen nástrojů, jejichž rozsah však byl omezen na několik publikací nebo na výuku softwarového inženýrství na vysokých školách, anebo se jedná o firemní know-how (Henderson-Sellers 1994, Choopy 2004).

### 3.1

## Otázka transformace zadání od uživatele do podoby objektového modelu

I když je tvorba objektového modelu usnadněna výše uvedenými metodami a nástroji, které říkají co a jak lze při modelování použít, jedná se o velmi obtížný a přitom klíčový problém. Zatím není k dispozici jednoznačný a všeobecně uznávaný návod, který by posloužil k efektivní transformaci zadání problému do formální podoby modelu. Velkou roli zde hraje zkušenost návrháře a jeho schopnost komunikovat se zadavateli. Značným problémem je rozpoznání objektů a jejich vlastností při modelování zadání pro vytvářený systém. K řešení těchto problémů byly vyvíjeny různé různě složité techniky.

Tou nejjednodušší – až naivní, a také samozřejmě nejméně účinnou – je návod vycházející z jazykové analýzy textu zadání (podstatná jména = objekty, slovesa = metody, přídatná jména = atributy) (Rumbaugh et al. 1991). Mezi sofistikovanější metody, které byly pro tento účel vyvinuty, patří:

1. Object Behavioral Analysis, která slouží k získávání prvotní představy o objektovém modelu. Jedná se o iterativní techniku používající především tzv. modelující karty. Touto technikou začíná projekt tvorby informačního systému a jejím výstupem je objektová informace dostatečná k sestrojení prvotního modelu (Rubin et al. 1992).
2. Behavioral Constraints (Goldberg 1995), která slouží k transformaci prvotního objektového modelu na konceptuální metodou „filtrace“ skrze formálně vyjádřená pravidla, která omezují použití jednotlivých možných hierarchií mezi objekty.
3. Applying Patterns – využívání vzorů (poprvé publikováno P. Coadem na internetu, dále například (Beck 1997) a poté řada dalších publikací). Technika slouží ke stanovení vhodné struktury objektů v konceptuálním modelu pomocí znovuvyužívání objektových vzorů, jenž jsou formalizované části znalostí z předchozích projektů. Tyto vzory jsou v podobě konceptuálních podgrafů a jsou schopny se vzájemně kombinovat a v nových podmínkách hrát nové role. Používání této techniky spočívá ve vytváření konceptuálního modelu jako orientovaného grafu metodou skládání a substitucí podgrafů představujících jednotlivé vzory.
4. Structural Transformations (Nierstrasz 1990, Goldberg 1995), jenž je zvláštním případem předchozí techniky pro potřeby postupné transformace objektového modelu od modelu popisujícího zadání k modelu popisujícímu řešení.

## Dnešní stav objektových nástrojů a technik

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 výsledky ve formě zpráv sdružení OMG (Object Management Group). Toto mezinárodní sdružení firem, vysokých škol a dalších institucí se zabývá standardizací OOP, podporou aplikovaného výzkumu a jeho uplatněním v praxi (OMG). V rámci norem ANSI existují schválené standardy nebo zatím jen návrhy objektových programovacích jazyků. Každým rokem ve světě probíhá několik konferencí, kde je diskutována jak praxe, tak i příspěvky k matematické teorii OOP.

### 4.1

#### Přínosy OOP

OOP již také není třeba obhajovat a vysvětlovat jeho přednosti, mezi které v první řadě patří znovupoužitelnost, komponentový přístup a celkově snadnější tvorba softwaru. OOP již není doménou nadšenců a postupně se stává hlavním stylem tvorby softwaru. Ve většině oblastí již dostatečně prokázal svoje přednosti a určitě jde pojmenovat nemálo oblastí, kde by dnes bez jeho využití nebylo možné aplikace v rozumném čase a s rozumnými náklady ani sestavit.

Dnešní roli OOP lze charakterizovat následovně:

1. OOP se stalo ve své smíšené formě hlavním paradigmatem tvorby softwaru. Téměř všechny dnešní programovací jazyky lze zařadit do kategorie smíšených jazyků podporujících různou měrou objektový přístup. Téměř všechny metody analýzy a návrhu informačních systémů využívají objektový přístup.
2. Došlo ke sjednocení notací používaných pro objektové modelování. Na rozdíl od ještě nedávné minulosti nyní existuje jen jeden všeobecně uznávaný standard UML.
3. Dnešní tvorba softwaru v praxi využívá pokročilá vývojová prostředí. Žádný v praxi používaný programovací systém dnes není tvořen pouze kompilátorem a knihovnou zdrojových kódů. Současná pokročilá vývojová prostředí dovolují pracovat s komponentami, obsahují prostředky pro inkrementální programování, generátory testů, vizuální nástroje modelování a programování, podporují používání návrhových vzorů a refactoring.
4. Těžiště výzkumu nových metod a metodologií se přesunulo od hledání způsobu tvorby jedné konkrétní softwarové aplikace z pohledu vývojáře k širším aspektům informačního managementu. Dnes je oblastí zájmu problematika řízení informačních projektů, spojení tvorby a údržby informačních systémů, otázky kvality atd.

## 4.2 Problémy OOP

Ne všechno je však bezproblémově vyřešeno. Současná doba řeší tři problémy, které vycházejí z předchozího vývoje OOP: odklon od původního OOP, nedokonalost UML a nedokonalost metod analýzy.

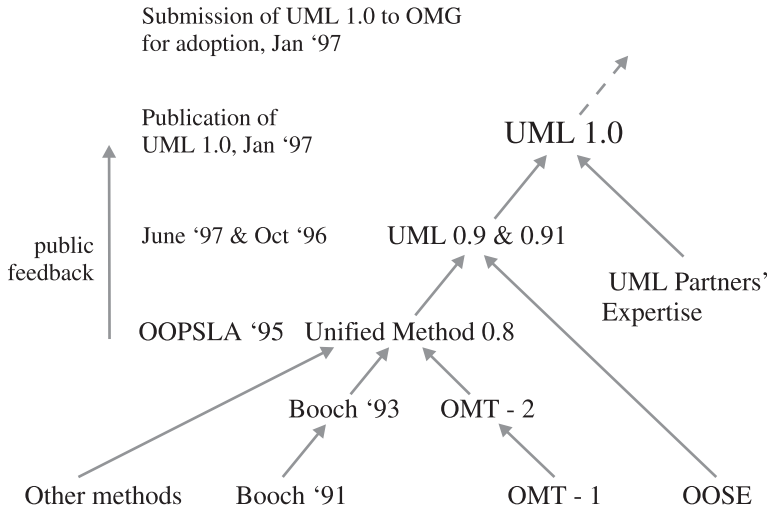
### 4.2.1 Odklon od původního OOP

Došlo k odklonu od „čistého“ OOP. Hybridní řešení, která byla zamýšlena původně jako dočasný kompromis z důvodu technologické nouze (jak bylo diskutováno v kapitole 3.1.2), se začala používat jako hlavní prostředek pro tvorbu programů. Tato skutečnost ale skrývá kromě svých praktických výhod jeden velký problém. Jde totiž o to, že po teoretické stránce je hybridní OOP jen ad-hoc vybranou podmnožinou celého paradigmatu tak, jak se jí povedlo implementovat v jazycích C++ a podobných.

Konkrétně to znamená, že některé vlastnosti, se kterými se velmi vážně pracovalo v průběhu 80. a 90. let (Shriver et al. 1987), se dnes nevyužívají, přestaly být předmětem výzkumu a jsou dokonce zapomenuty. Většinová komunita tvůrců softwaru, která zná jen hybridní OOP, již dokonce ve dvou případech některé vlastnosti „znovuobjevuje“. To je konkrétně případ aktorového výpočtu se softwarovými agenty a nebo aspektového programování. Obojí vychází z původních prací ze 70. let týkajících se objektového modelu výpočtu a mohlo by být logickým pokračováním těchto původních myšlenek, jak dokazuje například (Nierstrasz 1990, Hopkins 1992, Shoham 1993, Agha 1994). Dnes se ale prezentují jako nové přístupy, které OOP údajně překonávají, dokonce až negují.

### 4.2.2 Problém s UML

Další problém je otázka samotného modelovacího jazyka UML. Před vznikem UML, v první polovině 90. let, jsme měli několik mezi sebou soupeřících objektových metodologií s navzájem odlišnými notacemi. Jednalo se o tzv. objektové metodiky první generace. Mnoho softwarových firem nepoužívalo pouze jednu metodiku, ale kombinaci několika – nejčastěji objektové modely z OMT spolu s interakčními diagramy z metody Boochovy a Use-Case přístupem z Jacobsonovy metody OOSE. Většina z těchto metodik se poté stala základem pro jazyk UML, jak ukazuje schéma z dokumentace o UML.



Obr. 50: Vývoj UML

UML přinesl sjednocení dosavadních notací. Notace UML nejvíce vychází z OMT a stala se uznávaným standardem. UML ale do sebe zahrnul z původních metodik a notací mnoho navzájem různých prvků a stále je zahrnuje. Je to například tzv. „business extension“ z původní Jacobsonovy metody, která byla přidána ve verzi 1.2, a nebo nedávno uveřejněná verze 2.0, která pohltila metodiku SDL pro podporu real-time procesů (UML 2004).

UML je „jen“ grafický jazyk. To by samo o sobě nemělo vadit – je dobře, že od roku 1996 máme konečně pro objektové modelování k dispozici standard. Problém je ale v tom, že k „univerzálnímu“ jazyku nemáme odpovídající metodiku a tak se za metodiku mnohdy považuje samotná znalost UML. Drtivá většina odborné literatury o UML svým čtenářům předstírá, že metodika problém není, a že se naučí modelovat v UML tím, že se naučí kreslit jednotlivé diagramy. Autor má podobné negativní zkušenosti i s náplněmi školicích kurzů „moderního objektového modelování“.

UML také není rozhodně nástrojem, který laik za rozumně krátkou dobu (třeba během 15 minut na začátku schůzky s analytiky) pochopí tak, že je schopen číst a rozumět diagramům. Toto není nereálný požadavek, protože v minulosti bylo možné takto pracovat s entitně relačními a data-flow modely. Bohužel v objektově orientovaném modelování podobný elegantní a jednoduchý nástroj nemáme. Namísto toho jsou zadavatelé posílání na dlouhá školení o UML, kde jsou nuceni pracovat s CASE nástroji konkrétní firmy. Pro jedince, kteří neovládají programování, je UML příliš složitý a potom i nesprávně na základě této zkušenosti interpretují celý objektový přístup. Je samozřejmě možné pro neprogramátory vybrat z UML přijatelnou podmnožinu pojmů, ale většina odborné literatury i výklad v kurzech se příliš a často zbytečně opírá o programátorské znalosti. Tyto problémy UML popisuje podrobně (Graham et al. 2002) Jde především o následující:

- a) UML modely obsahují příliš mnoho pojmů. Pojmy jsou na různých úrovních abstrakce, někdy se i významem překrývají (např. vazby mezi use-cases), dokonce se jejich

definice v různé literatuře liší. Proto může stejný model jinak interpretovat analytik a jinak programátor (typickým příkladem jsou asociace mezi objekty).

- b) V UML diagramech je více variant pro zobrazení některých detailů v modelech (např. kvalifikátory a vazební objekty nebo stavové diagramy, které jsou mixem automatů Mealyho a Mooreova typu). Záleží na analytikovi, kterou variantu si vybere.
- c) Některé pojmy jsou nedostatečně definovány (např. události ve stavových diagramech), jeden symbol UML pokrývá více odlišných pojmů (např. v diagramu sekvenčí splývá datový tok mezi objekty s řídicím tokem, což kromě jiného komplikuje implementaci).
- d) I když je UML po grafické stránce celkově vydařený, tak podle některých analytiků například vadí totožný symbol obdélníka pro instanci a třídu (rozdělují se jen vnitřním popisem) a také směr šipky dědění, která vede směrem k rodičovskému objektu, i když v kódech programovacích jazyků (i při laických výkladech co to je dědění) je dědičnost realizována opačným směrem – od rodičovského objektu k jeho potomkovi.
- e) UML pracuje s typovanými datovými modely, což čistě objektově orientovaným jazykům, které jsou dynamické, nevyhovuje. Stejně je to i s objektovými databázemi.
- f) UML přímo nepodporuje některé vazby mezi objekty (např. závislost mezi objekty nebo polymorfismus bez přítomnosti dědění), které sice smíšené jazyky neznají, ale pro čisté objektové programování jsou důležité. V takových případech UML nabízí jen možnost definovat si vlastní stereotypy.
- g) Většina sofistikovaných objektově orientovaných algoritmů (například některé behaviorální návrhové vzory) je založena na spolupráci více objektů ve vhodné datové struktuře. V UML se však taková řešení musejí rozkreslovat do oddělených diagramů popisujících zvlášť statickou datovou strukturu, zvlášť výpočetní mechanismus a zvlášť stavy a přechody objektů a to ještě jen po jednotlivých objektech (nelze jedním diagramem znázornit vzájemné souvislosti operací, stavů a přechodů více objektů mezi sebou).
- h) Přestože dodavatelé CASE nástrojů a ostatní lidé zainteresovaní na trhu spojeném s UML tvrdí opak, UML velmi špatně podporuje první fáze analýzy informačních systémů, kdy je třeba modelovat business kontext budované aplikace. Zjednodušeně řečeno: pro tuto fázi UML má nedostatečné prostředky, nutí analytika modelovanou problematiku transformovat do podoby, která nedovoluje zachytit všechny potřebné detaily zadání a naopak jej nutí příliš brzy přemýšlet „softwarově“. Takto sestavený konceptuální model je potom samozřejmě nedostatečný a vývojáři se musejí při kódování vracet k modelované problematice a upřesňovat detaily systému.

Na závěr je třeba prohlásit, že UML jde samozřejmě použít. V tomto textu nebylo záměrem jej zpochybnit. Jde o to, že velmi záleží na schopnostech analytiků a manažerů projektů, aby si zajistili správnou interpretaci svých modelů, protože vývojáři mají vždy silnou tendenci všechny modely chápat jen jako zobrazení struktur pro konečný zdrojový kód. Diskutovanou problematiku lze shrnout do následujících bodů:

1. UML není metoda, je to „jen“ zobrazovací prostředek.
2. UML je komplikovaný – kdo neovládá programování, obtížně se ho učí a je velmi pravděpodobné, že ho nesprávně chápe.

3. UML nezdůrazňuje, které pojmy se mají používat ve fázi analýzy a které až ve fázi návrhu a implementace.
4. UML málo podporuje čistě objektové programovací jazyky a objektové databáze.
5. UML málo pomáhá tam, kde součástí projektu je hledání a upřesňování zadání.
6. Modelování se v UML často interpretuje jen jako zobrazování budoucího zdrojového kódu aplikace (nebo dokonce jen jako zobrazování dat uvnitř této aplikace).

### 4.2.3

## Nedostatečnost metod analýzy

Současné široce používané metody objektové analýzy selhávají v situacích, kdy je potřeba rychle a přesně zachytit specifické potřeby uživatele. Tento fenomén také souvisí s UML, ale podrobněji se jím budeme zabývat v samostatné kapitole.

## 4.3

### Pokrok v oblasti programovacích jazyků a prostředí

I když se dnes čisté objektové programovací jazyky používají méně často na úkor smíšených, což odborníci v době jejich vzniku nepochybně nepředpokládali, lze přesto prohlásit, že na konci 90. let došlo k velmi výraznému kvalitativnímu pokroku v oblasti nástrojů pro tvorbu softwaru. V celém minulém období vývoje počítačů až do 90. let (jen s výjimkou některých čistých objektových prostředí) byl základním nástrojem pro programování překladač příslušného programovacího jazyka (samozřejmě doplněný o debugger, knihovnu, nápovědu atd.).

Dnes je situace úplně jiná. V praxi se dnes téměř výhradně pracuje s vývojovými prostředími. Jsou to takové nástroje, kde je možné pracovat odděleně s jednotlivými komponentami, části kódu lze tvořit interaktivně a s podporou vizuálních prostředků, některá prostředí dovolují do programu zasahovat za jeho chodu, běžně se používají modelovací nástroje integrované do programovacího prostředí, automatické generátory testů, databázové sdílení kódů v rámci celého týmu, podpora pro balíčkování a verzování aplikací, refaktoring, reverzní inženýrství a další vymoženosti. Samotná znalost programovacího jazyka a knihoven pro praktickou práci zdaleka nestačí. Dnešní vývojáři musejí prakticky umět pracovat s návrhovými vzory, znovupoužitelnými komponentami, řeší problémy integrace nových částí informačního systému mezi stávající struktury atd.

## 4.4

### Objektový přístup v databázových systémech

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 (Bertino et al. 1995).

Po více než 15 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 tří největších dopravců mezi USA a Evropou), informačních systémech dodavatelů elektřiny (např. Florida 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.

Pro zajímavost lze podotknout, že objektově orientovaný databázový model vychází z prací nad čistým objektovým datovým modelem a ne hybridním. Dochází tak v praxi k paradoxním situacím, kdy používaný databázový systém (např. GemStone) poskytuje více možností využití OOP, než je k němu připojený klient (používající jazyk C++) schopen využít. Je docela dobře možné, že právě objektové databáze přinesou očekávanou renesanci čistého objektového programování.

## 4.4.1

### Čisté objektové a objektově relační databáze

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. Proto si zopakujeme jejich charakteristiku:

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 (Loomis 1994).

Relačně objektová technologie je dodnes 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, O<sub>2</sub>, 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 větší než Oracle).
3. Konservativní myšlení potenciálních uživatelů, které se konkrétně projevuje ve formulaci požadavků na databázové aplikace, 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 máme řadu důvodů 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 a znalostní systémy, 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.

Ještě je v tomto kontextu vhodné 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. Trend vývoje tedy spíš naznačuje, že se budou používat všechny dnes známé datové modely.

Pro zajímavost lze podotknout, že objektově orientovaný databázový model vychází z prací nad čistým objektovým datovým modelem a ne smíšeným. Dochází tak někdy v praxi k paradoxním situacím, kdy objektový databázový systém (např. GemStone) poskytuje více možností využití OOP, než je k němu připojený klient schopen využít. Je docela dobře možné, že právě objektové databáze přinesou očekávanou renesanci čistého objektového programování.

#### 4.4.2

### Situace v České republice a ve světě

Bohužel se dnes „nerelačními“ objektovými databázemi v České republice 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 dílčích teoretických prací, které jednotlivě dokazují účelnost objektově orientovaného datového modelu v databázových systémech, používají se zatím v praxi objektových databázových aplikací většinou jen postupy původně určené pro práci s relačními systémy a nebo jen intuitivní přístupy založené na zkušenosti s imperativními objektově orientovanými programovacími jazyky.

#### 4.4.3

### Formální techniky návrhu objektových databází

Přestože již existuje mnoho dílčích teoretických prací, které jednotlivě dokazují účelnost objektově orientovaného datového modelu v databázových systémech, používají se zatím v oblasti metod analýzy a návrhu objektových databázových aplikací vesměs jen postupy původně určené pro práci s relačními systémy a nebo jen intuitivní přístupy založené na zkušenosti s imperativními objektově orientovanými programovacími jazyky. Předpokládá se využití refaktoringu, návrhových vzorů a agilních metodik.

Bohužel pro objektový datový model zatím není žádná všeobecně uznávaná a používaná technika nebo metoda návrhu.

Je sice 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 algoritmy v softwarové aplikaci může podstatně komplikovat jejich efektivní databázové zpracování. Struktury mnoha návrhových vzorů totiž využívají dynamické vazby mezi objekty, řetězení metod atp.

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. Taková tvrzení se objevují i v renomované literatuře.

Tvrzení o větší rychlosti může velmi dobře platit, 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 formálními technikami to je ještě složitější. Velký pokrok učinili Scott Ambler a Kent Beck, jedni z pionýrů agilních metodik. Ve svých knihách prezentují návrhy prvních tří „objektových normálních forem“, které se týkají správného návrhu struktury tříd objektů a jsou odvozeny z pravidel normálních forem pro relační databáze. (Beck 2003, Ambler 1997)

## 4.5

### Metody řízení projektů informačních systémů

Jak již bylo řečeno, těžiště výzkumu nových metod a metodologií se přesunulo od hlediska tvorby jedné konkrétní softwarové aplikace k širším aspektům informačního managementu. Důležitou otázkou dneška je problematika řízení softwarových projektů využívajících objektovou technologii.

V oblasti řízení a podpory softwarových projektů je k dispozici velké množství metodik. Podrobný přehled podává například (Buchalceková 2005). Pokud se zaměříme na oblast OOP a zároveň vynecháme specifické typy softwaru, jako jsou například webová sídla nebo software pro řízení v reálném čase, lze prohlásit, že jsou u nás i ve světě nej-používanější dvě metodiky, které se staly uznávaným standardem. První z nich je „Object-Oriented Software Process Pattern“ od Scotta Amblera (Ambler 1998, 1999) a druhou je „Rational Unified Process“ (RUP) firmy Rational (Jacobson et al. 1999). Ve světě i u nás je známější RUP, pro který jsou již dokonce dostupné i podpůrné softwarové nástroje. Ve srovnání s amblerovou metodikou je RUP podrobnější a složitější, ale Amblerův přístup má podle názoru autora čtyři významné přednosti:

- a) ... je jednodušší a je důsledně procesně orientovaný ve srovnání s RUPem, na který lze nahlížet jen jako na obrovskou sadu jednotlivých objektových i neobjektových nástrojů a technik, ze kterých si projektový manažer musí umět správně vybrat.

- b) ... zabývá se i fází údržby a provozu již vyrobeného systému, čímž dobře do procesu začleňuje správu softwarové konfigurace, podporu uživatelům a vytváří předpoklady pro zpětnou vazbu na zahájení nových projektů. RUP se postimplementačními fázemi nezabývá.
- c) ... lépe podporuje tvorbu v čistých objektových jazycích a databázích (důraz na znovupoužitelnost, refactoring, komponentový přístup...) a lépe do celkového procesu začleňuje využívání metrik. RUP je šitý na míru konzervativnější technologii hybridních programovacích jazyků (např. C++ nebo Java) a relačních databází.
- d) ... lze využít i pro jiné metody analýzy a návrhu, než z dílny UML. Amblerův přístup je slučitelný například s metodou BORM a dokonce i s extrémním programováním.

Amblerova metodika byla samotným autorem nedávno doplněna a upravena. Kromě nesporné inovace a vylepšení ale došlo bohužel i k tomu, že autor „obrousil“ hrany některých svých tvrzení a do své metodiky zahrnul i některé sporné prvky z RUP. Tato varianta Amblerovy metody je známá pod názvem EUP (Enterprise Unified Process). V praxi se velmi často setkáváme s aplikacemi obou metodik současně.

## 4.5.1

### Iterativní a evoluční versus sekvenční model životního cyklu

Pro tvorbu softwaru lze použít různé modely životního cyklu. Podrobný přehled podává například (Larsson et al. 2002) nebo (Wilkie 1994). Modely životního cyklu lze rozdělit do tří základních kategorií: sekvenční, iterativní a evoluční.

### 4.5.1.1

#### Sekvenční model životního cyklu

Sekvenční model je založený na myšlence, že existuje systematický postup, jak dojít od zadání k řešení pomocí řady na sebe navazujících činností, které lze předem naplánovat. Výstup jedné aktivity je vstupem následující. Nejznámější varianta sekvenčního modelu je tzv. vodopádový model. Jde o nejstarší přístup k této problematice a je stále nejvíce používán. Pro zajímavost je třeba ještě uvést, že tento přístup se používá ve velké většině i u jiných inženýrských disciplín, kde je často jediným možným modelem tvorby – je tomu tak například v architektuře a stavebnictví.

Jeho nespornou výhodou je dobrá možnost řízení a sledování postupu řešení. Jeho nevýhodou, pro kterou je v oblasti OOP zatracován, je skutečnost, že vyžaduje mít na počátku přesně a úplně definované požadavky.

### 4.5.1.2

#### Iterativní model životního cyklu

Iterativní model je založený na myšlence, že je možné se vracet do předchozích fází vývoje projektu za účelem zpřesnění zadání. Nejznámější varianta tohoto přístupu používaná

v OOP je prototypový model a spirální model. Právě skutečnost, že lze zahájit tvorbu dříve, než jsou známy všechny požadavky na systém, a použít zkušenost z prvotní implementace pro upřesnění zadání, měla za následek velkou oblibu tohoto stylu v komunitě OOP.

Výhodou tohoto stylu je tedy možnost průběžně doplňovat a zpřesňovat zadání. Nevýhodou je horší možnost řízení projektu a sledování postupu řešení.

### 4.5.1.3

## Evoluční model životního cyklu

Evoluční model životního cyklu je založen na myšlence provádění projektu postupně po menších částech, přičemž celý systém vzniká postupně tak, jak se i během vývoje mění požadavky na něj. Tento model získal v komunitě OOP největší oblibu. Nejznámější varianta tohoto přístupu je inkrementální programování, component based development a agilní metodiky, i když samotní autoři agilních metodik vesměs prohlašují, že se jedná o něco úplně jiného a nového.

Výhody tohoto přístupu jsou v možnosti postupného vývoje a údržby a krátké době mezi zadáním a řešením dílčího požadavku. Velkou nevýhodou tohoto přístupu je velmi obtížné řízení a sledování projektu, a to především, jde-li o projekty většího rozsahu.

## 4.5.2

## Rigorózní versus agilní metodiky

### 4.5.2.1

## Rigorózní metodiky

Rigorózní metodika je pejorativní označení přístupu založeného na sekvenčním modelu od autorů agilních metodik. Někteří autoři ale pod rigorózní metodiky zahrnují jakoukoliv metodiku, ve které se objevují různé fáze během vývoje systému a ve které se pracuje s formální dokumentací (diagramy, tabulky, seznamy, ...), i když je tato metodika iterativní nebo evoluční.

Rigorózní metodiky jsou údajně příliš složité, málo účinné, vyžadují množství dokumentace a tím vším komplikují a oddalují výslednou implementaci. Rigorózní metodiky jsou také údajně neúčinné v podmínkách rychle se měnících požadavků a vyvíjejících se technologií.

### 4.5.2.2

## Agilní metodiky

Agilní metodiky jsou samotnými autory označovány jako metodiky, které umožňují vytvořit řešení velmi rychle a optimálně a toto řešení dovolují pružně přizpůsobovat. Jedná se o několik metodik, z nich nejpoblárnější je takzvané extrémní programování (XP). Agilní přístup (AP) byl s velkým nadšením přijat v komunitě programátorů pracujících s OOP.

Představitelé těchto přístupů z celého světa v roce 2001 vydali společný dokument Agile Manifesto a vytvořili alianci pro „agilní vývoj softwaru“. (Beck 2003, Beck 2002)

Tento manifest deklaruje následující priority – citace z (AP 2001):

*Odhalili jsme lepší způsob vývoje softwaru, sami jej používáme a chceme pomoci i ostatním, aby jej používali. Dáváme přednost:*

- a) individualitám a komunikaci před procesy a nástroji,
- b) provozuschopnému softwaru před obsažnou dokumentací,
- c) spolupráci se zákazníkem před sjednáváním kontraktu a
- d) reakci na změnu před plněním plánu.

Přestože doposud uvedená informace o agilním přístupu vypadá krajně podezřele, tak agilní přístup má v praxi dobré výsledky, což můžeme potvrdit i na základě vlastních zkušeností. Autoři agilních metodik jsou většinou pro věc nadšení, ale také velmi vzdělaní lidé. Někteří z nich odborně publikují i v oblasti formálních technik a jiných metod řízení projektů, což dokladují například i odkazy na práce Amblera a Becka v této knize.

Agilní přístup má velké přednosti v malých týmech, které mohou pravidelně komunikovat se zákazníkem/zadavatelem. Nezbytnou nutností pro úspěch je ale mít zajištěné:

- a) technickou podporu týmu (vývojové prostředí, které umožňuje práci s komponentami, inkrementální kompilaci, refaktoring, evidence a sdílení kódu, metriky a v neposlední řadě testování) a
- b) koordinovaný a průběžný kontakt se zadavatelem/zákazníkem.

Agilní přístupy nelze použít u velkých projektů, kde se naráží nejvíce na nedostatek průběžného kontaktu se zadavatelem a na potřebu projekt plánovat.

### 4.5.2.3

#### Příčina sporu

Příčina vzniku agilních metodik je samotnými autory popisována jako selhání rigorózních metodik. Příčinu jejich selhání vysvětlují v nepotřebné dokumentaci, zbytečném kreslení diagramů a vůbec v provádění aktivit, které oddalují „opravdové“ programování. Jako lék nabízejí tyto zdržující aktivity nedělat vůbec a soustředit se jen na vlastní tvorbu softwaru.

S tímto tvrzením však nelze souhlasit. Pokud se podíváme na jiné oblasti inženýrských aktivit (stavebnictví, strojírenství...), uvidíme, že se všude plánuje, analyzuje, dokumentuje, prototypuje, testuje atd. Proč by tedy měla být oblast tvorby softwaru výjimečná?

Základní myšlenka potřebnosti „rigorózních“ metodik nemůže být špatná. I v softwarovém inženýrství je potřeba projekty plánovat, řídit, analyzovat zadání atp. Na druhou stranu je pravda, že v mnoha případech „rigorózní“ přístup nevede k lepšímu, rychlejšímu a levnějšímu výsledku než při nasazení AP.

Rigorózní metodiky pro vývoj s pomocí OOP mají za sebou dlouhý vývoj. Většinou vznikly postupnou změnou z klasických strukturovaných metodik. Dnes používané

metodiky mají svůj původ ve výzkumu a zkušenostech s tvorbou softwaru pomocí OOP v 90. letech. Zde jsou příčiny, proč nesplňují očekávání a selhávají. Zároveň se zde skrývá vysvětlení, proč došlo ke vzniku AP. Příčiny jsou dvě:

- a) První spočívá ve změně stylu programování. Rigorózní metodiky selhávají proto, že se nedostatečně vyrovnaly se změnou nástrojů a technik pro programování a implementaci. Jednoduše řečeno; rigorózní metodiky stále dostatečně nezaznamenaly výraznou kvalitativní změnu ve způsobu tvorby programů a stále předpokládají, že ve fázi implementace je jen potřeba z celkového modelu úlohy vyprodukovat zdrojový kód, který se dodá kompilátoru k překladu. Vývojáři proto oprávněně hledí s nedůvěrou na rigorózní metodiky, když na jedné straně vyžadují tvorbu perfektního konceptuálního modelu a na druhé straně nevyužívají možnosti, které mají dnešní vývojové prostředí. Naproti tomu AP nejen umožňuje, ale přímo vyžaduje plnou podporu vlastností nových vývojových prostředí a navíc bez potřeby modelování.
- b) Další problém spočívá ve schopnosti rigorózních metodik správně, účinně a rychle zachytit, analyzovat a zobrazit zadání od uživatelů. K modelování se dnes téměř výhradně používá UML. V této knize je diskutována jeho nedostatečná podpora úvodních fází modelování. AP je totiž také odpovědí na to, že UML nepodporuje dobře úvodní analýzu. Proto AP radí raději žádnou analýzu zadání nedělat a místo toho řešit projekt po částech a pravidelně spolupracovat s uživatelem. Bohužel vinou nedostatků rigorózních metodik je dnes AP prakticky úspěšné i v oblastech, kde by uživatelé byli schopni a dokonce by preferovali na začátku projektu formulovat požadavky – nebo jejich podstatnou část. Ale protože je v UML obtížné je srozumitelně modelovat, raději se nemodeluje vůbec a přistupuje se k AP, přičemž uživatelé musí průběžně komunikovat během vývoje. Toto je problém větších projektů, například specifických aplikací pro business, systémů pro řízení technologií (v telekomunikačním průmyslu nebo distribuce energií), atd. Ve všech těchto oblastech je třeba projekty plánovat a zadání je z podstatné části známé předem.

## **4.6**

### **Tvorba informačních systémů v kontextu podnikového managementu**

Při práci na velkých projektech se analytici informačních systémů setkávají s problémem, kdy funkčnost budovaných rozsáhlých systémů má vliv 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, nová potřeba legislativní podpory... Proto je žádoucí se při práci na informačních systémech zabývat i změnou těchto souvisejících struktur. V praxi se však bohužel tato otázka podceňuje a problém zavádění a fungování informačních systémů se řeší „od opačného konce“, tedy například se provádějí výběrová řízení na konkrétní technologie (např. čipové karty nebo jiná koncová zařízení), aniž by došlo ke správnému pochopení a nastavení business procesů, související legislativní podpory atp.

## 4.6.1

### Procesy a procesní modely – requirement engineering

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...). Těmito problémy se zabývá poměrně nedávno konstituovaný obor aplikované informatiky, který je anglicky označován „requirement engineering“. Více lze nalézt kupříkladu v (Kotonya et al. 1999, Hammer et al. 1994). 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.

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

## 4.6.2

### 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, není nemožná myšlenka modelovat podnikový a informační systém ne jako modely dva, ale jako jeden model a změny a vlastnosti procesů přímo promítat do změn a vlastností softwaru a naopak. Tento přístup je podrobně popsán například v (Taylor 1995).

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:

- a) Zjednodušuje celý proces analýzy a návrhu a snižuje celkovou spotřebu práce, neboť se buduje jen jeden model namísto dvou.
- b) Řeší strukturální nesoulad mezi business procesy a jejich podpůrnými komponentami informačního systému.

- c) Souvislosti s řízením a organizací práce a strukturou informačního systému jsou srozumitelné, informační systém je lépe realizovatelný.
- d) Uspodňuje problémy a náklady spojené s návrhem provádění 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, 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 povinnosti s potřebnou mírou autority a pravomocí.

### 4.6.3

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

Podívejme se 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 neextrémnější podobě se ještě od některých prodejců softwaru můžeme dovědět, že informační systém je softwarový produkt, který právě nabízejí). Taková zjednodušení a nepochopení potom velmi často vedou k velkým problémům které lze popsat následujícím scénářem:

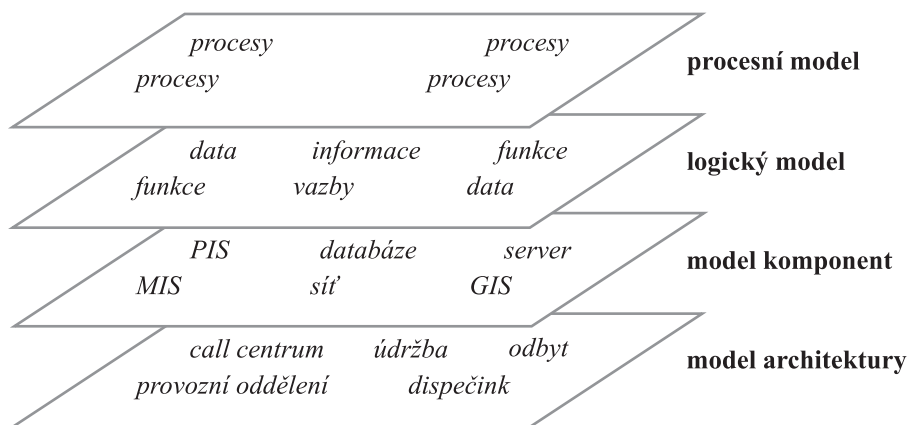
1. Očekává se, že problémy řídicí a organizační povahy, které organizace má, musejí být řešeny vybudováním či nákupem „nového informačního systému“ (nebo jeho nové komponenty). = Příčina problémů v řídicím systému se nesprávně chápe jen jako nedostatečné vybavení informačními technologiemi.
2. Očekává 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. (Nebyla nastavena nová legislativa, organizační podpora...) Tato skutečnost se samozřejmě tají a navenek se předvádí, jak je nový software moderní a funkční, čili 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:

<b>informační systém =</b>	<b>informační technologie</b> (programy, počítače, sítě...) + <b>zabezpečení</b> (správci systémů, údržba systémů, zajištění bezpečnosti informačních systémů, zajištění kvality...) + <b>uživatelé</b> (jaké služby jim systém poskytuje a také co od nich vyžaduje)
<b>řídící systém =</b>	<b>organizační struktura + pravidla, řídicí funkce, kompetence, legislativní podpora... + informační systém.</b>

Tab. 13: Informační a řídicí systém

Informační systém je tedy jen jednou součástí řídicího systému organizace. Analýza a návrh řídicího systému organizace je složitou záležitostí. V souladu s konvergenčním přístupem se doporučuje na jeho model nahlížet čtyřmi různými způsoby:



Obr. 51: Čtyři vrstvy modelu řídicích systémů podniku nebo organizace

- 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.
- 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.
- Dalším způsobem je sestavení modelu komponent. Jedná se sestavení modelu, jehož prvky jsou například konkrétní moduly informačních nebo jiných subsystémů systému jako například personální informační systém, evidence zásob, GIS, DWH, podnikový intranet apod. Vazbami v tomto modelu jsou vzájemné souvislosti a závislosti vyjmenovaných prvků na sobě.
- 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.

Dochází-li ke změně na jakékoliv úrovni, tak je vhodné provést rozbor dopadů této změny na okolní úrovně postupně až k procesům. Například rozhodnutí vedení organizace „postavíme si nové výpočetní středisko“ nebo „v budově XY uděláme oddělení Z pro styk se zákazníky“, 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. Podobně nebezpečí na úrovni komponent v sobě skrývá například záměr „začneme používat GIS“, zavedeme „webový portál na internetu“ atd. Jestliže se totiž rozhodnutí o informačních systémech provádějí pouze na jedné úrovni (například technologií) bez vazby na další související prvky jiných úrovní (například business procesy nebo organizační struktura), tak reálně hrozí, že jedině, co změna organizaci přinese, jsou zbytečně vyhozené peníze a pro mnoho lidí přidělení práce navíc.

#### **4.6.4**

### **Vztah k OOP**

Myšlenka konvergenčního modelování má původ v OOP. Změna v procesech má vliv na změnu v logické architektuře, ta zase na komponenty a ty nakonec mohou mít vliv na softwarovou architekturu a naopak. Tuto závislost je třeba při tvorbě informačního systému respektovat a při analýze se jí zabývat. OOP má všechny předpoklady k tvorbě takových analýz. Bohužel metodiky a nástroje využívající UML jsou v této oblasti stále na samotném počátku a vesměs předpokládají, že požadavky na informační systém není třeba evaluovat a analyzovat.

Situace s AP je ještě méně radostná. Filosofie AP jde zcela proti zde popsaným zásadám. Žádná analýza, žádné rozborů a reorganizace business procesů, žádné organizační změny. Jen co nejrychlejší implementace podle okamžitých potřeb uživatele.

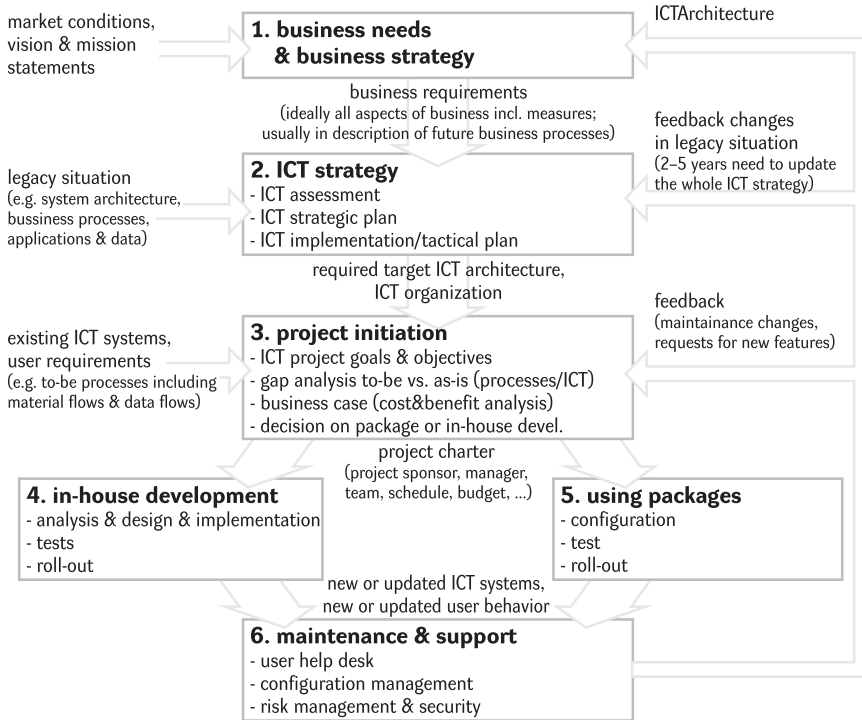
## Jak správně využít objektový přístup v projektech informačních systémů

Ve většině oblastí již OOP dostatečně prokázalo své přednosti a určitě bychom dokázali pojmenovat nemálo oblastí, kde dnes již bez využití OOP by nebylo možné aplikace v rozumném čase a s rozumnými náklady ani sestavit. To ale také znamená, že u velkých projektů se dnes nemůžeme spolehnout jen na slepou víru nadšenců (většinou čerstvých absolventů vysokých škol) ve všemocnost nových verzí objektových programovacích jazyků ani na samospasitelnost jednotného modelovacího jazyka UML a CASE nástrojů. CASE, UML a dobrý programovací jazyk je samozřejmě při tvorbě softwaru nutností (autor sám má velmi dobré zkušenosti se Smalltalkem a podílí se na vývoji modelovacího nástroje Craft.CASE), ale na problematiku „stavění“ softwaru není možné nahlížet jen pouze z perspektivy „zedníků“ byj jakkoli kvalifikovaných. Taková zjednodušení v praxi vedou k velkým rozčarováním. Někdy se dokonce stává, že krach přehnaných očekávání a následný neúspěch projektu vede u některých týmů k odsouzení OOP jako pro praxi nevhodného přístupu. Řešení zde naznačeného problému spočívá v umění podívat se na tvorbu softwaru také očima „architekta“ či „stavbyvedoucího“. U velkých projektů se není možné spoléhat jen na nesporné výhody objektového modelování a programování. Velké projekty je potřeba kvalifikovaně plánovat a řídit tak, aby se přednosti OOP doopravdy projevíly, a ne aby se staly ohrožením projektu.

### 5.1

#### Celopodnikový pohled

V dnešních organizacích, ať se zabývají nejrůznějším předmětem své činnosti, tvoří informační a komunikační systémy integrální součást samotných firem. Zároveň zde vlivem technologického rozvoje dochází k postupné konvergenci informačních a komunikačních technologií. Proto se oblast řízení těchto technologií označuje „ICT Management“. Jedná se o soubor činností, které zároveň realizují a zároveň mají vliv na vlastní podnikovou strategii. Jeden z možných přístupů je uveden na obrázku, který je popsán v knize (Hall et al. 2004).



Obr. 52: ICT management process podle (Hall et al. 2004)

Na tomto schématu jsou následující bloky činností:

1. Formulace podnikové strategie firmy – nejen na základě potřeb trhu, ale i na základě možností firemních informačních technologií.
2. Formulace informační strategie firmy – jako rozpracování formulované podnikové strategie na detail ICT.
3. Posuzování projektů – zde se rozhoduje o „spouštění“ jednotlivých ICT projektů jako postupná realizace cílů formulovaných v informační strategii s ohledem na potřeby uživatelů a znalosti z údržby stávajících systémů.
4. Tvorba projektů vlastním vývojem – jako jeden způsob realizace projektu.
5. Použití hotového softwaru – jako druhý způsob realizace projektu.
6. Údržba a podpora projektem vytvořeného či koupeného systému.

## 5.2

### Model životního cyklu projektu informačního systému

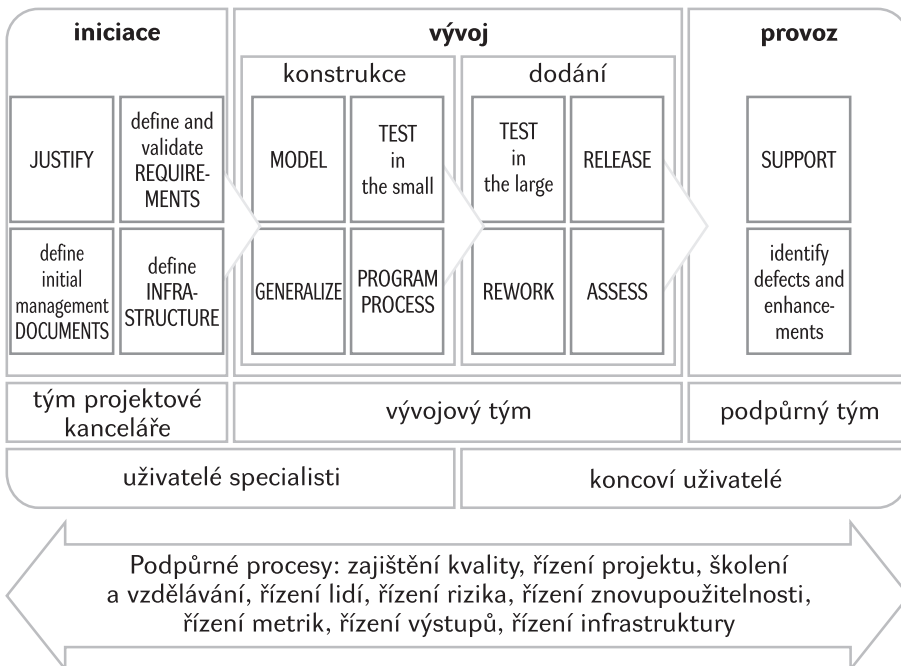
Přípravu i realizaci projektu informačního systému je nutno vidět jako proces. (Ewusi 2003) Následující návrh řešení je aplikací původního Amblerova přístupu popsáno v (Ambler 1997, 1998). Tento přístup, jak již bylo řečeno, nahlíží na tvorbu softwaru očima manažera z perspektivy projektového řízení. To znamená, že zde popisujeme něco

trochu jiného, než klasické metody analýzy a návrhu, které jsou v tomto chápání metodami „pro dělníky“. (Při určité míře zjednodušení to znamená, že například celá metoda „extrémního programování“ nebo OMT může být považována za konkrétní náplň jediné fáze konstrukce zde popisovaného přístupu).



Obr. 53: Čtyři hlavní fáze životního cyklu dle Amblera

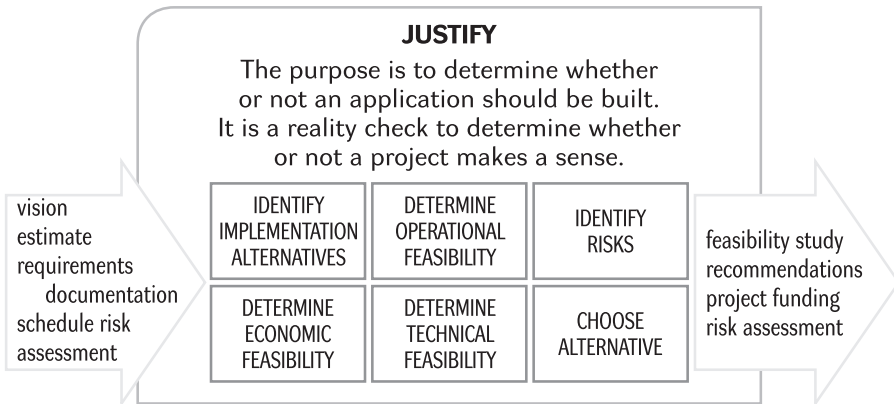
Jak je uvedeno na obrázku, celý proces je členěn na čtyři fáze pojmenované „iniciace“, „konstrukce“, „dobání“ a „provoz“. Každá z fází se skládá ze dvou až čtyř dílčích procesů.



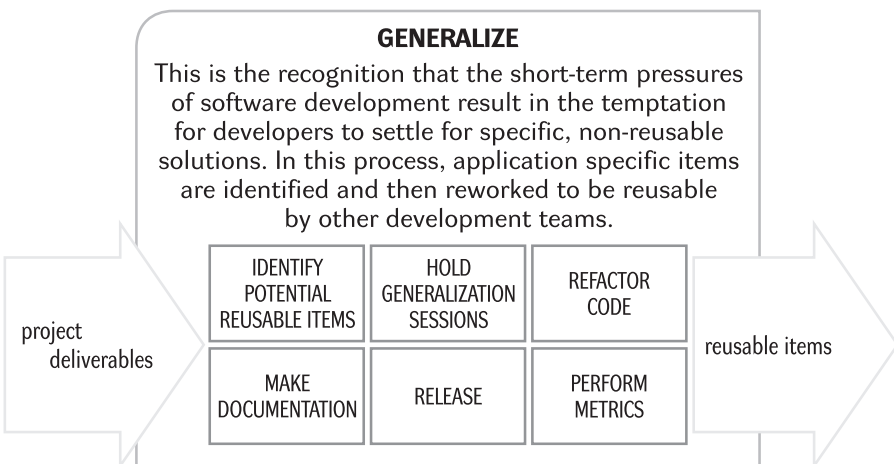
Obr. 54: Čtyři hlavní fáze životního cyklu v detailu dle Amblera

Čtyři hlavní fáze by se měly provádět sekvenčně po sobě. Protože na konci každé fáze dochází ke změnám týmů a platform, tak je doporučováno, aby manažer projektu neopomněl svolat pracovní setkání, kde prezentuje dosavadní postup projektu, provede kontrolu dokumentace a dalších dosud vytvořených výstupů a seznámí pracovní skupinu s novými členy týmu. Dílčí procesy uvnitř fází je možné provádět opakovaně – iterativně nebo evolučně. Tento model v sobě kombinuje výhody všech hlavních přístupů: Umožňuje plánování a projektování ve velkém (= celý projekt) a zároveň se nebrání experimentování v malém (= uvnitř fází).

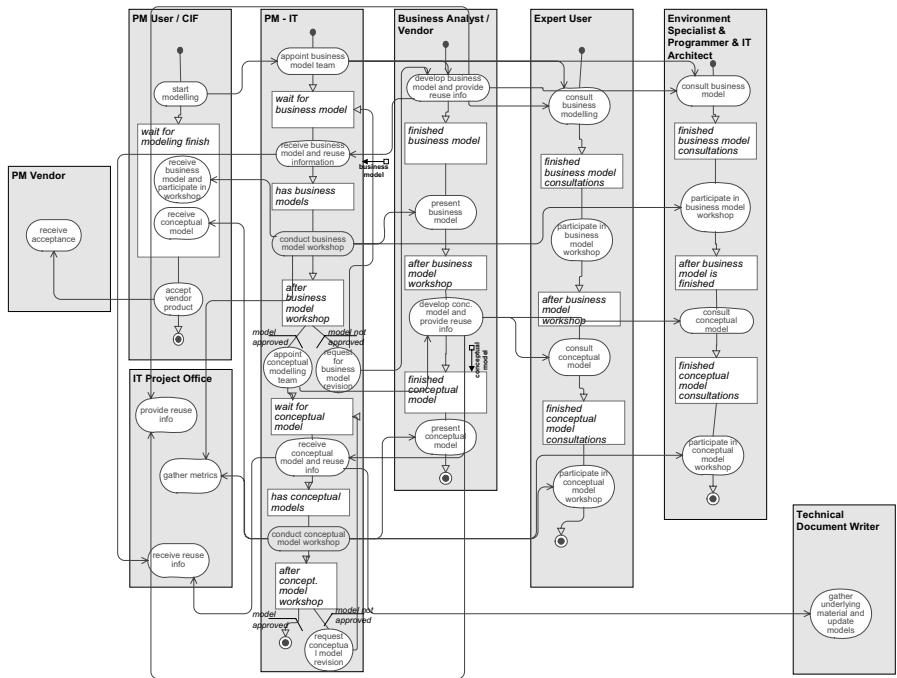
Každý z dílčích 14 procesů je možné v konkrétních podmínkách modelovat ve formě procesní mapy, kde lze vyznačit podrobný výčet potřebných aktivit a vymezení rolí účastníků procesu. Na obrázcích je příklad podrobnějšího obsahu dílčích procesů a příklad procesní mapy z konkrétního projektu firmy Deloitte&Touche pro velkou mezinárodní telekomunikační společnost.



Obr. 55: Obsah dílčího procesu INITIATE-JUSTIFY podle Amblera



Obr. 56: Obsah dílčího procesu CONSTRUCT-GENERALIZE podle Amblera



Obr. 57: Příklad nastavení procesu konstrukce softwaru v konkrétním projektu z praxe velké komunikační firmy

## 5.2.1 Iniciace

Proces iniciace (initiation) slouží k zajištění všech přípravných prací, nezbytných k zahájení tvorby software. Jeho cílem je připravit vše potřebné k zahájení projektu. Skládá se z procesů:

- a) Definice požadavků (define requirements). Prvotní sběr požadavků na nový systém, ke kterému dochází na zahajovací schůzce účastníků projektu.
- b) Příprava manažerské dokumentace (prepare management documents). Příprava plánu čerpání zdrojů a časového plánu celého projektu. Ten zahrnuje požadavky na kapacity, technické a finanční zdroje včetně sestavení časového harmonogramu řešení.
- c) Příprava infrastruktury (prepare infrastructure). Je to příprava podkladů, informačních zdrojů, případně i instalace dílčích systémů, které budou potřeba pro tvorbu systému.
- d) Proces zmocnění či narovnání (justify). Dílčí proces, během kterého je třeba vypracovat rizika celého projektu, právního zabezpečení a počítačové bezpečnosti. Zároveň by zde mělo dojít k nalezení alternativ řešení a výběru optimální varianty. Během této fáze se také může na základě zjištěných okolností rozhodnout systém neimplementovat.

## 5.2.2 Konstrukce

Proces konstrukce (construct) slouží k vytvoření požadovaného systému. Vlastní tvorba začíná teprve v této fázi. Vzhledem ke specifickým vlastnostem objektového software se předpokládá iterativní opakování těchto fází, protože zde může až v rámci prvotní implementace docházet ke zpřesnění a správnému pochopení zadání. Počet iterací se pohybuje okolo 2 až 3:

- a) Modelování (model). Dílčí proces, ve kterém dochází k podrobné definici zadání a úplnému získání podkladových materiálů. Během tohoto procesu by měly být modelovány a vyzkoušeny (simulovány) cílové uživatelské procesy. Vše by mělo být dokumentováno.
- b) Sestavování (program). Proces, kdy dojde k vytvoření (naprogramování) vlastního software. Tento podproces se odehrává na vývojové platformě. Kromě samotného software by měla být pro potřeby budoucích úprav a oprav vytvořena systémová dokumentace. Zároveň by měla být zahájena práce na uživatelské dokumentaci.
- c) Testy v malém (tests in small). Proces, který slouží k prověření funkčnosti. Tento podproces se odehrává na testovací (nikoli provozní) platformě a účastní se ho speciálně určení členové vývojového týmu, tedy nikoliv skuteční uživatelé a na vývoji nezúčastnění zadavatelé.
- d) Generalizace (generalize). Proces, kde dojde k optimalizaci vytvořeného software, jeho podrobnému dokumentování a identifikaci potenciálně znovupoužitelných a nahraditelných artefaktů. Tato „úklidová“ fáze si neklade za cíl vylepšení funkčnosti vytvářeného systému, ale jeho rozčlenění po technické stránce. Bez tohoto pořádku, který je nutné do systému vnést, jsou pozdější úpravy, opravy, rozšiřování nebo opakované využití při tvorbě dalších programů komplikované.

## 5.2.3 Dodání

Proces dodání (deliver) slouží k uvedení nového systému do provozu na provozní platformě. Tvoří jej dílčí procesy, které mohou běžet i souběžně.

- a) Vydání (release) je proces, který zajišťuje přechod na provozní platformu, řešení problémů s instalací na této platformě a přípravu provozní infrastruktury. Jeho součástí jsou i podpůrné a pomocné činnosti, jako například zveřejnění systému koncovým uživatelům (propagace) a organizace nezbytných zaškolení, vydání příruček atp.
- b) Testy ve velkém (tests in large) jsou testy, prováděné zástupci cílových uživatelů.
- c) Přepřerobání (rework) je opravný proces, který reaguje na výsledky testů. Je podobný konstrukci v malém, ale probíhá na provozní platformě. Nejde tu jenom o software, může obsahovat zásahy nebo změny do dokumentace atd.

d) Zhodnocení projektu (assessment) je závěrečný proces pracovního týmu a týmu projektové kanceláře, který slouží k jejich vnitřním potřebám. Je okamžik, kdy je možno poučit se z chyb, ocenit iniciativu účastníků, zhodnotit rizika, kvalitu, vyhodnotit metriky apod. Výsledky, alespoň rámcově, by měly být zveřejněny a vždy archivovány.

## 5.2.4

### Provoz

Proces provoz pokrývá časový úsek používání systému v praxi. Jeho součástí jsou dva dílčí procesy:

- a) Podpora (support), která reprezentuje množinu aktivit, prováděných v týmu „help desku“. Jedná se o různé rady uživatelům, týkající se ovládání a instalace systému, péče o průběžné doškolení a zaškolení (např. nově přichozících uživatelů, kteří nebyli zaškoleni). Dále sem patří řešení problémů, spojených s poruchami hardware, případně sběr podnětů, vedoucích ke zlepšení chodu systému.
- b) Údržba (maintenance), které představuje operativní odstraňování chyb v systému. Zde je důležité rozlišit alespoň podněty ke zlepšení systému, který je plně funkční, a hlášení chyb, které nedovolují pokračovat v chodu systému nebo chyb, které ovlivňují chod systému nesprávným způsobem.

Manažer projektu by se měl také postarat v průběhu projektu o řádné zajištění kvality. Jako jednoduchý a účinný prostředek jsou Amblemem doporučovány kontrolní seznamy (check-lists), kterými lze ověřovat, zda můžeme příslušnou fázi zahájit, zda vykonáváme co je právě potřeba a nebo zda můžeme fázi prohlásit za ukončenou.

- Matice požadavků je aktualizována.
- Projektový plán je aktualizován.
- Modely, zdrojové kódy a dokumentace byla přiřazena k příslušné verzi.
- Plán testů je připraven.
- Uživatelská, provozní a instalační dokumentace je hotova a může být testována.
- Aplikace je připravena k instalaci na testovací platformu.

Tab. 14: Kontrolní seznam k ukončení procesu konstrukce

## 5.2.5

### Jednotlivé týmy v procesech

Stejně jako fáze životního cyklu nelze zužovat pouze na schéma analýza / návrh / implementace, tak i problematiku týmů nelze omezovat pouze na zadavatele, uživatele, řešitelský tým a jeho manažera. Pro správný chod výše popisovaných procesů potřebujeme následující:

## Tým projektové kanceláře

Je to stálý tým pro všechny projekty, které se v softwarové firmě či oddělení provádějí. Jeho úkoly jsou tyto:

1. Organizuje a zahajuje tvorbu nového systému počínaje výběrem a jmenováním pracovního týmu.
2. Sbírá, archivuje a poskytuje nezbytné informace a dokumentaci vývojovým pracovníkům. Pro tento archív je vhodné používat softwarový nástroj. Tato „knihovna znovupoužitelných znalostí“ je anglicky označována jako „group memory“. Jde o velmi důležitý nástroj pro vývoj pomocí OOP.
3. Organizuje testování systému ve fázi dodání.
4. Zajišťuje školení.
5. Hodnotí projekty.
6. Organizuje ukončení činnosti pracovního týmu a předání jeho výsledků do provozu.
7. Řídí chod „help desku“.

## Tým „help desku“

Je to stálý tým s těmito úkoly:

1. Je kontaktním místem pro hlášení chyb, organizuje a zajišťuje jejich nápravu, provádí vyrozumění o způsobu nápravy chyby podle zásady: kdo chybu ohlásil, ten je vždy informován o způsobu řešení.
2. Sbírá podněty ke zlepšení od uživatelů.

## Zahajovací tým

Je to ad-hoc tým, sestavený projektovou kanceláří pro nastartování jednotlivého nového projektu. Jeho úkoly jsou:

1. Spolupracuje s týmem projektové kanceláře při jmenování pracovního týmu.
2. Přípravuje manažerské podklady pro projekt.
3. Vyhodnocuje rizika, aspekty bezpečnosti, práva, varianty řešení, ... (viz proces *justify*).
4. Sestavuje a zodpovídá za plán projektu, obsahujícího časové odhady, požadavky na zdroje a kapacity.
5. Koordinuje spolupráci se zástupci uživatelů.

## Pracovní tým

Je to ad-hoc tým, sestavený projektovou kanceláří pro vypracování jednotlivého projektu. Mezi hlavní odpovědnosti a pravomoci patří:

1. Zodpovídá za úspěšné vytvoření systému za dodržení vstupních omezujících podmínek
2. Produkuje všechny dohodnuté výstupy (software, dokumentace, příručky, rozčlenění na opakovatelně použitelné prvky, sběr hodnot příslušných metrik aj.).

3. Zodpovídá za projekt a jeho výsledek v osobě manažera projektu. Manažerem projektu by neměl být technik-programátor a při implementaci většího cizího systému ani šéf IT zadavatele.
4. Zajišťuje spolupráci manažera projektu s projektanty pro oblast řešení technických složek projektu.

V souvislosti s týmy je třeba se také zmínit o problému optimální alokace lidských zdrojů. Pro čtyři fáze zde uvedené metodiky se doporučují optimální poměry alokace pracovní síly 1:3:2:1. Klasickou chybou nezkušených projektových manažerů je plýtvání pracovní silou při zahájení projektu, která potom v průběhu projektu chybí. Manažeré v dobrém úmyslu ohromit zadavatele a tak co nejlépe prezentovat svoji firmu sestaví obrovský tým, který se představí při zahajovacím mítinku, ale jen někteří z něj dále pokračují v řešení. Autor dokonce zná případy z praxe některých českých firem, kdy původní mnohočlenný tým plný expertů a známých osobností včetně charismatického šéfa firmy do fáze konstrukce degradoval týden před závěrečným termínem pouze na jednoho až tři přetížené studenty-programátory.

Vzhledem k vlastnostem OOP se posiluje potřeba analytických profesí včetně business specialistů, datových architektů a dalších na úkor programátorských profesí až k poměru přesahující poměr 1:1.

## 5.2.6

### Provozní, testovací a vývojová platforma

Pro využití výhodných vlastností OOP ve tvorbě softwaru ve velkém je nezbytné nastavit odpovídající řídicí a podpůrné procesy a organizační řád. Bez těchto nezbytností nelze očekávat, že přechod na OOP sám o sobě povede ke zlepšením. Slepá víra ve výhody objektového přístupu může dokonce vést i ke zhoršení kvality tvorby velkých aplikací. Proto je potřeba podporovat pro OOP charakteristické vlastnosti jako znovupoužitelnost, komponentový přístup a nové paradigma programování. V předložené metodice k tomu slouží především proces justify z fáze iniciace, proces generalizace z fáze konstrukce a specifické požadavky na provoz projektové kanceláře.

Tvorba softwaru ve velkém také vede k požadavku zavést specifické platformy. Zde není možné se spoléhat na nějaké zázračné vlastnosti moderních informačních technologií. Jde o následující platformy:

1. Vývojová platforma, která představuje počítače vývojových pracovníků. Jsou to počítače a software, oddělené od provozních systémů tak, aby jejich pracovní cyklus nebo i případné problémy či nezbytné rekonfigurace nenarušily průběh rutinního provozu. Nejlepším řešením je umístit vývojové práce na jiný systém a jiný segment sítě, než je provozní platforma.
2. Testovací platforma, která slouží k prvotnímu testování. Ze stejných důvodů jako vývojová by i tato platforma měla být také oddělena od provozní platformy.
3. Provozní platforma je „ostré“ prostředí pro uživatele systému, kde je žádoucí vysoký stupeň robustnosti, spolehlivosti (zálohování, testování správnosti chodu, ...) a bezpečnosti.

## 5.3

### Postupná transformace datového modelu při projektování

V této kapitole bude popsán soubor objektivě orientovaných technik a metod, které je vhodné použít během projektování informačního systému. Postupy zde uvedené tvoří základ metody BORM, která je původní českou metodikou. BORM byl publikován především v (Merunka et al. 2000, Carda et al. 2003) a v zahraničí také v (Knott et al. 2003, Liu et al. 2005, Knott et al. 2000).

#### 5.3.1

#### Metoda BORM

Metoda BORM (Business and Object Relation Modeling) je vyvíjena postupně od roku 1993. Od počátku byla orientována na podporu tvorby objektivě orientovaných softwarových systémů založených na čistých objektivě orientovaných programovacích jazycích a vývojových prostředích, jakými jsou například prostředí Smalltalku a nerelační objektové databáze. BORM je možné využít nejen ve tvorbě softwaru, ale i k analýze požadavků na projektovaný systém a na modelování business procesů.

Práce na BORMu byla od svého počátku součástí grantu VAPPIENS (research project Various Programming Paradigms in Integrated Environments), který je součástí programu „Know How Fund of Czech Academic Link Programme“ Britské rady (British Council). Od roku 1996 je další vývoj podporován firmou Deloitte&Touche Czech Republic and Central Europe, kde je tato metoda také používána. BORM lze charakterizovat pomocí následujících tří vlastností:

- 1) BORM je navržen jako metoda, která pokrývá všechny fáze vývoje softwaru. Velká pozornost je v BORMu věnována úvodním fázím projektu a postupům, jak najít objekty v zadaném problému a zkontrolovat jejich správnost. Techniky z těchto fází BORMu lze používat samostatně pro modelování procesů i takových systémů, které nemají přímý vztah k tvorbě softwaru.
- 2) BORM pro každou jednotlivou fázi životního cyklu využívá v diagramech jen omezenou sadu pojmů. Předpokládá se totiž, že během projektování dochází k postupným přeměnám pojmů na jiné. Například ve fázi analýzy se nepoužívají pojmy jako agregace, jednoduchá či vícenásobná dědičnost, protože tyto pojmy jsou relevantní až pro implementaci. Naopak pojmy jako stav, přechod či asociace jsou používány během analýzy, ale ve fázi implementace, kdy se snažíme model přizpůsobit cílovému implementačnímu prostředí, se s nimi již nepracuje. Nejde jen o postupné zvyšování úrovně detailu ve vytvářeném modelu, ale skutečně o řadu transformací modelu v průběhu životního cyklu.
- 3) V BORMu je každý pojem reprezentován shodnými symboly bez ohledu na to, jestli se jedná např. o diagramy datové struktury nebo komunikací mezi objekty. BORM používá pro znázorňování konceptuálních a softwarových pojmů většinu symbolů shodně s jazykem UML, ale dovoluje v jednom diagramu znázornit například posílání zpráv mezi metodami různých objektů v různých stavech. Tento přístup dovoluje vyjádřit konzistentním způsobem některé žádoucí detaily softwarové konstrukce,

kteří lze výhodně aplikovat především při návrhu pro čistě objektově orientované programovací jazyky. Tento originální způsob nahrazuje tvorbu více od sebe oddělených třídních, stavových a kolaboračních diagramů a také dovoluje zobrazit větší množství spolu souvisejících informací. Samostatné stavové či interační diagramy jsou však samozřejmě v BORMu také používány.

BORM rozlišuje 6 fází životního cyklu vývoje systému:

- 1) Strategická analýza. Zde dochází k vymezení samotného problému, je stanoveno jeho rozhraní, jsou rozpoznány základní procesy, které se v systému a také v jeho okolí mají odehrávat.
- 2) Úvodní analýza. Zde dochází k rozpracování samotného problému, jsou mapovány požadované procesy v systému a vlastnosti základních objektů, které se na diskutovaných procesech podílejí.
- 3) Podrobná analýza. Je rozpracování analýzy do detailů jednotlivých typů objektů (sady objektů, třídy objektů) a objektových vazeb (skládání, dědění, závislosti, ...).
- 4) Úvodní návrh (design). Je to první fáze, ve které se začínáme snažit systém upravit tak, aby byl schopen softwarové implementace. Proto se zde již nehovoří o analýze, neboť z pohledu zadání by mělo již vše být hotovo a rozpoznáno. Úvodní návrh používá shodné nebo velmi podobné nástroje jako předchozí fáze, ale liší se způsobem práce s nimi.
- 5) Podrobný návrh (design). V této fázi dochází k přeměně prvků již existujícího modelu do takové podoby, která je podřízena cílovému implementačním prostředím. V této fázi se zohledňují vlastnosti konkrétních programovacích jazyků, databází apod.
- 6) Implementace (tvorba, sestavování programu). V této fázi se vytváří (programuje, sestavuje či generuje z CASE nástroje) požadovaný software.

### 5.3.1.1

#### Použití BORMu v praxi

BORM je určen k podpoře celého životního cyklu tvorby informačních systémů. Jiným neméně významným použitím je jeho možnost využití nikoliv za účelem pozdější implementace nějakého informačního systému, ale přímo pro účely organizačního poradenství. Objektové modely BORMu slouží k nalezení slabin ve stávající organizaci a procesech a k návrhu změn, které by tyto nedostatky odstranily.

### 5.3.2

#### Vývoj pojmu objekt během projektování

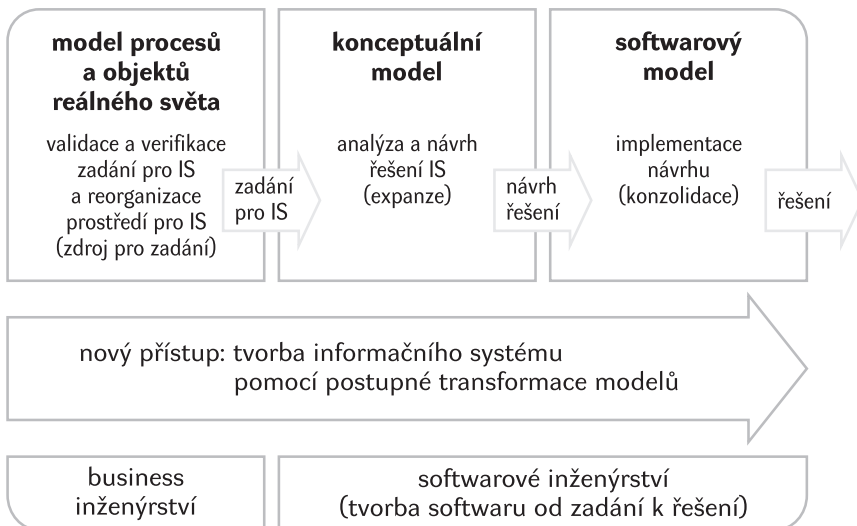
Samotný pojem objektu včetně jeho vlastností se v jednotlivých fázích projektu mění. Jinak chápe objekt programátor při implementaci v nějakém konkrétním programovacím jazyce a jinak chápe objekt zadavatel, protože pro něj je objekt zobrazením nějaké entity reálného světa, která je v okruhu jeho zájmu při formulaci zadání.

U každé z obou zainteresovaných skupin při vývoji IS lze rozlišit tři různé úrovně chápání zadání a realizace softwarové aplikace.

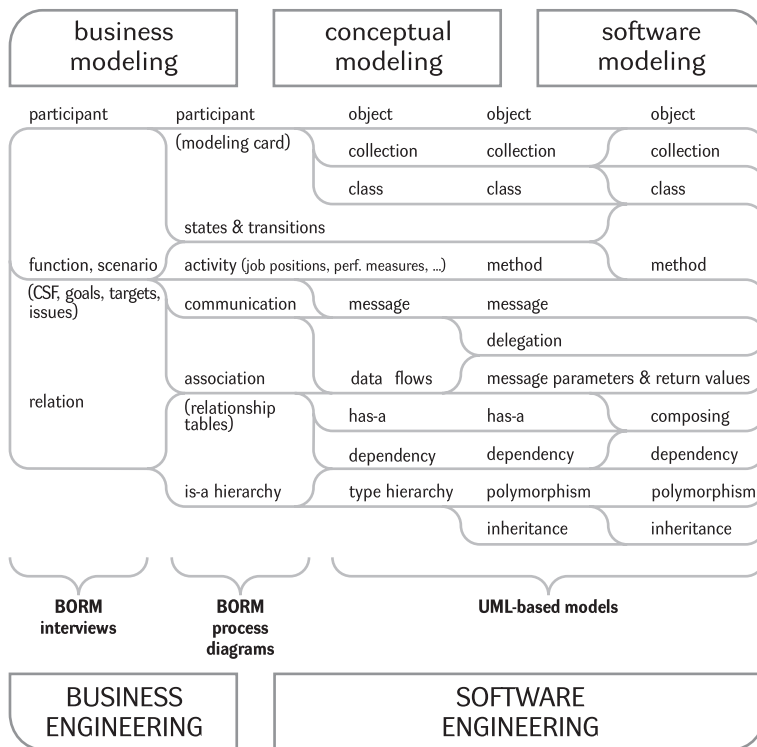
- a) Aplikační úroveň představuje okruh znalostí a dovedností, se kterými přichází jedna nebo druhá skupina do běžného pracovního styku. Pojmy z aplikační oblasti jsou proto pro příslušníka skupiny známé a srozumitelné. Bohužel jsou však nejvíce vzdálené pojmům aplikační úrovně skupiny druhé a informační systémy proto nelze stavět pouze na této úrovni.
- b) Základní úroveň představuje komunikační optimum mezi velmi od sebe vzdálenými aplikačními úrovněmi. Největší roli zde hraje konceptuální modelování, které dovoluje dostatečně srozumitelné a formální diagramové nástroje, které jsou na jedné straně ještě sledovatelné „neprogramátory“ z první skupiny a zároveň na druhé straně poskytují dostatek informací pro příslušníky druhé skupiny.
- c) Pokročilá úroveň je tvořena množinou nástrojů, technik a znalostí, které v první skupině dovolují najít správný konceptuální model a druhé skupině tento model umožňují transformovat do softwarové podoby. Rozdíl mezi dobrým a špatným analytikem je právě v největší míře dát mírou znalostí z této úrovně.

Z výše uvedených informací vyplývá, že není možné pracovat ve všech etapách tvorby IS se stejným pojmem objektu. Lze očekávat, že jednotlivé atributy a vazby mezi objekty se budou v průběhu vývoje IS měnit, a že každý následující pojem bude mít zřejmě svého abstraktnějšího předchůdce, ze kterého byl odvozen. Tyto transformace jednotlivých pojmů mezi sebou jsou obsahem jednotlivých technik v různých fázích tvorby informačního systému. Každý pojem proto má

1. okruh nadřazených pojmů, ze kterých může být na základě nějakého postupu odvozen,
2. okruh podřízených pojmů, které z něj mohou být pomocí nějakého postupu odvozeny,
3. okruh platnosti, neboť v jiných fázích vývoje IS, než které mu přísluší, je místo pro jemu nadřazené nebo podřízené pojmy a
4. sadu technik či pravidel, pomocí kterých je transformován na z něj odvozené pojmy.



Obr. 58: Postupná transformace objektového modelu v BORMu



Obr. 59: Postupná transformace objektového modelu v BORMu – detail

V průběhu modelování nelze libovolně přidávat nebo měnit prvky modelu, protože každá změna musí být vždy konzistentní a zdůvodnitelná s odpovídajícím předchozím stavem modelu. BORM rozděluje objekty na tři hlavní skupiny:

1. Softwarové objekty, se kterými se pracuje v závěrečných fázích vývoje IS za účelem softwarové implementace. Tyto objekty obsahují pojmy přímo odpovídající konstrukcím z objektových programovacích jazyků a nebo standardu UML.
2. Konceptuální objekty, se kterými se pracuje v prostředních fázích vývoje IS. Tyto objekty obsahují základní pojmy objektově orientovaného paradigmatu, jako například polymorfismus objektů, zapouzdření, skládání, delegování, klasifikace objektů podle různých dimenzí, závislost objektů, třídy a množiny objektů atd. Je pravda, že mnohé z konceptuálních pojmů jsou shodné se softwarovými pojmy, ale značnou část z nich je třeba při přechodu na softwarové objekty transformovat, protože současné používané programovací jazyky podporují pojmy OOP pouze omezeným způsobem. Zhruba řečeno je rozdíl mezi konceptuálními a softwarovými objekty závislý na použitém programovacím prostředí a je proto např. v případě C++ větší, než při použití Smalltalku. Smyslem tvorby modelu s konceptuálními objekty je snaha mít implementačně nezávislou ale dostatečně podrobnou dokumentaci softwarového návrhu, která by byla použitelná i pro inovace systému po změně technologie. To by nebylo možné, kdyby se modelovalo jen podle možností aktuálního programovacího prostředí.

3. Objekty reálného světa, (anglicky jako „business objects“). Tyto objekty vyplňují mezeru mezi zadáním – tj. chápáním na aplikační úrovni zadavatele a mezi konceptuálním objektovým modelem. Objekty reálného světa podporují pouze vybrané pojmy OOP a většinu pojmů ponechávají na pozdějších transformacích. Model objektů reálného světa je použitelný nejen pro zahájení analýzy softwarové aplikace, ale i pro podporu práce např. manažerů pro tvorbu modelů při rozhodování, aniž by projekt vždy nutně končil softwarovou implementací.

### 5.3.3

#### Fáze expanze a konzolidace

V jednodušším pohledu na 6 fází BORMu můžeme rozlišit dvě hlavní etapy; expanze a konzolidace.

Fáze expanze začíná analýzou modelu business objektů. Dochází zde ke hromadění informací potřebných pro vytvoření aplikace. Stadium expanze končí s dokončením analytického konceptuálního modelu, který na logické úrovni reprezentuje požadované zadání a v abstraktní podobě popisuje jeho řešení.

Zbývající fáze od konceptuálního modelu až k finálnímu systému složenému ze softwarových objektů se označují jako stadium konzolidace. Je tomu tak proto, že v těchto etapách se model, který je produktem předchozí expanze, postupně stává fungujícím programem. To 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. (Takto odstraněná informace však může být v budoucnu základem analýzy nové verze systému).

Umění rozlišit a vyváženě řídit expanzi a konzolidaci je klíčovým faktorem úspěchu softwarových projektů. Samotný iterativní model totiž nezkušeného manažera svádí k neúměrnému počtu iterací s dlouhými expanzemi. Konzolidace se potom odbývá a výsledný produkt nemá potřebnou kvalitu.

### 5.3.4

#### Objekty reálného světa (business objekty)

Techniky a nástroje uvedené v této kapitole se týkají prvotního objektového modelu. Tento model je velmi vzdálen od pojmů a vazeb, které se používají při objektově orientovaném programování a naopak je blízký pojmům reálného světa. Tuto část BORMu je možné použít dvojím způsobem:

- a) Jako metodu pro rozpoznání zadání pro informační systém a sestavení prvotního modelu tohoto řešení nebo
- b) Jako metodu pro modelování, analýzu a reinženýring organizační struktury a business procesů firmy, přičemž následné budování informačního systému zde není podmínkou.

### 5.3.4.1

## Metoda OBA

Metoda OBA (Object Behavioral Analysis) je technika sloužící k získávání strukturovaných podkladů ze zadání pro potřeby konstrukce prvotního objektového modelu. Právě proto je velmi vhodná pro nasazení v počáteční fázi tvorby IS podle zásad BORMu, kde výstupy OBA analýzy slouží ke konstrukci diagramů „business“ objektů.

Metoda vznikla počátkem 90. let na základě zkušeností s aplikacemi různých technik JAD (Joint Application Design) a CRC (Class-Responsibility-Collaborator) pro potřeby objektové analýzy a návrhu a implementace v objektově orientovaných programovacích jazycích. (Bellin 1997)

1. Zaměstnanci (referenti) používají auta pro svoje služební účely.
2. Vedoucí zaměstnanců potvrzují žádosti zaměstnanců na auta pro služební cesty.
3. Autoprovoz přiděluje auta zaměstnancům.
4. Správa systému (přístupová práva, role uživatelů).

Tab. 15: příklad seznamu funkcí informačního systému

Jedná se o iterativní techniku začínající řízeným interview se zadavateli a pracujícími s různými typy formulářů, tabulek a modelových karet, ke kterým přísluší sada postupů a pravidel. Podrobnější popis OBA analýzy lze například nalézt v (Rubin et al. 1992). V knize (Carda et al. 2003) je podrobně popsána OBA jako nástroj metody BORM. Jednotlivé kroky OBA analýzy, jak jsou použity v BORMu, jsou následující:

1. krok – rozpoznání procesů. V tomto kroku se na základě provedeného interview sestaví seznam požadovaných funkcí systému a klíčové objekty v systému. Jedná se vesměs o textové popisy. Cílem tohoto kroku je nejen zahájit stavbu modelu, ale vymezit zadání v rámci možného širšího kontextu řešeného problému.
2. krok – rozpoznání plánování scénářů jako detailního popisu již rozpoznávaných funkcí a popisů vlastností objektů. V tomto kroku se u každého scénáře rozlišuje původ procesu, vlastní popis procesu, participující objekty a popis výsledku procesu.
3. krok – definování vztahů mezi objekty navzájem a mezi objekty a procesy pomocí modelových karet. V tomto kroku se pro každý rozpoznávaný objekt z předchozího kroku vytvoří jeho modelová karta, která obsahuje jméno objektu, seznam aktivit objektu a s ním související seznam s modelovaným objektem spolupracujících objektů..
4. krok – modelování procesů. V tomto kroku se pro každý rozpoznávaný objekt s pomocí informací v tabulce scénářů a modelových kartách sestaví životní cyklus objektu jako sled jeho stavů a přechodů mezi těmito stavy v podobě procesního diagramu.
5. krok – verifikace a validace. Zde se kontroluje shoda mezi diagramy, tabulkami a skutečnými požadavky na systém. K tomu slouží dva nástroje. Jedním z nich je datový model obsahující skutečná data pomocí nichž lze prověřit správnost návrhu. Druhým nástrojem je simulátor procesů, který dovoluje „sehrát a vyzkoušet“ proces znázorněný diagramy.

<i>initiation:</i> Zaměstnanec potřebuje auto na cestu	<i>roles:</i> <b>Auto</b>
<i>action:</i> Podání žádosti, její posouzení a přidělení auta	<b>Referent</b> performs <b>Vedoucí</b> approves
<i>result:</i> Zaměstnanec dostává přidělené auto na cestu nebo žádné auto na cestu nedostane	<b>Vedoucí autoprovozu</b> cooperates
<i>initiation:</i> Referent dostal přidělené auto	<i>roles:</i> <b>Auto</b>
<i>action:</i> Referent si vyzvedl auto v autoprovozu a provádí cestu autem, nebo si ho nevyzvedl (vrátil přidělení) a autem nejel	<b>Referent</b> performs <b>Vedoucí autoprovozu</b> is responsible
<i>result:</i> Auto je po služební cestě vráceno do autoprovozu, nebo referent vrátil přidělení	

Tab. 16: Příklady scénářů (generováno z Craft.CASE)

Metoda OBA je přímo založena na předpokladu iterativního přístupu k analýze. Například jednotlivé scénáře z 1. kroku jsou v 5. kroku příslušným předepsaným způsobem konfrontovány s životními cykly jednotlivých objektů a kontroluje se jejich vzájemná úplnost a souvislost. Následné kroky OBA tedy mohou posloužit i jako podklady pro dodatečné upřesňování informace v krocích předchozích. (Pro varianty známých řešení se doporučuje provést 2 až 3 opakování všech kroků – ani zde jeden průběh nestačí).

OBA pomáhá získávat strukturovaným způsobem potřebné podklady k sestavení prvotních objektových diagramů. Má však i další zajímavé přínosy do procesu tvorby I.S.:

1. poskytuje prostředky pro dokumentování projektu od samého počátku,
2. modelové karty a další výstupy OBA jsou znovupoužitelné v dalších podobných projektech (například jako návrhové vzory) a
3. úsilí vynaložené při sestavování scénářů a životních cyklů objektů lze zužitkovat při návrhu optimální funkčnosti uživatelského rozhraní.

Collaborators in diagram with name 'výpůjčka auta':	Referent	Vedoucí	Vedoucí autoprovozu
v autoparku: přiděluje se			<<
ve službě: vrací se do autoparku	<<		>>
přiděleno: vyjíždí z autoparku	<<		

Tab. 17: Modelová karta pro auto (generováno z Craft.CASE)

Collaborators in diagram with name 'výpůjčka auta':	Auto	Referent	Vedoucí autoprovozu
rozhoduje žádost: konzultuje			
rozhoduje žádost: potvrzuje žádost		>>	
start: přijímá žádost o auto		<<	
eviduje schválenou žádost: ruší výpůjčku auta		>>	>>
eviduje schválenou žádost: ukončuje evidenci výpůjčky			<<
rozhoduje žádost: zamítá žádost		>>	

Tab. 18: Modelová karta pro vedoucího (generováno z Craft.CASE)

Metodu OBA lze provádět dokonce jen s tužkou v ruce a příslušnými předtištěnými formuláři a tabulkami na papíře. Samozřejmě lepším způsobem je použití CASE nástroje, který dokáže většinu rutinních operací (například různé vzájemné kontroly, udržování projektových dat v konzistentním tvaru a možnost tisku tabulek a formulářů) provádět automaticky.

### 5.3.4.2 Diagram ORD

Diagram ORD (Object-Relationship-Diagram) byl vyvinut k vizuální reprezentaci informace o procesech a objektech získané metodou OBA. Jedná se o jednoduchý diagram, který obsahuje jen malý počet pojmů a symbolů, které jsou plně postačující pro prvotní popis modelovaných procesů a tím je použitelný i pro konzultace se zadavateli/zákazníky. Pojmy ORD jsou následující:

pojmem	symbol	popis
Objekt = Participant	Obdélník se jménem zobrazeným uvnitř v levém horním rohu.	Objekt (je označován jako „Participant“) představuje účastníka modelovaného procesu.
Stav	Menší obdélník odlišený barvou a typem písma pro pojmenování stavu kreslený dovnitř symbolu pro objekt.	Stavy vyjadřují postupné změny participantů v čase.

Asociace	Silná černá šipka s plným zakončením mezi participanty a nebo stavy. U šipky se píše popis, který blíže specifikuje charakter vazby.	Asociace vyjadřují datově orientované vztahy mezi participanty a nebo jejich stavy, protože se mohou v čase měnit. Vyjadřují jednotným způsobem vztahy, které mohou být později upřesněny jako skládání, dědění nebo závislost objektů.
Aktivita	Ovál propojený čarou s participanem nebo jeho stavem. Ovály mohou být kresleny také dovnitř k nim příslušných objektů.	Aktivita reprezentují jednotlivé aspekty chování objektů tak, jak byly rozpoznány pomocí scénářů v modelových kartách.
Komunikace	Šipka, která propojuje aktivity mezi sebou. Malé pojmenované šipky kreslené rovnoběžně k hlavní šipce komunikace vyjadřují datové toky.	Komunikace vyjadřují sled provádění a vzájemnou závislost aktivit různých objektů mezi sebou. datové toky mohou být vedeny oběma směry.
Přechod	Šipka s nevyplněným trojúhelníkovým zakončením, která propojuje aktivity a stavy jednoho objektu.	Součástí přechodu je také aktivita, ze které přechod vychází. Přechod tedy představuje činnost, kterou je třeba vykonat, aby objekt změnil svůj stav.
Podmínka	Přeškrtnutí s textovým popisem u komunikace nebo u propojení aktivity a objektu.	Podmínkou se vyjadřuje omezení platnosti komunikace nebo aktivity.

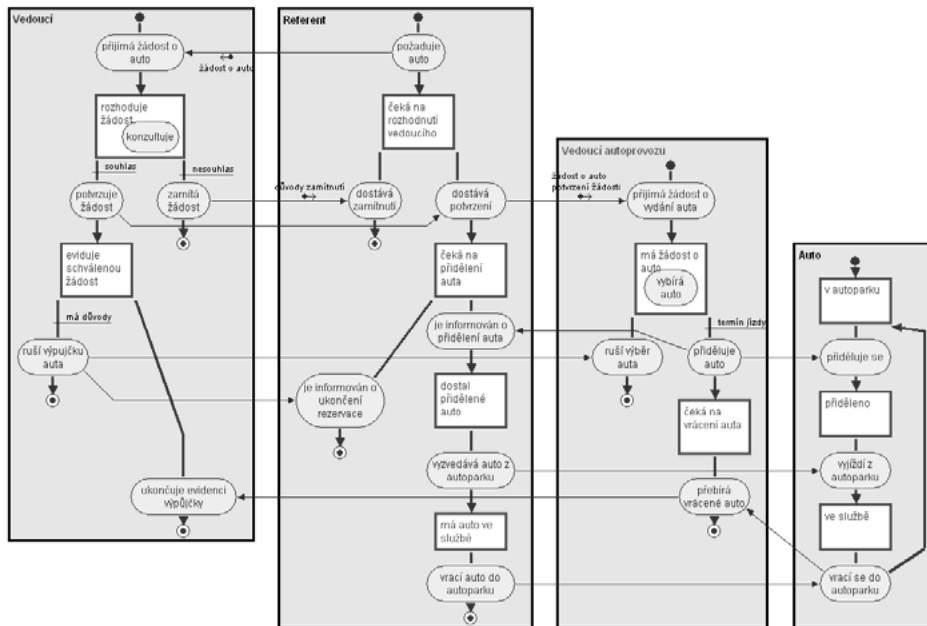
Tab. 19: Pojmy diagramu ORD

ORD dovoluje modelovat jednotlivé procesy současně dvojím způsobem:

1. Sekvence stavů a přechodů každého objektu, na které lze nazírat jako na jednotlivé stavové diagramy, vyjadřující roli daného objektu v modelovaném procesu. Tento pohled slouží ke kontrole celkového modelovaného procesu například při interview.
2. Sled komunikací mezi aktivitami různých objektů v různých stavech vyjadřuje průběh vlastního procesu. Celkový proces je tedy znázorněn jako propojení rolí objektů, které se tohoto procesu účastní. Nazíráme-li na participující objekty se svými stavy a přechody jako na automaty, jedná se o zobrazení průběhu procesu metodou komunikace automatů mezi sebou, kde výstup jednoho objektu je vstupem pro jiný objekt.

Vzhledem k tomu, že modelovaný proces je konstruován jako propojení rolí (stavů a přechodů) účastnících se objektů, tak ORD dovoluje jednoduchým a nenásilným způsobem zachytit přesný průběh modelovaného procesu a poskytuje tím i prostředky pro

ověřování jeho správnosti. (Do ORD totiž není například možné přidat aktivitu, která by nenavazovala na nějaký již přítomný stav nebo nebyla vázána nějakou komunikací s jinou aktivitou.) Tyto vlastnosti, které přímo vyplývají z použité teorie (ORD je diagram, kde každý objekt (participant) je modelován jako Mealyho automat. Při simulaci se využívá synchronizace pomocí Petriho sítě.), jsou velmi dobře využitelné v interview, ve kterých se diagram sestavuje nebo verifikuje.



Obr. 60: Příklad diagramu popisujícího proces (generováno z Craft.CASE)

### 5.3.4.3

## Podrobná analýza procesů

Pro podrobnou analýzu podnikatelských a správních procesů nelze vystačit se sadou základních pojmů. Součástí analýzy podnikatelských a správních procesů je v neposlední řadě analýza pracovních činností, systemizace pracovních míst, simulace procesů a návrh nové organizační struktury odvozené ze struktury procesů.

Pro konstrukci podrobného objektově orientovaného modelu podniku je stejně jako pro modely informačních systémů klíčový pojem procesu, participantu a aktivity, které v tomto kontextu interpretujeme následovně:

- Proces. Pro popis procesů nám slouží scénáře OBA a jejich podrobné rozpracování v podobě procesních diagramů ORD úplně stejně jako při modelování informačních systémů. V konkrétních modelech velkých organizací je takových procesů typicky 50 až 100.

- b) Participant. V perspektivě modelování organizačních a správních procesů jsou objekty – participanty jednotlivé funkční jednotky podniku. Může to být například útvar vedoucího provozu, oddělení obchodu, zákaznické centrum, nejrůznější provozní útvary, oddělení reklamace, úsek generálního ředitele apod. V konkrétních modelech velkých organizací je takových participantů nejčastěji 100 až 200. Mezi participanty je možné vytvářet asociace a vazby skládání. Participantem mohou být jak velké úseky – např. obchodní oddělení, tak i jednotky malé například na úrovni jednoho pracoviště. Kritériem pro rozpoznání participantu není velikost nebo oficiální zařazení v podnikové hierarchii ani prostorové vymezení, ale jen a pouze existence jednoznačné a pro participant charakteristické množiny aktivit.
- c) Aktivita je jedna konkrétní činnost, kterou provádí konkrétní participant v konkrétním procesu. Je to například „vyřízení objednávky“ nebo „posouzení reklamace“ nebo „vyjednávání stavebního povolení“. Aktivity mohou měnit stavy svých participantů. Aktivity by také měly mezi sebou komunikovat. V konkrétních modelech velkých organizací je takových aktivit (všech participantů ve všech procesech) asi 1000 až 5000. Pokud uvnitř aktivity dokážeme rozpoznat sled dalších aktivit, které různě komunikují, tak je účelné aktivitu rozdělit a mezi nimi ještě najít potřebné stavy. Naopak pokud je v modelu několik aktivit pohromadě, které komunikují stejným způsobem nebo nekomunikují vůbec, tak nemá smysl je rozlišovat a je vhodné je sloučit do jedné.

#### 5.3.4.4

### Rozšíření modelu business procesů směrem nahoru

Pro rozhodování nad podnikovými a správními procesy je třeba znát jejich souvislosti s dalšími atributy organizace. Takových atributů je několik druhů a zpravidla mezi ně patří cíle, úkoly, problémy a kritické faktory úspěchu. V konkrétních případech můžeme model rozšířit i o další.

Během podrobné analýzy při interview se uvedené atributy rozpoznávají a sleduje se jejich vliv a souvislosti s procesy. Nejčastějším způsobem prezentace těchto důležitých informací jsou tabulky – například tabulka procesů a cílů, kde řádky jsou jednotlivé procesy, sloupce jsou jednotlivé cíle a na průsečících procesů a cílů se vyznačuje míra ovlivnění příslušného cíle příslušným procesem.

#### 5.3.4.5

### Rozšíření modelu obchodních a správních procesů směrem dolů

Participanty a aktivity jsou při velmi podrobné analýze příliš hrubým nástrojem. Jejich úlohou je napomáhat při modelování procesů – především při jejich členění na menší části, které jsou spolu logicky propojeny. Pro podrobnou analýzu je však třeba získat informaci i o následujících údajích, které jsou užitečné pro úvahy o optimální podobě procesů a na jejich základě formulovaného zadání pro informační systém:

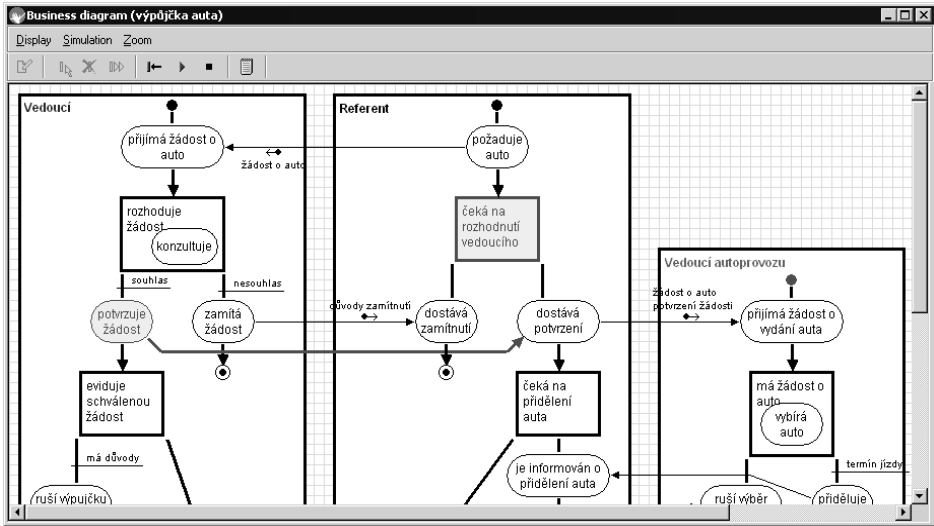
1. Pracovní místa. Je to konkrétní popis pracovního místa daného pracovníka s případným počtem konkrétních pracovníků ne této pozici. Je to například „Elektroinstalátor“, „Řidič stavebního stroje“ nebo „Sekretářka ředitele obchodního úseku“. Tuto úlohu nemůže převzít participant. Členění participantů do takových podrobností, které odpovídají pracovním místům, je sice možné, ale výsledný model by byl velmi komplexní, obsahoval by příliš mnoho opakujících se nebo podobných prvků, takže by byl velmi nepřehledný. V konkrétních modelech velkých organizací je takových pracovních míst nejčastěji 500 až 1000.
2. Popis pracovní činnosti. Je to konkrétní popis týkající se plnění jednoho pracovního úkolu. Je to například „zajištění pracoviště“ nebo „oprava vozidla“. Podobně jako u pracovních míst není vhodné spojit pojem pracovní činnosti s aktivitou. Průměrná aktivita v podrobném modelu obsahuje 2 až 5 takových pracovních činností. Celkem to znamená asi 1000 až 5000 činností. Mohou se ale i vyskytnout aktivity s jedinou pracovní činností.
3. Zařízení je konkrétní pracovní pomůcka nebo stroj, který je potřeba k vykonání dané aktivity (například „služební auto“ nebo „osobní počítač“ nebo „ochranný oděv“).
4. Software. Zde jsou na mysli komponenty nebo moduly zamýšlených nebo již existujících informačních systémů, které jsou potřeba pro vykonání dané aktivity.

Uvedené možnosti rozšíření modelů podnikových a správních procesů slouží například jako podpora návrhu optimální struktury procesů a od toho se odvíjející organizační struktury a vylepšených popisů pracovních činností jednotlivých pracovních míst. Možností, jak využít a dále rozpracovat modelovou informaci je opravdu mnoho. Jednou z možností je například její využití pro optimalizaci počtu pracovníků na jednotlivých pozicích metodou zjišťování časových ekvivalentů jednotlivých úkonů vzhledem k celkovému času daného pracovní dobou. Další možností je například využití informace jako podklady pro konstrukci uživatelských rozhraní budovaných komponent informačních systémů.

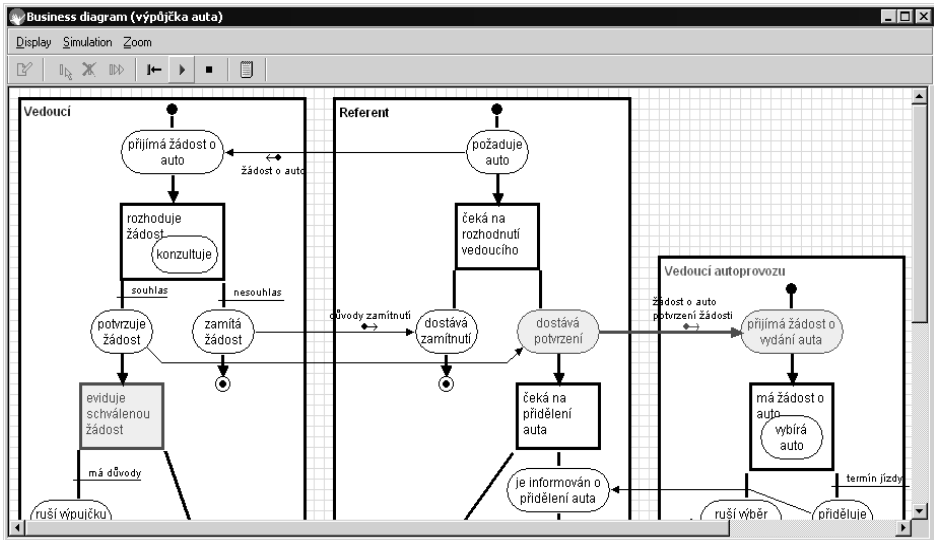
### 5.3.4.6

#### Simulace procesů

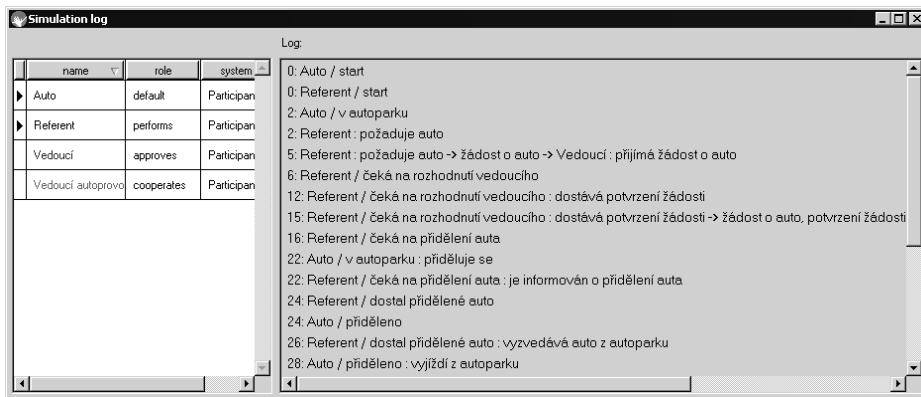
Modely obchodních a správních procesů, pokud jsme použili analytický nástroj, je možné použít i jako podklad pro sestavení simulačních modelů, se kterými lze provádět simulační experimenty. Do stavů a přechodů participantů může být doplněna informace o časovém průběhu. Máme-li k dispozici simulační software, tak můžeme modelovou informaci výhodně využít k podrobnému ověření funkčnosti a smysluplnosti navrhovaných procesů přímo se zadavatelem. Dokonce i v případě, kdy speciální simulační software není k dispozici, může být výhodné naprogramovat funkční prototypové řešení zamýšlené aplikace, které poslouží podobným způsobem, jakým by posloužil simulační model.



Obr. 61: Příklad simulace procesu v Craft.CASE – stav před obdržení potvrzení od vedoucího



Obr. 62: Příklad simulace procesu v Craft.CASE – stav po obdržení potvrzení od vedoucího



Obr. 63: Simulační záznam v Craft.CASE – seznam událostí objektu Referent a Auto

### 5.3.4.7

## Změna procesů – Business Process Reengineering

Modelování a změna podnikatelských a správních procesů (business procesů) je klíčovou technikou při provádění BPR (Business Process Reengineeringu). BPR lze definovat jako zamyšlení a zásadní přeměnu fungování dané organizace. [Hammer et al. 1994] Hlavní vlastnost charakteristická pro BPR je jeho primární zaměření na procesy – tedy na to, co podnik dělá, či musí dělat, a teprve sekundárně – na základě poznanych procesů – na organizační strukturu, která by měla činnosti realizovat. BPR se provádí se ve dvou etapách:

1. V první etapě (etapa AS-IS = tak, jak to je) se podrobně modeluje stávající stav podnikových procesů. To je důležité pro exaktní vymezení předností a především nedostatků podniku. Bez této nepopulární fáze totiž nelze zodpovědně modelovat budoucí stav. Pro analytika je kriticky důležité v této fázi udržet projekt tak, aby důsledně popisoval pouze stávající stav se všemi nedostatky i bizarnostmi, neboť interview se zadavateli mají zpravidla tendenci sklouznout k povídání o chtěných nebo zamýšlených strukturách a k zatajování popisu problémů se stávající strukturou.
2. Po precizním vyhodnocení výsledků první etapy (měření, porovnávání, konzultace) se přistupuje k druhé etapě (etapa TO-BE = jak by to mělo být). Zde se navrhuje a podrobně popisují nové procesy a z informací v nich uložených se nakonec provádí nový návrh.

V BORMu se přístup BPR doporučuje u složitějších projektů k upřesnění a verifikaci prostředí, do kterého se informační systém projektuje, a na základě kterého se formulují funkční požadavky na tento informační systém.

### 5.3.5

## Logické – konceptuální objekty

S nimi se v BORMu setkáváme ve středních fázích vývoje systému. Model tvořený konceptuálními neboli logickými objekty stojí jakoby napůl cesty mezi zadáním a řešením. Proto se také zpočátku na tyto objekty a jejich vazby nahlíží z perspektivy analýzy, kdy je ještě dovoleno použít model z konceptuálních objektů použit k upřesnění modelu zadání, ale zároveň model tvořený konceptuálními objekty slouží pro zahájení návrhu, kdy již považujeme problém za rozpoznáný a začínáme jej konsolidovat do počítačově realizovatelné podoby.

### 5.3.6

## Přechod od business objektů ke konceptuálním objektům

Přechod od objektů reálného světa ke konceptuálním objektům je možné stručně popsat následovně:

1. Nejprve se provede podrobný popis objektů reálného světa a naleznou se vazby „je jako“ (is-a) a asociace.
2. Na základě znalosti objektů reálného světa se naleznou třídy a množiny objektů.
3. Vazby „je jako“ (is-a) poslouží pro sestavení hierarchie typů.
4. Asociace mezi objekty a provázání objektů pomocí komunikací poslouží pro nalezení vazeb skládání (agregace) a pro posílání zpráv mezi metodami objektů.

#### 5.3.6.1

### Diagramy konceptuálních objektů

Diagramy konceptuálních objektů jsou na rozdíl od diagramů objektů reálného světa poměrně známé a používané již od začátku 90. let. Někdy se však nesprávně ztotožňují s diagramy objektů softwarových, protože se v nich objevují velmi podobné nebo zcela shodné pojmy. Rozdíl je však ve způsobu nazírání na tyto pojmy a vazby. V BORMu je pro tyto diagramy použit upravený UML. Jsou to tyto úpravy a doplnění:

- a) UML nerozlišuje mezi polymorfismem, děděním, hierarchií typů a hierarchií „je jako“ (IS-A hierarchie). V našem přístupu tyto vazby graficky rozlišujeme.
- b) UML nerozlišuje mezi pojmem třída objektů a množina objektů. V našem přístupu tyto vazby rozlišujeme a zavádíme nový grafický symbol pro množinu objektů a pro třídu jako objekt sám o sobě. Pojmy třída a množina v BORMu se z důvodu jednoduchosti a srozumitelnosti modelují jednotně pouze pro objekty reálného světa.
- c) V našem přístupu klademe důraz na dynamickou stránku problému. Metody objektů včetně zobrazených podrobností jejich vazeb (zpráv) na jiné metody jiných objektů jsou nedílnou součástí popisu systému a používají se ve všech typech diagramů. Proto jsme zavedli samostatný grafický symbol pro metodu a pro zprávu poslanou z metody k jiné metodě.

d) UML dovoluje v jednom konkrétním objektovém diagramu použít současně vazby a pojmy na různé úrovni abstrakce – tedy míchat pojmy objektů reálného světa s konceptuálními a softwarovými. V jednom diagramu je možné mít například současně vyznačené asociace spolu s děděním atd. V našem přístupu doporučujeme používat pojmy postupně, tak jak se během modelování od sebe odvozují.

### 5.3.7

## Softwarové – implementační objekty

Se softwarovými objekty se v BORMu setkáváme až v závěrečných fázích životního cyklu vývoje systému, kdy je třeba model postupně transformovat do takové podoby, která je vyžadována pro fyzickou realizaci systému v podobě programu v daném programovacím jazyce. Právě dokončení modelu tvořeného softwarovými objekty na takové úrovni podrobností, které již odpovídají výrazovým prostředkům použitého jazyka, se považuje za okamžik ukončení objektově orientovaného návrhu a zahájení implementace. Přeměnu modelu tvořeného strukturou konceptuálních objektů a model softwarových objektů lze stručně popsat následujícím způsobem:

- a) Přeměna hierarchie typů na hierarchii dědění mezi třídami a případná transformace vícenásobné dědičnosti na jednoduchou.
- b) Doplnění tříd o atributy a metody, které umožní realizovat stavy a přechody (současné smíšené i čisté objektové jazyky se stavy a přechody přímo nepracují).
- c) Využití návrhových vzorů pro optimalizaci a pro kontrolu proveditelnosti modelu v daném programovacím jazyce.
- d) Upravit strukturu modelu podle dostupných výrazových prostředků implementačního prostředí.
- e) Napojení modelu na struktury v existujících systémech, se kterými náš systém musí spolupracovat (například pro výměnu dat).

### 5.3.8

## Přínos rozdělení modelu na business, konceptuální a softwarové objekty

Podívejme se ještě jednou na důvody používání tří objektových modelů během tvorby systému:

### „business“ modelování

Nejprve je doporučeno sestavit model zadání v kategoriích business objektů. Zde je výhodné nepoužívat žádné specificky softwarové pojmy, protože je na ně příliš brzy. Důležitější je zde dosažení porozumění mezi zadavatelem a analytikem a jeho řádné dokumentování. Programátorské pojmy zde nepoužíváme také proto, že některá vazby známé v reálném světě mají jiný význam, než ve světě softwarového kódu (např. dědičnost) a nebo je dnes používané programovací jazyky nepodporují vůbec (například delegování nebo závislost).

V této fázi modelování se také zabýváme i vztahy mezi participanty, které nakonec nebudou součástí funkčnosti budované aplikace. Jejich modelování ale může být důležité pro upřesnění zadání a pro provedení nezbytné reorganizační změny v okolí projektovaného informačního systému.

## Konceptuální modelování

Konceptuální model vzniká transformací z předchozího modelu, ze kterého se vyberou ty objekty a vazby, které reprezentují budovaný informační systém. Tyto objekty a vazby jsou potom základem pro sestavení modelů popisujících konceptuální analýzu budovaného systému. (Což je místo, až od kterého klasické metodiky – jako např. OMT – začínají projektování). I když je tento konceptuální model již popisem budovaného systému, tak není vhodné „skočit“ přímo do světa implementace a při výběru pojmů a vazeb se omezit jen na vlastnosti cílového implementačního prostředí. Tato dokumentace totiž musí sloužit i při možných pozdějších změnách a rozšířeních systému, přechodu na jinou technologii atd.

## Softwarové modelování

Tato poslední fáze je procesem, kdy se sestavený konceptuální model konsoliduje do takové podoby, že je ho možné implementovat v příslušném programovacím prostředí. Rozdíly mezi konceptuálním a softwarovým modelem tedy nespočívají jen v různé míře zobrazeného detailu, ale hlavně v zohlednění konkrétních implementačních omezení jak ze strany programovacího prostředí, tak i ze strany dříve vyrobených softwarových komponent, se kterými musí nový systém spolupracovat, a které téměř nikdy nemají ideální znovupoužitelnou strukturu.

### 5.3.9

## Evoluce hierarchií objektů

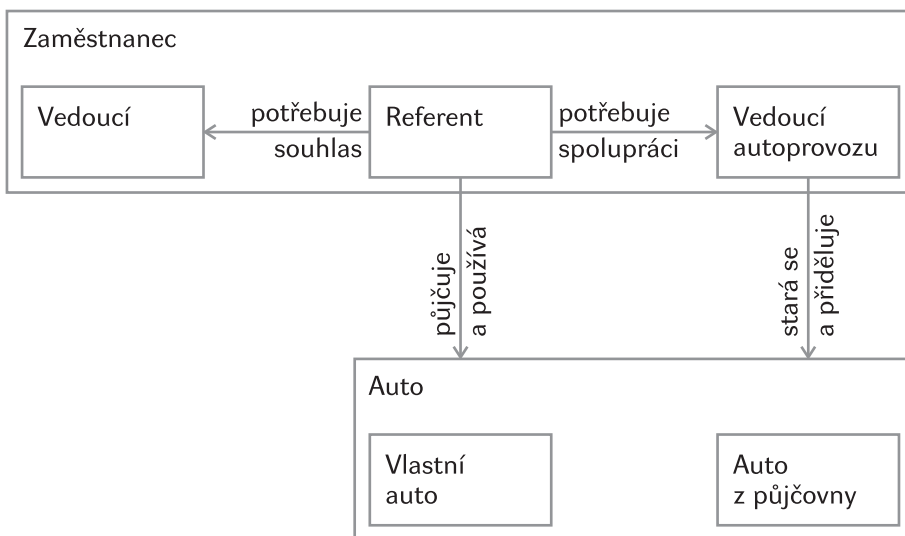
Nové typy se v objektových systémech realizují většinou pomocí tříd, přičemž ale programátoři vědí, že novou třídu do systému lze vyrobit nejen pomocí dědění, ale i skládáním. Z toho proto vyplývá, že hierarchie dědění v implementačním modelu a hierarchie typů v analytickém modelu jednoho systému nemusí vždy znamenat totéž. Navíc při používání dědičnosti máme v programovacích jazycích nástroje pro skrývání metod a dat děděných objektů. Na hierarchii tříd objektů se proto v BORMu nahlíží trojím způsobem podle následujících kritérií:

1. Z pohledu návrháře – tvůrce nových objektů. Tato hierarchie je hierarchií dědičnosti, protože dědičnost je programátorským nástrojem pro tvorbu nových tříd. Její místo je ale až ve fázi softwarového modelování.
2. Z pohledu uživatele – analytika nebo aplikačního programátora, který potřebuje již hotové objekty použít ve svém systému. Tento pohled, který předchází implementaci, 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. Její místo je ve fázi konceptuálního modelování.
- 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 i datovým obsahem objektu a jeho konkrétní rolí v modelovaném systému. Její místo je ve fázi „business“ modelování.

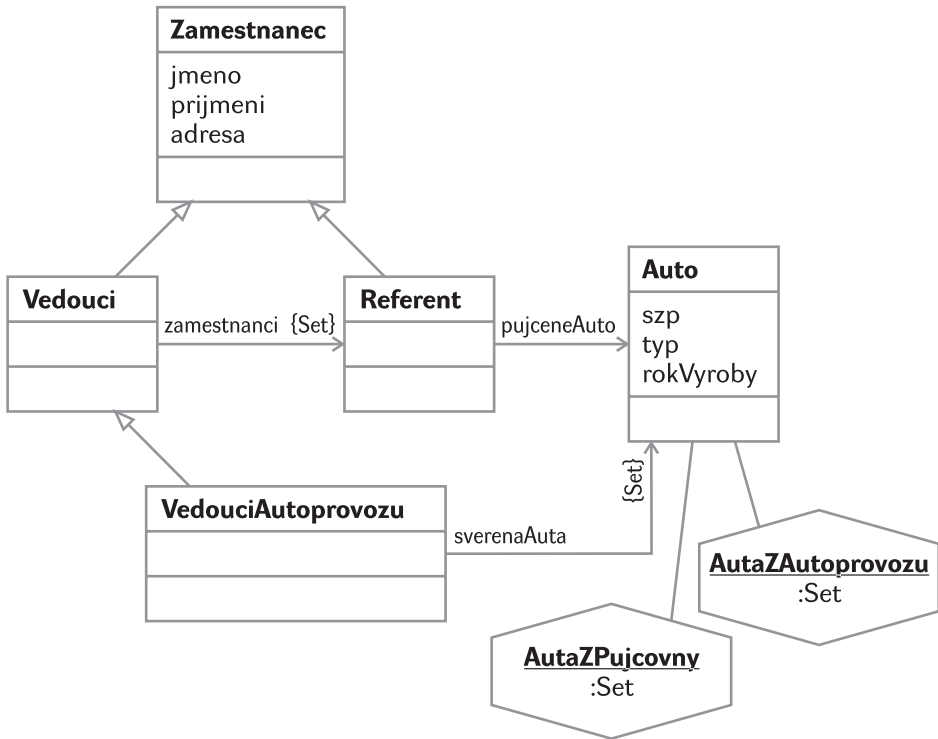
U jednoduchých úloh je samozřejmě pravda, že uvedené tři hierarchie jsou totožné. Proto se tímto problémem programátorské kuchařky příliš nezabývají a všechny tři typy hierarchií považují za dědičnost. U komplexnějších úloh však toto tvrzení neplatí a to především při návrhu systémových knihoven, které se opakovaně znovupoužívají při návrhu konkrétních systémů. V některých objektových programovacích jazycích lze dokonce nalézt prostředky pro oddělení typů a tříd.

V BORMu se nejprve pracuje pouze s hierarchií „je jako“ (anglicky IS-A), z ní se později odvodí hierarchie typů a nakonec se na základě typů navrhne hierarchie dědičnosti. Tento přístup, kdy se s hierarchiemi objektů pracuje v různých fázích vývoje různě, zabraňuje nadměrnému a nesprávnému použití dědičnosti v průběhu implementace. Postupnou evoluci hierarchií budeme demonstrovat na následujícím příkladu, do kterého patřil i příklad procesu simulace a OBA a je popsán v (Carda et al. 2003). Ve fázi business modelování byla sestavena hierarchie objektů, kde v horní části diagramu je participant vedoucí, referent a vedoucí autoprovozu jako tři disjunktní podmnožiny pod participantem zaměstnanec:



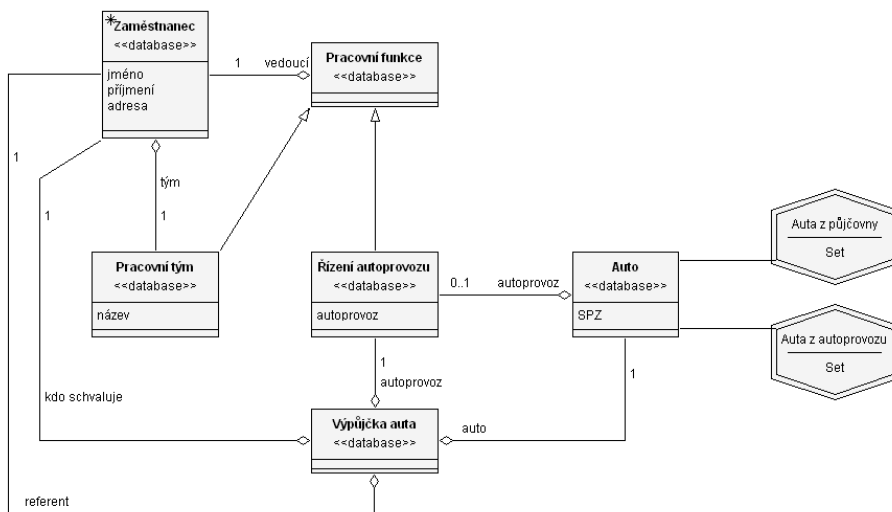
Obr. 64: Příklad proměny datového modelu – fáze business modelování

V následné fázi konceptuálního modelování došlo nejen ke zpřesnění modelu, ale také ke změně hierarchie. Kritériem už totiž není příslušnost do domény, ale chování objektů. Proto je teď vedoucí autoprovozu „podtypem“ vedoucího. (vedoucí autoprovozu se chovají stejně jako vedoucí a k tomu mají ještě další vlastnosti). Tuto hierarchii nebylo vhodné použít dříve, protože bychom touto abstrakcí zmátli zadavatele systému z praxe, kam se systém projektuje. V reálném světě totiž žádný vedoucí autoprovozu není zároveň vedoucím zaměstnanců, ale jde o jinou funkci. Ve fázi business modelování totiž hierarchie znázorňuje vztah domén (kdo je kdo), tady ale znázorňuje vztahy v chování a vlastnostech (kdo je jako).



Obr. 65: Příklad proměny datového modelu – fáze konceptuálního modelování

A nakonec ve fázi softwarového modelování muselo dojít k dalšímu přepracování, protože systém nevznikal na zelené louce, ale jeho implementace se musela napojit na databázi všech zaměstnanců (v diagramu třída označená hvězdičkou). Nebylo proto možné z předchozí fáze rozpoznané podtypy implementovat jako různé třídy. Proto se musela použít transformace podle zásad návrhového vzoru Stav tak, aby různé typy a podtypy objektů byly implementovány bez dědění jen pomocí skládání (v příkladu to je skládání k objektu pracovní funkce).



Obr. 66: Příklad proměny datového modelu – fáze softwarového modelování

Jak ukazuje tento příklad z praxe, tak není v praxi často možné realizovat „krásný“ objektový návrh právě z důvodu nutnosti zachování a napojení se na předchozí struktury. Na druhou stranu by bylo chybou, kdyby analytici přeskakovali prostřední fázi konceptuálního modelování a z analýzy zadání by se rovnou začali věnovat implementaci. Vždy je totiž prospěšné vědět a mít dokumentované, jaký je „ideální“ konceptuální model systému. Tato znalost je totiž velmi užitečná při údržbě, rozšiřování, opravách apod.

### 5.3.10

#### Tři dimenze objektového modelu – zjednodušení složitosti

V ideálním případě si lze v BORMu představit jediný souhrnný diagram pro celý systém. Rozsáhlost modelu u velkých systémů však prakticky znemožňuje s takovými diagramy pracovat. Proto je třeba pracovat s menšími diagramy, které vždy popisují pouze část systému. Široce používanou možností zjednodušení složitosti modelů jsou dekompozice a hierarchie. Je však také možné použít ještě jeden způsob, jak snížit složitost:

Prvky a vazby v objektově orientovaném systému lze třídit a filtrovat podle toho, jestli nesou informaci o datech nebo o funkcích nebo o změnách v čase. Tato tři kritéria si potom můžeme představit jako tři rozměry abstraktního znalostního prostoru, ve kterém je objektově orientovaný model vytvářen:

1. Vynecháním časového rozměru získáme pohled, který zobrazuje vztah dat a funkcí mezi sebou. Zobrazíme-li potom funkce ve větší podrobnosti před daty (objekty), tak dostaneme pohled, který se v standardním UML nazývá „object interaction diagram“. Zobrazíme-li naopak data (objekty ve větší podrobnosti) před funkcemi

(metodami), tak do této skupiny také může patřit i diagram, který zobrazuje objekty, třídy, atributy a metody včetně objektových hierarchií (dědění, skládání) a je v standardní UML nazýván jako „object-class diagram“.

2. Vynecháním datového rozměru získáme pohled, který zobrazuje vztahy funkcí (metod) v závislosti na čase. Typickým představitelem je sekvenční diagram z UML.
3. Vynecháním funkčního rozměru získáme pohled, který zobrazuje chování dat v čase. Takový typ diagramu v standardním UML není. V tomto pohledu jde totiž o zobrazení, jak se v jednotlivých stavech v čase mění datová struktura systému. V BORMu k tomuto slouží již dříve popsaný procesní diagram. Pro každý stav, který budeme chtít zobrazit, lze jako jeho dekompozici sestrojít samostatný zjednodušený „object-class“ diagram definující data a vazby daného stavu. Právě odlišností mezi těmito diagramy jednotlivých stavů (jiné vazby, jiné hodnoty proměnných, změny v kardinalitě, ...) názorně ukazují průběh datových změn systému v čase a napomáhají jeho softwarové implementaci.

### 5.3.11

#### Chyby, kterých je třeba se vyvarovat při modelování

Při vytváření struktury objektového modelu nejčastěji dochází vlivem podcenění výše uvedených postupů a ovlivnění neobjektovými technikami návrhu k následujícím chybám, které jsou diskutovány například v (Abadi 1996, Larsson et al. 2002, Molhanec 2004):

- ⊗ 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 „ukaz“ 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.
- ⊗ Zjednodušení modelu výpočtu směrem k fyzické architektuře počítače projevující se v omezení parametrů zpráv a atributů objektů na pouze skalární data typu „znak“ nebo „číslo“ atp.
- ⊗ Chybné stanovení hranice, která určuje, kdy se různé objekty mají ještě modelovat hierarchií různých tříd a kdy se již jedná o objekty s různým datovým obsahem ale stejné třídy.
- ⊗ Nerozlišení pojmů třída objektů a kolekce objektů v konceptuálním modelu a z toho vyplývající nevhodná implementace typů pomocí tříd a chápání kolekcí jen jako extentů tříd.

## 5.3.12 Zkušenosti

Metoda BORM a především její možnosti analýzy v počátečních fázích vývoje projektu byla v letech 1996 až 2000 prakticky použita firmou Deloitte&Touche například v projektech pro pražské zdravotnictví, Ústav pro státní informační systém ČR, elektroenergetiku, zemědělství, telekomunikace a plynárenství. Ve všech projektech se ukázalo, že tento přístup lze dobře využívat jako nástroj pro provádění business process reengineeringu a pro analýzu znalostí o business a workflow procesech. Proto se používá i na jiných projektech, než z oblasti softwarového inženýrství. Výsledky takové analýzy ale mohou velmi dobře sloužit pro pozdější podrobnou a úplnou specifikaci zadání softwarového projektu

Přínosy této metody ve tvorbě softwaru byly ověřeny při použití programovacích nástrojů VisualWorks/Smalltalk-80, Visual Basic, .NET a Control Web, přičemž vytvářené aplikace jsou vesměs typu klient-server a využívají relační databázové systémy Oracle, rozhraní ODBC a nebo objektovou databázi Gemstone. Metoda je využívána ve firmě e-Fractal s.r.o., která je jedním z největších obchodníků na českém internetu a zabývá se také tvorbou softwaru na zakázku, především webových informačních systémů využívajících objektovou databázi.

Součástí vývoje BORMu byl i výběr vhodného analytického a modelovacího CASE nástroje. V současné době je BORM již implementován v nástroji Metaedit Plus., finské firmy MetaCase. MetaEdit je výsledkem výzkumného projektu na univerzitě v Jüvaskylä. Od listopadu 2004 také probíhá pod patronací Deloitte implementace vlastního nástroje Craft.CASE,,. Autor této knihy se podílel v letech 1998-2000 na implementaci BORMu do systému MetaEDIT a v současné době vede analytický tým projektu CraftCASE.

## Závěr

Objektově orientovaný přístup se dnes stal hlavním způsobem tvorby softwaru i analýzy systémů. I když ale dnes máme k dispozici CASE nástroje, výkonná vývojová prostředí, sofistikované knihovny a komponenty, tak není pravda, že vytvářené systémy jsou bez problémů. Nejen studenti, ale i vývojáři ve firmách občas vytvářejí velmi bizarní výtvo-ry. Setkáváme se s nesprávným používáním vazeb a hierarchií mezi objekty, krkolomnými triky v kódu atp. Problém takových aplikací není ale vždy v tom, že nefungují. Naneštěstí jsou mnohdy v chodu díky moderním komponentám, vývojovým prostředím a výkonnému hardwaru opravdu podivné konstrukce. O to je potom horší debata s tvůr-cem, že by měl ve vlastním zájmu systém přepracovat, protože argumentům o obtížné údržbě, rozšiřitelnosti, spolehlivosti, nebezpečí nekonzistence a redundance dat, ... není ochoten naslouchat.

Materiál předložený v této knize si proto kladl za cíl podat jednotlívý přehled o proble-matice OOP a předložit názor autora na diskutovaný obor, zamyslet se nad způsobem výkladu jednotlivých nástrojů a technik, u kterých se může na první pohled zdát, že spolu vzájemně nijak nesouvisí. Je velká škoda, že tak perspektivní a již prakticky používaná technologie, jako je OOP, nemá dosud srozumitelný a všeobecně uznávaný teoretický základ a na něm vybudované formální techniky, které by se výborně uplatnily ve výuce a inspirovaly aplikovaný výzkum. Je nepochybné, že se tímto tématem zabývá řada pra-covišť ve světě, ale zatím nebyly publikovány ucelené a všeobecně uznávané výsledky. Proto se domnívám, že blízká budoucnost přinese možná i několik alternativních přístu-pů. Jedním z nich se snaží být i myšlenky popisované v této práci.

Mezi původní a nové myšlenky popisované v této práci patří především návrh využít datové modelování jako nástroj pro lepší analýzu systémů a přesnější formulaci požadav-ků na informační systémy.

V první části knihy je podán přehled nástrojů tohoto nového oboru. Jednotlívým prv-kem je zde formální aparát založený na lambda-kalkulu, pomocí kterého jsou vykládány vlastnosti objektově orientovaných dat. V této části knihy lze nalézt také úvod do jazyka Smalltalk, výukové modely Daskalos a LambdaTalk, modelování v čistě objektové data-bázi Gemstone a na základě vlastních zkušeností z praxe modifikovanou techniku nor-malizace tříd.

Druhá část knihy se zabývá vztahem datového modelování k metodám projektování. Praktické shrnutí výsledků práce autora za období cca 15 let v diskutovaném oboru před-stavuje metodika BORM, která byla nejen několikrát publikována u nás i v zahraničí, ale již se začala i prakticky používat. Nejnovější výsledek zde diskutovaných postupů je také projekt modelovacího nástroje Craft.CASE.

Materiál předložený v této knize vznikl za přispění výzkumného záměru MSM6046070904 zabývajícího se informační podporou pro řízení.

## Použitá literatura

### 7.1

#### Vlastní publikace

- Carda et al. 2003 Carda A., Merunka V., Polák J.: *Umění systémového návrhu*, Grada 2003, ISBN 80-247-0424-2
- Hall et al. 2004 Hall J.et al.: *Accounting Information Systems 3rd edition*, South-Western Publishing, 2004, ISBN 0538877960. (spoluautor kapitoly 4. – System Development Activities)
- Knott et al. 2000 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 ICRE 2000, Chicago 2000. IEEE Computer Society Press ISBN 0-7695-0565-1
- Knott et al. 2003 Knott, Roger P.; Merunka, Vojtěch; Polák, Jiří: „*The BORM methodology: a third-generation fully object-oriented methodology*“, Knowledge-Based Systems 3(10) 2003, Elsevier Science Publishing, New York.
- Liu et al. 2005 Liu L. et al: *Management of the Object-Oriented Development Process*, Virgin Island 2005, ISBN 1-59140-605-6 (spoluautor kapitoly 15. – The BORM Methodology)
- Merunka et al. 2000 Merunka, V.; Polák, J.; Kofránek J.: *Úvod do metody BORM*, ve sborníku konference Objekty 2000, ISBN 80-213-0682-3
- Merunka 2001 Merunka V.: *Zkušenosti s objektovou analýzou a návrhem*, ve sborníku konference Objekty 2001, ISBN 80-213-0829-X
- Merunka 2002A Merunka V.: *Objekty v databázových systémech*, skriptum ČZU Praha, Praha 2002. ISBN 80-213-0318-2
- Merunka 2002B Merunka V.: *Řídící a podpůrné procesy v objektově orientované tvorbě softwaru*, ve sborníku konference Objekty, listopad 2002 Praha, ISBN 80-213-0947-4
- Merunka 2003 Merunka V.: *Objektový databázový systém Gemstone*, sborník konference OBJEKTY 2003. Ostrava 2003. ISBN 80-248-0274-0
- Merunka et al. 2004 Merunka V., Pergl R., Pícka M., *Objektově orientovaná tvorba software*, ČZU Praha, 2004, ISBN 80-213-1159-2

## 7.2

### Ostatní

- Abadi 1996 Abadi M., Cardelli L.: *A Theory of Objects*, Springer 1996, ISBN 0-387-94775-2
- Agha et al. 1994 Agha Gul, Hewitt Carl: *Actors – A Conceptual Foundation for Cocurrent Object-Oriented Programming*, pp 49-74, Research in OOP 1994, MIT Press ISBN 0-262-19264-0
- Ambler 1997 Ambler Scott: *Building Object Applications That Work, Your Step-By-Step Handbook for Developing Robust Systems Using Object Technology*, Cambridge University Press/SIGS Books, 1997, ISBN 0521-64826-2
- Ambler 1998 Ambler, Scott W.: *Process Patterns – Building Large-Scale Systems Using OO Technology*, Cambridge University Press – Managing Object Technology Series 1998, ISBN 0-521-64568-9
- Ambler 1999 Ambler, Scott W.: *More Process Patterns – Delivering Large-Scale Systems Using OO Technology*, Cambridge University Press – Managing Object Technology Series 1999, ISBN 0-521-65262-6
- Ambler et al. 2004 Ambler Scott, Beck Kent: *Object Orientation – Bringing data professionals and application developers together*, prosinec 2004, <http://www.agiledata.org/essays/objectOrientation101.html>
- Ambler 2004 Ambler: *The Object Primer – Agile Model-Driven Development with UML 2.0*, Ronin International, ISBN 9780521540186.
- AP 2001 *AP manifesto*, <http://www.agilealliance.org>, December 2004
- Barry 1998 Barry D.: *The Object Database Handbook: How to Select, Implement, and Use Object-Oriented Databases*, ISBN 0471147184
- Bartoška et al. 2004 Bartoška Jan, Adámková Milena, Hnátková Běla, *Tvorba softwaru nad biometrickou strukturou dat*, ve sborníku konference Objekty 2004
- Beck 1997 Beck K.: *Smalltalk Best Practice Patterns*, Prentice Hall 1997, ISBN 0-13-476904-X
- Beck 2002 Beck K.: *Extrémní programování (český překlad)* Grada 2002, ISBN 80-247-0300-9
- Beck 2003 Beck K.: *Agile Database Techniques – Effective Strategies for the Agile Software Developer*, John Wiley & Sons; 2003, ISBN 0471202835
- Bellin et al. 1997 Bellin, David; Simone, Susan Suchman: *The CRC Card Book*, Addison-Wesley 1997 ISBN: 0201895358
- Bertino et al. 1995 Bertino E., Martino L: *Object-Oriented Database Systems – Concepts and Architectures*, Addison Wesley 1995, ISBN 0-201-62439-7
- Biermann 1990 Biermann Alan W.: *Great Ideas in Computer Science*, MIT Press 1990, ISBN 0-262-02302-4
- Blaha et al. 1995 Blaha M., Premerlani M.: *Object-Oriented Modeling and Design for Database Applications*, Prentice Hall 1995, ISBN 0-13-123829-9

- Booch 1994 Booch Grady: *Object-Oriented Analysis and Design with Applications*, Second Edition, Benjamin Cummings 1994, ISBN 0-8053-5340-2
- Buchalcevová 2005 Buchalcevová A.: *Metodiky vývoje a údržby informačních systémů*, Grada 2005, ISBN 80-247-1075-7
- Burleson 1999 Burleson D.: *Inside the Object Database Model*, CRC Press 1999, ISBN 0-8493-1807-6
- Coad 1993 Peter Coad: *Object-Oriented Programming*, Yourdon Press, 1993
- Coad et al. 1995 Peter Coad, David North and Mark Mayfield: *Object Models: Strategies, Patterns and Applications*, Yourdon Press, 1995
- Catell 1998 Catell R. G.: *The Object Data Standard: ODMG 3.0*, ODMG 1998, ISBN 1558606475
- Choppy 2004 Choppy C.: *Improving Use-Case Based Requirements*, in proceedings of FASE – Fundamental Approaches to Software Engineering 2002, pp. 244-261, Springer ISSN 0302-9743
- Darnton 1997 Darnton, Geoffrey, Darnton, Moksha: *Business Process Analysis*, International Thomson Publishing 1997 ISBN: 1861520395
- Davis 1993 Davis, A.: *Software Requirements – Objects, Functions and States*, Prentice Hall 1993
- Davis 1986 Davis M D, Weyuker E J.: *Computability, Complexity and Languages – Fundamentals of Theoretical Computer Science*, Academic Press 1986, ISBN 0-12-206380-5
- Derr 1995 Derr K.W.: *Applying OMT – A Practical Guide to Using the Object Modelling Technique*, Sigs Books 1995, ISBN 1-884842-10-0, Prentice Hall 1995 ISBN 0-13-231390-1
- Doskočil 2004 Doskočil Tomáš, Merunka Vojtěch, *Zkušenosti z výuky s objektovým databázovým systémem Gemstone/S v předmětu DBII na KII PEF ČZU*, ve sborníku konference Informatika o výuce informatiky, Zlín 2004, ISBN 80-7302-066-1
- Drbal 1996 Drbal Pavel, *Proudy v objektově orientovaných metodikách*, ve sborníku konference Tvorba softwaru 1996, Ostrava.
- Drbal 2002 Drbal P.: *Extrémní programování a metodický přístup*, ve sborníku konference Tvorba softwaru 2002, ISBN 80-85988-74-7
- Ewusi 2003 Ewusi K.: *Software Development Failures*, MIT Press 1993, ISBN 0-262-05072-2
- Gamma et al. 2003 Gamma E. et al.: *Návrh programů pomocí vzorů*, český překlad z Design Patterns – Elements of Reusable Software, Grada 2003, ISBN 80-247-0302-5
- Gemstone 2004 *Gemstone Object Server – documentation & non-commercial version download*, <http://www.gemstone.com>, <http://smalltalk.gemstone.com>, prosinec 2004
- Goldberg 1995 Goldberg Adele, Kenneth Rubin S.: *Succeeding with Objects – Decision Frameworks for Project Management*, Addison Wesley 1995, ISBN 0-201-62878-3

- Graham et al. 2002 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>, květen 1992
- Hankin Hankin Chris.: *Lambda Calculi – A Guide for Computer Scientists*, Clarendon Press – Oxford 1994, ISBN 0-19-853841-3
- Hay 2006 Hay David: *Data Model Patterns*, Morgan Kaufman 2006, ISBN 0120887983
- Hammer et al. 1994 Hammer, M. – Stanton, A.: *The Reengineering Revolution*, Collins 1994
- Henderson-Sellers Henderson-Sellers B, Edwards JM.: *MOSES – A Second Generation Object-Oriented Methodology*, pp 68–73, Object Magazine 4(3) 1994
- Hopkins 1992 Hopkins T.: *Animating an Actor Programming Model*, Computer Science Dept. University of Manchester 1992
- HOPL 1988 Wexelblat Richard L.: *History of Programming Languages*, Academic Press Pennsylvania 1988, ACM Monographic Series, from the ACM SIGPLAN History of Programming Languages Conference, June 1-3, 1988, ISBN 0-12-745040-8
- Hruška 1995 Hruška T.: *Objektově orientované databázové technologie*, sborník konference Tvorba softwaru 1995, ISBN 80-901751-3-9
- Jacobson 1992 Jacobson Ivar: *Object-Oriented Software Engineering – A Use Case Driven Approach*, Addison Wesley 1992, ISBN 0-201-54435-0
- Jacobson et al. 1999 Jacobson I., Booch G., Rumbaugh J.: *The Unified Development Process*, Addison Wesley 1999, ISBN 0201571692
- Khodorkovsky 2002 Khodorkovsky V. V.: *On Normalization of Relations in Databases*, Programming and Computer Software 28 (1), 41–52, January 2002, Nauka Interperiodica.
- Kotonya 1999 Kotonya G. – Sommerville I.: *Requirements Engineering – Processes and Techniques*, 1999, J. Wiley and Sons
- Kroha 1995 Kroha P.: *Objects and Databases*, McGraw Hill, London 1995, ISBN 0-07-707790-3.
- Lacko 1996 Lacko B.: *Komparativní analýza strukturovaného a objektově orientovaného přístupu*, sborník konference OBJEKTY 1996. ČZU Praha 1996 str.69-76
- Larsson 2002 Larsson M., Crncovic I.: *Building Reliable Component-Based Software Systems*, Artech House 2002, ISBN 1-58053-327-2
- Loomis 1994 Loomis M.: *ODBMS – Hitting the Relational Wall*, pp 56-60, JOOP January 1994
- Loomis et al. 1994 Loomis M., Chaundri A.: *Object Databases in Practice*, Prentice Hall 1994, ISBN 013899725X
- Martin et al. 1995 Martin James, Odell James J.: *Object-Oriented Methods – A Foundation*, Prentice Hall 1995, ISBN 0-13-63856-2
- Molhanec 2004 Molhanec M.: *Několik poznámek k porozumění objektového paradigmatu*, ve sborníku konference Objekty 2004

- Nassi, Shneiderman 1973 I. Nassi , B. Shneiderman, *Flowchart techniques for structured programming*, ACM SIGPLAN Notices, v. 8 n. 8, p. 12–26, August 1973
- Nierstrasz et al. 1990 Nierstrasz Oscar, Papathomas Michael: *Viewing Objects as Paterns of Communicating Agents*, pp 255–266 , in Object Management 1990, Centre Universitaire d' Informatique – Universite de Geneve, also at <ftp://cui.unige.ch/OO-articles/viewingObjectsAsPatterns.ps.Z>
- Nootenboom 2004 Nootenboom Henk Jan: *Nut's – a online column about software design*. <http://www.sum-it.nl/en200239.html>, prosinec 2004
- OMG Webové stránky sdružení Object Management Group. <http://www.omg.org>
- Papazoglou 2000 Papazoglou M., Spaccapietra S., Tari Z: *Advances in Object-Oriented Data Modeling*, MIT Press 2000, ISBN 0-262-16189-3
- Partridge 1996 Partridge C.: *Business Objects – Reengineering for Reuse*, Butterworth-Heinemann 1996, ISBN 07506-2082X
- Rashid 2004 Rashid A.: *Aspect Oriented Database Model*, Springer 2004, ISBN 3-540-00948-5
- Riecken 1994 Riecken D.: *Software Agents*, pp 18–147, Communications of ACM, 37 (1994)
- Rubin et al. 1992 Rubin K.S., Goldberg A.: *Object Behavioural Analysis*, pp 48–62 Communications of the ACM 35(9) 1992
- Rubin et al. 1993 Rubin K.S, Goldberg A.: *Object Behavior Analysis*, pp 48–57 Communications of the ACM, vol 35 no 9. 1993
- Rubin et al. 1994 Rubin Kenneth S, McLaughry Patrick, Pellerini David.: *Modelling Rules using Object Behavior Analysis and Design*, pp 63–68, Object Magazine 4(3) 1994
- Rumbaugh et al. 1991 Rumbaugh James, Blaha Michael, Premerlani William, Eddy Frederic, Lorensen William: *Object-Oriented Modelling and Design*, Prentice Hall 1991, ISBN 0-13-630054-5
- Shiver et al. 1987 Shriver Bruce, Wegner Peter.: *Research Directions in OOP*, MIT Press 1987, ISBN 0-262-19264-0
- Shlaer 1992 Mellor Stephen, Shlaer Sally: *Object Lifecycles: Modeling the World in States*, ISBN 0136299407
- Shoham 1993 Shoham Y.: *Agent Oriented Programming*, pp 51-92, Artificial Inteligence 60 (1993)
- Silbershatz 2004 Silbershatz A et al.: *Database Systems Concepts – 4th Edition*, McGraw Hill 2004, ISBN 0-07-228363-7
- Taylor 1995 Taylor, David A.: *Business Engineering with Object Technology*, John Wiley 1995 ISBN: 0471045217
- UML 2004 *The Unified Modeling Language Documents*, <http://www.uml.org>, prosinec 2004
- Wai 1992 Wai Y. Mok, Yiu-Kai Ng and David W. Embley, *An Improved Nested Normal Form for Use in Object-Oriented Software Systems*. Proceedings of the 2nd International Computer Science Conference: Data and Knowledge Engineering: Theory and Applications, pp. 446-452, Hong Kong, December 1992.

- Wilkie 1993 Wilkie G.: *Object-Oriented Software Engineering*, Addison Wesley 1993, ISBN 0-201-6276-1
- Wirfs-Brock 1990 Rebecca Wirfs-Brock, Brian Wilkerson and Lauren Wiener: *Designing Object-Oriented Software*, Prentice Hall, 1990
- Yonghui et al. 2001 Yonghui Wu, Zhou Aoying: *Research on Normalization Design for Complex Object Schemes*, Info-Tech and Info-Net, vol 5. 101–106, Proceedings of ICII 2001, Beijing.
- Yourdon 1994 Yourdon E.: *Object-Oriented System Design – An Integrated Approach*, Prentice Hall 1994, ISBN 0-13-176892-1
- Yourdon 1995 Yourdon E.: *Mainstream Objects – An Analysis and Design Approach for Business*, Prentice Hall 1995 ISBN 0-13-209156-9
- Zahir et al. 1997 Tari Zahir, Stokes John, Spaccapietra Stefano: *Object Normal Forms and Dependency Constraints for Object-Oriented Schemata*, ACM Transactions on Database Systems 513-569, Vol 22 Issue 4, December 1997.

## Seznam obrázků

Obr. 1: Xerox Dorado, 1976	23
Obr. 2: Tektronix 4404, 1982	24
Obr. 3: Symbol třídy podle UML	33
Obr. 4: Konkrétní třída podle UML	33
Obr. 5: Instance třídy podle UML	33
Obr. 6: Symbol kolekce	34
Obr. 7: Konkrétní kolekce	34
Obr. 8: Dědění podle UML	35
Obr. 9: Skládání podle UML	36
Obr. 10: Diagram UML s kolekcí	37
Obr. 11: Příklad asociace mezi objekty podle UML	46
Obr. 12: Tři možnosti skládání	46
Obr. 13: Workspace	57
Obr. 14: Browser	58
Obr. 15: Debugger	59
Obr. 16: Systém Squeak	60
Obr. 17: Ovládání aplikací v trojrozměrném prostředí	61
Obr. 18: Pohled z jednoho světa to druhého	61
Obr. 19: Spuštění Daskalu z VisualWorks	62
Obr. 20: Tvorba tříd a metod	62
Obr. 21: Manipulace s objekty	63
Obr. 22: Pracovní panel	63
Obr. 23: Panel zdrojového kódu	64
Obr. 24: Panel s diagramem	64
Obr. 25: Dokumentace ve formátu HTML zobrazená webovým prohlížečem	65
Obr. 26: Přehled symbolů pro řízení výpočtu	66
Obr. 27: Zvýrazňování syntaxe	66
Obr. 28: Příklad předdefinovaných zpráv	67

Obr. 29: Start simulace	67
Obr. 30: Průběh simulace	68
Obr. 31: výpočet faktoriálu	68
Obr. 32: Euklidův algoritmus	69
Obr. 33: Počáteční model úlohy s třídami a asociacemi	72
Obr. 34: Datový model úlohy	73
Obr. 35: Gemstone v prostředí VisualWorks	76
Obr. 36: Přátelé – datový model	81
Obr. 37: Obchod s pivem – datový model	85
Obr. 38: Příklad úlohy v nenormalizovaném tvaru	92
Obr. 39: Příklad úlohy v 1ONF	93
Obr. 40: Příklad úlohy v 2ONF	94
Obr. 41: Příklad úlohy v 3ONF	95
Obr. 42: Obchod s pivem – datový model	96
Obr. 43: Obchod s pivem – jiný datový model	97
Obr. 44: Vzor Adaptér	100
Obr. 45: Příklad použití Adaptéru	100
Obr. 46: Vzor Skladba	101
Obr. 47: Vzor Dekorátor	102
Obr. 48: Vzor Stav	103
Obr. 49: Příklad použití Stavů	104
Obr. 50: Vývoj UML	122
Obr. 51: Čtyři vrstvy modelu řídicích systémů podniku nebo organizace	135
Obr. 52: ICT management process podle (Hall et al. 2004)	138
Obr. 53: Čtyři hlavní fáze životního cyklu dle Amblera	139
Obr. 54: Čtyři hlavní fáze životního cyklu v detailu dle Amblera	139
Obr. 55: Obsah dílčího procesu INITIATE-JUSTIFY podle Amblera	140
Obr. 56: Obsah dílčího procesu CONSTRUCT-GENERALIZE podle Amblera	140
Obr. 57: Příklad nastavení procesu konstrukce v konkrétním projektu z praxe	141
Obr. 58: Postupná transformace objektového modelu v BORMu	148
Obr. 59: Postupná transformace objektového modelu v BORMu – detail	149
Obr. 60: Příklad diagramu popisujícího proces (generováno z Craft.CASE)	155
Obr. 61: Příklad simulace procesu v Craft.CASE – stav před obdržení potvrzení od vedoucího	158
Obr. 62: Příklad simulace procesu v Craft.CASE – stav po obdržení potvrzení od vedoucího	158
Obr. 63: Simulační záznam v Craft.CASE – seznam událostí objektu Referent a Auto	159
Obr. 64: Příklad proměny datového modelu – fáze business modelování	163
Obr. 65: Příklad proměny datového modelu – fáze konceptuálního modelování	164
Obr. 66: Příklad proměny datového modelu – fáze softwarového modelování	165


## Seznam tabulek

Tab. 1: Srovnání datového modelování a programování. . . . .	14
Tab. 2: Objekty v tabulce. . . . .	28
Tab. 3: Kolekce občanů. . . . .	41
Tab. 4: Kolekce občanů po výběru. . . . .	41
Tab. 5: Srovnání jazyků Smalltalk-80 a Smalltalk DB. . . . .	71
Tab. 6: Výsledek výběru. . . . .	83
Tab. 7: Výsledek dotazu. . . . .	83
Tab. 8: Výsledek dotazu. . . . .	83
Tab. 9: Výsledek dotazu. . . . .	89
Tab. 10: Výsledek dotazu. . . . .	89
Tab. 11: Výsledek dotazu. . . . .	89
Tab. 12: Porovnání relačního a objektového datového modelu. . . . .	105
Tab. 13: Informační a řídicí systém. . . . .	135
Tab. 14: Kontrolní seznam k ukončení procesu konstrukce. . . . .	143
Tab. 15: příklad seznamu funkcí informačního systému . . . . .	151
Tab. 16: Příklady scénářů (generováno z Craft.CASE). . . . .	152
Tab. 17: Modelová karta pro auto(generováno z Craft.CASE). . . . .	152
Tab. 18: Modelová karta pro vedoucího (generováno z Craft.CASE). . . . .	153
Tab. 19: Pojmy diagramu ORD. . . . .	153



**Vojtěch Meruňka**

# D a t o v é **modelování**

Nakladatelsví: ©Alfa Publishing, s. r. o.,  
U garáží 1436/6, 170 00 Praha 7 → Rok prv-  
ního vydání: 2006 → Návrh obálky: Aleš  
Leznar a Bohumil Janda → Sazba obálky:  
Darina Lepišová → Návrh sazby: Kateřina  
Šlehoferová a Bohumil Janda → Sazba:  
 *Pencil design* → Odpovědný redaktor: Pavla  
Berglová → Edice: Management Studium →  
Tisk: Tiskárna Alfa → Doporučená cena:  
Kč 299,- Sk 449,- Eur 16,- → Made in EU →  
[www.alfaknihy.cz](http://www.alfaknihy.cz) → [info@alfaknihy.cz](mailto:info@alfaknihy.cz)  
ISBN 80-86851-54-0



**D**evelopment  
**I**nnovation  
**R**esearch  
**E**rgonomy  
**C**reativity  
**T**raining

## Inovativní a kreativní přístup v součinnosti se zákazníkem při :

- konzultacích a definování business procesů
- návrhu systémů až po implementaci
- ověření přínosů systémů pro zákazníka
- využití metod BORM a eXtreme Programming



**Partnerství CINCOM (VisualWorks) a Gemstone (Gemstone/S)**

**Vývoj projektů v moderních prostředích (Smalltalk, Erlang...)**

**Vývoj aplikací Web 2.0 s využitím výhod prostředí SeaSide, CSS, AJAX**

# ■ Cincom Smalltalk™

## The Smart Choice for Software Development

- Object oriented development at its best – “everything is an object”
- Modern dynamic platform with latest features for client/server, intranet, web, service oriented and mobile applications
- Based on open industry standards
- Binary portable across MS Windows, Mac, Linux and UNIX
- Enables true agile software development – eXtreme Programming has been developed in Smalltalk
- Runs business critical applications with millions of users worldwide across all industries such as banking/insurance, automobile, electronics, ... – up to 15 years in use

## What's your benefit?

- Easy to learn
- Intuitive to use
- Less errors
- More fun
- Enthusiastic community

## Happy Smalltalking!

E-mail: [infode@cincom.com](mailto:infode@cincom.com)

<http://www.cincom.com>

<http://www.cincom.com/smalltalk>

