NĚKTERÉ ASPEKTY PŘECHODU OD FÁZE ANALÝZY K NÁVRHU PŘI OBJEKTOVĚ ORIENTOVANÉ TVORBĚ SOFTWARU

Vojtěch Merunka

Katedra informačního inženýrství PEF ČZU v Praze, merunka@pef.czu.cz

Motto: Dědění je dobrý sluha, ale zlý pán

Abstract

Skills of move from the analysis stage to the design stage are very important in the software system development process. In current object-oriented approaches, howewer, there exist problems caused by maltreatment of object concepts. Some of them related to the object inheritance and object classes are discussed in this paper.

Od světa konceptuálních objektů k softwarovým objektům

Během tvorby systému je třeba jeho konceptuální model postupně transformovat do takové podoby, která je nezbytná pro fyzickou realizaci systému v podobě programu v daném programovacím jazyce. Právě dotvoření modelu do takové úrovně podrobností, které již odpovídají výrazovým prostředkům použitého prostředí (programovací jazyk, databáze, …), lze považovat za okamžik ukončení objektově orientovaného návrhu a možné zahájení implementace.

Ve většině metodik a nástrojů včetně UML se „konceptuální“ a „softwarové“ objekty příliš nerozlišují a používají se pro ně také stejné diagramy, ale jsou i jiné postupy, kde tomu tak není [1,2,3]. Ve fázi analýzy totiž potřebujeme co nejlépe poznat zadání a tam mohou být implementační podrobnosti na obtíž. A naopak ve fázi návrhu se potřebujeme zaměřit na realizaci výstupů z analýzy, ale nepotřebujeme již znát některé aspekty modelované reality. Diskutovaný přechod lze proto popsat pomocí následujících kroků:

1. Přeměna hierarchie typů na hierarchii dědění mezi třídami a případná transformace vícenásobné dědičnosti na jednoduchou.
2. Doplnění tříd o atributy a metody, které umožní realizovat stavy a přechody (*současné objektové programovací jazyky se stavy a přechody přímo nepracují*[1]).
3. Využití techniky strukturálních transformací pro modifikaci, optimalizaci a kontrolu proveditelnosti modelu v použitém implementačním prostředí.
4. Pokud to cílové prostředí vyžaduje, tak nahradit vazby závislosti (*např. jazyk Smalltalk závislost podporuje, ale C++ ne*).
5. Pokud to cílové prostředí vyžaduje, tak nahradit vazby delegování (*ve většině objektových programovacích jazyků totiž nelze posílat jednotlivé zprávy jako paralelní procesy*).
6. Využití návrhových vzorů.
7. Napojení na případné struktury v existujících systémech.

[1] Výjimkou je jazyk Eiffel a dialekty CLOSu a Smalltalku.

Dva způsoby jak implementovat novou třídu do systému

O dědění se v kuchařkách objektového programování praví, že slouží k tvorbě nových typů pomocí již existujících v systému. Typem se v tomto kontextu rozumí jeho implementace pomocí třídy objektů. Způsoby, jak může vzniknout v objektovém systému nová třída, však jsou dva, což nejlépe ukáže následující příklad.

Mějme návrh třídy objektů, která implementuje datový typ FIFO - fronta (Queue) pomocí znovupoužité třídy OrderedCollection. Třída OrderedCollection je standardní součástí knihovny většiny objektových systému (ve Visual Basicu to je např. podobný typ Collection, v C++ to je CObList atd.) - v našem příkladě použijeme názvy tříd a metod ze Smalltalku.

Objekty této nové třídy Queue budou muset provádět metody size (zjištění počtu prvků ve frontě), isEmpty (test prázdnosti), push: aValue (přidání na konec fronty) a pop (výběr prvku ze začátku fronty). Jak již bylo řečeno, tak objekty třídy OrderedCollection obsahují ve svých metodách dostatečné množství operací potřebných pro vytvoření naší fronty, a proto je použijeme.

Využijeme-li tedy implementaci děděním od třídy OrderedColection, tak metody pop (vyber jeden ze začátku) a push: anObject (přidej jeden na konec) se sice musí napsat nově, ale protože ve svých kódech můžou zužitkovat děděné metody removeFirst a addLast: anObject, tak je jejich implementace velmi snadná. Metody size a isEmpty jsou do instančních metod třídy Queue přímo děděny. Jak je vidět, tak implementace nové třídy objektů pomocí dědění je jednoduchá a elegantní. Má však nevýhodu. Instance třídy Queue totiž dědí od třídy OrderedCollection kromě metod i všechny ostatní metody třídy OrderedCollection, tedy i například addFirst: anObject, removeLast, což by mohlo způsobit například při týmové tvorbě programů nedovolené zacházení s nově naprogramovaným objektem (ve frontě by se mohlo předbíhat, šlo by odstraňovat zevnitř fronty, …). Jednou z možností je nežádoucí děděné metody ošetřit stíněním novými metodami podle následujícího příkladu:

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

Jinou v praxi mnohem používanější možností je zásah do třídy OrderedCollection a prohlášení nežádoucích metod za metody privátní, které se nedědí, což ale může narušit využití třídy OrderedCollection v jiných částech systému.

Dědění naštěstí není jedinou možností, jak v systému sestrojit nový typ objektu pomocí existujícího. Tento druhý způsob spočívá v tom, že každá instance naší nové třídy bude obsahovat jako svoji část (= proměnnou) jednu instanci třídy OrderedCollection například se jménem buffer. Tato implementace je bezpečná v tom, že všechny ostatní nepoužité metody z rozhraní vnitřního objektu jsou navenek **zakryté**, protože o práci s celým objektem rozhoduje pouze jeho vlastní rozhraní bez ohledu na vlastnosti rozhraní v něm uschovaných objektů (částí). Máme tedy zaručeno, že s objekty třídy Queue bude zacházeno pouze předepsaným způsobem, a že s těmito objekty nebude možné provádět pro frontu nepřípustné operace (např. přímý vstup doprostřed fronty, předbíhání ve frontě apod.), i když tyto operace jsou v možnostech instancí třídy OrderedColection, kterou jsme v programu použili. Musíme ale napsat nejen nové metody pop a push: anObject - nyní podobně snadno jako v předchozím

případě, jen s využitím vnitřního objektu buffer, ale i metody size a isEmpty v neposlední řadě i další metodu initialize, která nám nový objekt uvede do žádoucího počátečního stavu s připraveným prázdným objektem buffer uvnitř.

I když je, jak ukazuje náš příklad, implementace nových typů pomocí skládání, v mnohých případech z pohledu OOP čistší, tak je v programátorské praxi málo oblíbená nebo dokonce neznámá, protože vyžaduje více programátorské práce, než je tomu při použití dědění.

Zhruba řečeno, dědění je vhodné ve většině případů, kdy potřebujeme rychleji realizovat nějaké nové objekty, a kdy se nemusíme zabývat možnými důsledky dědění, protože máme jistotu, že s objekty se bude nakládat jen předepsaným způsobem a stačí nám, že software bude správně fungovat. To je například oblast prototypového návrhu. Pokud však potřebujeme sestrojit nové objekty, které budou dále využívány a mají být součástí stabilní a robustní aplikace, je třeba zvážit, zda by využití skládání nebylo vhodnější.

Dědění, hierarchie typů a taxonomie nejsou vždy totéž

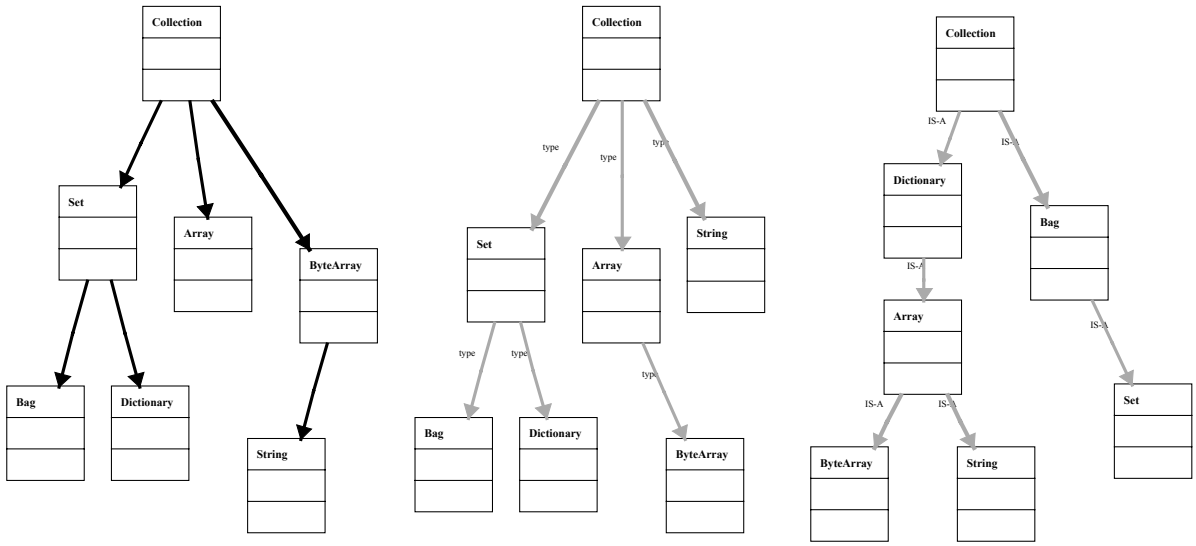
Ukázali jsme si, že nové typy se v objektových systémech realizují pomocí tříd, přičemž ale novou třídu do systému lze vyrobit nejen pomocí dědění, ale i skládáním. Z toho ale vyplývá, že hierarchie dědění a hierarchie typů v jednom systému nemusí vždy znamenat totéž. Navíc při používání dědičnosti máme v programovacích jazycích nástroje pro skrývání metod a dat děděných objektů. Na objektovou hierarchii lze proto nahlížet trojím způsobem podle následujících kritérií:

1. *Z pohledu tvůrce* – návrháře nových objektů. Tato hierarchie je **hierarchií dědičnosti**, protože dědičnost je výborným nástrojem pro tvorbu nových tříd (Ale jak jsme si ukázali, není jediným).

2. *Z pohledu uživatele.* Tento pohled lze ještě podrobněji rozdělit:
   2.1. *Z pohledu polymorfismu* – pohled aplikačního programátora, který potřebuje objekty ve svém systému použít, ale jejich tvorbou se ještě nezabývá. Objekty na nižších úrovních hierarchie potom musejí být schopny dostávat stejné zprávy, jako objekty vyšších úrovní. Právě tato hierarchie je **hierarchie typů**.
   2.2. *Z pohledu aplikační domény* – pohled analytika. Instance tříd na nižších úrovních potom musejí být prvky stejné domény, kam patří instance tříd nadřazené třídy. To znamená, že doména nižší úrovně je podmnožinou domény vyšší úrovně. Tato hierarchie je anglicky označována jako IS-A, česky ji můžeme přeložit „**je jako**“ (nebo „**patří k**“). Od hierarchie typů se může v konkrétních případech lišit proto, že se nezabývá jen chováním objektů na rozhraní, ale objektem celkově.

U jednoduchých úloh je samozřejmě pravda, že uvedené tři hierarchie jsou totožné. Proto se tímto faktem programátorské kuchařky příliš nezabývají. U komplexnějších úloh však toto tvrzení neplatí a to především při návrhu systémových knihoven, které se opakovaně znovupoužívají při tvorbě konkrétních systémů. Pěkným příkladem je následující ukázka:

DĚDĚNÍ, HIERARCHIE TYPŮ A VAZBA IS-A



Na obrázku je (zleva doprava) hierarchie dědění, hierarchie typů a hierarchie „je jako“ pro část systémové knihovny jazyka Smalltalk týkající se sad (collections) objektů. Jsou to následující třídy:

- **Collection** (česky „sada“). To je abstraktní třída, ze které jsou odvozovány jednotlivé konkrétní třídy. Společnou vlastností všech těchto je objektů je schopnost obsahovat jako svoje data další objekty. Sady jsou tedy schránky na objekty.
- **Dictionary** (česky „slovník“). Slovník je taková sada, kde každá hodnota v ní uložená má přiřazenou jinou hodnotu (takže dohromady tvoří dvojici), která slouží jako přístupový klíč k dané hodnotě. Slovníky můžeme použít opravdu jako slovníky pro jednoduché překlady z jednoho jazyka do druhého. Dalším často uváděným příkladem použití objektových slovníků je například jejich použití pro telefonní seznam – klíčem jsou jména osob a hodnotami s klíči spojenými jsou telefonní čísla.
- **Array** (česky „pole“). Pole lze jednoduše popsat jako slovník, kde klíče hodnot mohou nabývat pouze přirozených čísel 1 až velikost pole. Na hodnoty pole se přistupuje tedy také jakoby pomocí klíčů.
- **ByteAray** (česky „bajtové pole“). Jedná se o takové pole, kde povolený okruh hodnot je omezen na celá čísla v intervalu 0 až 255. Ostatní vlastnosti se nemění.
- **String** (česky „řetězec znaků“). Na řetězec znaků lze nahlížet jako na pole (Array), kde povolený okruh hodnot je omezen na znaky.
- **Bag** (česky „pytel“). Bag je taková sada, do které lze objekty ukládat nebo z ní vybírat. Hodnoty uvnitř pytle ale nejsou přístupné pomocí klíčů, ale pouze skrze svoji hodnotu – pokud hodnotu objektu neznám, nemohu jej odebrat.
- **Set** (česky „množina“). Množina je taková sada typu pytel, do které lze stejnou hodnotu vložit jen jednou. Pokud množina hodnotu již obsahuje, tak další vložení stejné hodnoty

je ignorováno na rozdíl od výše uvedeného pytle, který násobné výskyty stejné hodnoty dovoluje. Objekty množina svojí funkčností odpovídají matematickému chápání množin. Proto se tak jmenují.

Uvedený popis tříd na obrázku sleduje pohled analytika – hierarchii „je jako“ (IS-A). Pokud bychom ale popis soustředili na popis funkčnosti – rozhraní objektů, které je vymezeno okruhem přípustných zpráv, dojdeme k poněkud odlišné **hierarchii typů**. například slovníky mohou dostávat stejné zprávy jako množiny a lze tedy na ně nahlížet jako na podtypy množin. A naopak s řetězci znaků se pracuje značně odlišným způsobem než s poli, takže je můžeme považovat za typ, který s poli souvisí jen málo.

A do třetice hierarchie dědění, která je podstatná pro programovou realizaci uvedených tříd, je také trochu jiná. Řetězce znaků je výhodné implementovat děděním z bajtových polí a přidáním či pozměněním potřebných metod. Naopak pole a bajtová pole se implementačně dost liší, protože obyčejná pole se v paměti realizují jako pole odkazů na objekty kdežto bajtová pole jsou jednoduché úseky paměti. Dědění mezi poli a bajtovými poli proto užitečné není.

Jak bylo ukázáno, problematika typů a tříd a jejich vztah k dědění je složitá. Při návrhu systému je proto nejlepší dědění odsunout na pozdější fáze a zacházet s ním jen jako s implementačním nástrojem. Nejprve je třeba na úrovni objektů reálného světa rozpoznat hierarchii „je jako“ (is-a), potom ji upřesnit vymezením příslušných typů pro konceptuální objekty a až nakonec přemýšlet o optimální implementaci typů pomocí dědění softwarových objektů. Ve většině programovacích jazyků lze proto nalézt prostředky (které se ale bohužel velmi málo nebo nesprávně používají) pro oddělení typů a tříd.

Úspora za každou cenu

Dědění se často mylně považuje za povinný nástroj, který je nutně nedílnou součástí každého objektového systému. Objektový program, který neobsahuje dědění, je automaticky považován za podezřelý či dokonce za špatný. Výsledkem jsou křečovité snahy programátorů najít podobnosti mezi objekty za každou cenu a dědit nejrůznější bizarnosti.

Nesprávné použití dědění si ukážeme na příkladě, který v nejrůznějších variantách čtenáři jistě dobře znají. Mějme za úkol sestavit aplikaci pro evidenci knih v knihovně. Lze si představit, že analýza nás dovede ke třídám **Kniha** (popis knihy s názvem, autorem, ISBN, …), **ExemplářKnihy** (konkrétní výtisk s evidenčním číslem, který se půjčuje a vrací), **Čtenář** (ten, kdo si exemplář půjčuje a vrací) a **Knihovna** (systém jako celek, který spravuje knížky a čtenáře). V takovém systému je několik vazeb skládání (např. mezi čtenářem a exemplářem kvůli uchování informace o výpůjčce, …), ale uvedené čtyři třídy jsou mezi sebou typově odlišné. Například kniha není nikdy čtenářem, čtenář není nikdy knihovnou, atd. Celá řada programátorů je však tímto faktem vyvedena z míry a ve snaze učinit program „řádně“ objektovým zoufale hledá, co by se dalo dědit, a například najde podobné atributy u čtenáře, knihy a knihovny a vyrobí jim společného předka. Takto lze například extrahovat atribut **název** a dědit ho jako jméno čtenáře, název knihy, označení celé knihovny. Ještě přísernější je dědění mezi třídami navzájem; například **Kniha** může dědit ze **Čtenáře**, koneckonců kniha má vždy svého autora, a když tedy pomocí dědění rozšíříme osobu o ISBN, rok vydání a název, máme **Knihu** se vším všudy a ještě jsme v programu ušetřili dalších pár řádek zdrojového kódu v našem oblíbeném C++, o čemž přeci OOP je, nebo ne?

Největší chybou takových přístupů není to, že programy nefungují. To není pravda – pokud jsou používány v souladu se zadáním, tak mohou fungovat docela dobře (tím obtížněji se pak vysvětluje, že program je špatný). Problém je v tom, že nadměrné užívání dědění kromě požadovaných funkcí do objektů přidává i funkce, které přesahují zadání a které mohou být zneužity a také komplikují pozdější údržbu či modifikace.

Vícenásobná dědičnost

S vícenásobnou dědičností je třeba zacházet velmi opatrně. Na úrovni hierarchie typů ve fázi konceptuálního návrhu je sestavování nových typů z více existujících současně správné. To ale neznamená, že v implementaci budeme otrocky podle hierarchie typů dědit třídy mezi sebou. Jen málo programovacích jazyků dovoluje vícenásobné dědění – je to například CLOS a Eiffel nebo některé dialekty Smalltalku. Z běžně používaných jazyků dovoluje vícenásobné dědění jen C++, ale při bližším prozkoumání vychází najevo, že objekty tříd, která vícenásobně dědí, jsou ve skutečnosti objekty poskládané z příslušných částí. V C++ lze proto na vícenásobné dědění nahlížet také jako na syntaktickou zkratku pro skládání objektů, které přináší úsporu zdrojového kódu a těm, kteří ji ovládají, dodává pocit magie.

Třídy versus množiny objektů

Na pojem třídy se může nahlížet dvojím způsobem:

1. Třída je **realizace objektového typu**; ve třídě uchováváme popis struktury objektů tohoto typu a množinu jeho operací/metod. V čistě objektových jazycích (např. CLOS, Smalltalk) se s třídou na rozdíl od hybridních/méně objektově orientovaných jazyků (např. C++, VB, Object Pascal v Delphi a Java) může pracovat jako s objektem se vším všudy; diskutovaný popis struktury spolu s operacemi jsou jeho data, se kterými lze za chodu programu pracovat, měnit je, doplňovat přidávat nebo mazat.
2. Úplně jiný pohled na třídu je **chápání třídy jako množiny**, která jako prvky obsahuje objekty, které dané třídě náleží jako její instance. Tento pohled na třídu je více abstraktní a ztotožňuje pojem třídy s pojmem množiny - domény, která vymezuje výskyt objektů daného typu.

Naneštěstí se v běžných metodách, jako je UML, oba způsoby nazírání na pojem třídy mísí dohromady a nerozlišuje se mezi třídou jako objektem sám o sobě a mezi třídou jako pomyslnou množinou tvořenou všemi objekty, které do dané třídy patří.

Nesprávné splynutí diskutovaných dvou pojmů vede potom k tomu, že se všechny množiny objektů v modelovaném systému modelují jako samostatné třídy. Problém si ukážeme na příkladu.

V systému, kde jsou objekty třídy **Osoba**, potřebujeme vymezit z nějakého důvodu dvě množiny osob: **Zákazníky** a **Dodavatele**. Pro analytika, který ovládá např. jen UML a nezná do hloubky problematiku OOP, se nabízí jednoduché řešení: sestrojit dvě nové podtřídy **Zákazník** a **Dodavatel**, které dědí ze třídy **Osoba**. Toto řešení však není optimální minimálně ze dvou důvodů:

1. Pokud nepotřebujeme v nových podtřídách nové atributy a operace, tak se nové třídy zavádí jen kvůli potřebě zanesení do modelu skutečnosti, že některé osoby jsou a jiné nejsou zákazníky a dodavateli.
2. Přináší potíže v případě, kdy se jedna a ta samá osoba stane zákazníkem a dříve byla dodavatelem a dokonce nebo kdy bude současně zákazníkem i dodavatelem.

Splývání pojmu třída a množina je možné z důvodu jednoduchosti a srozumitelnosti modelovat jednotně na úrovni hierarchie „je jako“ (is-a) pro objekty reálného světa. Pro konceptuální objekty, kde se tato hierarchie převádí na hierarchii typů, jsou však množiny potřeba a pro objekty softwarové, kde hierarchie typů vede k hierarchii dědění je to už nezbytnou nutností.

Podtřídy nebo instance

Dalším problémem, který souvisí s děděním a týká se objektového modelování, je otázka, do jaké míry je účelné vytvářet nové a nové podtřídy a kdy se rozhodnout ukončit vlnobití nových tříd a různost objektů realizovat pomocí různých hodnot atributů ve třídách. Problém si popišme na příkladu:

Mějme část knihovního informačního systému, která se týká časopisů. Časopisy lze dělit na odborné, vědecké, populárně naučné, … - vytvoříme tedy jednotlivé třídy. Odborné časopisy lze dělit na počítačové, lékařské, ekonomické, … - takže další podtřídy. Počítačové časopisy jsou Softwarové noviny, Bajt, Computer World, … . Softwarové noviny lze dělit na ročník 1991, 92, … a tak dále až na jednotlivé výtisky. Kde ale ukončíme tvorbu nových tříd a budeme vzájemné odlišnosti zapisovat jen do atributů objektů stejné třídy? Program lze sestavit jak jedním extrémem: jedna třída časopis a všechny výtisky různých časopisů jsou její instance, i druhým extrémem: Každé vydání každého časopisu je samostatná třída, která má tolik instancí, kolik čísel daného vydání v knihovně máme. Obojí lze naprogramovat tak, že bude dobře fungovat.

Je potřeba přiznat, že neexistuje jednoznačné pravidlo, jak se v takovém případě rozhodnout. Hlavním kritériem tu je analýza **chování a prezentace objektů navenek**. Pokud nedokážeme najít odlišnosti v chování, operacích a vnějších atributech, tak nemá smysl model štěpit na více tříd. A nezapomínat na to, že stále ještě můžeme používat i množiny objektů – viz. předchozí kapitola. Dalším podkladem pro rozhodování jsou ještě

- zohlednění potenciální databázové realizace a
- potřeba usnadnit či alespoň nezkomplikovat možné úpravy a rozšiřování systému.

Náš příklad lze proto například vyřešit tak, že třídy budou končit na úrovni počítačového, lékařského, … časopisu. Jednotlivé výtisky jednotlivých časopisů budou potom objekty lišící se svými atributy a náležející do různých množin.

Závěr

Tvorba objektových konceptuálních modelů během tvorby systému má svoje vlastní pravidla a okruh pojmů, které nejsou vždy totožné s pravidly a pojmy objektového programování, jak je známe v objektových programovacích jazycích. Kdyby tomu tak nebylo, tak by nám objektové diagramy sloužily pouze ke grafickému zobrazení cílového programového kódu a ne k analýze a návrhu.

Dědičnost je jednou z mála výlučných vlastností objektově orientovaného přístupu. Nenacházíme jej v technologiích, které objektovému přístup předcházely. Proto je dědění tak oblíbené v řadě příruček a při laických výkladech, co to je objektový přístup. Samozřejmě je pravda, že dědění je velmi mocný nástroj při tvorbě softwaru objektovým způsobem. Jde ale o to si uvědomit, kde má svoje místo, že přináší s sebou určité problémy a že není všelék.

Literatura

1. 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
2. Merunka, Vojtěch; Polák, Jiří: Experience of Substantial and Detailed Process Mapping as First Phase of Reengineering Requirement Definition in a Gas Distribution Company, in Proceedings of Fourth International Conference on Requirements Engineering ICRE 2001, Toronto 2001. IEEE Computer Society Press.
3. Merunka, Vojtěch; Polák, Jiří; Louis, Garcia Rivas: BORM – Business Object Relation Modeling, in Proceedings of WOON 2001 Conference – White Object-Oriented Nights, St. Petersburg 2001.
4. MetaCASE Ltd: Domain Specific Modelling – White Paper, http://www.metacase.com
5. Metodologie SDL, Telelogic Corp. http://www.telelogic.com
6. Taylor, David A.: Business Engineering with Object Technology, John Wiley 1995 ISBN: 0471045217
7. Yourdon E.: Mainstream Objects - An Analysis and Design Approach for Business, Prentice Hall 1995 ISBN 0-13-209156-9
8. Goldberg Adele, Kenneth Rubin S.: Succeeding with Objects - Decision Frameworks for Project Management, Addison Wesley 1995, ISBN 0-201-62878-3
9. The Unified Modeling Language Documents, Rational Software Corporation, http://www.rational.com