ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE
Provozně ekonomická fakulta
Katedra informačního inženýrství

Obor informační management

Vývoj objektově orientovaných informačních systémů pomocí metody postupných transformací

Doktorská disertační práce

Doktorand: **Ing. Marek Pícka**
Školitel: Prof. Ing. Ivan Vrana, DrSc.

2010

Děkuji svému školiteli Prof. Ing. Ivanu Vranovi, DrSc. za vedení této práce.

Obsah

Seznam obrázků a tabulek

Obrázky

Tabulky

Zdrojový kód

1 Úvod

1.1 Motivace

V dnešní době jsou velké informační systémy typicky navrhovány pomocí rozsáhlých a složitých objektově-orientovaných metodik. Výstupem těchto metodik je velké množství dokumentů, převážně diagramů a textových dokumentů v kterých je zachycen model informačního systému. Metodiky nám říkají, jaký artefakt (tj. dokument, diagram či program) kdy a pro co použít a typicky obsahují i metodu, jak správný výstup vytvořit. Tyto metodiky jsou častokrát velmi složité a rozsáhlé – mají mnoho výstupů a častokrát se do formalismu metodiky zamotáme tak, že pro samé dokumenty, diagramy, schůzky a termíny zanedbáváme ten hlavní úkol – dodat software včas, za daný rozpočet a s požadovanou funkcionalitou. Udržování konzistence mezi těmi všemi dokumenty, diagramy a zdrojovým kódem je samostatný, náročný proces (viz kapitola 4), který se nakonec z kapacitních a časových[*] důvodů buď neprovádí vůbec nebo se ošidí.. Přece to pro prvotní dodání softwaru, a (to je nejdůležitější) pro proplacení projektu, není důležité. To se nám však pravděpodobně vymstí. Patrně sice nezapomeneme implementovat nějakou požadovanou funkčnost (to si jistě zákazník před platbou zkontroluje) , ale při budoucí údržbě informačního systému nastanou problémy. Dokumentace bude nepřesná[†] a neposkytne nám patřičnou oporu. Velké problémy také vzniknou při následných změnách systému. Zákazník změní nějaký požadavek a my nebudeme vědět, jak se tato změna promítne do našeho systému – nebudeme mít žádné spojení mezi kódem a požadavky. Ale v současnosti nevadí[‡] a stejně zákazník počítá s tím, že budoucí úpravy budou drahé (v roce 2004 podle Standish Group 71% projektů neúspěšných nebo s překročenými náklady viz Obr. 2).

S dalšími problémy se potýkají, v současnosti moderní, méně formální přístupy určené pro tvorbu projektů menšího rozsahu. Ty mají sice výhodu v krátkých iteracích vývoje a tím ve větší zpětné vazbě od zákazníka, ale tyto projekty jsou typicky dělány „programátory“ s žádnou nebo minimální analýzou. Typicky se přechází rovnou od požadavků přímo k implementaci (kód je přeci nejpřesnější dokumentace), nebo se použije minimální podmnožina modelovacího jazyka UML –

[*] Času je přeci vždy málo.
[†] Špatná dokumentace je horší než žádná [Wiegers 2003].
[‡] Psáno před finanční krizí.

nejlépe diagramu tříd vygenerovaného z CASE nástroje. „Programátor“ je spokojen a zákazník dostane svou dokumentaci. Častokrát se tento postup maskuje tvrzením, že se používá velice moderní metoda návrhu softwaru – extrémní programování [Beck 2002].

Špatný přístup* je častokrát používán i v knihách a učebnicích ([Page-Jones 2000] nebo [Schmuller 1999]). Tam je objektově orientovaný přístup k tvorbě softwaru vykládán pouze jako syntaxe a sémantika jazyka UML. Málokdy jsou zdůrazněny jednotlivé závislosti mezi diagramy a už vůbec není vyloženo, že existuje něco jako požadavek a že by mělo být verifikovatelné, že jsme ho splnili.

Typickým výsledkem těchto přístupů je, že vidíme, (v lepším případě) celý projekt jako nesouvisející výstupy – zadání, nějaké UML diagramy a kód. Ztrácíme tak konzistenci mezi jednotlivými částmi projektu a častokrát tak dochází k nesplnění požadavků, či (v horším případě) k vypracování nepotřebných funkčností.

Jedním z řešení je samozřejmě důsledně provádět vývoj a správu požadavků (Requirement Engineering). Ta nám poskytuje nástroje, jako je např. matice dohledatelnosti požadavků (Requirement Traceability Matrix – viz část 4.5.4) . Konstrukce této matice je však náročná na přesnost a pečlivost† a dodatečně je jí časově náročné zkonstruovat či uvést do souvislosti s realitou.

Já v této práci navrhuji řešení, jak se na model informačního systému nedívat jako na soubor vzájemně málo souvisejících artefaktů (psané dokumentace, diagramů , zdrojového kódu atd.), ale dívat se na tento model jako na soubor vzájemně provázaných prvků, které postupně jsou do modelu přidávány. Každý nový prvek přidaný do modelu, musí mít důvod své existence v prvcích již v modelu obsažených (třeba nějakém požadavku). Když si u každého prvku v modelu zaznamenáme, z jakých prvků vzniknul, tak dosáhneme stav, že u něj budeme vědět z jakých důvodů (tedy požadavků) se do modelu dostal.

* Kritika tohoto přístupu například [Molhanec 2007]
† Musí se konstruovat průběžně.

1.2 Cíle práce

Hlavním cílem této práce je navrhnout metodu návrhu informačních systémů, která zajistí vzájemnou dohledalnost jejích prvků. Tj. u každého prvku modelu budeme schopni zjistit, na základě kterých prvků vzniknul.

Vedlejší cíle vedoucí k dosažení tohoto hlavního cíle:

1. Navrhnout základní terminologii, postuláty a vlastnosti této metody.
2. Typizovat způsoby vytváření nových prvků v modelu informačního systému.
3. Navrhnout modely této metody a jejich grafickou prezentaci.
4. Prozkoumat vlastnosti a použití této metody.
5. Navrhnou způsob implementace této metody do CASE nástrojů a
6. Aplikovat tuto metodu na vybranou metodu návrhu informačního systému.

1.3 Metodika

Tato disertační práce vychází z mého dlouhodobého zájmu (jak teoretického, tak praktického) o zlepšení metod návrhu softwaru. Postup řešení této disertační práce byl následující:

1. Provedl jsem rešerši s tímto obsahem:
    a. Analýza používaných metod návrhu informačních systémů.
    b. Analýza problémů při návrhu informačních systémů a způsoby jejich dnešního řešení (kapitoly 2 a 3).
    c. Požadavky – typy požadavků a vlastnosti požadavků, se zaměřením na jejich dohledatelnost.
    d. Transformace – obecná definice a vlastnosti transformací a jejich transformací při návrhu softwarových systémů – MDE (Model Driven Engineering) a MDA (Model Driven Architecture).
2. Na základě rešerše a zkušeností s postupem práce tvůrce informačních systému formuloval základní principy metody postupných transformací. Tyto principy jsou shrnuty v postulátech.

3. Zavedl jsem pojmy metody postupných transformací.
4. Z postulátů jsem odvodil vlastnosti metody postupných transformací prvků a navrhl jsem možné využití těchto vlastností při tvorbě informačního systému.
5. Navrhnul jsem metamodel metody postupných transformací a jeho grafickou reprezentaci (diagram záznamu transformací).
6. Zobecnil jsem metodu postupných transformací prvků, tím že jsem si zavedl její model. Tento model obsahuje pojmy (tj. typy prvků) a přechody (tj. možné typy transformací*, také předpisy transformací). Navrhnul jsem metamodel přechodů mezi pojmy a na základě něj jeho grafickou podobu (diagram).
7. Navrhnul jsem možný popis těchto předpisů transformací a jejich použití
8. V praktické ukázce použití této metody jsem vytvořil:
    a. Algoritmy generující matici dohledatelnosti požadavků z modelu postupných transformací prvků.
    b. Navrhnul jsem způsoby implementace této metody do CASE nástrojů.
    c. Provedl jsem analýzu metody BORM a vytvořil její model přechodů mezi pojmy a tyto možné transformace jsem popsal.

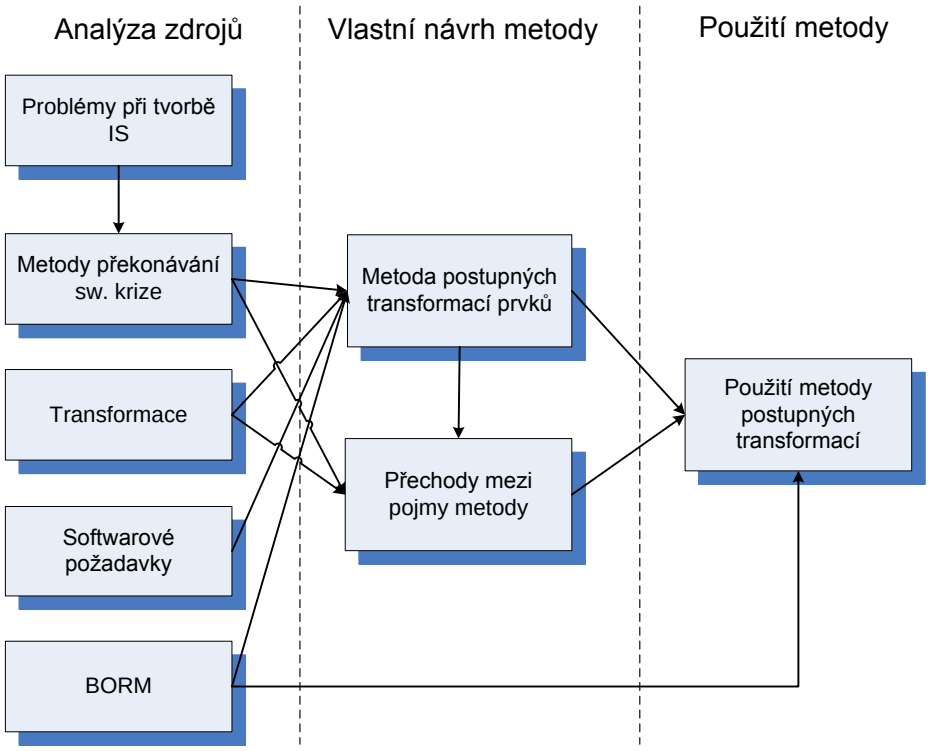
Části této disertační práce byly publikovány a diskutovány na mnoha domácích (cca 8 článků mezi lety 2005-9) i zahraničních konferencích [Pícka 2006c, 2008b] a v odborném časopise [Pícka 2008a].

1.4 Struktura práce

Disertační práce je členěna v souladu s postupem řešení stanovené problematiky. Jednotlivé kapitoly práce lze rozdělit do těchto tří skupin:

1. analýza dostupných zdrojů – literární rešerše (vstupy práce)
2. návrh metody postupných transformací (jádro řešení práce)
3. implementace a aplikace navržené metody (možnosti využití práce)

* Možné typy transformací vycházejí z metody analýzy a návrhu informačního systému.



Obr. 1 – Struktura řešení disertační práce

1.4.1 Analýza dostupných zdrojů

V této části práce je shrnut současný stav poznání v oblastech, které jsou využity při analýze a následné konstrukci vlastního řešení. Přehled mapuje problémy při tvorbě informačních systémů a metody jejich řešení . Dále se zaměřuje na první fáze konstrukce informačního systému – tvorbu požadavků a jejich souvislost s vytvářeným modelem systému. Nakonec jsou popsány transformace a jejich použití (jako MDE) při tvorbě softwarových systémů.

Kapitola „**Problémy při tvorbě IS**“ ukazuje na problémy se kterými se potkáváme při tvorbě informačních systémů. Na základě příkladů a studií ukazuje, že při vytváření informačních systémů se častokrát setkáváme s překročením rozpočetu, zpožděním proti plánu, nedostatečnou kvalitou a nedodržením požadavků.

Kapitola „**Metody překonávání softwarové krize**“ popisuje historii i současnost softwarového inženýrství, jako úspěšné metody pro zmírněni následků softwarové krize. V této kapitole jsou uvedeny jednotlivé fáze modely životního cyklu vývoje softwaru i jejich možné modely. Jsou zde uvedeny možná paradigmata vývoje softwaru i rozdělení metodik, z těchto paradigmat

vycházejících. Nakonec se tato kapitola věnuje metamodelování, jako prostředku pro modelování metod a metodik návrhu informačních systémů.

Kapitola „**Softwarové požadavky**“ popisuje softwarové požadavky, jejich typy a vlastnosti. Požadavky jsou základem tvorby jakéhokoliv informačního systému. Pokud jsou požadavky na systém nesprávně formulovány nebo dokonce chybí, tak dochází k velkým časovým i finančním ztrátám při tvorbě softwaru. Dále se věnuje vztahu jednotlivých částí (modelu) systému a požadavků, na základě tyto části vznikly – dohledatelnosti požadavků.

Kapitola „**Transformace**“ popisuje obecné vlastnosti transformací, jejich klasifikaci, charakteristiky i mechanizmy použité k jejich modelování. Dále je v této kapitole popsán jeden ze způsobů použití transformací při tvorbě softwaru – MDE (Model Driven Engineering) a jeden konkrétní přístup, využívající jeho myšlenek – MDA (Model Driven Architecture).

Kapitola „**BORM**“ se věnuje metodě BORM (Business Object Relationship Modeling), která je jednou z moderních metod návrhu informačního systému, zejména jeho procesní části. Tato metoda sloužila jako částečná inspirace k této práci i pro praktickou ukázku metody postupných transformací prvků.

1.4.2 Návrh metody postupných transformací prvků

V této části práce jsou aplikovány poznatky z předchozích kapitol s cílem navrhnout metodu tvorby informačních systému s použitím postupných transformací.

Kapitola „**Metoda postupných transformací prvků**“ – cílem kapitoly je definovat základy metody postupných transformací prvků, která je založena na postupném přidávání nových prvků do již existujícího modelu informačního systému na základě již existujících prvků. Cílem kapitoly je zavést základní pravidla metody postupných transformací, diskutovat její vlastnosti, definovat způsob zachycení provedených transformací a jejího použití při tvorbě modelu systému.

Kapitola „**Přechody mezi koncepty metody**“ spojuje posupné transformace prvků a metody analýzy a návrhu informačních systémů. Tyto metody popisují, jaké všechny možné transformace mezi prvky jsou v jejich rámci přípustné. Kapitola zobecňuje prvky použité, v rámci vytváření modelu informačního systému, do pojmů (neboli typů prvků) a mezi těmito pojmy popisuje typy přípustných transformací. Toto je zachyceno v modelu přechodů mezi pojmy (metody). Dále je v této kapitole navržen popis typů transformací a jejich možné využití.

1.4.3 Implementace a použití metody postupných transformací

V této části jsou ukázány možnosti použití metody postupných transformací prvků při tvorbě informačních systémů.

Kapitola „**Použití metody postupných transformací**“ uvádí příklady použití metody postupných transformací. Jsou zde uvedeny algoritmy generující matice dohledatelnosti požadavků na základě modelu informačního systému vytvořeného pomocí metody postupných transformací prvků. Dále jsou zde probrány a navrženy způsoby implementace této metody do CASE nástrojů. Nakonec je zde uvedeno jak postupovat při vytváření modelu přechodů mezi koncepty metody.

V příloze **„BORM z pohledu metody postupných transformací prvků“** je aplikována tato metoda na příkladu metody BORM – jsou popsány typy transformací, které lze použít při tvorbě modelu informačního systému za použití metody BORM.

2 Problémy při tvorbě IS

Tato kapitola pojednává o problémech se kterými se potkáváme při tvorbě rozsáhlých informačních systémů. Na příkladech a studiích ukazuje na typické problémy při tvorbě informačních systémů - překročení rozpočetu, zpoždění proti plánu, nedostatečnou kvalitou a nedodžování požadavků. Tyto stupňující problémy byly pojmenovány softwarová krize.

2.1 Úvod

V roce 1986 Alfred Spector a David Gifford [Spector 1986] napsali článek porovnávající stavění mostů a vývoj softwaru. Základní myšlenka byla:

> *Mosty se obvykle stavějí včas, za stanovenou cenu a nepadají. Na druhou stranu u softwaru je málokdy dodržen termín i rozpočet. A dokonce téměř vždy selhává*[*].

Hlavním důvodem, že jsou mosty postaveny včas, za dohodnutou cenu a nepadají, je jejich extrémně detailní návrh. Plány mostu jsou dané a stavebník má velmi omezenou možnost je měnit. Na rozdíl od mostů musí softwarové projekty reflektovat rychle se měnící okolní složitý svět. Programové vybavení se musí vyrovnat s neustálou změnou jak technologií tak i požadavků na něj. Detailní a neměnný design není možný.

Dalším důvodem je to, že tvorba softwaru je poměrně mladá disciplína. Se stavěním mostů máme zkušenosti již přes 3000 let. Když most spadne, tak se vytvoří velká komise, která zkoumá příčiny a zveřejní doporučení, aby se to příště neopakovalo. Softwarový průmysl se většinou snaží chyby a selhání bagatelizovat a skrývat. Výrobce softwaru téměř nikdy neručí za ztráty způsobené chybami programů. Podrobné vyšetřování selhání a zveřejnění doporučení se děje je v případech kdy jde o hodně peněz nebo o život [Mellor 1994]. Výsledkem tohoto přístupu je to, že se stejné chyby dělají znovu a znovu.

Výsledkem tohoto neustálého růstu komplexnosti softwarových systémů, neustálého tlaku na jejich změnu a relativní nedospělosti postupů jejich vzniku je jev nazývaný softwarová krize.

[*] Nicméně stavění mostů není také bez problémů. Mnoho mostů je nakonec dražší, staví se delší dobu a dokonce nějaký most občas i spadne.

2.2 Softwarová krize

Pojem softwarová krize byl poprvé použit F. L: Bauerem na konferenci NATO v roce 1968. Termín vešel následně v povědomost zejména díky článku Edsgera Dijkstry „The Humble Programmer“ [Dijkstra 1972], kde popisuje příčiny tohoto problému takto:

> *„Hlavní příčinou softwarové krize byl nárůst výkonu hardware. Jinak řečeno, programování nemělo problémy, dokud neexistovaly počítače. Dokud jsme měli slabé počítače, mělo programování jen snesitelně těžké problémy. Nyní máme gigantické počítače a k nim gigantické problémy se softwarem“.*

Tato softwarová krize byla způsobena tím, že se programy staly tak rozsáhlými a jednotlivci již nebyli schopni obsáhnout celý program. Od té doby, tedy za téměř 40. let vzrostl výkon a tím i možnosti počítačů (a s tím samozřejmě také složitost programů) mnohonásobně – jejich výkon roste víceméně exponenciálním* tempem. Tomu, že jsme vůbec schopni dnes úspěšné řešit rozsáhlé softwarové projekty, vděčíme zavedení inženýrských postupů do procesu tvorby softwaru (viz kapitola 3.2).

Hlavní příznaky softwarové krize jsou:

- Projekty překračují rozpočet.
- Projekty překračují čas.
- Software nemá dostatečnou kvalitu.
- Software neodpovídá požadavkům.
- Projekt není dobře řiditelný a software je obtížně udržovatelný

2.3 (Ne)úspěšnost IT projektů

Pojem softwarová krize byl nejpopulárnější na přelomu 60. a 70. let kdy jsme poprvé narazili na strop tehdy používaných metod. Již mnohokrát jsme se domnívali (většinou v souvislosti s objevením nové technologie či metody), že softwarová krize je již zažehnána, ale její příznaky

* Takzvaný Moorův zákon – v roce 1965 předpověděl spoluzakladatel firmy Intel Gordon E. Moore, že se počet tranzistorů (a tím i výkon) na integrovaném obvodu každé dva roky zvýší dvojnásobně [Moore 1965]. Moore považoval tento odhad za nadnesený, ale tato předpověď platí stále a firna Intel předpokládá, že bude platit cca. do roku 2021, kdy se narazí na možnosti křemíku.

přetrvávají dál. V následující tabulce (Tab. 1) jsou uvedeny příklady ztrát (větších než $50 mil. USD) způsobenými neúspěšnými IT projekty .

Rok	Společnost	Ztráta (v USD)
2004-05	UK Inland Revenue	Softwarová chyba způsobila (neoprávněné) vrácení daňového přeplatku ve výši $3,4 mld.
2004	Avis Europe PLC [UK]	ERP systém zrušen – utraceno $54,5 mil
2004	Ford Motors Co	Nákupní systém zrušen po zavedení – škoda $400 mil
2004	J Saintsburry PLC [UK]	Supply-chain management systém zrušen po zavedení – škoda $527 mil.
2004	Hewlett-Packard	Problémy s ERP – $160 mil.
2003-04	AT&T Wireless	Problémy s upgrade CRM systému – $100 mil.
2002	McDonald's Corp.	Vývoj inovovaného nákupního is zrušen – $170 mil.
2002	CIGNA Corp	Problémy s CRM systémem – $445 mil.
2001	Nike Inc.	Problémy s supply-chain management systémem – škoda $100 mil.
2001	Kmart Corp.	Vývoj Supply-chain management systému zrušen – škoda $130 mil.
1999	Stát Missisippi	Vývoj daňového systému zrušen po utracení $11,2 mil, stát následně utrpěl škodu ve výši $185 mil.
1999	Hershley Foods Corp.	Problémy s ERP – $151 mil.
1998	Snap-on Inc.	Problémy s objednávkovým systémem – $160 mil.
1997	U.S. Internal Revenue Service	Neúspěšná snaha o vylepšení is. výběru daní – $4 mld.
1997	Oxford Health Plans Inc.	Chyba v účetním is vyústila ve ztrátu následkem toho akcie společnosti poklesly o $3,4 mld.

Rok	Společnost	Ztráta (v USD)
1996	Arianespace [Francie]	Chyba v sw. specifikaci – škoda $350 mil. při výbuchu rakety Ariane 5
1994	U.S. Federal Aviation Administration	Pokročilý automatizovaný systém zrušen, utraceno $2,6 mld.
1993	London Stock Exchange [UK]	Zrušen vývoj burzovního is po utraceni $600 mil.
1993	Allstate Insurance Co	Vývoj office-automation systému zrušen – škoda $130 mil.
1993	Greyhound Lines Inc.	Vývoj rezervačního systému zrušen, utraceno $165 mil.

Tab. 1 – Příklady neúspěšných IT projektů [Charette 2005]

Bylo provedeno mnoho studií ukazujících přetrvávající projevy softwarové krize. Některé (další jsou třeba [Evusi 2003], [IT-UK 2003] nebo [Keller 2004]) z nich jsou uvedeny v následujících odstavcích. Z nich byl vytvořen závěr o úspěšnosti softwarových projektů [Cortex]:

- IT projekty jsou spíše neúspěšné než úspěšné
- jen 1 z 5 projektů je plně úspěšný
- čím větší je projekt, tím je větší naděje že selže

2.3.1 CHAOS Report 1995

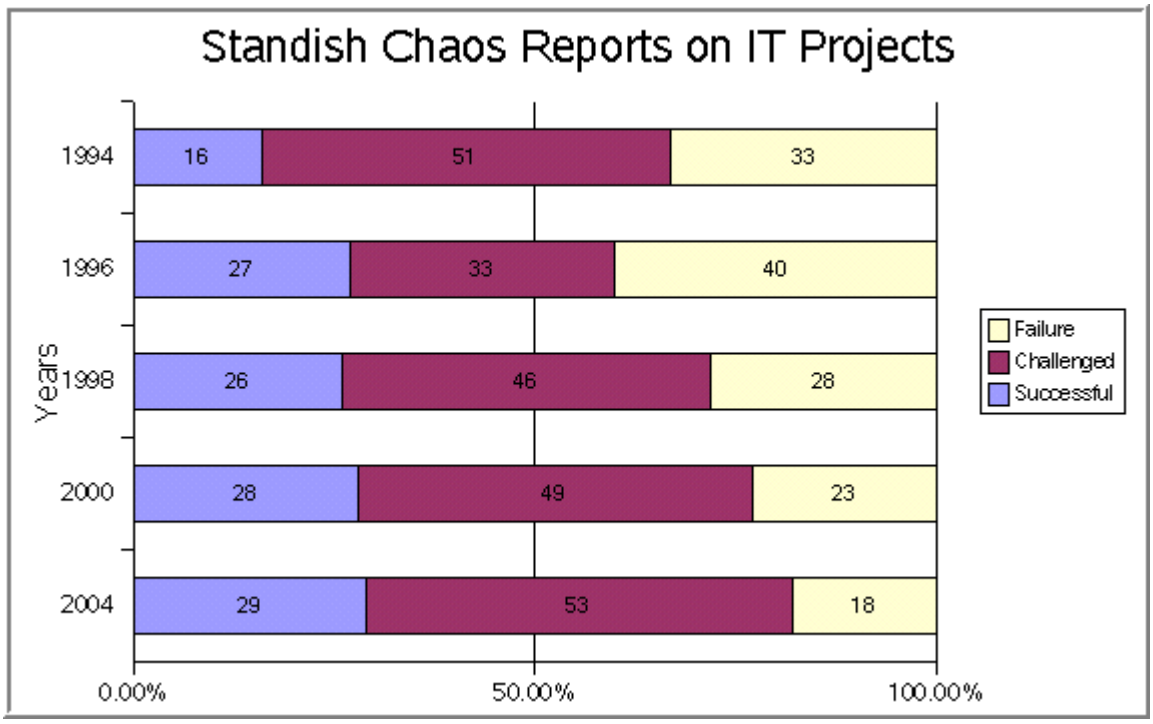
Zpráva CHAOS [Standish 1995], kterou v roce 1995 vydala Standish Group, je asi nejznámější studií o selhání IT projektů. Tato zpráva vznikla na základě 365 účastníků z různých oborů (např. bankovnictví, zdravotnictví, pojištění, akcie, státní správa) provozujících 8380 aplikací. Projekty byly rozděleny do 3 kategorií:

- úspěšný – projekt byl dokončen včas, byl dodržen rozpočet a měl všechny plánované vlastnosti.
- problémový – projekt byl dokončen, ale později, nebo za více peněz, nebo nebyly implementovány všechny vlastnosti.
- neúspěšný – projekt byl zrušen před dokončením.

V roce 1995 31,1% projektů bylo zrušeno před dokončením. 52,7% projektů bylo problémových a průměrně byl překročen rozpočet o 189% a průměrné zpoždění bylo 222%. Na základě těchto čísel Standish Group odhadla, že americké firmy a vláda zaplatily 81 mld. USD za zrušené projekty a zaplatily navíc 59 mld. USD za zpoždění v projektech.

Pouze 16,2% bylo dokončeno včas a za danou cenu. Ve větších společnostech byly čísla ještě horší – pouze 9% projektů bylo dokončeno včas a bez překročení rozpočtu. Projekty dokončené největšími americkými společnostmi měly implementováno v průměru pouze 42% původně předpokládaných vlastností. Menší společnosti na tom byly lépe – dokončeno bylo 78,4% projektu a bylo implementováno 74,2% zamýšlených vlastností.

Tato studie se každoročně opakuje a shrnutí jejích výsledků z let 1994 až 2004 je na Obr. 2. Je možno sledovat zejména snižující se počet neúspěšných projektů (až na 18% v roce 2004). Toto je vysvětlováno zejména zmenšující se velikostí projektů.



Obr. 2 – Závěry studií CHAOS v letech 1994 – 2004 [theopensourcery.com]

2.3.2 Průzkum společnosti Robbins-Gioia

Společnost Robbins-Gioia udělala v roce 2001 průzkum [Robbins 2001] týkající se zkušeností podniků se zaváděním ERP systémů (Enterprise Resource Planning – česky podnikový informační

systém). Bylo osloveno 232 společností různých oborů (informační technologie, komunikace, zdravotnictví, vláda atd.). Celkem 36 % procent společností mělo zkušenosti se zaváděním ERP systémů.

Hlavní závěry byly tyto:

- 51 % se domnívalo, že implementace ERP systému byla neúspěšná.
- 46 % účastníků uvedlo, že dle jejich názoru jejich organizace neumí využít ERP systém k zlepšení své činnosti.

2.3.3 Průzkum organizace The Conference Board

Tento průzkum [Conf 2001] byl proveden v roce 2001. Zúčastnilo se ho 117 společností s implementovaným ERP systémem.

Hlavní závěry byly tyto:

- 34 % bylo velmi spokojeno.
- 58 % bylo částečně spokojeno.
- 8 % bylo nespokojeno.
- 40 % projektů nepřineslo očekávané výsledky do 1 roku.
- Společnosti uvedly, že očekávané přínosy se dostavily o půl roku později než bylo očekáváno.
- Implementace byla o 25 % dražší než byl plán.
- Náklady na následnou podporu byly podceněny o 20 %,

2.3.4 Studie OASIG

Studie [OASIG 1995] byla vytvořena skupinou OASIG (Organizational Aspect of IT – Special Interest Group) v roce 1995. Byl vykonán průzkum u cca 14000 organizací. Hlavní závěry byly ty, že v nejoptimističtějším případě bylo pouze 20-30% projektů úspěšných – nebo-li 7 z 10 projektů končí z různých důvodů neúspěšně.

2.4 Faktory selhávání IT projektů

Nejobvyklejší faktory [Charette 2005] proč tyto projekty selhávají jsou tyto:

- Nerealistické nebo neupřesněné cíle

- Nepřesný odhad potřebných zdrojů.
- Špatně definované systémové požadavky.
- Špatné monitorování (reporting) stavu projektu.
- Špatné řízení rizika.
- Špatná komunikace mezi zákazníky, vývojáři a uživateli.
- Použití nevhodné nebo nezralé technologie.
- Neschopnost se vypořádat s komplexností projektu.
- Špatné vývojářské praktiky.
- Špatné řízení projektu.
- Obchodní a investorova politika.
- Komerční tlaky.

Samozřejmě, projekty typicky neselhávají jenom z jednoho nebo ze dvou výše uvedených důvodů. Typicky je to kombinací více důvodů. Většina neúspěchů je způsobena kombinací technických, manažerských a obchodních rozhodnutí.

3 Metody překonávání softwarové krize

Tato kapitola popisuje historii i současnost softwarového inženýrství, jako úspěšné metody pro zmírněni následků softwarové krize. Jsou zde probrány jednotlivé fáze modely životního cyklu vývoje softwaru i jejich možné modely. Jsou zde uvedeny možná paradigmata vývoje softwaru i rozdělení metodik, z těchto paradigmat vycházejících. Nakonec se tato kapitola věnuje metamodelování, jako prostředku pro formální popis metod a metodik návrhu informačních systémů.

3.1 Úvod

Softwarová krize vznikla proto, že softwarové systémy se stávaly stále složitější a jednotlivec je nebyl schopen v celé jejich šíři pochopit podrobně. Hlavní postupy pro překonávání této krize tedy jsou:

- Umožnit jednotlivci více pochopit a vytvořit, sem patří programovací paradigmata, programovací jazyky, modelovací jazyky, CASE nástroje, vizuální programování, návrhové vzory , oddělení programů od dat (databáze) atd.
- Umožnit efektivní spolupráci více jednotlivců, sem patří zejména metody organizace a řízení projektů, metodiky tvorby softwaru atd.

Obor, který se těmito postupy zabývá se nazývá softwarové inženýrství.

3.2 Softwarové inženýrství

Termín softwarové inženýrství byl poprvé použit na konferenci NATO v Garmish-Partenkirchenu roku 1968. Tento termín byl zvolen, aby se naznačila nutnost zavedení opakovatelných postupů do tvorby softwaru. Následující rok na konferenci v Římě se účastníci shodli na následující definici softwarového inženýrství [Brooks 1995]:

> *„Softwarové inženýrství je disciplina, která se zabývá zavedením a používáním řádných inženýrských principů do tvorby software tak, abychom dosáhli ekonomické tvorby software, který je spolehlivý a pracuje účinně na dostupných výpočetních prostředcích.“*

40 let historie vývoje softwaru a softwarového inženýrství zachycuje tabulka Tab. 2.

Etapa		
	Charakteristiky	
do poloviny 60. let 20. stol.: pionýrské doby softwarových projektů		
	Hardware	sálové počítače, velmi drahý, omezený výpočetní výkon, specializovaný na hromadné zpracování dat.
	Uživatelé	armáda, vědecká pracoviště, státní instituce, největší podniky
	Software	specializovaný software s dávkovým zpracováním, úlohy numerické a datového zpracování, první databáze (IMS - 1966), terminálová struktura aplikací
	Programovací jazyky, vývojové nástroje	strojový kód (asembler), první počítačové jazyky FORTRAN (1956), COBOL (1959), LISP(1958), Algol (1960)
	Metody a metodiky	software vytvářen jednoúčelově bez ohledu na budoucí potřeby, programy většinou neudržovatelné a neměnné
1968 – konference NATO v Garmish-Partenkirchenu – první použití pojmů softwarové inženýrství a softwarová krize		
70. léta		
	Hardware	HW se stává dostupnějším, vznik UNIXu (1969-73), vznik mikroprocesoru (1971), v podniku obvykle jeden výkonný počítač k němuž se uživatelé připojují pomocí terminálů.
	Uživatelé	uživatelská obec se rozšiřuje větší podniky a vysoké školy
	Software	první aplikace umožňující interakci se svými uživateli, počátek produktového softwaru (sw. se opakovaně prodává), hierarchické a síťové databáze, první relační

		databáze (konec 70. let)
	Programovací jazyky, vývojové nástroje	strukturované programování Pascal – (1970), C (1974). deklarativní Prolog, vznik objektově-orientovaných jazyků (Smalltalk – 1971), dotazovací jazyk SQL (do 1979).
	Metody a metodiky	vývoj není řízen, v druhé polovině 70. let se začínají využívat techniky softwarového inženýrství (specifikace, návrh, testování, modely životního cyklu, atd.), první strukturované metodiky – např. Structured Analysis (Cnstantine, Yourdon – 1975), Jackson (1975), Structured Analysis (DeMarco, Yourdon – 1978)
80. léta		
1981 – osobní počítač IBM PC		
	Hardware	rozšíření osobních počítačů (IBM PC – 1981), miniaturizace výkonných počítačů, výkonné pracovní stanice, nástup lokálních počítačových sítí
	Uživatelé	uživatelská obec se dále rozšiřuje do průmyslu i služeb, osobní počítače se začínají široce používat při kancelářské práci
	Software	rozvoj „krabicového“ softwaru – zejména pro osobní počítače. rozvoj softwaru pro kancelářské účely (textový a tabulkový procesor, atd.). Vznik GUI.
	Programovací jazyky, vývojové nástroje	používají se zejména strukturované jazyky, rozvoj objektově-orientovaných jazyků například C++ (1979-83). První CASE (Computer Aided Software Engineering) nástroje (1982), první programy pro správu verzí (CVS – 1986)
	Metody a metodiky	velký nástup softwarově-inženýrských metodik, strukturované metodiky (SSADM – 1980) ke konci 80. let zkoumání objektově orientované analýzy a návrhu, první objektově-orientované metodiky – OOSA (Shlaer-Mellor – 1988)
90. léta		
1993 – v CERNU vzniká World Wide Web		

1996 – sdružení OMG přijímá jazyk UML za svůj standard		
	Hardware	dramatický nárůst výkonnosti HW při značném poklesu ceny, výpočetní technika i příslušenství se stávají běžnou záležitostí dostupnou pro všechny firmy i jednotlivce, rozvoj globálních sítí – internet, web (1992)
	Uživatelé	instituce, školy, firmy i jednotlivci
	Software	široká nabídka „krabicového“ softwaru, nejen pro kancelářské využití (databáze, grafika, DTP, hry, zpracování textu aj.). Současně rozvoj softwaru na zakázku, značně roste komplexnost SW a rozsáhlost projektů. Široké používání relačních databází.
	Programovací jazyky, vývojové nástroje	velký rozvoj objektově-orientovaných jazyků – Java (1994), integrovaná vývojová prostředí (IDE – vizuální programování, rozsáhlé standardní knihovny programovacích jazyků
	Metody a metodiky	velký rozvoj objektově-orientovaných metodik, modelovací jazyky (UML – 1996), RAD – Rapid Applications Development). Měření (míry).
1997 softwarové inženýrství uznáno jako obor s certifikátem v USA		
od r. 2000		
2003 první akreditovaný bakalářský program v oboru Softwarové inženýrství (USA).		
	Hardware	velmi levný a výkonný hardware, počítače se rozšiřují téměř všude – od těch nejmenších zařízení (telefony, PDA, atd.) až po největší zařízení.
	Uživatelé	všichni
	Software	software má mnoho podob: • podnikové aplikace, • „klasický“ uživatelský software, • webové aplikace, • „embeded“ software (ve spotřební elektronice) aj.

		velký důraz internetové aplikace, SOA (Service Oriented Architecture)
	Programovací jazyky, vývojové nástroje	díky sofistikovaným programovacím nástrojům je software vyvíjen velmi rychle, stoupá jeho komplexnost, velký důraz na web, prosazují se objektově-orientované skriptovací programovací jazyky (Python, Ruby, PHP),velký důraz na znuvupoužitelnost (komponenty). Programátor spíše program skládá z již existujících komponent než programuje. Objevují se aspektově-orientované jazyky (AspectJ – 2001), jazyk XML. důsledné používání návrhových vzorů.
	Metody a metodiky	Plně se prosazují objektově-orientované metodiky, kromě klasických metodik se začínají prosazovat i „lehké“ metodiky umožňující pracovat efektivněji v dynamicky se měnícím prostředí.

Tab. 2 – Vývoj softwarového inženýrství

Softwarové inženýrství má blízkou souvislost s různými disciplami [Richta 2006]:

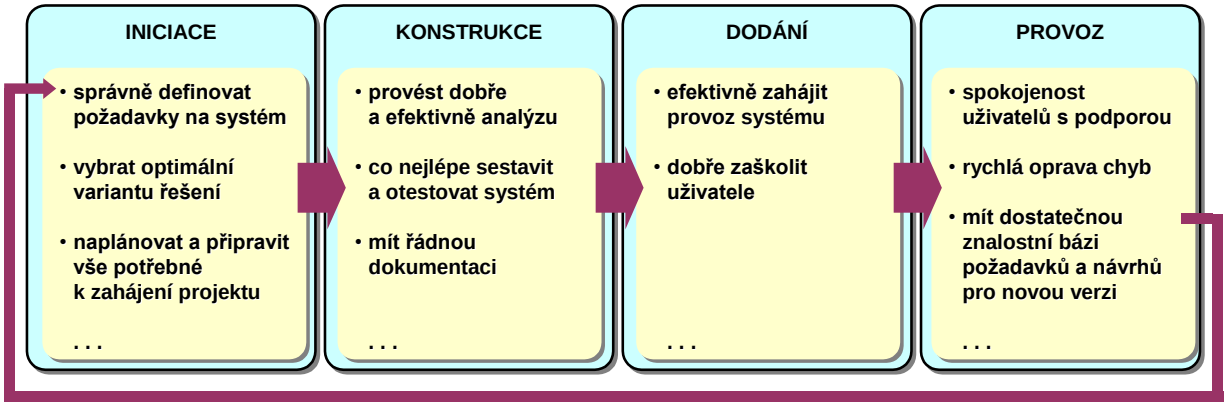
- Inženýrství – jedná se o disciplinované využívání pragmatických zkušeností, tj. rutinní postupy očekávané od inženýra.
- Věda – softwarové inženýrství zahrnuje v sobě rozvoj matematiky a logiky, které jsou potřebné k řešení úloh. Návrhář používá exaktní metody a formální postupy.
- Umění – softwarové inženýrství lze považovat také za umění, neboť v sobě zahrnuje aspekty přisuzované umění návrhu (např. krásu, eleganci) – návrh vzhledu, návrh rozhranní, ovládání, návrh architektury, atd.
- Tovární výroba – v některých firmách funguje návrh softwaru jako tovární výroba. Mnoho programátorů spolupracuje na programu podobně jako by byl na výrobní lince. Jedem programátor přidává tlačítka, další programuje připojení do databáze, atd. Program skládají ze standardizovaných dílů.
- Řízení projektů – v současné době se jsou realizovány velmi rozsáhlé projekty, na kterých ze mohou účastnit desítky analytiků, designérů, programátorů a testerů. Je potřeba naplánovat kapacity, přidělit zdroje, zajistit koordinaci a pod.

3.3 *Fáze životního cyklu vývoje softwaru*

Celý vývojový cyklus vývoje softwaru se obecně skládá z těchto fází [Ambler 1998] (viz obrázek Obr. 3 – Hlavní fáze životního cyklu [Merunka et al 2005]):

1. **Iniciace** (Initiate).
2. **Konstrukce** (Construct).
3. **Dodání** (Deliver).
4. **Provoz** (Maintenance and support).

Dle charakteru projektu se dává na různé fáze rozdílný důraz. U rozsáhlých a rizikových projektů se například dává velký důraz na úvodní fázi přípravy. U menších projektů může převládnout fáze vývoje, u životně kritických projektů je kladen důraz na fázi správy a podpory.



Obr. 3 – Hlavní fáze životního cyklu [Merunka et al 2005]

3.3.1 *Iniciace*

Proces iniciace slouží k zajištění všech přípravných prací, nezbytných k zahájení tvorby software. Jeho cílem je připravit vše potřebné k zahájení projektu. V rámci iniciace mohou proběhnou tyto procesy [Pergl 2008]:

- **Vymezení a ověření počátečních požadavků** – různé metodiky mají specifický přístup ke shromáždění a dokumentaci uživatelských požadavků.
- **Klasifikace požadavků** – požadavky je třeba klasifikovat (viz např. [FURPS 2007]), např. jim podle důležitosti přidělit priority.
- **Posouzení projektu** – rozhodujeme o realizovatelnosti např. pomocí studie proveditelnosti (Feasibility study) [Keyes 2002].

- **Vymezení typu a obsahu dokumentace** – je třeba vymezit rozsah a formu projektové dokumentace.
- **Vymezení infrastruktury projektu** – před zahájením projektu je třeba vytvořit potřebnou infrastrukturu.

3.3.2 Konstrukce

Proces konstrukce slouží k vytvoření požadovaného systému. Vlastní vývoj začíná teprve v této fázi. Vývoj může probíhat jak sekvenčně, tak i jak iterační proces [Ambler 1998]. Konstrukce se skládá z těchto fází:

- **Analýza** – analýza je prováděna tímto způsobem [Pergl 2008] – modelování (podnikové, doménové, architektury – výstupem je konceptuální model), identifikace funkčních celků, rozhodnutí o znovupoužitelnosti, či nákupu, či vývoji.
- **Návrh** – modelování návrhu implementace. Výstupem je například digram tříd (definuje strukturu) a sekvenční diagram (definuje chování) [Rumbaugh at al. 1998].
- **Implementace** – se skládá z těchto činností – psaní kódu realizující vlastní činnost programu (obchodní logiku), návrh uživatelského rozhranní, integrace komponent, psaní vývojářské dokumentace, psaní příruček, atd.
- **Testování** – testovat můžeme všechny fáze životního cyklu (metoda FLOOT [Ambler 2004]):testování požadavků, testování analýzy, testování návrhu, testování kódu. V současné době se prosazuje tzv. Test Driven Development [Beck 2004], kdy testujeme průběžně při implementaci.

3.3.3 Dodání

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

- **Testování ve velkém** – v [Ambler 2004] jsou tyto testy rozděleny na – testy systému (funkční testování, testování instalace, zátěžové testování, operační testování, testování podpory) a uživatelské testování (alfa testování, beta testování, pilotní testování, akceptační testování).
- **Přepracování** – vady objevené při testování ve velkém jsou zaznamenány a následně opraveny. Tato oprava je vlastně novou fází konstrukce a proběhnou v ní znova všechny fáze.

- **Nasazení** – jádrem je vlastní instalace systému, ať již celého nového, nebo pouze aktualizací. Nasazení se skládá z těchto částí – příprava k předání (vypracování plánů školení uživatelů, akceptace dokumentace, migrace dat), nastavení procesů (nastavení provozních procesů a podpory,školení pracovníků provozu a podpory) a předání uživatelům (školení uživatelů, instalace aplikace).
- **Vyhodnocení** - slouží pro projektový tým k vyhodnocení zkušeností z projektu.

3.3.4 Provoz

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

- Uživatelská podpora – reprezentuje množinu aktivit, prováděných v týmu „help desku". Jedná se o různé rady uživatelům, týkající se ovládání a instalace systému, péče o průběžné doškolování a zaškolování. Dále sem spadá řešení chyb s hardwarem, sběr nových podnětů k vývoji systému (change management) apod.
- Údržba – představuje operativní odstraňování chyb v systému.

3.4 Modely životních cyklů vývoje softwaru

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 [Merunka et al. 2005]:

- **sekvenční,**
- **iterativní,**
- **adaptivní**.

3.4.1 Sekvenční model

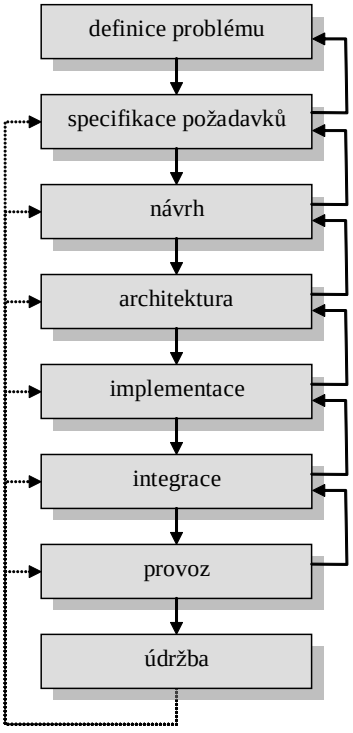
Sekvenční model je založený na myšlence, že existuje systematický postup, jak dojít od zadání k řešení pomocí řady na sebe navazujících činností, které lze předem naplánovat. Výstup jedné aktivity je vstupem následující. Tento přístup se často používá (a většinou je i jediný možný) ve velké většině ostatních inženýrských disciplín. u Nejznámější varianta sekvenčního modelu je tzv. vodopádový model (viz. obrázek Obr. 4) jehož autorem je Winston Royce [Royce 1970], další varianta je striktní posloupnost fází.

Sekvenční se skládá z posloupnosti fází:

- Definice problému.
- Specifikace požadavků.
- Návrh.
- Architektura.
- Implementace.
- Integrace.
- Provoz.

U sekvenčního modelu provádíme vyhodnocení na úplném konci procesu, až po dodání produktu a získání zkušeností s nim, a na základě tohoto vyhodnocení se vracíme do příslušné fáze a provádíme nový vývoj.

U vodopádového modelu je navíc zavedena zpětná vazba, kdy na konci každé fáze provádíme vyhodnocení a případně se vracíme.



Obr. 4 – Model vodopád

Nevýhodou* tohoto modelu je, že mezi zadáním projektu a jeho dodáním uplyne velmi dlouhá doba, při které nemáme zpětnou vazbu od zadavatele – požadavky na systém se mohou zatím změnit.

Na základě vodopádového modelu je založena většina strukturovaných metodik.

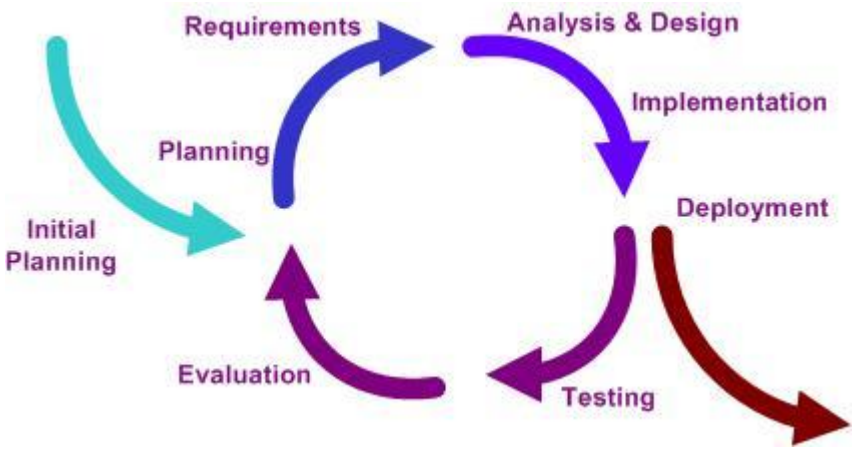
3.4.2 Iterativní model

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

Obecně jsou iterativní modely založeny na střídání těchto čtyřech fází (Obr. 5):

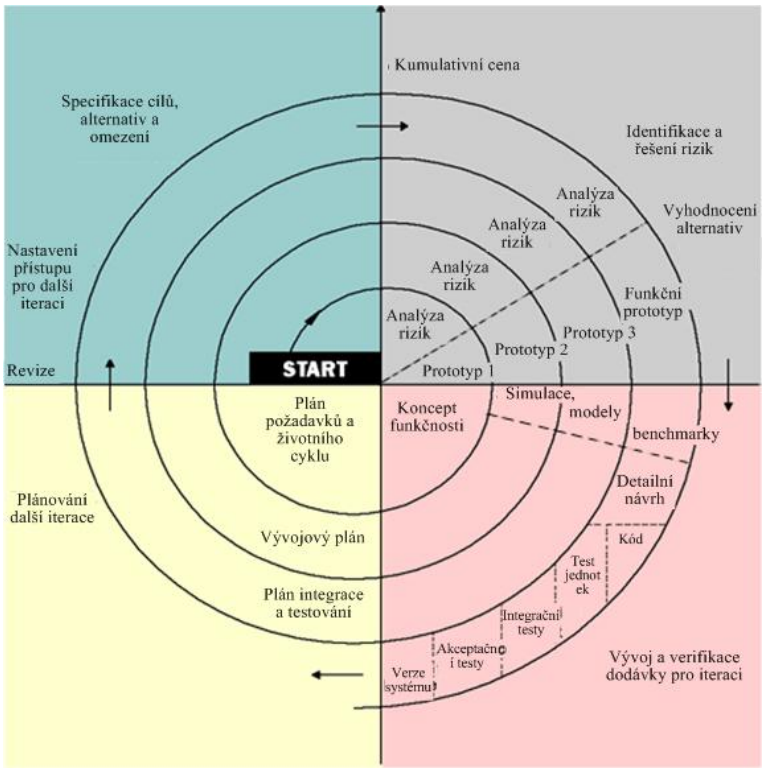
1. Specifikace.
2. Analýza a implementace.
3. Testování .
4. Vyhodnocení.

Výhodou tohoto modelu je zpětná vazba po proběhnutí každé iterace.



Obr. 5 – Iterativní model vývoje [wikipedia.org]

* Sám autor tohoto modelu v [Royce 1970] hodnotí jeho nejjednodušší formu jako „riskantní a náchylnou k chybám“.



Obr. 6 – Model spirála [Pergl 2008]

Spirálový model navrhl Barry Boehm v roce 1988 [Boehm 1988]. Nebyl to sice první iterativní model, ale autor poprvé vysvětlil proč provádět iterace. Iterace typicky trvají několik měsíců (Boem původně navrhoval 6 měsíců až 2 roky), na konci každé z nich je vyroben prototyp, který se vyhodnotí a následně se přechází do nové spirály. Spirálový model se typicky používá pro větší projekty[*].

V dnešní době jsou objektově-orientované metodiky založeny typicky na základě nějakého typu iterativního modelu.

3.4.3 Adaptivní model

Adaptivní model vychází z iterativního (čili pro něj platí také obrázek Obr. 5). Hlavním rozdílem je zkrácení iterací na minimum, maximálně v řádech týdnů (spíše týdne). Častokrát bývá doba iterace pevně daná. Jeho základní principy jsou vyjádřeny v Agilním manifestu [Agile 2001] :

[*] Pro zajímavost – tento model se používá nejen pro softwarové projekty. Spirálový model vývoje je použit také pro (zatím ne moc úspěšný) vývoj Americké protiraketové obrany [Boese 2008].

- Individuality a iterace spíše než procesy a nástroje.
- Fungující programy spíše než vyčerpávající dokumentaci.
- Spolupráci zákazníka nežli dohodu ve smlouvě.
- Reagovat na změnu nežli dodržovat plán.

Adaptivní model nezávisí na plánu, ale průběh další iterace je určován v dohodě se zákazníkem. Výhodou tohoto modelu je veliká pružnost a připravenost na změnu. Jeho nevýhodou je jeho nepředvídatelnost a nevhodnost pro velké projekty. V adaptivním modelu se častokrát používá tzv. Rapid Application Development [Martin 1990].

Na adaptivním modelu jsou založeny agilní metodiky.

3.5 Paradigmata

Paradigmata* jsou základní principy, na nichž jsou založeny metody vývoje softwaru. Používáme zejména tyto paradigmata:

- Strukturované – častokrát se používá synonymum imperativní – musíme specifikovat přesné kroky, jak se máme dostat do požadovaného stavu. Strukturované paradigma zakazuje konstrukci skoku, takže nám z programových konstrukcí zbývají podmínka, cyklus a volání procedury. Strukturovaný program má odděleny datové a programové struktury.
- Objektové – základním pojmem je objekt, což je struktura spojující data s funkcemi (zde nazývanými metody), které s nimi pracují. V dnešní době je objektové paradigma nejpoužívanější při tvorbě softwaru.
- Deklarativní – v deklarativním paradigmatu neříkáme jak přesně má systém pracovat, ale popisujeme co má dělat. Při vývoji softwaru se nejčastěji z deklarativních jazyků používá jazyk SQL pro definici dat a manipulaci s nimi.
- Ostatní – mezi ostatní shrnuji taková paradigmata se příliš nepoužívají v metodách vývoje a návrhu softwaru a to, protože jsou buď příliš nová (například aspektové nebo metaprogramování) anebo se používají pro specializované účely (funkcionální – umělá inteligence).

* Tuto část schválně nenazývám obvyklejším „Programovací paradigmata“, protože tyto paradigmata (tj. základní vzorce) se netýkají jenom programování, ale téměř celého procesu vývoje softwaru

V průběhu tvorby konkrétního softwaru typicky nepoužíváme pouze jedno paradigma, ale kombinaci několika paradigmat.

3.6 Metodiky

Metodiky vývoje softwaru jsou rámec, který je použit pro strukturování, plánování a řízení procesu vývoje informačního systému [CMS 2005].

Vlastnosti metodik vývoje softwaru jsou:

- každá metodika je založena na jednom, nebo i více, základních paradigmat (viz. 3.5 Paradigmata).
- každá metodika klade různý důraz na jednotlivé fáze životního cyklu vývoje softwaru, některé může pomíjet, jiné bude zpracovávat v celé šíři. Typicky neobsahuje všechny procesy vyjmenované v odstavcích 3.3.1 až 3.3.4.
- Fáze životního cyklu metodiky jsou různě za sebou organizovány (viz. část 3.4).
- Metodika obsahuje různé nástroje, modely a metody, které nám pomáhají při vlastním procesu vývoje softwaru.

Dále se dají metodiky rozdělit podle složitosti:

- Těžké (heavyweight) – jsou to komplexní a složité metodiky s mnoho technikami a pravidly. Typicky se snaží zachytit proces vývoje v celé šíři. Kladou velký důraz na dokumentaci.
- Lehké (lightweight) – obsahují pouze několik technik a pravidel, která se dají snadno dodržovat. Kladou největší důraz na implementaci.

Podle těchto hledisek můžeme metodiky klasifikovat. Například:

- metodika UP [Jacobson et al. 1999] – je „těžká“ objektově-orientovaná metodika, založená na iterativním procesním modelu, zabývá se všemi fázemi vývoje, a používá především jazyk UML.
- SSADM [SSADM] – je „těžká“ strukturovaná metodika, založená na vodopádu, zabývá se všemi fázemi vývoje a používá zejména data-flow a ER modelování.
- extrémní programování [Beck 2004] – je lehká objektově-orientovaná metodika, založená na evolučním modelu, soustředí se zejména na fázi implementace

Aby bylo možné metodiky a metody vytvářet, formalizovat a následně modelovat, k tomu slouží metamodelování (tj. modelování modelování).

3.7 Metamodelování

Předpona *meta-* pochází z řečtiny (μετά) a znamená za, po, vně, mimo. Meta- se používá k vyjádření skutečnosti, že koncept (tedy v tomto případě metakoncept) je abstrakcí jiného konceptu – metakoncept tedy popisuje koncept. Z hlediska modelování tedy metamodel znamená model popisující model a metamodelování zjednodušeně znamená modelování modelování.

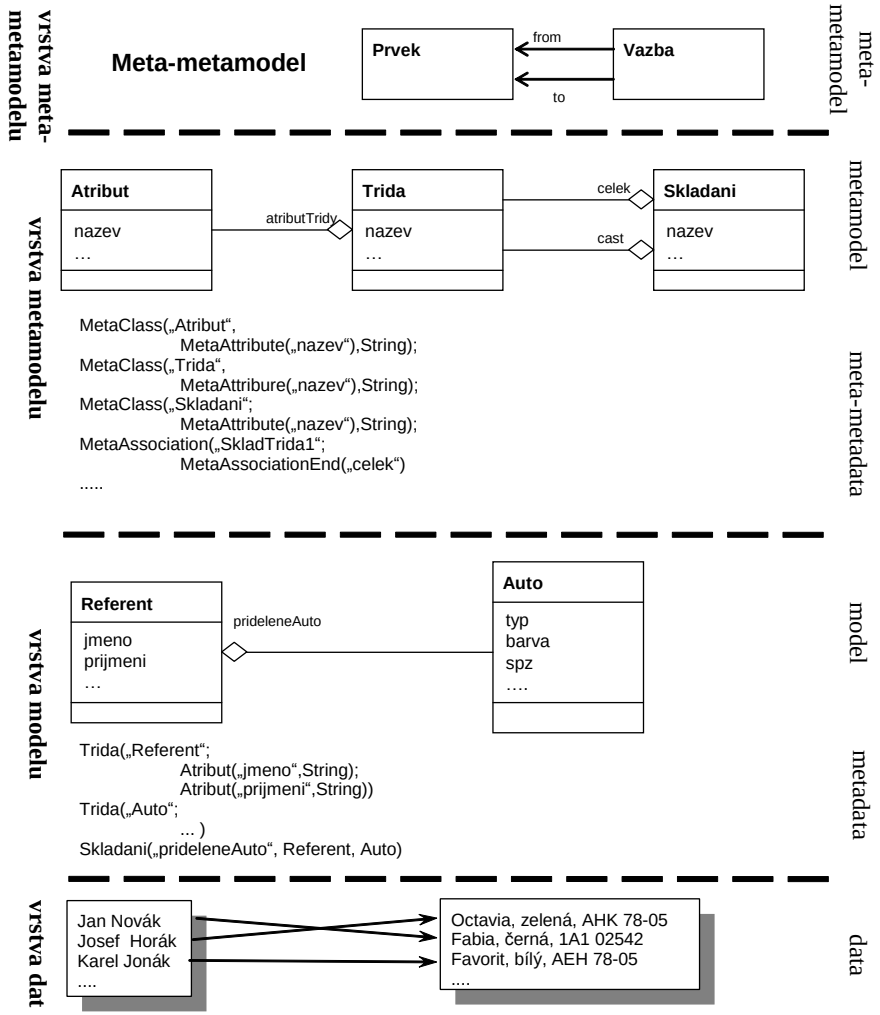
Podle [Mili et al. 1995] má metamodelování tyto 3 dimenze:

1. Metamodelování, jako modelování modelovacího jazyku.
2. Metamodelování, jako znázornění vícenásobných úrovní modelu.
3. Metamodelování, jako modelování způsobu, jak manipulovat s aplikačním modelem a používat ho.

Klasický rámec pro metamodelování je podle [OMG 2006] založen na čtyřvrstvé architektuře (Obr. 7). Tyto vrstvy jsou obvykle popisovány takto:

- Informační (datová) vrstva zahrnuje popisovaná data.
- Vrstva modelu zahrnuje metadata popisující data v informační vrstvě. Metadata jsou neformálně shrnuta do modelů.
- Vrstva metamodelu* zahrnuje popis (tj. meta-metadata) definující strukturu a sémantiku metadat. Z meta-metadat jsou neformálně tvořeny metamodely. Metamodel je abstraktním jazykem pro popis různých druhů dat.
- Vrstva meta-metamodelu definuje strukturu a sémantiku meta-metadat (tj. popisuje metamodel).

* V dnešní době se metamodely obvykle používají jako:
1. schéma, pro definici sémantiky dat pro jejich uložení nebo výměnu
2. jazyk k podpoře speciálních metod a procesů.
3. jazyk k rozšíření sémantiky existující informace



Obr. 7 – Čtyřvrstvá architektura pro metamodelování [Picka 2003]

3.7.1 Prostředky pro metamodelování

Metamodelovací metody definují rámec pro metamodelování, který obvykle obsahuje definici meta-metamodelu a metamodelovacího jazyku, pomocí kterého je definován metamodel. Pro potřeby metamodelování v informačním a softwarovém inženýrství bylo vyvinuto mnoho přístupů pro tvorbu metamodelu – COMMA [Bulthuis et al. 1998], GOPRR [Kelly 1997], MOF [OMG 2006], OPRR [Rossi at al. 2006] atd.. V dalších odstavcích jsou popsány tři metamodelovací rámce, které jsou vhodné pro definici metodik (více o metamodelování a metamodelech v [Pícka 2005a]).

3.7.1.1 COMMA

Projekt COMMA (Common Object Methodology Architecture) se snažil identifikovat společné jádro všech objektově orientovaných metodologií, následně reprezentovat tyto základní pojmy pomocí metamodelu a vytvořit metamodely nejrozšířenějších objektově orientovaných metodologií (více viz [Bulthuis et al. 1998]).

COMMA používá tyto základní pojmy:

- **Pojem** (Concept) – má jméno a atributy,
- **Dědění** (Inheritance) – vyjadřuje relaci specializace,
- **Asociace** (Association) – vyjadřuje vztah mezi pojmy,
- **Agregace** (Aggregation) – vyjadřuje sklání, je to speciální případ asociace,
- **Role** (Role) – objevuje se, když objekt přijímá charakteristiky jiného objektu. Role je dočasná a objekt může mít i více rolí najednou.

Hlavním výsledkem projektu COMMA je vytvoření velmi jednoduchého (ale mocného) objektově orientovaného metamodelovacího jazyka.

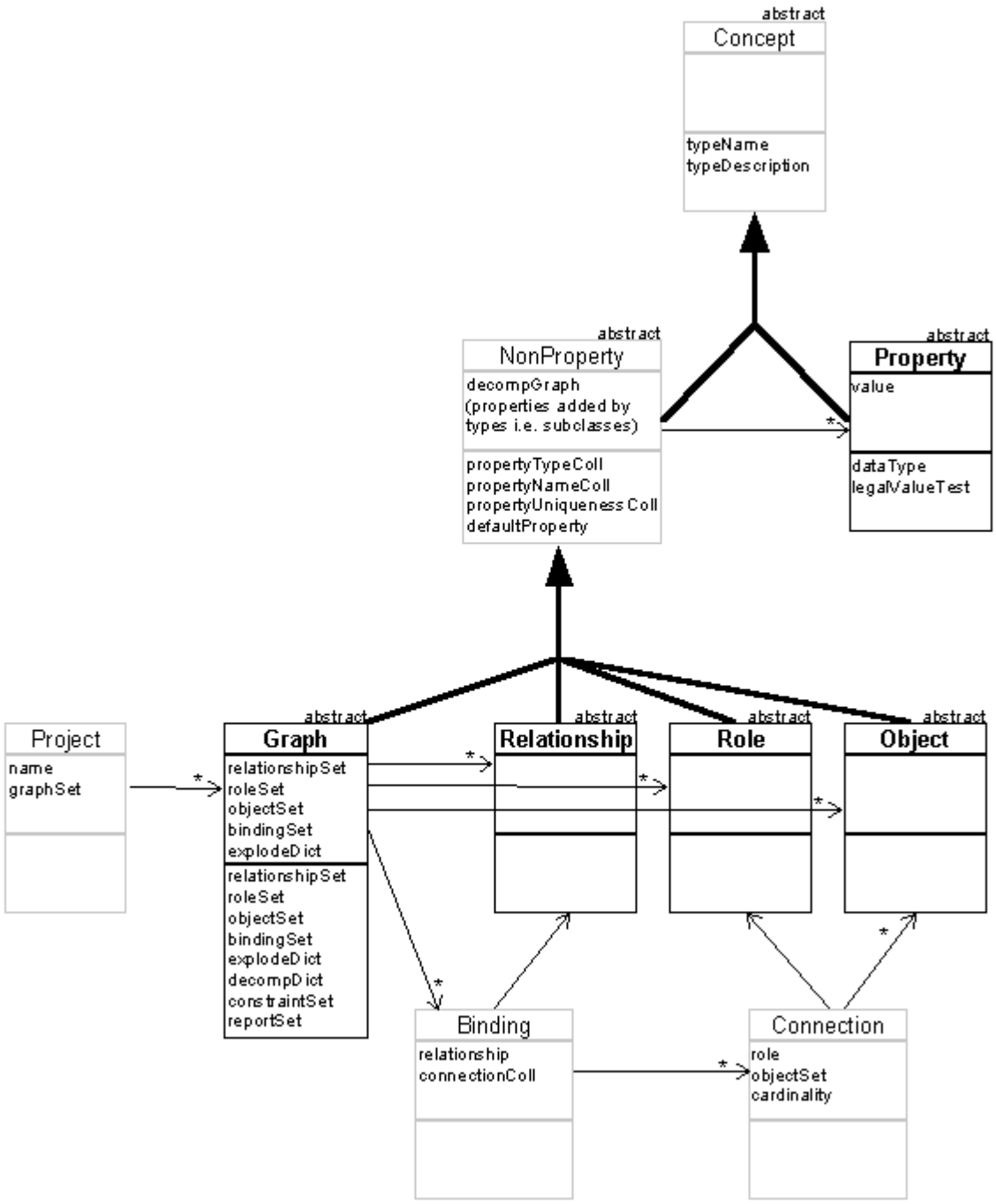
3.7.1.2 GOPRR

Metamodelovací jazyk GOPRR (Graph-Object-Property-Role-Relationship) vznikl, jako součást disertační práce pana Kellyho (viz [Kelly 1997]), rozšířením jazyka OPRR. Hlavním úkolem bylo na základě GOPRRu vytvořit CAME (Computer Aided Method Engineering) nástroj MetaEdit+ [Kelly et al. 2008].

Základními prvky metamodelovacího jazyka GOPRR, už podle názvu, jsou:

- **Diagram** (Graph) – je kolekce objektů, vztahů a rolí, která definuje co a jak lze spojovat dohromady.
- **Objekt** (Object) – definuje co entitu která může existovat samo o sobě.
- **Vlastnost** (Property) – charakterizuje graf, objekt, roli nebo vztah.
- **Vztah** (Relationship) – existuje mezi dvěma a více objekty.
- **Role** – existuje mezi vztahem a objektem.

Meta-metamodel GOPRR je na znázorněn na Obr. 8.



Obr. 8 – GOPRR meta-metamodel [Kelly 1997]

3.7.1.3 Rodina standardů OMG

Standardy OMG (Object Management Group) jsou založeny na čtyřvrstvé architektuře (viz Obr. 7). Vrstvu meta-metamodelu popisuje standard MOF (Meta Object Facility – viz [OMG 2006][*]) . OMG definuje několik metamodelů založených na MOF. Například:

- metamodel UML (Unified Modeling Language) – standard pro objektový modelovací jazyk (více [OMG 2009]),
- metamodel IDL (Interface Definition Language) – standard popisující objektová rozhraní tříd pro standard distribuovaných objektů CORBA, a jejich mapování do různých programovacích jazyků.
- metamodel CDW (Common Data Warehouse) – standard definující architekturu datových skladů.

Data mezi metamodely založenými na MOF mohou být vyměňována pomocí formátu XMI (XML Metadata Interchange – [OMG 2007]).

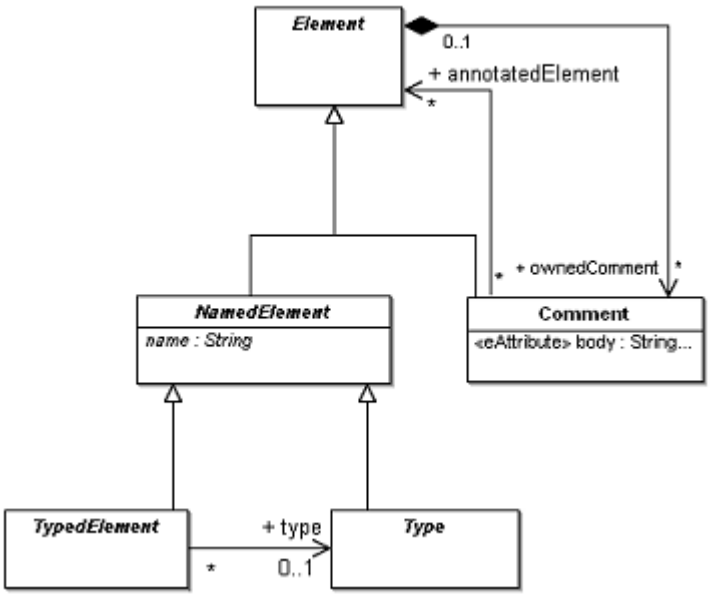
MOF je samovysvětlující, tj. definuje sám sebe (a tedy neexistuje meta-meta-metamodel). Základními koncepty MOF jsou:

- **třídy** (Class)– modelují metaobjekty,
- **asociace** (Assotiation)– modelují binární relace mezi metaobjekty,
- **datové typy** (data Type)– modelují primitivní data,
- **balíček** (Package) – slouží k modularizaci modelu.

Standard MOF 2 obsahuje dvě základní části – EMOF (Essential MOF) a CMOF (Complete MOF) – viz [OMG 2006]. EMOF představuje podmnožinu CMOF a umožňuje jednodušší způsob vytváření nových metamodelů a následnou manipulaci s nimi.

Metamodel UML (viz [OMG 2009]) úzce vychází z MOF a liší se jenom v drobnostech (např. umožňuje vícenásobné asociace). Například třída v UML je definována jako instance třídy „Class“ v UML metamodelu. Tato třída je definována jako instance třídy „Class“ z MOF modelu (meta-metamodelu). A nakonec třída „Class“ z MOF modelu je definována sama sebou.

[*] Starší verze MOF (v současnosti je aktuální verze 2.0) je standardizována i jako ISO standard ISO/IEC 19502:2005



Obr. 9 – Část balíčku Core:Basic standardu MOF [OMG 2006]

4 Softwarové požadavky

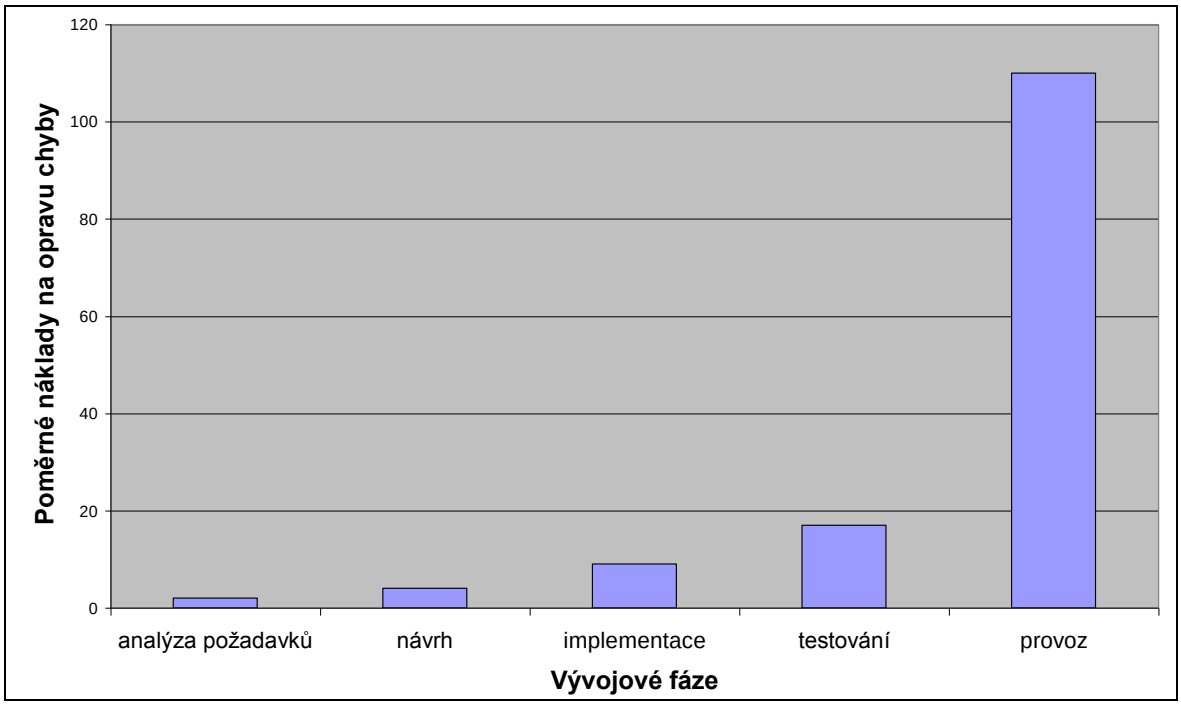
Tato kapitola popisuje softwarové požadavky, jejich typy a vlastnosti. Požadavky jsou základem tvorby jakéhokoliv informačního systému. Pokud jsou požadavky na systém nesprávně formulovány nebo dokonce chybí, tak dochází k velkým časovým i finančním ztrátám při tvorbě softwaru. Dále se tato kapitola věnuje vztahu jednotlivých částí (modelu) systému a požadavků – dohledatelnosti požadavků.

4.1 Význam požadavků

Požadavky a práce s nimi hraje kritickou roli při tvorbě softwaru. Význam požadavků popsal F. Brooks [Brooks 1987] takto:

> *Nejtěžší samostatnou fází tvorby softwarového systému je rozhodnout, co přesně dělat. Žádná z ostatních částí konceptuální práce není tak složitá, jako vybudování podrobných technických požadavků. Žádná z ostatních částí práce systém tak nezmrzačí, když ji uděláte špatně. Žádná z ostatních částí se tak těžko neopravuje později.*

Špatně napsané požadavky zapříčiňují předělávání a náklady na něj mohou dosáhnout 30-50% celkových nákladů [Boehm 1988], 70-80% nákladů na předělávání je způsobeno špatnou prací s požadavky [Leffingwell 1997]. Čím později se na chyby v požadavcích přijde, tím dražší je jejich náprava (viz. [Grady 1999] na obrázku Obr. 10).



Obr. 10 – Poměrné náklady na chybu, podle toho, kdy byla objevena [Grady 1999]

4.2 Definice softwarových požadavků

Pojem softwarový požadavek je definován v [Sommerwille 1997] takto:

> *Softwarowé ožadavky jsou popis toho , co všechno by se mělo implementovat. Popisují žádané chování systému a jeho vlastnosti a mohou představovat nějaká omezení procesu vývoje systému*

Požadavek je tedy „cokoliv, co ovlivňuje rozhodování při návrhu“ [Lawrence 1997].

4.3 Typy požadavků

Požadavků máme několik druhů [Wiegers 2003]:

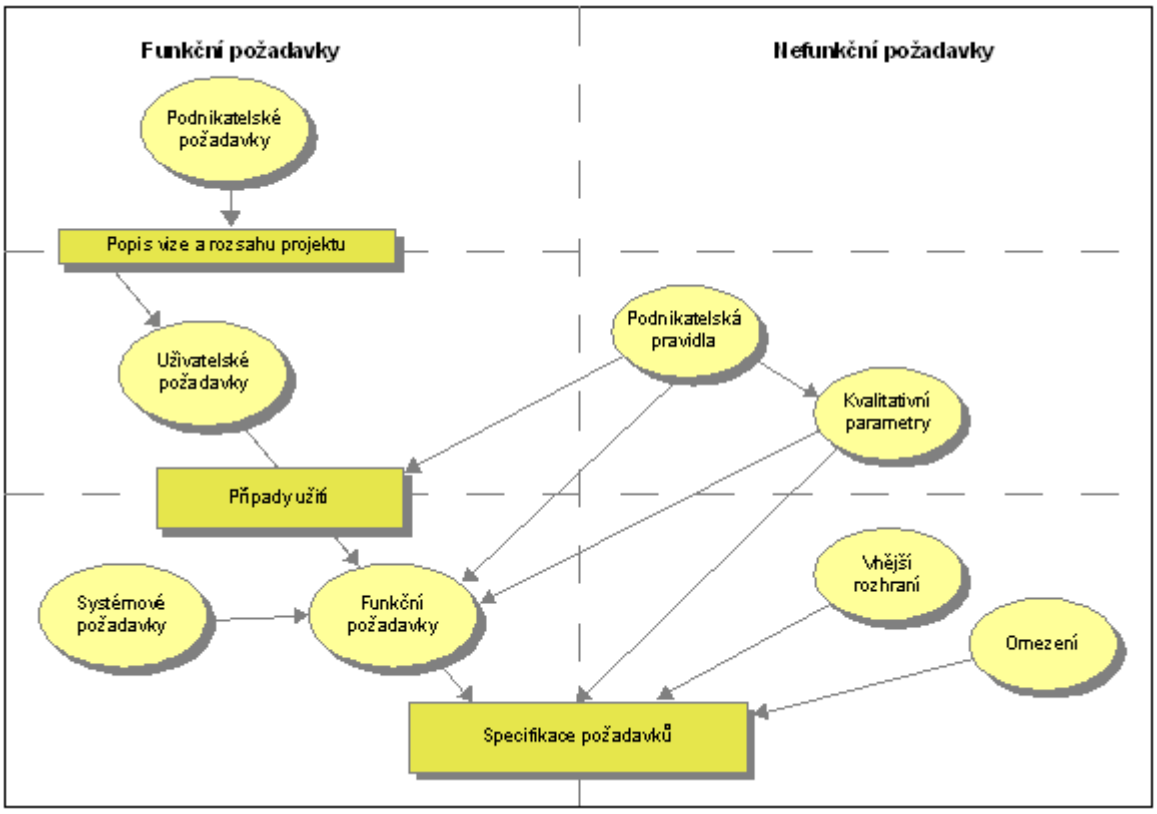
- **Funkční požadavky** – představují přímé požadavky systém, tedy co se má doopravdy implementovat. Funkčních požadavků máme několik druhů:
    - **Podnikatelské (business) požadavky** – Formulují cíle zadavatele na nejvyšší úrovni. Podnikatelské požadavky říkají, čeho se má prostřednictvím vyvíjeného systému dosáhnout. Tyto požadavky ohraničují projekt.

- **Uživatelské požadavky** – popisují cíle uživatelů a úkoly, které mají uživatelé prostřednictvím systému řešit.
- **Požadavky na funkčnost** – popisují softwarovou funkcionalitu, která řeší uživatelské požadavky. říká se jim také požadavky na chování systému.
- **Systémové požadavky** – jsou celkové požadavky na systém složený z více podsystémů.

- **Nefunkční požadavky** – jedná se o požadavky vysvětlující. Nefunkční požadavky zachycují vliv okolí, vliv omezení na proces vývoje
  - **Kvalitativní požadavky** – zachycují požadavky na kvalitu softwaru [Vaníček 2004].
  - **Omezení** – popisuje omezení systému daná vnějšími podmínkami.

Závislost a návaznosti mezi požadavky zachycuje obrázek Obr. 11. Požadavky jsou zachyceny ve specifikaci požadavků (software requirements specification).

Mezi požadavky nepatří podrobnosti o návrhu nebo implementaci (mimo omezení), informace o plánování a informace o testování [Leffingwell 2000].



Obr. 11 – Vztahy mezi různými typy požadavků [Wiegers 2003]

4.3.1 Vlastnosti dobře napsaných požadavků

Vlastnosti dobře napsaných požadavků shrnuje [IEEE 830] takto:

- Vlastnosti jednotlivých požadavků
  - **Úplnost** – každý požadavek musí popisovat danou funkcionalitu úplně. Musí obsahovat všechnu informaci, kterou vývojáři potřebují pro jeho návrh a implementaci.
  - **Správnost** – každý požadavek musí popisovat příslušnou funkcionalitu přesně, o přesnosti rozhoduje příslušný zdroj požadavku (typicky uživatel).
  - **Proveditelnost** – každý požadavek se musí dát realizovat v rámci známých možností a příslušných omezení systému a jeho provozního prostředí. Jedna možnost jak zajistit proveditelnost je přítomnost softwarového vývojáře při sběru požadavků, další možnost je inkrementální vývoj.
  - **Nepostradatelnost** – každý požadavek by měl představovat funkci, kterou si opravdu vyžádal zadavatel, či byla vynucena vnějšími podmínkami. Pomůže také se pokusit přiřadit obchodní (či uživatelský), odkud daný požadavek pochází.
  - **Priorita** – každému funkčnímu požadavku je přiřadí priorita, která ukazuje jak je daný požadavek důležitý. Na základě této priority lze pak sestavit kvalitní časový plán.
  - **Jednoznačnost** – daný požadavek by měl mít jednoznačný význam. Požadavky je dobré popisovat jednoduchým, srozumitelným jazykem.
  - **Ověřitelnost** – správná implementace každého požadavku by měla být prověřena nějakým testem. Když ze požadavek nedá ověřit, tak jeho implementace není objektivní analýzy, ale náhody. Ověřit se nedají neúplné, nejednotné, nejednoznačné a neproveditelné požadavky [Drabick 1999].
- Vlastnosti celé specifikace
  - **Úplnost** – v specifikaci by neměli žádné požadavky chybět ale ani žádný požadavek přebývat. Žádný požadavek se také nemůže vyskytnou vícekrát.
  - **Bezrozpornost** – požadavky by si neměli na stejné úrovní vzájemně odporovat, ani odporovat požadavkům na úrovni vyšší (tj. například obchodním požadavkům).

- **Přizpůsobitelnost** – specifikaci musí být možno v případě potřeby přepsat – tj. musíme vést historii změn požadavků. To znamená že, každý požadavek musí být přesně identifikovaný.
- **Dohledatelnost** (traceability) – u dohledatelného požadavku se dá vyhledat jeho zdroj, návrh, příslušná implementace i testovací scénář. Více o dohledatelnosti a o technikách k jejímu zabezpečení v části 4.5

4.4 Metody zachycení požadavků

Pro získání uživatelských požadavků se mohou použít tyto metody:

- interview,
- JAD (Join Application Development) session [Wood 1995],
- prototypování uživatelského rozhranní.

Uživatelské požadavky lze zachytit v těchto strukturách:

- kontextový diagram [Rumbaugh et al. 1998]
- případy užití [Rumbaugh et al. 1998]
- CRC modelování [Ambler 1998],
- product backlog – metoda SCRUM [Schwaber at al. 2001].
- user stories (metodika Extrémní programování) [Beck 2002a] a dalších.

Více např. v [Kano 1994] .

4.5 Dohledatelnost požadavků

Malé změny požadavků mají často velké důsledky, které vyžadují úpravy velké části systému. Abychom byli schopni dohledat všechny části systému postižené změnou, musíme mít zaznamenáno souvislost požadavků s ostatními částmi systému (např. s podnikatelskými pravidly, s částmi architektury a návrhu, s moduly zdrojového kódu, s testovacími scénáři).

Sledování požadavků zabezpečuje a dokumentuje, že životní cyklus je dodržován [Baldwin 2001, Antoniol et al 2002]. Je to z těchto příčin:

- demonstruje vztah mezi návrhovými vstupy a návrhovými výstupy.
- zabezpečuje, že je návrh založen na předchůdci, který je podložen nějakým požadavkem.

- zabezpečuje že návrhová specifikace je ověřená a funkční požadavky ověřené.

4.5.1 Sledování požadavků

Vztahy mezi požadavky a z nich vzniklými pracovními výstupy sledujeme pomocí vzájemných vztahů - odkazů. Díky nim můžeme sledovat celý životní cyklus požadavku, od jeho vzniku až po implementaci [Gotel 1994] a nasazení. Takto dostaneme možnost dohledat u každého výstupu jeho zdroje vzniku (tedy původní požadavky). Dohledatelnost požaduje, aby byl každý požadavek jednoznačně identifikován, aby se na něj dalo, pomocí odkazů, jednoznačně odkazovat v průběhu celého projektu.

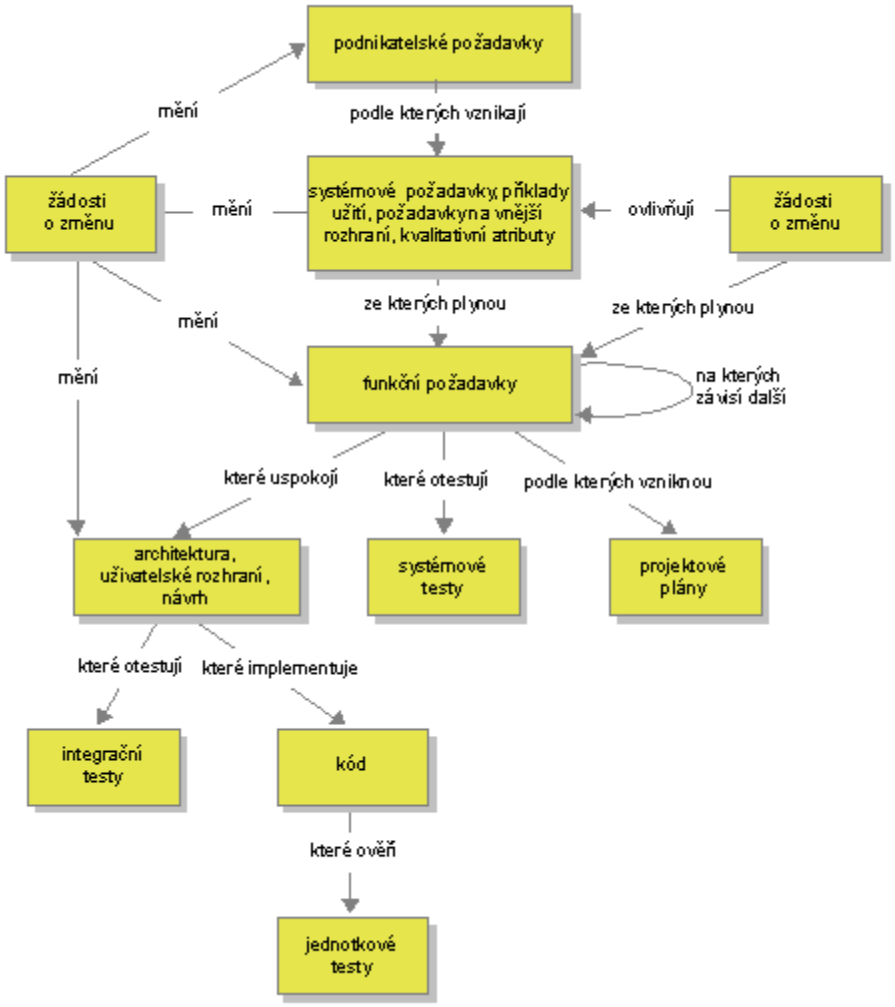
Odkazy se dají rozdělit obecně do čtyř skupin [Jarke 1998]. Ty jsou znázorněny na obrázku Obr. 12. Odkazy tedy mohou být:

- Odkazy od zákazníka k požadavkům – podle těchto odkazů lze poznat, které požadavky se musí změnit při změně uživatelských potřeb. Déle lze podle nich poznat, jestli jsme nezapomněli na nějakou zákazníkovu potřebu.
- Odkazy od požadavků k zákazníkovi – vedou opačným směrem. Lze podle nich identifikovat zákazníka, od kterého požadavky přišly.
- Odkazy o požadavků k výstupům – tyto odkazy identifikují výstupy vzniklé na základě uživatelských požadavků. Podle těchto odkazů lze identifikovat, zda jsme nezapomněli na implementaci nějakého uživatelského požadavku.
- Odkazy od výstupů k požadavkům – tyto odkazy identifikují z jakého zdroje (tedy proč) daná část systému (modelu) vznikla.

Obr. 12 – Čtyři druhy odkazů [Jarke 1998]

Odkazy pomáhají najít zdroje požadavků a jejich vzájemné vztahy a závislosti. Je z nich dobře vidět, kudy se budou šířit změny po úpravě nebo smazání nějakého uživatelského požadavku.

Některé typy odkazů, které se mohou vyskytnou v projektu jsou znázorněny na obrázku Obr. 13.



Obr. 13 – Některé používané odkazy [Wiegers 2003]

4.5.2 *Výhody dohledatelnosti uživatelských požadavků*

Dohledatelnost požadavků nabízí tyto výhody [Wiegers 2003]:

- **Certifikace** – dohledatelnost požadavků se hodí při bezpečnostní certifikaci systému, protože díky ní můžeme ukázat, že všechny požadavky byly skutečně implementovány.
- **Analýza dopadu** – bez odkazů mezi požadavky máme velkou šanci přehlédnout nějakou část systému, na kterou bude mít analyzovaná změna vliv.
- **Údržba** – dobrá dohledatelnost usnadňuje správné provádění změn při údržbě systému což zvyšuje produktivitu .

- **Sledování projektu** – když se bude při vývoj pečlivě zaznamenávat všechny odkazy, tak bude k dispozici přesný ukazatel průběhu implementačních prací. Chybějící odkazy zaznamenají, že daný výstup ještě není hotový.
- **Úpravy starších systémů** – při úpravě starších systémů je možné si nechat vypsat ty jejich části, které jsou pokryty funkcemi novéno systému.
- **Recyklování** – odkazy usnadňují opakované použití komponent systému, protože spojují související požadavky, návrh, zdrojové kódy a testy.
- **Snižování rizika** – dokumentace závislosti mezi komponentami systému zmenšuje problémy, jež by mohly nastat, když nějaký z klíčových členů s nepostradatelnými znalostmi opustí projekt.
- **Testování** – když nějaký test skončí neúspěšně, tak jsou podle odkazů snadno dohledatelné ke zdrojovým kódům příslušející požadavky. Když jsou spojeny testy s požadavky, tak se je možno vyhnout nadbytečným testům.

4.5.3 Způsoby zachycení dohledatelnosti požadavků

Pro zachycení dohledatelnosti požadavků se používají dva typy technik [Pinheiro 2000]:

- Techniky založené na matici dohledatelnosti požadavků (Requirement Traceability Matrix) [Sommerswille 1997] – více následující části (4.5.4).
- Techniky založené na křížových referencích a indexech. Při této technice se vytváří reference mezi několika artefakty pro indikaci vztahů mezi nimi, nebo se vytváří seznam obsahující příslušné artefakty [Smith 1993].

4.5.4 Matice dohledatelnosti požadavků

Nejobvyklejší způsob zachycení odkazů mezi požadavky a ostatními částmi systému je matice dohledatelnosti požadavků (Requirement Traceability Matrix).

Nejobvyklejší typ [Sommerswille 1997] matice dohledatelnosti požadavků se vytvoří tak, že si vytvoříme tabulku, kde ve sloupcích budou postupně uživatelský požadavek (případně jen jeho jednoznačný identifikátor), dále odkaz na odpovídající funkční požadavek, v dalším sloupci

budeme evidovat, v jaké části kódu či modulu je požadavek implementován a dále přiřadíme příslušné testy*.

Tuto tabulku následně naplníme ze specifikace seznamem všech uživatelských a k nim příslušejících funkčních požadavků. Dále tuto tabulku vyplňujeme průběžně během vývoje.

Nefunkční požadavky, jako například požadavek na výkonnost nebo kvalitu, se nemusí týkat žádného kódu a tak se nemusí v této tabulce objevit. Existují však i nefunkční požadavky (například na bezpečnost) které se budou implementovat a tak budou zaznamenány i v této tabulce. Příklad tohoto typu matice (tabulky) dohledatelnosti je v tabulce Tab. 3 .

Odkazy (traceability links) mezi částmi návrhu mohou obecně být vztahu 1:1, 1:N i N:M. Příklady těchto možných kardinalit odkazů jsou[Wiegers 2003]:

- 1:1 – jeden element návrhu je implementován jedním modulem kódu.
- 1:N – jeden funkční požadavek je verifikován pomocí několika testů.
- M:N – každý případ užití vede na několik funkčních požadavků a jeden funkční požadavek odpovídá několika případům užití. Ideální by bylo, zachytit všechna spojení mezi elementy modelu, ale v praxi vztahy M:N vedou k složitému výsledku a je těžké je zachytit v plné šíři.

Uživatelský požadavek	**Funkční požadavek**	**Část návrhu**	**Modul Kódu**	**Testovací scénář**
UP-28	katalog.dotaz.třídění	třída Katalog	katalog.setřiď()	hledání 7, 8
UP-29	katalog.dotaz.import	třída Katalog	katalog.importuj() katalog.zkontroluj()	hledání 12,13 hledání 14

Tab. 3 – Jeden typ tabulky dohledatelnosti požadavků [Wiegers 2003]

* Toto je jen příklad, co by tak v tabulce dohledatelnosti požadavků mohlo být. V této tabulce můžeme sledovat artefakty jaké budeme potřebovat - například odkaz do dokumentace, na online nápovědu, na hardware na kterém je funkčnost provozována atd. Více sledovaných věcí však znamená více práce, lae záznamy nám mohou ušetřit čas a práci při úpravě systému.

Odkazy se dají řešit i skupinou obousměrných matic, ve kterých jsou zapsané závislosti mezi dvojicemi výstupů (viz tabulka Tab. 4), například mezi dvěma požadavky stejného (různého) typu, požadavky a testovacími scénáři a podobně.

Pomocí těchto matic se dají dobře reprezentovat vztahy mezi dvojicemi požadavků [Sommerswille 1997].

Funkční požadavek	**Případ užití**			
	PU-1	PU-2	PU-3	PU-4
FP-1	•			
FP-2	•			
FP-3			•	
FP-4			•	
FP-5		•		•
FP-6			•	

Tab. 4 – Obousměrná matice dohledatelnosti požadavků

5 Transformace

Tato kapitola popisuje obecné vlastnosti transformací, jejich klasifikaci, charakteristiky i mechanizmy použité k jejich modelování. Dále je v této kapitole popsán jeden ze způsobů použití transformací při tvorbě softwaru – MDE (Model Driven Engineering) a jeden konkrétní přístup, využívající jeho myšlenek – MDA (Model Driven Architecture).

5.1 Co transformovat

Hlavní otázkou u transformace je „Co má být transformováno a na co?“. Odpověď na tuto otázku záleží na zdrojovém a cílovém artefaktu. Protože se pohybujeme v oblasti softwarového inženýrství, tak budeme transformovat program nebo model.

Můžeme uvést tuto definici transformace modelu [Kleppe el al. 2003]:

> *Transformace modelu je generování cílového modelu ze zdrojového modelu podle definice transformace.* ***Definice transformace*** *je množina transformačních pravidel, která popisují jak model v zdrojovém jazyku může být transformován do modelu v cílovém jazyku.* ***Transformační pravidlo*** *je popis, jak se jeden nebo více konstruktů zdrojového jazyka transformuje do jednoho nebo více konstruktů cílového jazyka.*

5.2 Klasifikace transformací

Transformace lze obecně klasifikovat podle těchto hledisek [Mens et al. 2005]:

- Endogenní versus exogenní.
- Horizontální versus vertikální.

Toto rozdělení je ukázáno v tabulce Tab. 5.

	horizontální	**vertikální**
endogenní	Refactoring	Formal refinement
exogenní	Jazyková migrace	Generování kódu

Tab. 5 – Dimenze transformace modelu

5.2.1 Exogenní versus endogenní transformace

Pro transformaci musí být modely vyjádřeny i v nějakém modelovacím jazyku (například UML pro návrhové modely, programovací jazyk pro model zdrojového kódu). Syntaxe a sémantika modelovacího jazyku je vyjádřena pomocí metamodelu.

Na základě toho, zda zdrojový a cílový metamodel je stejný, tj. zda oba dva modely jsou vyjádřeny pomocí stejného modelovacího jazyka, rozlišujeme.

- **Endogenní transformaci**. U endogenní transformace jsou zdrojový i cílový modelovací jazyk stejné – provádíme transformaci v rámci jednoho metamodelu. V [Visser et al. 2004] je tento typ transformace nazván přeformulování (rephrasing). Typické příklady endogenní transformace například jsou:
  - Optimalizace – transformuji s cílem zvýšit kvalitu (typicky výkonu), při zachování sémantiky (typicky programovacího jazyka).
  - Refactoring – je to změna vnitřní struktury pro zvýšení kvalitativních charakteristik (srozumitelnost, modifikovatelnost, znovupoužitelnost, modularita, adaptabilita) bez viditelné změny chování [Fowler 1999].
  - Zjednodušení a normalizace – se používá ke snížení syntaktické komplexnosti, tj. pro překlad syntaktických „složitostí“ primitivních konstrukcí.
- **Exogenní transformaci**. U endogenní transformace jsou zdrojový i cílový modelovací jazyk různé – provádíme transformaci mezi dvěma metamodely. V [Visser et al. 2004] je tento typ transformace nazván překlad. Typické příklady exogenní transformace například jsou:
  - Syntéza – je „překlad“ abstraktnější specifikace (například analytický nebo návrhový model) do konkrétnější (například zdrojový kód v nějakém programovacím jazyce). Typický příklad syntézy je generování kódu, kde zdrojový kód je překládán do spustitelného kódu, nebo přístup, kdy překládáme návrhový model do spustitelného kódu.
  - Reverzní inženýrství - je opak syntézy ze specifičtější formy vytváříme abstraktnější formu.
  - Migrace – z jednoho programovacího jazyka na jiný při zachování té samé úrovně abstrakce.

5.2.2 *Horizontální versus vertikální transformace*

Dále lze transformace klasifikovat podle toho je zdrojový i cílový model na stejné úrovni abstrakce:

- **Vertikální transformace** – je transformace, kde zdrojový i cílový model jsou na různé úrovni abstrakce. Příkladem je generování kódu z návrhového modelu, případně „zdokonalování“ (refinement), kde postupně zdokonalujeme specifikaci do implementace ([Wirth 1973] nebo [Back 1998]).
- **Horizontální transformace** – je transformace, kde zdrojový i cílový model jsou na stejné úrovni abstrakce. Typický příklad je refaktoring.

5.3 *Charakteristiky transformace*

Transformace lze charakterizovat pomocí:

- **Úrovně automatizace.** Lze rozlišovat transformace, které lze provést automaticky a transformace, které musí být provedeny manuálně. Příkladem manuální transformace je transformace z dokumentu zachycujícího požadavky do analytického modelu. Tato transformace se musí provést ručně, abychom vyřešili mnohoznačnost, nekompletnost a nekonzistenci v požadavcích, které jsou vyjádřeny pomocí textu.
- **Komplexnost transformace.** Některé transformace jsou malé, jako třeba refaktoring, jiné jsou mnohem složitější. Příkladem složitějších transformací jsou například parsery, kompilátory generátory kódu. Rozdíly mezi jednoduchými a složitými transformacemi jsou veliké a tak musíme používat jiné techniky a nástroje pro každou z nich.
- **Uchování informace.** Existuje mnoho různých typů transformace. Každá zachovává jiné aspekty zdrojového modelu v cílovém. Zachované vlastnosti se liší podle druhu aplikované transformace. Například při refaktoringu

5.4 *Mechanizmy použitelné při modelování transformace*

Mechanizmus je zde použit ve velmi široké významu. Mechanizmem může být technika, jazyk, metoda, atd. Pro specifikování a vykonání transformace, se může použít jakékoliv programovací paradigma (procedurální, funkcionální, logické, objektové atd.).

Hlavní rozdíl mezi transformačními mechanizmy je však ten, jestli se k popisu transformace použije imperativní nebo deklarativní přístup [Mens et al. 2005]. Mechanizmus transformace tedy může být:

- **Imperativní** – popisujeme *jak* transformaci máme dělat. V praxi popisujeme algoritmus transformace v nějakém (imperativním) programovacím jazyce.
- **Deklarativní** – popisujeme ne jak, ale co máme dělat. Deklarativní přístup zahrnuje (ne jenom) funkcionální programování (např. [Gerber et al. 2002]), logické programování (např. [Courcelle 1990]), grafové transformace (např. [Kuske 2000]).

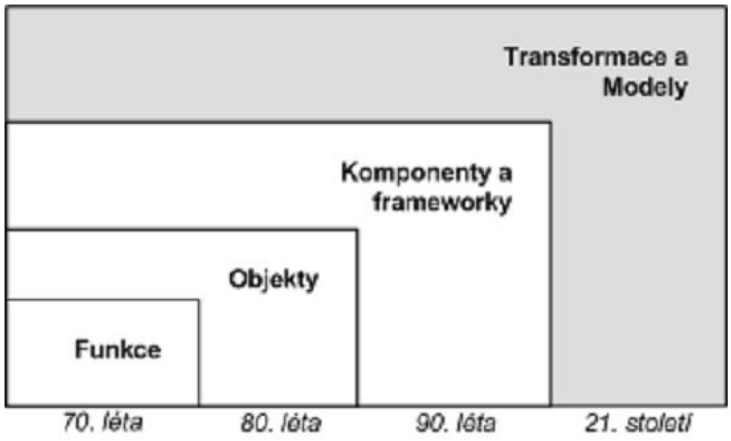
Z teoretického hlediska pohledu se zdá deklarativní přístup nadějnější – je založen na formálních základech, je obousměrný a nabízí jednodušší sémantický model pro specifikaci transformace modelu. Například pořadí aplikovaných pravidel je implicitní a procházení vstupního modelu a následné vytváření výsledného modelu je také implicitní. Toto umožňuje udržovat předpis transformace menší, přehlednější a lépe udržovatelný. Na druhou stranu imperativní přístup je vhodnější při rozhodování, jaká transformace má být použita při inkrementálních změnách modelu. Díky vlastnostem imperativního programování (explicitní stav, sekvence příkazů, iterace apod.) lze lépe řídit pořadí aplikování transformací z jejich množiny [Mens et al. 2005].

Deklarativní přístup zahrnuje:

- **Funkcionální programování.** Funkcionální přístup k transformacím je velmi přímočarý, protože každou transformaci si lze představit jako funkci, která převádí vstup (zdrojový model) na nějaký výstup (cílový model). Ve většině funkcionálních jazyků, lze s transformací nakládat také jako s modelem (tj. transformovat transformace). Nevýhodou funkcionálního přístupu je nešikovná manipulace se stavem transformace, v průběhu jejího vykonávání.
- **Logické programování.** Jazyky, pro logické programování, zahrnují mnoho užitečných vlastností pro transformace: backtracking, omezující podmínky [Surynek 2009], obousměrnost pravidel a pod. Navíc tyto jazyky obsahují nějaký dotazovací mechanizmus a tak není potřeba implementovat speciální dotazovací jazyk.
- **Grafové transformace.** Grafové transformace jsou techniky (a k nim příslušné formalizmy), které jsou přímo aplikovatelné pro transformaci modelů ([Mens at al. 2002, Gerber at al. 2008]). Mají mnoho výhod: vizuální notaci, jsou formalizované, nabízejí mechanizmus, jak sloučit jednodušší transformace [Kuske 2000]. Nevýhodou je, že různé techniky používané ke grafovým transformacím nemusí být spolu kompatibilní.

5.5 MDE – Model Driven Engineering

Jeden ze způsobů použití transformací při procesu tvorby softwaru je modelem řízené inženýrství (Model Driven Engineering – MDE). Základem MDE je soustředit se na vytváření modelů, či abstrakcí, které vysvětlují části systému. MDE klade hlavním pozornost na model, z kterého jsou následně generovány další výstupy (jako například zdrojový kód, dokumentace či další model). MDE představuje způsob, jak se vypořádat s rostoucí složitostí vývoje softwaru. MDE vede ke zvýšení znovupoužitelnosti (Obr. 14) a tím k zvýšení efektivity.



Obr. 14 – Vývoj znovupoužitelnosti v IT

MDE kombinuje následující přístupy [Smith 2006]:

- Doménově specifické modelovací jazyky (Domain specific modeling language [Kelly et al. 2008]) – pomocí kterých specifikuji strukturu, chování a požadavky jazykem příslušné domény (např. tvorba softwaru, finanční služby, řízení avioniky, řízení skladovacích služeb atd.).
- Transformační mechanizmy, které analyzují aspekty modelu a následně z modelu vytvoří požadovaný výstup (zdrojový kód, simulační model atd.)

Pro provádění transformací jsou k dispozici (kromě jiných) tyto transformační jazyky, postupy a nástroje:

- QVT [OMG 2008] – transformační jazyk standardizovaný v rámci MDA konsorciem OMG. Více o něm v části 5.6.1.
- Fujaba [Fujaba] – open source CASE nástroj podporující MDE, který obsahuje transformační jazyk založený na grafových transformacích.

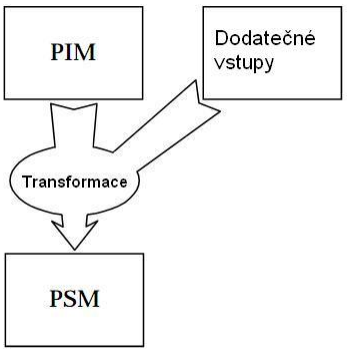
- GReAT (Graph Rewriting and Transformation Tool) [Agrawal 2003] – je to transformační jazyk založený na grafových transformacích, který je k dispozici v prostředí GME (Generic Modeling Enviroment).
- Tefkat [Hibbert at al 2007] – deklarativní transformační jazyk a engine založený na F-logice. Dostupný v prostředí Elipse.
- Kermeta [Kermeta] – je modelovací a programovací jazyk, jehož základní charakteristiky jsou: imperativní i funkcionální, aspektově a objektově orientovaný, založený na principech design-by-contract. Dostupný v prostředí Elipse.
- a mnoho dalších. Například lze použít (téměř všechny) standardní programovací jazyky, pokud mají přístup do repozitáře CASE nástroje, (Java, Prolog, Lisp, atd.), nebo jazyky zabudované v CASE nástrojích (např. C.C v Craft.CASE [Merunka 2006]), nebo jazyky pro manipulaci s XML[*] (XQuery + XSLT), nebo i nástroje z oblasti umělé inteligence (STRIPS [Nouza 2009]) atd.

5.6 MDA – Model Driven Architecture

MDA (Model Driven Architecture [OMG 2003]) je přístup k tvorbě informačních systémů, založený na myšlenkách MDE, vytvářený konsorciem OMG a založený na jeho standardech[†].

Základní myšlenkou je postupná transformace z modelu s vyšší úrovní abstrakce na model na nižší úrovni abstrakce. Na obrázku Obr. 15 je ukázána obecná transformace z platformově nezávislého modelu (PIM) na platformově specifický model (PSM). Jako jednoduchý konkrétní případ lze uvést transformaci z konceptuálního na implementační třídní diagram (složitější příklad je na Obr. 17).

[*] Repositáře CASE nástrojů, jsou často přístupné pomocí XML rozhraní (např. pomocí XMI u UML)
[†] MDA je založeno na těchto standardech OMG – UML (Unified Modeling Language), MOF (Meta Object Facility), QVT (Query/View/Transform), OCL (Object Constraint Language), CMW (Common Data Warehouse) aj. (viz [OMG 2003,2006,2007, 2008, 2009, 2010])

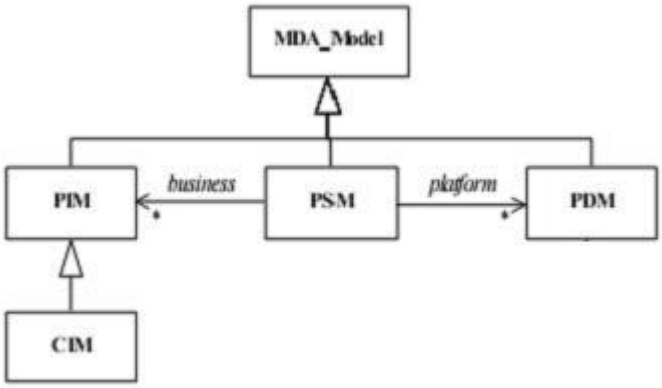


Obr. 15 – Transformace v MDA [OMG 2003]

V rámci MDA se uplatňují tyto modely [OMG 2003]:

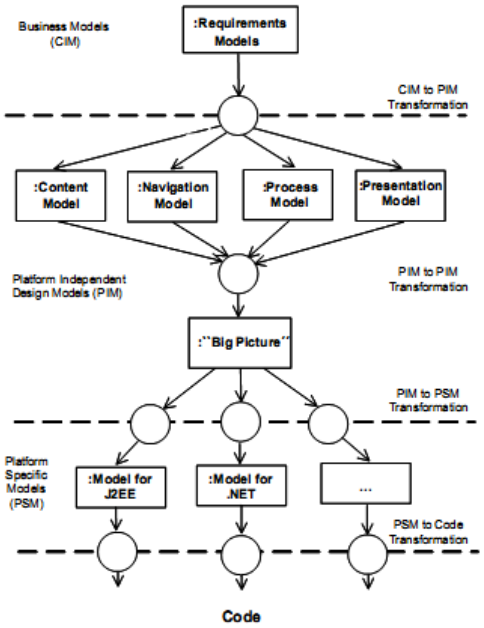
- **Výpočetně nezávislý model** (Computational Independent Model – CIM). Tento model se také označuje za doménový. Zaměřuje se na obecné požadavky na systém a jeho okolí. Reflektuje zákaznické („business“) požadavky a pomáhá popisovat, co od systému zákazník očekává. Součástí CIM může být: hierarchická struktura zaměstnanců, business proces objednávky popsaný v BPML, atd.
- **Platformově nezávislý model** (Platform Independent Model – PIM). Tento model je pohledem na systém abstrahující od konkrétní platformy, implementačních detailů a nasazení. Popisuje chování a strukturu jen v těch mezích, které zaručují přenositelnost mezi platformami. Hraje hlavní roli v přenositelnosti (tj. jeden PIM pro objednávku pro webovou aplikaci, mobilní atd.)
- **Platformově specifický model** (Platform Specific Model – PSM). PSM je pohled na systém s ohledem na konkrétní platformu, na které bude realizován. Finální PSM by měl poskytnout maximum informací pro vytvoření (vygenerování) cílového kódu aplikace.
- **Definiční model platformy** (Platform Definition Model – PDM) – PDM obsahuje popis cílové platformy, který je využit při transformaci PIM do PSM.

Tyto modely, a jejich vzájemné vztahy, používané v MDA jsou znázorněny na Obr. 16.



Obr. 16 – Metamodel MDA [Bezivin 2005]

Obecně, jakákoliv transformace (CIM→PIM, PIM→PSM) může proběhnout pomocí několika (pod)transformací (např. PIM→PIM) – viz Obr. 17, kde jsou znázorněny postupně tyto transformace CIM→PIM_1→PIM_2→PSM→zdrojový kód.



Obr. 17 – Příklad použití MDA při tvorbě web. aplikace [Koch et al. 2006]

Obecné přístupy, jakým způsobem provést transformaci z PIM do PSM, jsou podle [OMG 2003] tyto:

1. Označkování modelu.
2. Transformace metamodelu.
3. Transformace modelu.
4. Aplikace vzorů.

5. Sloučení modelů.

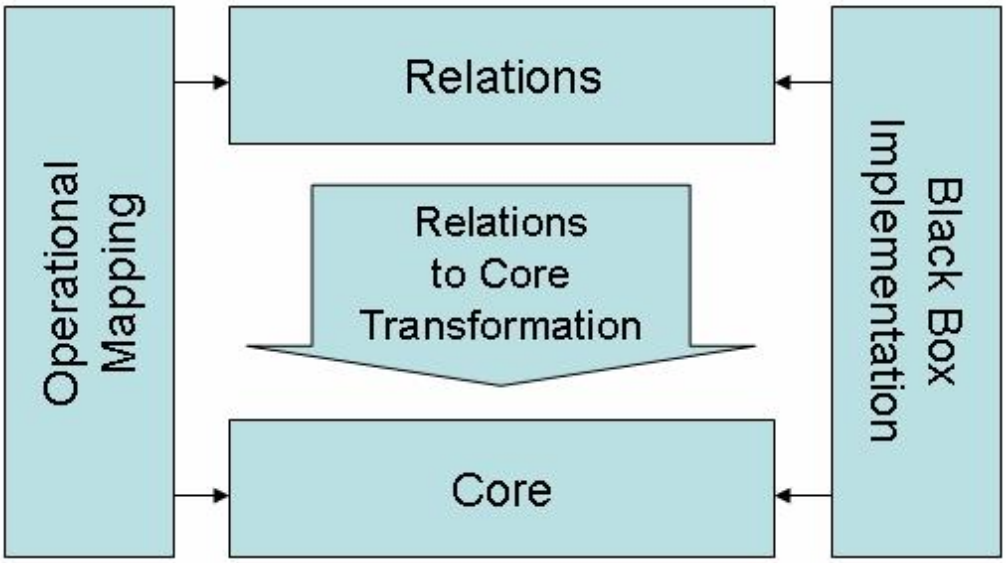
V praxi se častokrát používají kombinace těchto přístupů (transformace PIM pomocí profilu, transformace s použitím aplikování vzorů a značek, přídadně i manuální transformace[*]). Vstupem do transformace mohou být např. vzory, technické rozhodnutí, či kvalitativní požadavky.

Pro vlastní předpis transformace vytvořilo sdružení OMG standard QVT (Query/View/Transform [OMG 2008]).

5.6.1 Transformační jazyk QVT

QCT (Query/View/Transform [OMG 2008]) představuje standard sdružení OMG pro manipulace s modely a jejich transformace v rámci architektury MDA. QVT je schopno fungovat nad všemi modely definovanými pomocí MOF.

Struktura QVT je znázorněna na Obr. 18 .



Obr. 18 – Struktura QVT

QVT ve skutečnosti představuje několik doménově specifických jazyků a mechanizmů:

[*] Manuální transformace se příliš neliší od toho, jak se již dlouho „správně“ navrhuje software. MDA pouze přidává explicitní rozlišení mezi PIM a PSM, a záznam transformace.

- jazyk **Relations** – umožňuje vytvářet deklarativní specifikace relací (vztahů) mezi modely založenými na MOF. Tento jazyk umožňuje komplexní vyhledávání si odpovídajících vzorů (pattern matching) a implicitně vytváří záznam (trasovací třídy a jejich instance) o tom, co se děje během transformace.
- jazyk/model **Core** – je malý a jednoduchý jazyk, který pouze podporuje pattern matching na množině proměnných pomocí vyhodnocování podmínek přes všechny prvky modelu. Je stejně mocný jako Relations jazyk, Díky jeho relativní jednoduchosti může být jeho sémantika definována mnohem jednodušeji, ale transformace jím vyjádřené budou vyjádřeny delším programem, než u jazyka Relations. V rámci QVT je definováno jednoznačné mapování mezi jazyky Core a Relations – čili transformace napsaná pomocí Relations se může přeložit do Core a pak se následně provede.
- jazyk **Operational Mapping** – tento jazyk představuje cestu, jak standardně imperativně provést transformaci mezi modely. Tento jazyk představuje rozšířené OCL s vedlejšími efekty, které umožňují procedurální zápis transformace. Případně je možné „míchat“ deklarativní a imperativní způsob transformací dohromady a implementovat některé, deklarativně špatně vyjádřitelné, relace imperativně.
- **Black Box Implementation** – specifikace QVT umožňuje definovat některá MOF operace jako plug-in modul v některém jiném programovacím jazyce se všemi výhodami (větší výkon, použití již vytvořených specializovaných knihoven, …) i nevýhodami (přímý přístup k modelu a tím větší možnost udělat chybu, nutnost implementovat alespoň část jazyka Relations, …).

OMG poskytuje specifikací QVT pouze standard a implementaci nechává na ostatních.

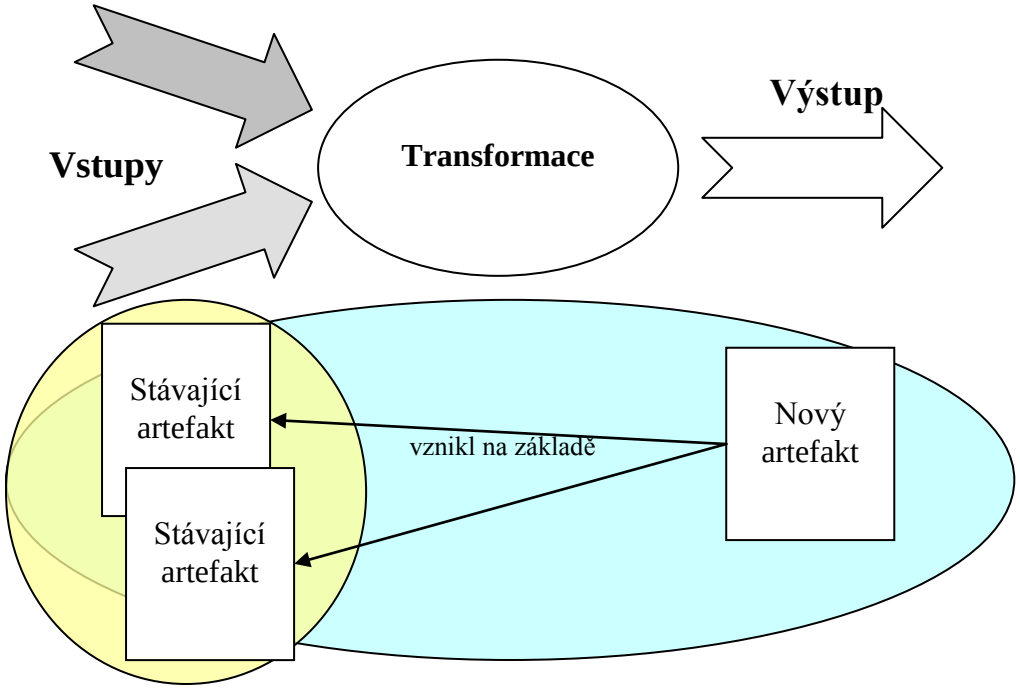
6 Metoda postupných transformací prvků

Cílem kapitoly je definovat základy metody postupných transformací prvků, která je založena na postupném přidávání nových prvků do již existujícího modelu informačního systému na základě již existujících prvků. Cílem kapitoly je zavést základní pravidla metody postupných transformací, diskutovat její vlastnosti, definovat způsob zachycení provedených transformací a jejího použití při tvorbě modelu systému.

6.1 Motivace

Základem metody postupných transformací prvků je způsob, jak vývojář postupně tvoří model informačního systému. Přidává vždy pouze (v jednom okamžiku) jeden nový artefakt – prvek (objekt či vztah) do vytvářeného modelu. Tento nový prvek (třídu, metodu, diagram, use-case, generalizaci atd.) tam přidává, v ideálním případě, pouze na základě rozhodnutí, vycházejícího z prvků již v modelu obsažených.

Každý prvek v modelu tedy vzniká na základě těch prvků, které jsou již v modelu obsaženy. Model by tedy měl obsahovat všechny informace, na základě kterých do něj přidáme nový prvek. Samozřejmě potřebujeme nějaké vstupy zvnějšku do modelu. Potřebujeme nějaké pevně dané prvky, na základě kterých pak vytváříme celý model. Tyto vstupní prvky jsou zadáním – požadavky na modelovaný systém. Můžeme tedy říci, že postupně *transformujeme* stávající model na nový model přidáním nového prvku (viz Obr. 19).



Obr. 19 – Transformace vstupních artefaktů na výstupní

Každý nový artefakt modelu tedy vznikne právě jednou transformací a v modelu bude mít prvky-předky na základě kterých vzniknul. Pokud bude v modelu existovat nějaký prvek, který nebude mít žádného předka, tak je buď nějakým požadavkem na systém, nebo byl do modelu přidán chybně (tj. prvek nevznikl žádnou transformací).

Vybrat a provést transformaci a tím přidat do modelu nový prvek musí vývojář. Transformaci provede na základě svých zkušeností, pravidel pro tvorbu modelu (tj. například nějaké metody tvorby modelu). Při tvorbě modelu má samozřejmě vývojář velký stupeň volnosti[*], tak také bude přesná transformace a jí přidaný prvek do modelu záviset přímo na něm (jeho zkušenostech, zvycích, atd.).

Pro zachycení jednotlivých transformací, tj. pro „záznam“ činnosti vývojáře (a tím zaznamenání a zdokumentování procesu tvorby informačního systému), lze použít diagram, ve kterém budeme zaznamenávat nové prvky a jejich vztahy z prvky na základě nichž vznikl. U každého nově vzniklého prvku si poznamenáme z kterých prvků vzniknul. V tomto diagramu tedy bude zachycen vztah (nový) prvek vznikl transformací *na základě* jiných prvků.

[*] Vlastně téměř jediné omezení je, aby nově přidaný prvek nešel proti požadavkům a posunul model směrem k cíli – hotovému modelu, plnícímu požadavky.

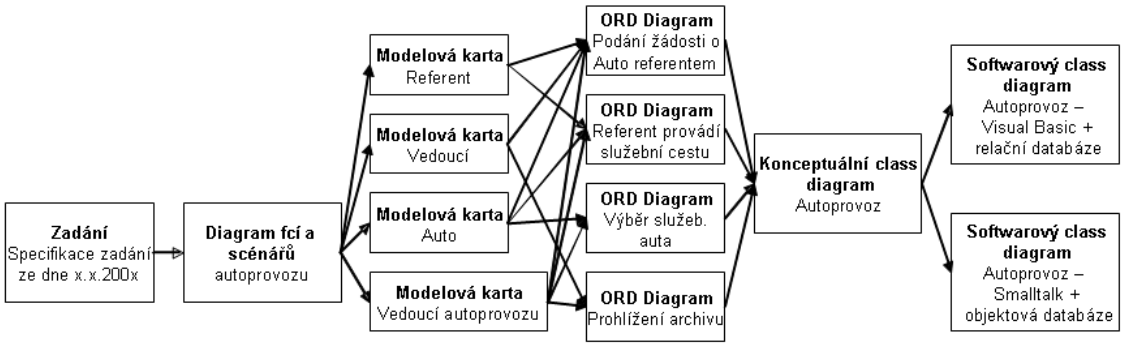
Pokud takovýmto způsobem vznikne model, tak budeme vědět u každého prvku na základě kterých vzniknul a tím i na základě kterých požadavků je tento prvek v modelu. Tak zajistím dohledatelnost požadavků (více kapitola 4.5 – [Antoniol et. al 2002]) u každého prvku v modelu.

6.1.1 Motivační příklad

V následujícím příkladu (a také téměř ve všech příkladech v této kapitole) budou použity části projektu, týkající se provozu autoparku společnosti, která půjčuje referentům na jednotlivé pracovní cesty automobily ze svého autoparku. Pracovní cesta podléhá schválení vedoucím a konkrétní automobily přiděluje na tyto cesty vedoucí autoparku. Je použita metoda BORM (Business Object Relation Modeling).. Podrobněji o BORMu v [Merunka 2003] nebo v tomto textu v části Příloha A.

Na Obr. 20 je zachycen konkrétní záznam procesu tvorby projektu autoprovozu. Našimi sledovanými prvky budou výstupy jednotlivých pracovních postupů BORMu[*]. U každého nového prvku si zaznamenáme (a zakreslíme do diagramu), na základě kterých prvků vzniknul. Nejdříve vzniknul zadávací dokument (pro nás je to vstupní prvek), z kterého je následně odvozen diagram, zachycující vztahy mezi funkcemi a scénáři. Následně na základě tohoto diagramu vznikly modelové karty, zachycující informace o jednotlivých účastnících provozu autoprovozu (zde konkrétně Referent, Vedoucí, Auto, Vedoucí autoprovozu). Participanti (modelovaní pomocí modelové karty) se účastní procesů. Ty jsou zachyceny pomocí procesních object-relationship diagramů (ORD). Z těchto diagramů je následně odvozen konceptuální třídní diagram autoprovozu, zachycující konceptuální třídy a vztahy mezi nimi. A nakonec z tohoto konceptuálního diagramu jsou odvozeny softwarové – implementační, třídní diagramy, zde konkrétně pro implementaci pomocí relační a objektové databáze.

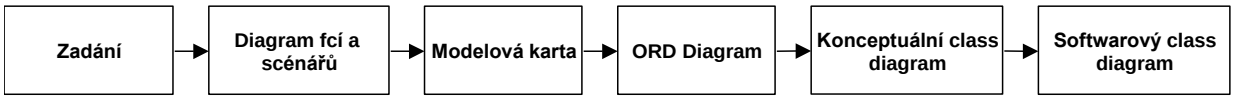
[*] Tento motivační příklad slouží pouze k naznačení principů a proto jsem zvolil pohled z méně podrobnější pohled – sledovanými artefakty tedy nejsou ty nejmenší prvky přidávané do modelu (tedy funkce, scénáře, stavy, přechody …), ale větší celky – modelové karty, diagramy, …



Obr. 20 – Záznam procesu tvorby autoprovozu

Tento příklad je velmi zjednodušený, a ve skutečnosti zadání – zejména nefunkční požadavky – budou ovlivňovat každý nově přidaný artefakt do modelu.

Když se podrobněji podíváme na obrázek Obr. 20, tak vidíme jisté pravidelnosti a vzory. Vidíme, že diagram funkcí a scénářů v metodě BORM vzniká ze zadání, modelové karty z diagramu funkcí a scénářů atd. Tyto pravidelnosti jsou dané použitou metodou návrhu modelu. Tyto vzory případně můžeme vyjádřit pomocí diagramu (zde na Obr. 21), v kterém vyjádříme pouze „dovolené“ typy transformací. Takto jsme schopni zachytit vztahy v příslušné metodě návrhu softwaru[*].



Obr. 21 – zjednodušené zachycení transformací v metodě BORM

6.1.2 Pohled na model

Pohled zachycený na Obr. 20, tj. pohled, kdy u každého výstupu (typ dokument, diagram atd.) dnes dokážeme zachytit pomocí matice dohledatelnosti požadavků (viz část 4.5.4).

Pro podrobný záznam průběhu procesu vývoje (a z toho vyplívajících výhod, jako je například dohledatelnost požadavků u každého prvku modelu) potřebujeme sledovat mnohem jemnější části modelu než jsou dokumenty a diagramy. Potřebujeme pracovat na úrovni vlastních nejmenších

[*] Tomuto zachycení dovolených typů transformací v metodě návrhu modelu se podrobně věnuje následující kapitola této práce.

prvků modelu – objektů, tříd, asociací atd. Pak získáme podrobnou informaci o vzniku jednotlivých pojmů, kterou pak lze použít k dohledatelnosti.

6.2 Zavedení pojmů

Nejdříve si musíme zavést významy nových pojmů použitých v této a následujících kapitolách. Jsou to pojmy:

Česky **Anglicky**	**Termín**	**Příklad**
Prvek *element*	Reprezentuje konkrétní, dále již nedělitelnou, část modelu informačního systému. Podobně je zaveden pojem prvek v definici jazyka UML viz. [OMG 2001]. Prvek je uložen v repositáři modelu. Prvky mají obvykle nějakou grafickou podobu obsaženou v diagramech používaných při modelování.Nový prvek v modelu vzniká pomocí transformace z již existujících prvků modelu. Prvek může mít atributy. Typicky je prvek entitou, kterou používáme při modelování – například ji zakreslujeme do diagramu. Otázka je, co všechno je prvkem. To záleží na jak podrobný záznam o vzniku modelu potřebujeme. V některých případech mohou být i některé atributy prvkem. Existují dva druhy prvků – objekty a vztahy (rozdělení na objekty a vztahy je inspirováno meta-metamodelem GOPRR [Kelly 1997]).	konkrétní třída, metoda, funkce, scénář, dědění, asociace atd.
Objekt *Object*	Objekt je speciálním druhem prvku. Je základním konceptem modelování. Objekty jsou hlavními elementy návrhu. Typickým příkladem jsou třídy, komponenty, rozhraní, balíčky (z UML), Entity (ER modelování), scénáře, participanti (BORM) atd. Objekty samozřejmě mají všechny vlastnosti jako prvky.	konkrétní třída, metoda, funkce, scénář, entita atd

Česky **Anglicky**	**Termín**	**Příklad**
Vztah *Relationship*	Vztah je speciální prvek, který představuje vlastnost objektu, která se váže k jinému objektu. Jinak má samozřejmě stejné vlastnosti jako prvek, tedy má atributy, je uložen v repozitáři a (může) mít vlastní grafickou podobu. Vlastní vztahy mezi objekty se typicky implementují pomocí atributů odkazem na objekty, které jsou ve vztahu.	konkrétní dědění mezi objekty, asociace
Vstupní prvek *input element*	Vstupní prvek, je speciální prvek, který je do modelu vložen zvnějšku, typicky jako důsledek analýzy zadání a potřeb. Vstupní prvky jsou do modelu přidávány pomocí vstupní transformace. Vstupním prvkem je vždy objekt.	typicky konkrétní požadavek.
Transformace *Transformation*	Je proces změny modelu. Přidává právě jeden prvek do modelu na základě již v modelu existujících prvků a vztahy tohoto prvku s . Vstupem do transformace je tedy část stávajícího modelu (prvky) a je nový model s právě jedním novým prvkem a jeho vztahy s prvky vstupujícími do transformace.	vytvoř dědění
Vstupní transformace *input transformation*	Je speciálním druhem transformace, která pouze přidává vstupní prvek do modelu. Tato transformace nemá žádný vstup a výstupem je nový model s jedním přidaným vstupním prvkem.	přidání nového požadavku
Atribut *Attribut*	Prvek může mít své atributy. Atribut má své jméno a svou hodnotu. Hodnotou může být nějaký základní datový typ (číslo, textový řetězec, …), nebo odkaz na jiný prvek. Hodnotou atributu může být také kolekce jiných atributů. Atribut se často používá k implementaci vztahů mezi prvky.	String jmeno
Model	Zjednodušený obraz skutečnosti, který zachycuje ty	Model

Česky **Anglicky**	**Termín**	**Příklad**
	stránky skutečnosti, které nás zajímají. Model systému jsou formalizovaně zpracované vědomosti o daném systému. Tyto vědomosti jsou uloženy v repositáři. Model je typicky vytvořen pomocí modelovacího jazyka (např. UML). Model se skládá z prvků a vztahů mezi nimi.	informačního systému
Záznam transformací *Transformations record*	Obsahuje seznam transformací, postupně aplikovaných na model. Každá transformace zná své vstupní i výstupní prvky a vztahy během jejího provádění vzniklé. Tento seznam může být uspořádaný i neuspořádaný (tj. množina).	
Diagram *Diagram*	Diagram je strukturovaný, graficky zachycený, pohled na část modelu (tj. podmnožinu prvků modelu a jejich vztahy). Tento pohled ukazuje některé aspekty modelu,například statické vztahy mezi prvky v modelu (třeba diagram tříd v UML), nebo dynamické aspekty modelu (například stavový diagram, nebo object-relationship diagram v BORMu).	Třídní diagram, Object-relationship diagram
Diagram transformací prvků *Transformation record diagram*	Diagram (postupných) transformací prvků (nebo také diagram záznamu procesu tvorby modelu) – je vrstva modelu, která obsahuje všechny záznamy transformací v modelu. Tento záznam zachycuje „rodokmen“ všech prvků modelu (tj. vztahy typu předchůdce-následník mezi prvky modelu). Tento diagram je grafickou reprezentace záznamu transformací.	
Metamodel *Metamodel*	Metamodel je model, který definuje abstraktní jazyk pro tvorbu modelů [OMG 2005]. Metamodel typicky pracuje s entitami typu pojem (viz definice v části 7.3). Tj. například Třída, atribut, stav, funkce, participant,	metamodel UML

Česky **Anglicky**	**Termín**	**Příklad**
	scénář.	
Metoda *Method*	Označuje konkrétní postup vedoucí k vyřešení dílčího problému (viz [Kadlec 2004]).	OBA (Object Behavior Analysis)
Metodika *Methodology*	Označuje komplexní postupy a návody pro vývoj softwarové aplikace. Pod metodikou se skrývají všechny etapy řešení (fáze životního cyklu).Metodika pro vyřešení dílčích problémů používá metody (definice viz. [Kadlec 2004]).	RUP

Tab. 6 – Pojmy metody postupných transformací prvků

6.3 Konstrukce modelu informačního systému

Teď určíme základní postuláty metody postupné konstrukce modelu informačního systému. Postuláty vychází z toho, jak vývojář ve skutečnosti vytváří model informačního systému – jak postupně přidává do modelu jednotlivé prvky. Prvky může přidávat pouze po jednom.

Abychom dodržovali zásady postupné konstrukce informačního systému pomocí transformací je nutné při konstrukci modelu dodržovat tyto zásady:

1. Každý nový prvek, přidávaný do modelu informačního systému, musí mít smysl.

2. Nový prvek, mimo vstupních, vzniká na základě již v modelu existujících prvků pomocí transformace.

3. Transformace mění původní model na nový přidáním právě jednoho nového prvku do stávajícího modelu.

4. Do modelu mohu zvnějšku přidat pomocí (vstupní) transformace nový prvek, tzv. vstupní. Tento prvek nevznikl na základě prvků v modelu již existujících.

Tab. 7 – Základní postuláty metody postupných transformací

6.3.1 Komentář k postulátům

Ad 1. Dodržet tuto zásadu je nejobtížnější. Musíme si určit, co to znamená, že má prvek smysl:

- Prvek má smysl, pokud je součástí nějakého modelu řešícího zadání. a prvek není nadbytečný.
- Prvek je nadbytečný a je ho (a tím i transformaci, kterou vznikl) vynechat a model stále řeší zadání.
- Model řeší zadání, pokud splňuje na něj předem dané požadavky (tj. zadání).

Tato zásada je velmi obecná (a nejhůře splnitelná) a říká to, že do modelu nelze přidávat nesmyslné prvky a každý nově přidaný prvek by nás měl přiblížit k cíli – modelu řešícího zadání. To je obtížné, protože to, jestli prvek měl v modelu smysl jsme schopni zjistit pouze zpětně až po vytvoření modelu. Právě ve výběru vhodného přidávaného prvku do modelu spočívá největší know-how návrhu modelu informačního systému a činí ho tak náročným. Doporučení při výběru „správného“ prvku je více v následující kapitole (č. 7).

Prvek tedy není nadbytečný, pokud dopomohl* ke vzniku prvku řešícího zadání. Zbytečné prvky sice nemění řešení, ale zbytečně zesložiťují a znepřehledňují model.

Ad 2. V tomto bodě říkáme, že nový prvek přidávaný do modelu (tedy mimo speciálních – vstupních) vznikne na základě prvků, které již v modelu existují. Vycházíme z úvahy, že něco (tedy nové prvky) nemůže vznikat z ničeho. Říkáme že nové prvky nemohou samovolně bez příčiny vznikat, důvod jejich vzniku musí být již v modelu obsažen (zadání). Tomuto ději (procesu vzniku prvků) říkáme transformace. Zatím je pro nás transformace černou skříňkou, která probíhá v hlavě tvůrce modelu informačního systému, a o které pouze známe její vstupy (vstupní prvky

* Matematicky bychom řekli, že prvek není zbytečný, pokud leží v tranzitivním uzávěru prvků řešení.

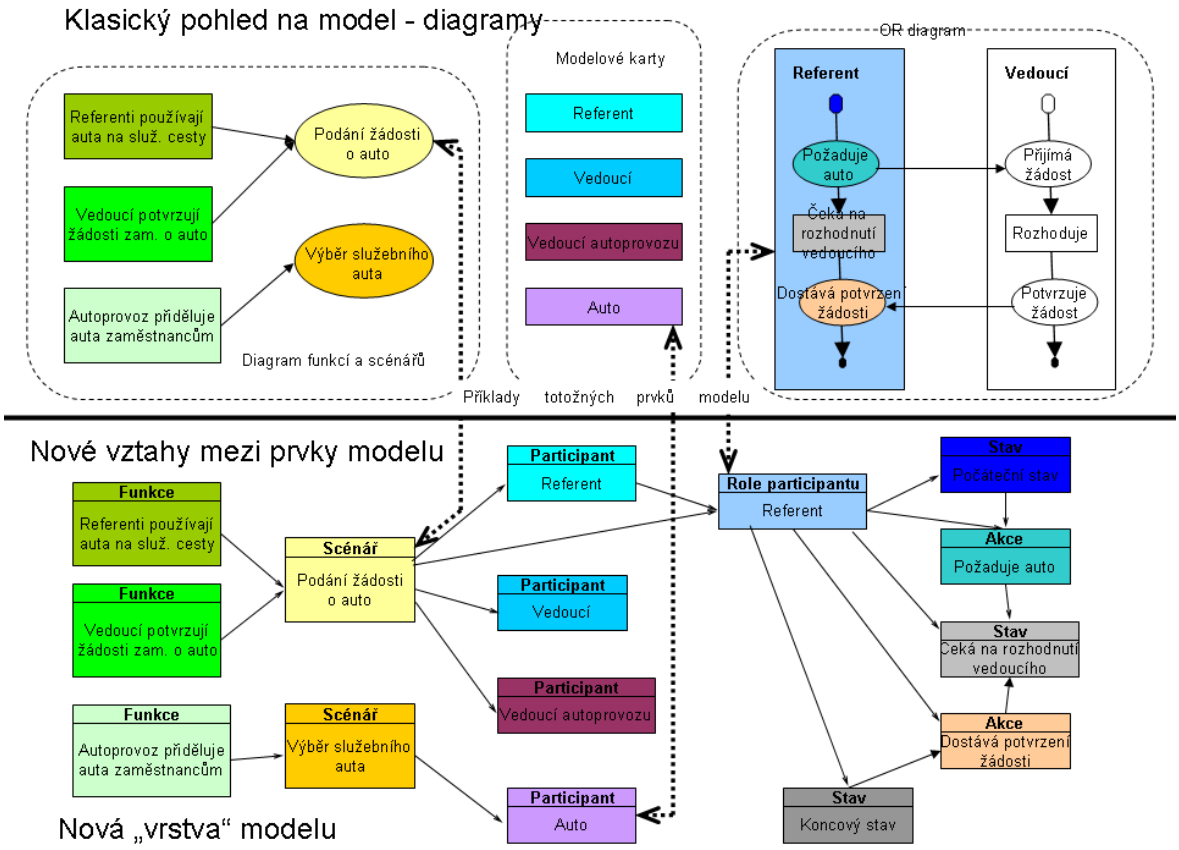
transformace) a výstupy (zde prvek a jeho vztahy přidávané do modelu). Transformace je pro nás čtveřice (M , I, p , V), kde M je Model vstupující do transformace, I je množina vstupních prvků transformace (model M obsahuje prvky z I), p je do modelu nově přidávaný prvek, V je množina vztahů prvku p s prvky z I.

Ad 3. Zde jsou důležité dvě věci. Nejdříve říkáme, že transformace je typu $M \rightarrow M$ (kde *M* je model) tedy vyhovuje obecné definici transformace (viz kapitola 5) a následně určujeme, kolik nových prvků transformací může vzniknout. V principu transformace může do modelu přidávat i více prvků, ale zde bylo zvoleno řešení s právě jedním prvkem. To má výhodu v jednodušších transformacích a v jejich jednodušším popisu a složitější transformace jdou složit z těchto jednodušších. Nevýhodou je, že transformací musí být provedeno více než v případě s více přidávanými prvky. Dalším argumentem, proč přidávat pouze jeden prvek je, že to více odpovídá způsobu práce skutečného vývojáře – v jednom okamžiku je vývojář fyzicky schopen přidat do modelu pouze jeden prvek. Vztahů mezi nově vytvořeným prvkem a stávajícím modelem může obecně vzniknout více.

Ad 4. V modelu musí existovat prvky dané zvnějšku (typicky ze zadání). Tento způsob konstrukce informačního systému je induktivní, tj. potřebujeme nějaký počátek. Další již mohou vznikat na jejich základě pomocí transformací. Vstupní prvky nejvíce ovlivňují výsledný model systému – jsou z nich odvozeny všechny ostatní prvky modelu. Přidávání nového vstupního prvku do modelu odpovídá přidání novému požadavku na model.

6.3.2 Nová vrstva modelu

Pokud budeme postupovat při konstrukci modelu postupovat podle výše zavedených postulátů, tak zkonstruujeme novou vrstvu modelu. Na obrázku Obr. 22 je tato nová vrstva znázorněna. Vzájemně si odpovídající prvky původního modelu a nově zkonstruované vrstvy jsou znázorněny stejnou barvou. V této vrstvě jsou zaznamenány všechny transformace, pomocí kterých model vzniknul. Pro každý prvek dokážeme najít z jakých prvků vznikl. Dokážeme tedy najít vstupní požadavky na prvek – to jsou prvky ze kterých postupně daný prvek vznikl. To, že jsme schopni dohledat, z jakého důvodu je prvek v modelu, to nám poskytuje silný nástroj pro kontrolu těchto prvků.



Obr. 22 - Vrstva transformací, jako nová vrstva modelu

6.3.3 *Vlastnosti modelu transformací prvků informačního systému*

Vrstva modelu, zaznamenávající transformace vzniklá podle zásad uvedených v části 6.2, bude mít následující vlastnosti:

1. Každý prvek, který není vstupní, musí vzniknout relevantní (tj. ne zbytečnou) transformací z již v modelu existujících prvků.

2. Každý prvek vzniká pomocí právě jedné transformace.

3. Po přidání nového požadavku (tj. zde nového vstupního prvku) na stávající model, který je v rozporu s ostatními požadavky, neexistuje žádný model, který splňuje všechny požadavky.

4. Nově přidaný prvek může mít vztahy pouze s těmi prvky, které vstupují do transformace.

5. Vstupním prvkem je vždy objekt.

6. Transformace může proběhnout pouze tehdy, pokud jsou k dispozici všechny prvky potřebné k jejímu provedení.

7. Mohu si vytvořit záznam, který zaznamenává všechny změny modelu, proběhlé pomocí transformací.

8. Diagram záznamu tvorby informačního systému tvoří orientovaný graf bez kružnic, kde uzly jsou prvky a hrany vyjadřují záznamy transformací.

9. Mezi prvky existuje částečné uspořádání.

10. Nově přidávaný prvek může mít atributy, jejichž hodnota je odvozena z prvků (a z jejich atributů), které se přímo podílejí na transformaci.

Tab. 8 – Odvozené vlastnosti modelu transformací prvků

Komentář k těmto vlastnostem:

Ad 1. Trochu problém dělá slovo relevantní. Ale když vyjdeme z postulátů, kde tvrdíme, že každý prvek přidaný do modelu dává smysl (není zbytečný), transformace, kterou tam je takový prvek přidán, má smysl.

Ad 2. Tento bod vyplývá z postulátů 2 a 3 (Tab. 7) metody postupných transformací. Transformace, kterou prvek vznikne, může být „normální“ nebo vstupní.

Ad 3. Pokud přidáme nový požadavek na model (ty jsou v našem případě reprezentovány vstupními prvky), který je v rozporu s ostatními požadavky, pak dojde k rozpadu modelu na 2 a více modelů, které řeší bezrozporné části zadání. Pokud například přidám požadavky, že informační systém musí být implementován pomocí objektové a relační databáze, tak řešením

budou 2 modely, jeden řešící implementaci systému relačně a druhý objektově. Velká část obou modelů může být stejná, ale od určitého okamžiku se rozdělí na dvě nezávislá řešení.

Ad 4. V postulátech jsem zavedl, že každý nový prvek vzniká na základě prvků v modelu již obsažených. Z tohoto vyplývá, že při transformaci mám k dispozici pouze ty prvky, z nichž nový prvek vzniká – tedy i pro vznik jakéhokoliv vztahu tohoto prvku s modelem, mám k dispozici pouze tyto prvky.

Ad 5. Vztah musí být mezi alespoň dvěma objekty. Tedy pokud do modelu přidám prvek zvnějšku, který není odvozen z nějakého jiného (a tím zatím nemá vztah k jinému prvku v modelu), tak to musí být objekt. Toto též vyplývá z předchozího bodu a postulátu číslo 4.

Ad 6. Zde pouze říkáme zřejmou věc, že transformace může proběhnout pouze, když jsou všechny potřebné vstupy dostupné. Toto také určuje „data-flow“ průběh návrhu informačního systému, kdy můžeme paralelně přidávat nové prvky do modelu podle toho, jaké prvky jsou dostupné. Atomickou operací je zde provedení transformace, tj. přidání nového prvku a jeho vztahů do modelu.

Ad 7. Záznam se vytváří tak, že si postupně zaznamenáváme, jak byly aplikovány jednotlivé transformace za sebou, jaké nové prvky a jejich vztahy vznikly. Dokonce ani nezávisí na pořadí jednotlivých transformací – záznam tedy může být množina. Toto je dáno „data-flow“ charakterem transformací – transformaci můžu vykonat pouze v okamžiku, kdy mám všechny potřebné prvky k dispozici.

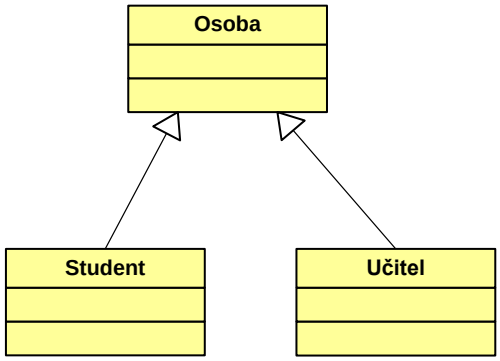
Ad 8. Diagram záznamu tvorby modelu, který je vytvořen vizualizací záznamu transformací v modelu.Tento graf je orientovaný a bez kružnic, kde orientace jde od „starých“ prvků k novým. Diagram tedy tvoří acyklický graf. To je dáno způsobem jeho vzniku, kdy každý prvek (mimo vstupních) vzniká právě jednou transformací.

Ad 9. Částečné uspořádání je dáno acykličností grafu.

Ad 10. Odůvodnění vychází z bodu 4 – nový prvek může mít vztahy pouze s prvky, které jsou vstupem transformace, kterou daný prvek vzniknul. Podobné je to s atributy prvku – během transformace mám k dispozici pouze prvky do ní vstupující. Atributy jsou součástí nově vzniklého prvku, a tak k jejich vytvoření mohu použít jen prvky vstupující do transformace a jejich atributy. Atributy často slouží k modelování vztahu.

6.3.4 Vztahy

Teď se pokusíme popsat úlohu vztahů v metodě postupných transformací. Obecně je vztah obecná vlastnost konkrétního objektu, která se váže k jinému objektu [wikipedia]. Na Obr. 23 je znázorněna část třídního diagramu v jazyce UML – můžeme tedy říci, že obsahuje 3 objekty (třídy Osoba, Student, Učitel) a 2 vztahy (dědičnost mezi osobou a studentem a druhou dědičnost mezi Osobou a Učitelem). Z jiného pohledu můžeme však také říci, že na Obr. 23 je znázorněno 5 prvků (3 prvky znázorňující třídy a 2 prvky znázorňující dědičnosti) a 4 vztahy – vztah mezi třídou Student a prvkem reprezentujícím dědičnost , třídou Osoba a prvkem dědičnost, třídou Učitel a druhou dědičností a třídou Osoba a druhou dědičností.



Obr. 23 – Část třídního diagramu

Možné jsou oba přístupy:

- v prvním ale tvrdíme, že obecný vztah (jako například dědičnost) je speciálním druhem prvku – může mít své atributy, může mít svou grafickou reprezentaci, dokonce může vznikat jako jediný výstup transformace. Výhodou tohoto přístupu je, že více odpovídá obecnému chápání pojmu vztah. Nevýhodou je, že musíme rozlišovat 2 druhy prvků – objekty a vztahy.
- druhý případ zredukoval vztah pouze na to, že nějaké dva prvky spolu sousedí a hlavní entitou při modelování se stává prvek. Výhodou tohoto přístupu je, základním prvkem modelování je pouze prvek modelu a vztahy mezi nimi jsou řešeny pomocí atributů* prvků.

* Tím, že atribut jednoho prvku odkazuje na jiný prvek.

Pro tuto práci byl nakonec zvolen první způsob[*] – tedy že vztah a objekt je speciálním druhem prvku. Diagram na Obr. 23 mohl být vytvořen tímto způsobem:

1. V modelu byly postupně pomocí transformací vytvořeny tyto 3 prvky – třída Osoba, Student a Učitel.
2. Transformací byl vytvořen prvek typu Dědičnost a jeho vstupními prvky byly třídy Osoba a Student.
3. Podobně se transformací vytvořila i druhá dědičnost mezi Osobou a Učitelem.

6.4 Metamodel modelu transformací prvků

Průběh tvorby modelu informačního systému lze zachytit pomocí seznamu transformací[†] postupně prováděných na modelu. Pro přesný popis modelu transformací prvků potřebujeme jeho metamodel.

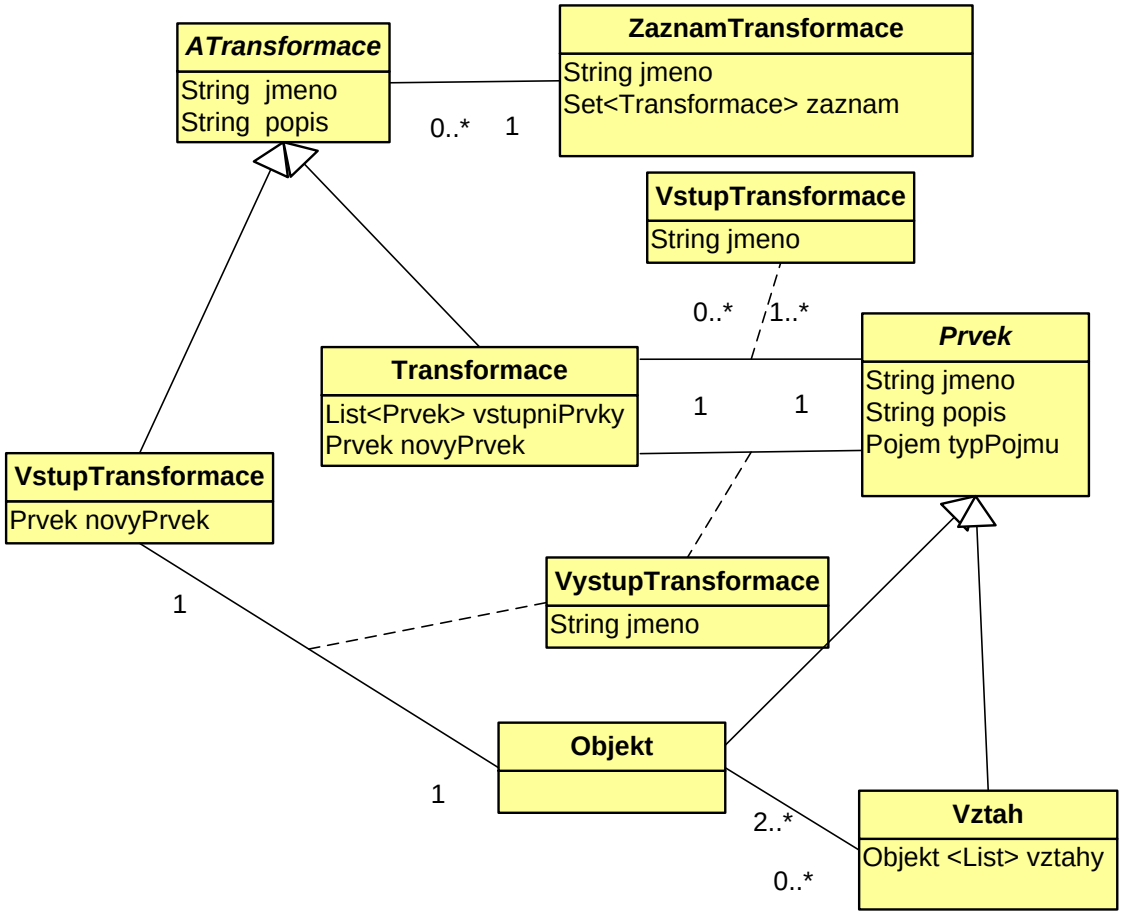
Na obrázku Obr. 24 je znázorněn obecný metamodel transformací prvků. Tento metamodel se skládá se z těchto tříd:

- Třída ATransformace – Abstraktní Transformace. Může být Standardní transformací nebo vstupní.
- Třída Transformace – reprezentuje vlastní transformaci mezi prvky. Obsahuje seznam odkazů na své vstupní prvky (0 až n) a jeden prvek výstupní, vzniklý pomocí této transformace.
- Třída VstupTransformace – reprezentuje vstupní transformaci. Vstupní transformace přidává do modelu právě jeden objekt.
- Třída Prvek – reprezentuje prvky obsažené v modelu. Nové prvky v modelu vznikají pomocí transformací. Prvek obsahuje i záznam o svém typu. Prvek je abstraktní třídou, čili ve skutečnosti se bude používat Objekt či Vztah.
- Třída Objekt – speciální typ prvku.

[*] Ale až potom, co byl nejdříve zvolen druhý způsob, ale následně se ukázalo, že se vztahem, jako speciálním druhem prvku se mnoho věcí zjednoduší.
[†] A při transformacích použitých vstupních prvků a v při nich vzniklých nových prvcích a vztazích.

- Třída Vztah – speciální typ prvku. Reprezentuje vztah (alespoň binární) mezi objekty.
- Třída VstupTransformace – může podrobněji popsat vstupní prvek transformace.
- Třída VýstupTransformace – může podrobněji popsat nový prvek vytvořený pomocí transformací.
- Třída Vztah – popisuje případné vztahy mezi vstupními prvky transformace a výstupním prvkem vzniklým během transformace. Vztah může vzniknout mezi neprázdnou podmnožinou vstupních prvků a výstupním prvkem[*]. Jeden vztah se týká pouze jedné transformace a jeho vstupních a výstupního prvku.
- Třída ZáznamTransformace – obsahuje seznam transformací, postupně aplikovaných na model. Každá transformace zná své vstupní i výstupní prvky během jejího provádění vzniklé. Tento seznam může být uspořádaný i neuspořádaný (tj. množina). Pokud ZáznamTransformace je uspořádaný, tak máme definované pořadí transformací a tak můžeme přesně říci v jakém pořadí byly nové prvky do modelu přidávány. Pokud je tento seznam neuspořádaný, tak sice nevíme pořadí transformací, ale víme jaké transformace můžeme v daném okamžiku provést (ty pro které máme všechny vstupní prvky).

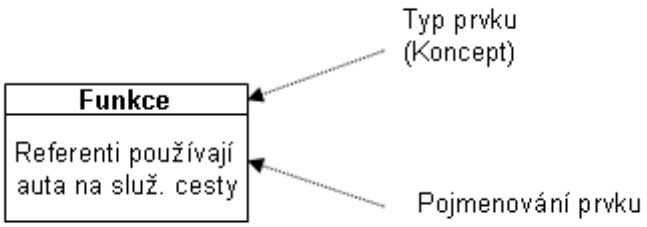
[*] V tomto metamodelu vztah odkazuje na asociační třídy VstupníPrvek a VýstupníPrvek, které následně odkazují na Prvek vstupující (či vystupující) do transformace.



Obr. 24 – Obecný metamodel transformací prvků

6.4.1 *Diagram postupných transformací prvků*

Pro snadnější orientaci a možnost práce se záznamem transformací navrhneme jeho „lidsky“ čitelnější podobu. Ta bude mít podobu diagramu postupných transformací prvků.

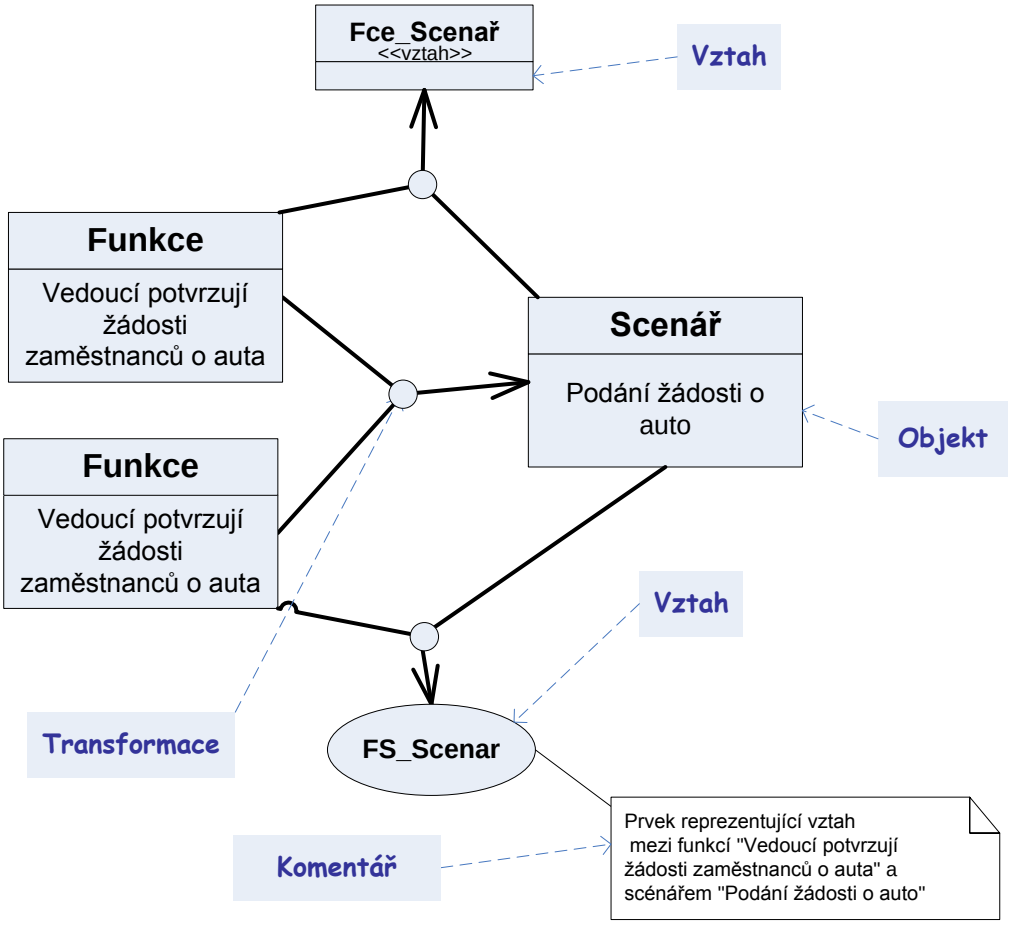


Obr. 25 – Reprezentace prvku

Vyjdeme z metamodelu na Obr. 24 a navrhneme pro každou jeho třídu grafickou reprezentaci. Máme tedy třídy:

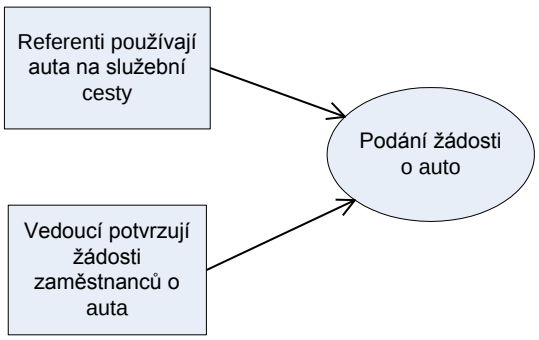
- Objekt – má grafickou reprezentaci podobnou třídě v UML. Obsahuje svůj typ (tedy v našem případě jméno pojmu (konceptu) - viz kapitole 7) a popis (případně jméno), který ho charakterizuje (viz Obr. 25).
- Vztah – vztah má grafickou reprezentaci buď pomocí elipsy, či stejnou jako objekt*, ale s přidaným stereotypem (<<vztah>>).
- Transformace – je znázorněna kroužkem. Tento kroužek je připojen pomocí úseček (reprezentujících VstupníPrvek) k vstupním prvkům a následně pomocí šipky v výstupnímu prvku transformace. Alternativní znázornění transformace je pomocí množiny šipek od vstupního prvku v výstupnímu.
- VstupníTransformace – obvykle není znázorněna vůbec, pokud o ní potřebujeme nějaké podrobnosti tak je zaznamenáme ke vstupnímu prvku (objektu vytvořeného vstupní pransformací), případně do Komentáře.
- VstupTransformace – tato třída je znázorněna pomocí úsečky mezi Prvkem a Transformací. Může být u ní zaznamenány podrobnosti o typu výstupu, případně jiné skutečnosti.
- VýstupTransformace – je znázorněn pomocí šipky směřující od Transformace k (výstupnímu) prvku. U vstupní transformace jí nezaznamenáváme.
- Komentář – lze přidat ke každému prvku diagramu. Syntaxe je stejná jako v UML (tj. obdélník s ohnutým pravým horním rohem).

* Tj. jako třída v UML.



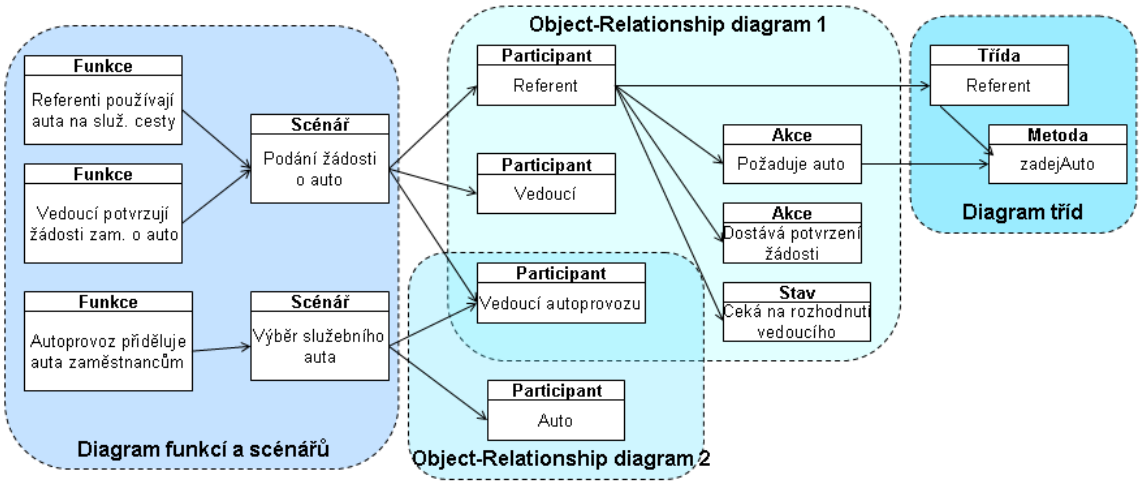
Obr. 26 – Grafické znázornění záznamu transformace

Na obrázku Obr. 26 je záznam transformací, který znázorňuje transformace provedené pří vzniku diagramu funkcí a scénářů na Obr. 27.



Obr. 27 – část diagramu funkcí a scénářů z BORMu

Na obrázku Obr. 28 je uveden příklad části modelu informačního systému autoprovozu. Pro návrh tohoto systému byla použita metodologie BORM (více viz Příloha A). Na tomto obrázku jsou uvedeny postupné transformace vstupních prvků (funkcí) ne výstupní (zde třídy). V tomto obrázku je pro zjednodušení použit způsob znázornění transformace pomocí množiny šipek mezi jejími vstupy a výstupem. Také nejsou znázorněny vztahy. Navíc je znázorněna příslušnost jednotlivých prvků k diagramům modelu. Z toho obrázku je vidět, že typicky nerozlišujeme pořadí vykonávaných transformací, neboli transformace se mohou vykonat v tom okamžiku, kdy máme k jejímu provedení všechny potřebné informace (prvky).



Obr. 28 – Část IS autoprovozu pomocí metody BORM (zjednodušeno)

6.5 *Použití modelu transformací*

Pokud budeme vytvářet model informačního systému výše zmíněných principů (viz odstavce 6.3 a 6.3.3), tak v modelu bude ke každému prvku možnost dohledat požadavky* (tedy v našem případě vstupní prvky do modelu), na základě kterých vznikl. Zajistíme tedy sledování požadavků na úrovni prvků modelu.

Sledování požadavků zabezpečuje a dokumentuje, že životní cyklus je dodržován. Podle [Baldwin 2001, Antoniol et al 2002] sledování požadavků má tyto vlastnosti :

* Dokonce nejen požadavky, ale i všechny prvky které ovlivnili vznik nového prvku.

- demonstruje vztah mezi návrhovými vstupy a návrhovými výstupy.
- zabezpečuje, že je návrh založen na předchůdci, který je podložen nějakým požadavkem.
- zabezpečuje že návrhová specifikace je ověřená a funkční požadavky ověřené.

Tato dohledatelnost požadavků nám může přinést výhody zejména v těchto oblastech [Wieners 2003]:

- **Certifikace** – dohledatelnost požadavků se hodí při bezpečnostní certifikaci systému, protože díky ní můžeme ukázat, že všechny požadavky byly skutečně implementovány.
- **Analýza dopadu** – bez odkazů mezi požadavky máme velkou šanci přehlédnout nějakou část systému, na kterou bude mít analyzovaná změna vliv.
- **Údržba** – dobrá dohledatelnost usnadňuje správné provádění změn při údržbě systému což zvyšuje produktivitu .
- **Sledování projektu** – když se budou při vývoj pečlivě zaznamenávat všechny odkazy*, tak bude k dispozici přesný ukazatel průběhu implementačních prací. Chybějící odkazy znamenají, že daný výstup ještě není hotový.
- **Úpravy starších systémů** – při úpravě starších systémů je možné si nechat vypsat ty jejich části, které jsou pokryty funkcemi nového systému.
- **Recyklování** – odkazy usnadňují opakované použití komponent systému, protože spojují související požadavky, návrh, zdrojové kódy a testy.
- **Snižování rizika** – dokumentace závislosti mezi komponentami systému zmenšuje problémy, jež by mohly nastat, když nějaký z klíčových členů projekčního týmu s nepostradatelnými znalostmi opustí projekt.
- **Testování** – když nějaký test skončí neúspěšně, tak jsou podle odkazů snadno dohledatelné ke zdrojovým kódům příslušející požadavky. Když jsou spojeny testy s požadavky, tak se je možno vyhnout se nadbytečným testům.

Konstrukce modelu informačního systému za použití metody postupných transformací nám tedy může zejména přinést tyto výhody:

- Lepší dokumentaci průběhu tvorby informačního systému.
- Zprůhlednění vztahů v modelu a tím lepší zajištění konzistence modelu.

* Zde jsou odkazy myšleny jako vztahy mezi prvkem modelu a požadavky, na základě kterých vzniknul – viz část 4.5.

- Různé kontroly modelu informačního systému.
- Najít požadavky, na základě kterých je daný prvek v modelu.
- Automatické vytvoření matic dohledatelnosti požadavků.
- Lokalizovat změny v modelu

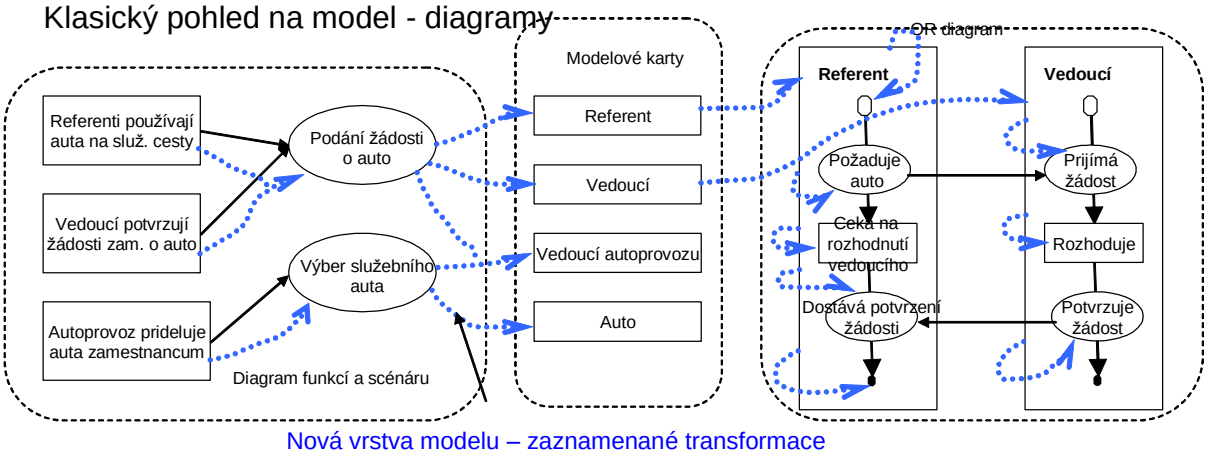
Tyto výhody podrobněji popíšeme v následujících odstavcích.

6.5.1 *Dokumentace průběhu tvorby informačního systému*

Zaznamenáváním původu všech nových prvků v modelu získáme nové, dříve nezachytitelné, informace o postupném vzniku modelu informačního systému. Jsme schopni zjistit původ jakéhokoliv prvku – jakou transformací daný prvek vzniknul a kdo byly jeho předkové. Možný příklad je na obrázku číslo Obr. 29 .

Tyto informace o vzniku prvků tvoří novou vrstvu modelu. Na obrázku Obr. 29 je zachycena část modelu autoprovozu pomocí diagramů funkcí a scénářů, modelovacích karet a object-relatioship diagramu. Toto je klasický pohled na model. Jsou zde však také navíc zaznamenány transformace, pomocí kterých tato část modelu postupně vznikla (transformace jsou zaznamenány modře). Tyto záznamy transformací tvoří novou, dosud nezachytitelnou, vrstvu modelu.

Zaznamenáním této nové vrstvy získáme představu (a dokumentaci) o tam, jak model vznikal, případně i popis a zdůvodnění použitých transformací.



Obr. 29 – Vrstva transformací (modře), jako nová vrstva modelu

6.5.2 Zprůhlednění vztahů v modelu informačního systému

Při konstrukci modelu informačního systému pomocí metody postupných transformací prvků jsme nuceni stále kontrolovat vytvářený model a o nově přidávaných prvcích je nutno přemýšlet v souvislostech s prvky již v modelu přítomnými.

Toto by mělo pomoci k zlepšení konzistence modelu informačního systému. Ještě více nám pomůže, že nový prvek může vzniknout pouze jednou z několika možných transformací (více viz následující kapitola č. 7).

6.5.3 Kontrola modelu informačního systému

Vytvářený model informačního systému můžeme kontrolovat při vytváření vůči pravidlům uvedeným v odstavcích 6.3 a 6.3.3. Typickou kontrolou bude například, zda všechny prvky, které nevznikají pomocí transformace jsou prvky vstupními. Zda byly dodrženy podmínky transformace, zda prvek vznikl pomocí právě jedné transformace atd.

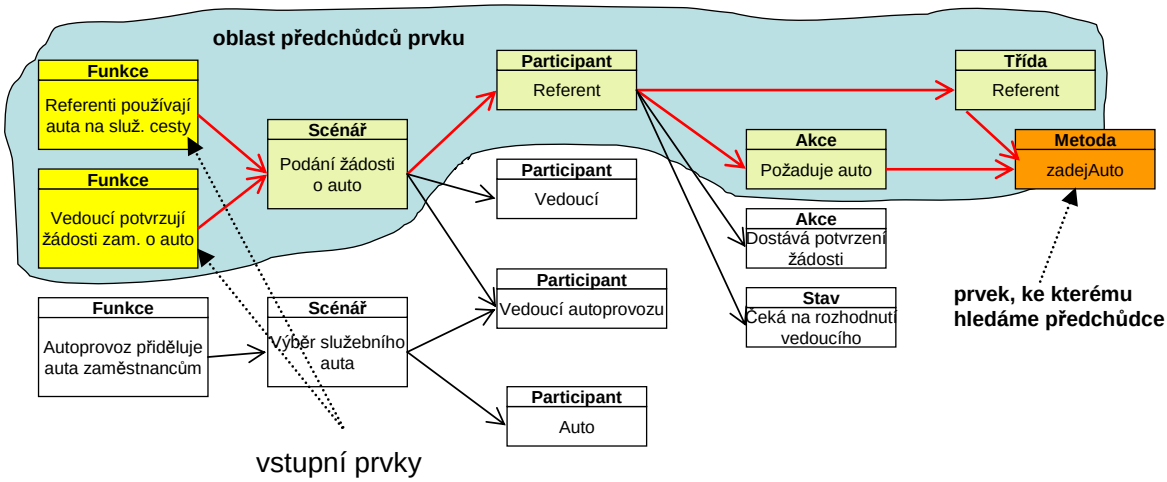
Typickou chybou jsou například prvky, které nevznikly pomocí žádné transformace a zároveň tyto prvky nejsou vstupní.

6.5.4 Hledání požadavků a vstupních prvků

V modelu zkonstruovaném podle metody postupných transformací jsme schopni najít vstupní transformace a vstupní prvky z kterých daný prvek vzniknul. Tyto vstupní prvky typicky odpovídají nějakému požadavku na model.

Na Obr. 30 je uveden příklad, kdy k metodě „zadejAuto“ hledáme odpovídající vstupní prvky (zde jsou to funkce „Referenti používají auta na služ. cesty“ a „Vedoucí potvrzuje žádosti o auto“). Jejich nalezení je jednoduché – vstupní prvky se musí nacházet v tranzitivním uzávěru* daného prvku a musí to být vstupní prvek.

* Tedy při konstruování tranzitivního uzávěru musíme jít proti směru šipek a ne klasicky po směru.



Obr. 30 – hledání požadavků k prvků

Algoritmus* (Výpis 1) najde všechny vstupní prvky (ty poznáme podle toho, že nemají žádného předchůdce). Algoritmus je založen na rekurzivním prohledávání modelu do šířky. Vstupní prvek poznáme podle toho, že nemá žádné předchůdce. V množině `seznamVstupPrvků` ukládáme vstupní prvky a v množině `navštívenéPrvky` zase prvky, které jsme již algoritmem zpracovali. Vstupem do tohoto algoritmu je prvek, u kterého hledáme vstupní prvky z kterých je odvozen.

```
NajdiVstupPrvky(prvek,seznamVstupPrvků:=Set(),navštívenéPrvky:=Set())
   if (prvek.dejPředchudce().isEmpty)
      return seznamVstupPrvků.add(prvek)
   for prv in prvek.predchudce()
      if (not prv in navštívenéPrvky)
         NajdiVstupPrvky(prv, seznamPožadavků, navštívenéPrvky.add(prv))
```

Výpis 1 – funkce hledající vstupní prvky k danému prvku

Pokud hledáme požadavky z kterých je prvek odvozen, tak použijeme algoritmus uvedený na Výpis 2. Je velmi podobný předchozímu algoritmu s tím rozdílem, že se každého zpracovávaného prvku ptáme, zda je vstupní a nebereme automaticky jen vstupní prvky.

* Algoritmy v této práci jsou psány pomocí pseudokódu, který vychází z programovacího jazyku Python (http://www.python.org), který je svou jednoduchostí na toto použití vhodný.

```
NajdiPožadavky(prvek,seznamPožadavků:=Set(),navštívenéPrvky:=Set())
   if (prvek.jePožadavek)
      seznamPožadavků.add(prvek)
   if (prvek.dejPředchudce().isEmpty)
      return seznamPožadavků
   for prv in prvek.predchudce()
      if (not prv in navštívenéPrvky)
         NajdiPožadavky(prv, seznamPožadavků, navštívenéPrvky.add(prv))
```

Výpis 2 – funkce hledající požadavky k danému prvku

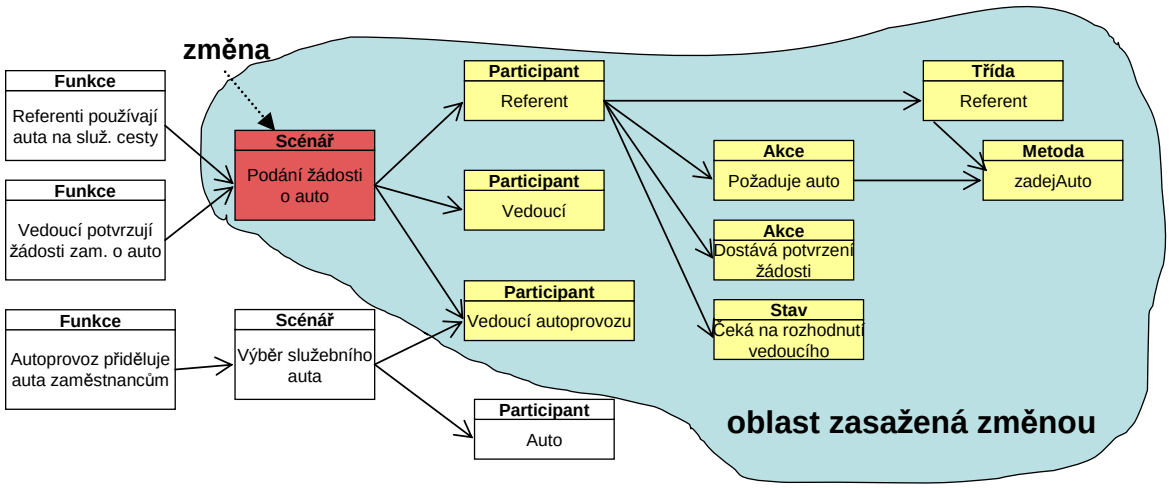
6.5.5 *Vytvoření matice dohledatelnosti požadavků*

V modelu zkonstruovaném podle metody postupných transformací jsme schopni automaticky vytvářet matice dohledatelnosti požadavků (viz část 4.5.4). Zjednodušeně lze tento postup popsat tak, že si vybereme nějaké prvky a k nim hledáme v jejich tranzitivním uzávěru k nim odpovídající požadavky.

Podrobnější popis vytváření obou dvou druhů matic dohledatelnosti požadavků je uveden v části 8.1.

6.5.6 *Lokalizace změn*

Častým problémem je, jak při nějaké změně modelu poznat, jaká část modelu je jí postižena a je kandidátem na opravu či předělání. V modelu vytvořeném pomocí metody postupných transformací je to snadné. Prvky postižené změnou se nachází v tranzitivním uzávěru změněného prvku. Na Obr. 31 je uveden příklad, kdy při změně prvku „Podání žádosti o auto“ jsou touto změnou postiženy všechny označené prvky.



Obr. 31 – Lokalizace změny v modelu

V již (alespoň částečně) vytvořeném modelu můžeme snadno automaticky zjišťovat všechny prvky, které byly změnou v modelu postiženy. Algoritmus, který najde všechny změnou postižené prvky je na Výpis 3. Tento algoritmus implementuje jednoduché rekurzivní prohledávání modelu do šířky a postupně přidává do množiny `zasaženéPrvky` ty prvky, které jsou touto změnou postiženy. Prohledávání skončí v okamžiku, kdy prvek ji nemá žádného následníka.

```
NajdiZměny(prvek,zasaženéPrvky:=Set())
   if (prvek.naslednik().isEmpty)
      return zasaženéPrvky.add(prvek)
   for prv in prvek.naslednik()
         if (not prv in zasaženéPrvky)
            NajdiZměny(prv, navštívenéPrvky.add(prv))
```

Výpis 3 – funkce hledající požadavky k danému prvku

6.6 *Závěrečné shrnutí kapitoly*

V této kapitole jsou zavedeny základní pojmy týkající se postupného modelování pomocí transformací prvků v modelu. Jsou navrženy zásady konstrukce modelu informačního systému. Tyto zásady jsou:

- Každý nový prvek, přidávaný do modelu informačního systému, musí mít smysl.
- Nový prvek mimo vstupních, vzniká na základě již v modelu existujících prvků pomocí transformace.
- Transformace mění původní model na nový přidáním právě jednoho nového.
- Do modelu mohu zvnějšku přidat pomocí (vstupní) transformace nový prvek, tzv. vstupní prvek. Tento prvek nevznikl na základě prvků v modelu již existujících.

Dále jsou na základě těchto zásad navrženy pravidla, která budou v takto navrženém modelu platit a lze jich využít například pro kontroly.

Je navržen metamodel diagramu záznamu transformací a také jeho grafická podoba.

V závěru kapitoly je navrženo využití tohoto principu postupné transformace prvků. Pomocí modelu takto zkonstruovaného lze:

- Lépe dokumentovat průběh tvorby informačního systému.
- Zprůhlednit vztahy v modelu a tím zlepšit jeho konzistence.
- Provádět různé kontroly modelu informačního systému.
- Najít požadavky, na základě kterých je daný prvek v modelu.
- Automaticky vytvořit matice dohledatelnosti požadavků.
- Lokalizovat změny v modelu.

7 Přechody mezi koncepty metody

Tato kapitola spojuje postupné transformace prvků a metody analýzy a návrhu informačních systémů. Tyto metody popisují, jaké všechny možné transformace mezi prvky jsou v jejich rámci přípustné. V rámci vytváření modelu informačního systému kapitola zobecňuje použité prvky do pojmů (neboli typů prvků) a mezi těmito koncepty popisuje typy přípustných transformací. Toto je zachyceno v modelu přechodů mezi koncepty (metody). Dále je v této kapitole navržen popis typů transformací a jejich možné využití.

7.1 Úvod

Konstrukce záznamu transformací prvků, definovaná v předchozí kapitole nám pomáhá pouze omezeně Při vlastním návrhu informačního systému. Dokonce zesložiťuje vlastní návrh – musíme o každém novém prvku přemýšlet (a zaznamenávat) na základě jakých prvků vzniká. Hlavní praktický význam záznamů postupných transformací spočívá zejména v tom, že takto vytvořený model má některé výhodné vlastnosti, které nám umožňují automaticky provádět některé kontroly modelu, lepší dokumentaci jeho návrhu, hledání změn a generování různých matic dohledatelnosti požadavků atd. (více o výhodách v odstavci 6.3.3).

Hlavní myšlenkou metody postupných transformací je, že každý nově přidávaný prvek* do modelu má své předchůdce. To nás nutí přemýšlet o kontextu každého nově přidávaného prvku a tím snižovat pravděpodobnost chybného návrhu. Tento postup ale neříká, jakou transformací tento nový prvek vzniká. V části 6.3.1 byly uvedeny tyto důležité skutečnosti:

- Každý nový prvek, přidávaný do modelu informačního systému, musí mít smysl.
- Prvek má v modelu smysl, pokud tento prvek je součástí nějakého modelu řešícího zadání a prvek není nadbytečný.
- Prvek je v modelu nadbytečný, pokud ho lze z modelu vypustit a model stále řeší zadání.

Z tohoto je vidět, že tato metoda nám nic neříká o tom, jaký prvek můžu v tomto okamžiku použít při vytváření modelu systému – dokonce po nás chce, aby nově přidávaný prvek měl smysl!! A lze to zjistit až zpětně. Aby nám tato metoda pomohla, tak potřebujeme vědět, jaké prvky mohou

* Samozřejmě mimo vstupních prvků.

v dané chvíli a kontextu vzniknout. Nové prvky vznikají pouze pomocí transformací – potřebujeme tedy určit jaké transformace je možno nad daným modelem využít.

Abychom mohli zjistit, jaké transformace jsou přípustné, musíme se zamyslet nad tím jak vzniká model informačního systému. Tvůrce informačního systému postupně přidává jednotlivé prvky (třídy, rozhraní, participanti, funkce atd.) do modelu. Tyto prvky přidává na základě myšlenkového procesu, jehož provedení jsme v předchozí kapitole nazvali transformací.

Jak tedy vytváříme (aplikujeme) při tvorbě informačního ty správné transformace? Transformace vytváříme na základě svých znalostí a zkušeností s tvorbou modelu informačního systému. Tyto znalosti a zkušeností můžeme získat náročnou (a hlavně nákladnou) metodou pokusů a omylů, nebo využijeme cizí zkušenosti. Takový souhrn zkušeností („best practices“) o tvorbě informačních systémů jsou shrnuty v metodách (a metodikách[*]) analýzy a návrhu informačních systémů. Pokud se budeme při návrhu informačního systému řídit těmito metodami, tak pravděpodobně budeme (v průměru) rychleji, kvalitněji a laciněji navrhovat informační systémy.

V těchto metodách tvorby informačního systému nalezneme opakující se vzory, na jejichž základě vznikají nové prvky modelu informačního systému. Tyto vzory nám popisují probíhající typy transformací během návrhu modelu pomocí těchto metod. Tyto metody návrhu IS nám pomáhají při tvorbě nových prvků v modelu – vznik nových prvků v modelu je řízen metodou analýzy a návrhu informačního systému. Metody návrhu IS určují jaké transformace mohu v daném kontextu použít a jaké nové prvky mohou vzniknout.

V okamžiku, kdy získáme možné transformace (a jejich předpisy) z metod návrhu IS, tak se trochu omezíme – nemůžeme provádět libovolné transformace, jsme omezeni jen na vstupy[†] a výstup transformace předepsaný metodou. Toto omezení nám zejména přinese tu výhodu, že tuto transformaci můžeme podrobněji popsat (kdy se provádí, vstupní prvky, jaký prvek vzniká, algoritmus transformace atd.).

Potom budeme umět znázornit typy prvků používané v metodě (koncepty) a typy transformací (přechody) mezi nimi. Můžeme tedy vytvořit „procesní mapu“ metody, na jejímž základě budeme

[*] Zde je dobré si uvědomit rozdíl mezi metodami a metodikami. Zjednodušeně – metoda je konkrétní postup, který něco vytváří a metodika je souhrn metod a rad a poznatků, kdy je použít, jak o rganizovat celý vývojový proces atd. Vlastní model ale vždy vytváříme aplikováním jednotlivých metod.

[†] Bez omezení metodou mohou být vstupní prvky transformace přidány z různých důvodů. Tyto důvody odpovídají typům odkazů (vztahů) mezi prvky – viz. Obr. 11. Po omezení metodou je typicky hlavní důvod použití nějakého vstupního prvku ten, že to metoda vyžaduje.

vědět jaké transformace můžeme v dané chvíli použít. Tuto mapu budeme nazývat diagram přechodů mezi koncepty.

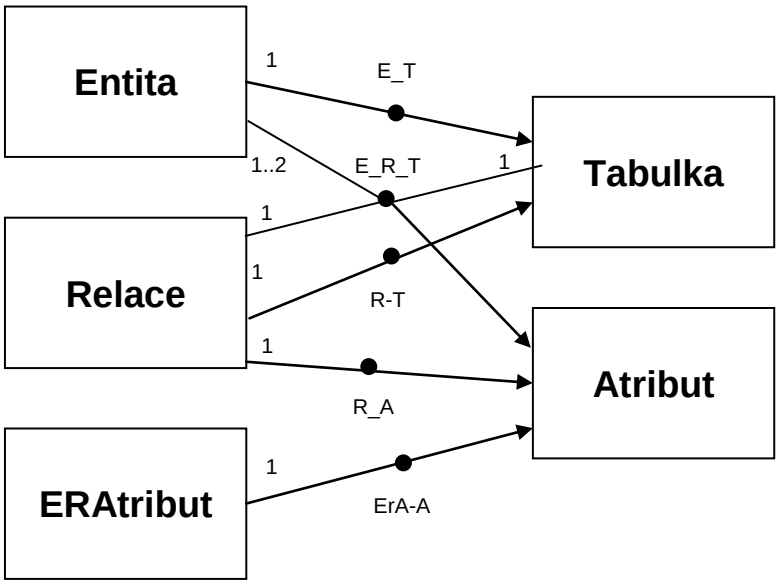
V současné době používané metody analýzy a návrhu informačních systémů nemají, tuto procesní mapu vytvořenu. Tyto metody nemají možné typy transformací mezi svými koncepty explicitně uvedeny. V lepším případě jsou tyto typy transformací v metodě jen naznačeny v nějakém, ne příliš podrobném, slovním popisu. Pro implicitní znázornění transformací potřebujeme nový aparát. Ten je popisován v následujících odstavcích.

7.2 Motivace - přechody mezi koncepty metody – praktická ukázka

Pro názornost si ukážeme nejdříve příklad modelu možných přechodů mezi koncepty metody. Pro jednoduchost jsme zvolili transformaci mezi Chenovým entitně-relačním diagramem a fyzickým modelem relační databáze. Tato transformace je dobře známá a často používaná a (téměř) každý CASE nástroj používaný pro modelování relačních databází jí provádí automaticky. Nejdříve si připomeneme její algoritmus:

1. Všechny entity převeď na tabulky.
2. Pokus je relace mezi entitami binární a typu 1:N a bez atributů, potom transformuj relaci na nový atribut (cizí klíč) a přidej ho k atributům tabulky u které je N. Pokud je relace typu 1:1 přidej cizí klíč k jedné z tabulek.
3. Jinak převeď relaci na tabulku, k jejímž atributům přidej cizí klíče ukazující na tabulky, mezi kterými je relace.
4. Převeď zbývající atributy u entit a relací na atributy u tabulek.

Tento slovní popis je znázorněn pomocí diagramu přechodů mezi koncepty metody na obrázku Obr. 32. Na tomto obrázku je celkem 5 různých přechodů mezi koncepty (neboli předpisů transformací). Je zde vidět, že (jedna) entita se transformuje na (jednu) tabulku. Relace se může transformovat buď na atribut (cizí klíč), nebo na tabulku s dvěma nebo více (podle stupně relace) cizími klíči (atributy). Atributy u entit a relací se transformují na atributy u tabulek. U každého předpisu transformace je uvedeno jeho jméno.



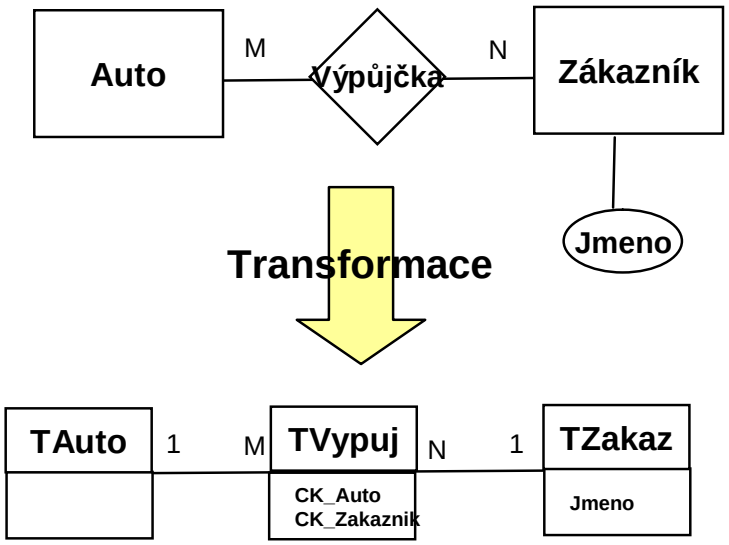
Obr. 32 – Model transformací pojmů metody transformace ER konceptuálního diagramu do fyzického relačního diagramu

Tyto transformace je možno provádět automaticky, protože známe její přesný algoritmus (viz např. [Silbershatz 2004]). Toto však v metodách analýzy a návrhu informačních systémů není typické. V nich typicky známe vztahy mezi koncepty metody (tj. model metody), ale konkrétní transformace a jejich vstupy volí tvůrce IS podle své zkušenosti.

Vlastní převod (transformaci) z ER diagramu na tabulky provedeme postupným aplikování jednotlivých předpisů transformací. Abychom dostali výsledek jako na Obr. 33 provedli jsme postupně tyto transformace:

1. E_T(Auto) – provedením této transformace vznikl nový prvek TAuto typu Tabulka.
2. E_T(Zakaznik) – vzniká tabulka TZakaz
3. R_T(Vypujcka) – vzniká tabulka TVypuj
4. ErA-A(jmeno) – vzniká atribut jmeno, a přiřazuje se do tabulky TZakaz
5. E_R_T(Auto, Vypujcka, TAuto) – vzniká atribut (cizí klíč) ck_Auto (TVypuj)
6. E_R_T(Zákazník, Vypujcka, TZakaz) – vzniká atribut (cizí klíč) ck_Zakaznik (TVypuj)

Seznam těchto transformací tvoří Záznam transformace.



Obr. 33 – Transformace z ER diagramu do tabulek

7.3 Zavedení pojmů

Zde zavedeme nové pojmy týkající se modelu přechodů mezi koncepty:

.

Česky **Anglicky**	**Termín**	**Příklad**
Koncept *(Pojem)* *Concept*	je entita s kterou pracujeme v rámci metody. Typickým konceptem jsou: třída, package, use-case, funkce, scénář, generalizace apod. Koncepty jsou obsaženy v metamodelu metody. Koncept je vlastně „datovým“ typem prvku. Místo pojmu lze v češtině používat koncept. Instancí pojmu je prvek (tedy ve skutečnosti objekt nebo vztah)	Třída, funkce, scénář, generalizace
Přechod mezi koncepty	Neboli předpis transformace – je to předpis, určující možnou přeměna (několika) konceptů na nové koncepty,	Vznik scénáře (viz Obr. 38)

Česky **Anglicky**	**Termín**	**Příklad**
Transition between Concepts	která je dovolená metodou. Tyto přechody definují návaznosti v metodě. Instancí transformace (tj. záznamem o skutečném provedení) je záznam transformace.	
Diagram přechodů konceptů v metodě Diagram of Transitions between Concepts	Model (možných, dovolených) přechodů konceptů v metodě (či zkráceně model metody) – je model zachycující všechny koncepty metody a vzájemné, metodou dovolené, transformace mezi nimi. Tento model lze graficky vyjádřit pomocí diagramu transformací pojmů metody.	Diagram transformací metody BORM (viz Obr. 37)

Tab. 9 – Pojmy metody postupných transformací prvků II.

Další potřebné pojmy byly zavedeny v části 6.2.

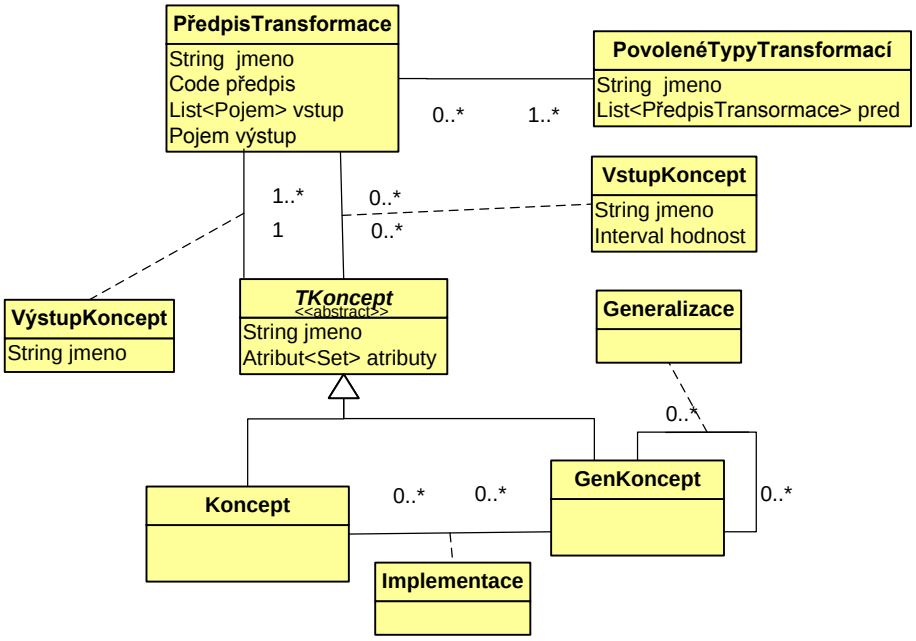
7.4 Vlastnosti

Pro praktické použití modelu (povolených) přechodů mezi koncepty je důležité si uvědomit, co je tento model a v čem nám pomáhá. Tento model definuje všechny typy prvků (neboli koncepty) a typy transformací mezi nimi můžeme, které můžeme při modelování využít. Typicky nám tento model neříká, jaké konkrétní transformace a tím jaký konkrétní prvek můžeme do modelu informačního systému přidat. Tento model nám dává pouze na výběr seznam všech možných transformací povolených danou metodou návrhu systému. Výběr typu transformace a konkrétní vstupy do ní nechává na tvůrci daného modelu informačního systému.

Metoda postupných transformací není schopna sama model informačního systému vytvořit, ale pomáhá nám při jeho tvorbě. Hlavním a nejdůležitějším elementem při tvorbě systému zůstává člověk.

7.4.1 *Metamodel modelu přechodu mezi koncepty*

Model přechodů mezi koncepty (a tím dovolené typy transformací) lze zachytit pomocí seznamu typů transformací, které můžeme při modelování využít. Pro přesný popis tohoto modelu jsme vytvořili jeho metamodel.



Obr. 34 – Metamodel modelu přechodu mezi koncepty

Na Obr. 34 je uveden metamodel modelu přechodu mezi koncepty. Obsahuje tyto třídy:

- TKoncept – abstraktní třída sdružující vlastnosti Koncept a GenKoncept
- Koncept – je metatřídou k třídě Prvek z metamodelu transformací prvků (viz Obr. 24). Koncept představuje typ prvku[*].
- GenKoncept – sdružuje společné vlastnosti a použití více konceptů. Je to abstraktní element, nemá tedy v modelu IS svou instanci. Nejlépe se dá přirovnat k rozhraní[†]. Slouží k zjednodušení předpisů transformací. Jako vstupní typ do transformace můžeme uvažovat celou množinu konceptů.

[*] Mezi konceptem a prvkem je stejný vztah jako mezi třídou a objektem.
[†] Interface v Javě.

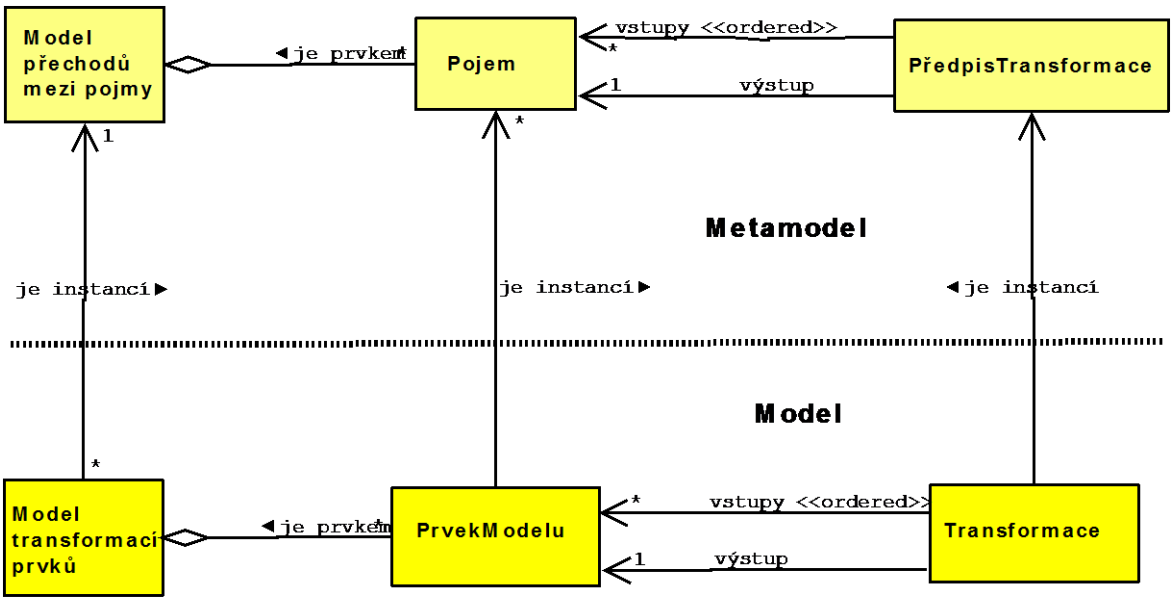
- Implementace – tato třída nám udává, jaké GenKoncepty implementuje koncept.
- Generalizace – tato třída nám umožňuje vytvářet hierarchie generalizovaných konceptů (GenKoncept). Samozřejmě v těchto hierarchiích platí Liskovové substituční pravidlo[*] (viz [Liskov 1994]). Generalizace má směr a platí, že GenKoncepty (jako vrcholy) a Generalizace (jako orientované hrany) tvoří acyklický graf. Pro zjednodušení musí pro každý Genkoncept platit, že žádný z jeho následníků (i zprostředkovaných, přes více generalizací) nesmí mít mít stejně pojmenované atributy.
- PředpisTransformace – třída PředpisTransformace v sobě sdružuje deklaraci transformace, tj. předepisuje jakých typů (neboli pojmů) budou vstupy do transformace. Pozor, počet vstupů jednotlivého pojmu může být určen intervalem. Dále určuje typ výstupu. Výstup je jen jeden. Dále PředpisTransformace může obsahovat předpis algoritmu vlastní transformace – tedy program, který provádí vlastní transformaci. Pokud nemá PředpisTransformace žádné vstupy, je vstupním a použije se pro vstupní transformaci.
- VstupKoncept – upřesňuje jednotlivé vstupy, tedy parametry transformace. Tj. jejich typ a případně množství prvků vstupujících do transformace. Atribut násobnost určuje, kolikrát (může se vyskytnout třeba 1..N krát) se daný typ prvku vyskytne na vstupu.
- VýstupKoncept – obsahuje informace o výstupním prvku transformace.
- PovolenéTypyTransformací – Tato třída obsahuje seznam všech povolených typů transformací, které můžeme při modelování využívat. Pokud jsme typy transformací (reprezentované třídou PředpisTransformace) vytvářeli podle nějaké metody návrhu IS, tak se dá říci tato třída představuje (z hlediska metody postupných transformací) druh modelu této metody. Tato třída odpovídá modelu přechodů mezi koncepty.

Každý koncept může implementovat 0 až n generalizovaných konceptů a mezi generalizovanými koncepty používáme vícenásobnou generalizaci. Jeden TKoncept[†] může být vstupem pro 0 až n předpisů transformace. Pokud koncept se není na vstupu žádného předpisu transformace, tak jeho instance (tedy prvky) budou „výstupními“[‡]. Předpis transformace může mít na vstupu 0 až N různých konceptů nebo generalizovaných konceptů. Pokud má 0 tak vytváří vstupní prvky. Každá transformace vytváří na svém výstupy pouze jeden prvek, a tedy každý předpis transformace bude

[*] Tj. potomka můžeme použít na všech místech jako předka.
[†] Tj. Koncept nebo GenKoncept.
[‡] Tedy budou vytvořeny a již se dále nepoužijí.

mít na výstupu prvek, který je instancí jednoho konceptu nebo generalizovaného konceptu. Třída PovoleněTypyTransformací bude obsahovat 0 až N různých předpisůTransformace.

Vztahy mezi modelem přechodů mezi koncepty a modelem transformací prvků jsou zachyceny (ve zjednodušené podobě) na Obr. 35. Dále jsou zde zachyceny souvislosti mezi metamodelem přechodů mezi koncepty a metamodelem transformací prvků (viz Obr. 24)



Obr. 35 – Vztah mezi metamodelem přechodů mezi koncepty a modelem transformací prvků

7.5 *Diagram přechodů mezi koncepty metody*

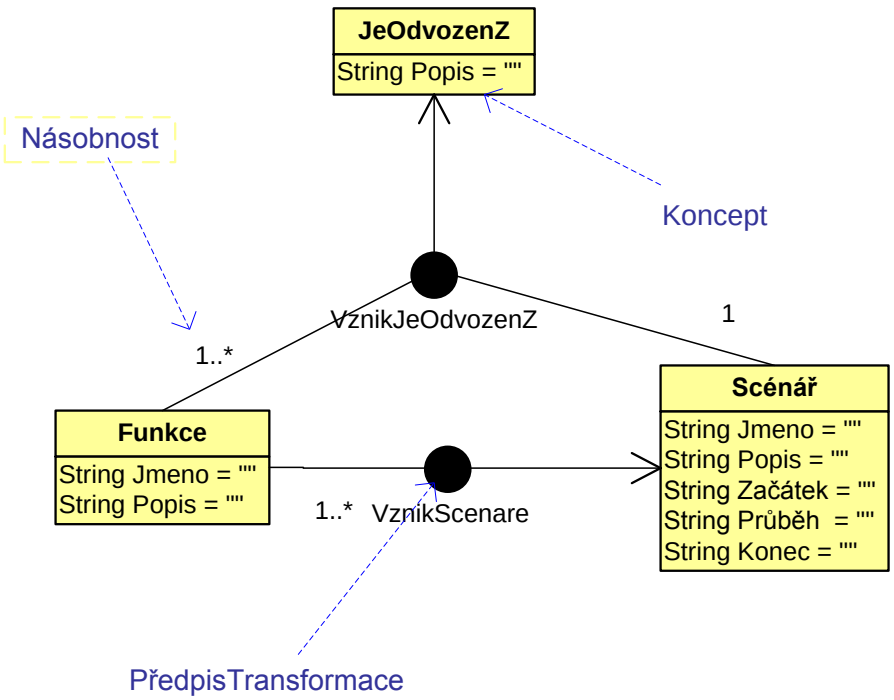
Pro zachycení toho, jaké transformace se mohou v rámci nějaké metody* použít použijeme model přechodů mezi koncepty metody. Víme, že když používáme nějakou metodu (třeba návrhu is), tak její provedení sestává z jednotlivých malých krůčků (my jsme je nazvali transformace) a k nim jsem schopni vytvořit jejich předpisy (v našem případě jsme je pojmenovali předpisy transformací, či přechody mezi koncepty). Tyto transformace na sebe navazují – jsme schopni se pomocí nich dostat od vstupů v výstupům, tedy i jejich předpisy na sebe musejí navazovat. Neboli v modelu musí existovat takové prvky (a také transformace a předpisy k nim), abychom mohli postupně vytvořit prvky výstupní a tedy jednotlivé transformace (a tím i jejich předpisy) musí na sebe navazovat. Můžeme tedy vytvořit „procesní mapu“ metody, na jejímž základě budeme vědět jaké

* Zde metodou myslíme nějaký konkrétní postup vedoucí k vyřešení dílčího problému [Kadlec 2004].

transformace (zadané jejich předpisy) můžeme v dané chvíli použít. Tuto mapu budeme nazývat diagram přechodů mezi koncepty metody, případně používat název diagram (povolených) transformací metody (případně zkráceně model metody).

Abychom mohli definovat podobu diagramu, potřebujeme grafickou podobu jednotlivých prvků diagramu:

- Koncept – použijeme stejný zápis jako má třída v UML. Atributy u tohoto pojmu znamenají, že stejné atributy bude mít i jeho instance (čili prvek).
- GenKoncept – použijeme stejný zápis jako pro třídu v UML a přidáme stereotyp, případně použijeme přerušovanou čáru.
- Implementace – použijeme zápis jako pro implementaci z UML (tj. stejný jako pro generalizaci, ale použijeme přerušovanou čáru).
- Generalizace – použijeme zápis jako pro generalizaci (dědění) z UML.
- PředpisTransformace – na diagramu bude předpis transformace znázorněn pomocí plného kroužku. U něj může být uvedeno jméno předpisu. Předpis vstupní transformace nebudeme do diagramu zapisovat, či pokud bychom ho chtěli více popsat, tak dáme popis do komentáře a spojíme ho se vstupním prvkem. Do předpisu transformace může vstupovat několik různých množin pojmů (každá z nich reprezentovaná třídou VstupKoncept)
- VstupKoncept – reprezentací bude úsečka mezi pojmem vstupujícím do předpisu transformace. U této úsečky a u jejího konce směrem k pojmu, bude napsáno kolik prvků tohoto typu bude při transformaci použito.
- VýstupKoncept – bude znázorněn jako šipka od předpisu transformace k (jednomu) výstupnímu pojmu.
- PovolenéTypyTransformací – tato třída bude odpovídat celému diagramu přechodů mezi koncepty
- Komentář – lze přidat ke každému prvku diagramu. Syntaxe je stejná jako v UML.



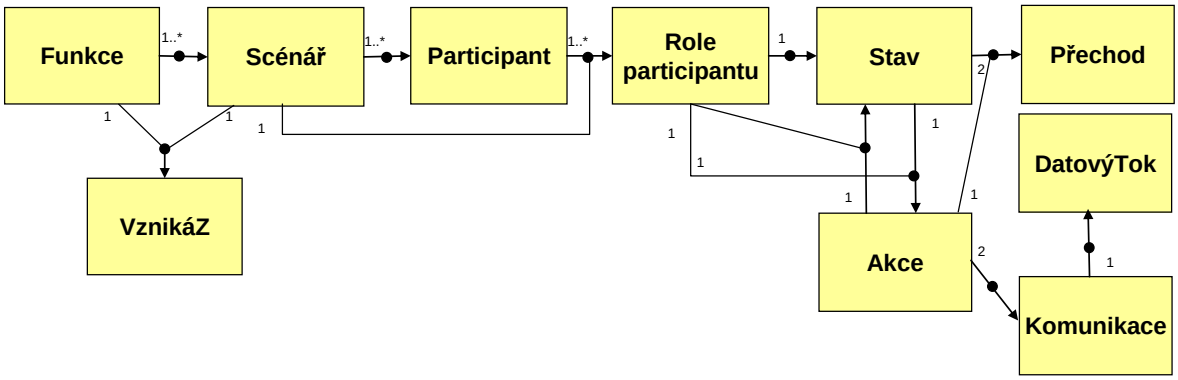
Obr. 36 – Část diagramu přechodu mezi koncepty metody BORM

Na Obr. 36 je ukázána část diagramu přechodů mezi koncepty. Je tam popsán (pomocí 2 předpisů transformací) vznik scénářů a vztahů mezi funkcemi a scénáři. Na začátku v modelu máme nějaké funkce. Nový scénář vzniká transformací na základě 1 až N funkcí. Tato transformace má předpis VznikScénáře. Pokud je v modelu už nějaká funkce i scénář, tak případně můžeme aplikovat druhou transformaci JeOdvozenZ (s předpisem VznikJeOdvozenZ*), která vytvoří prvek JeOdvozenZ, který reprezentuje vztah mezi Funkcemi a Scénářem (Scénář je vytvořen z daných Funkce/cí). Příklad záznamu transformací pro tento příklad je na Obr. 26 a na Obr. 27 je uvedena část diagramu funkcí a scénářů vytvořená pomocí tohoto postupu.

Pomocí takto vytvořeného diagramu můžeme zobrazit metodu jako síť pojmů s možnými (dovolenými) transformacemi mezi nimi. Dovolené transformace v dané metodě lze zjistit z popisu průběhu dané metody. Takto vytvořený diagram pak ukazuje vztahy mezi jednotlivými koncepty metody a lze z něj určit, jaké nové prvky mohou ve vytvářeném modelu vzniknout. Tento diagram nazýváme diagram transformací mezi koncepty.

Příklad tohoto (zjednodušeného) diagramu pro (zjednodušenou) metodu BORM je na Obr. 37.

* Pro zajímavost, tento předpis transformace může proběhnout automaticky.



Obr. 37 – Model transformací (mezi koncepty) metody BORM (zjednodušeno)

7.6 Definice předpisu transformace

Abychom mohli definovat předpis transformací mezi koncepty, potřebujeme k tomu znát jaké vstupy a výstupy může z předpisu odvozená transformace mít. Potřebujeme tedy znát typy (tedy koncepty) vstupů a jejich násobnosti a koncept výstupu. Dále potřebujeme znát, kdy může být daný předpis transformace proveden – potřebujeme tedy znát podmínku, která musí být před provedením transformace splněna. Na jejím základě vybíráme předpisy transformací, které mohou být v daný okamžik provedeny.

Pro definici vlastního algoritmus transformace musíme znát atributy, které modelují vztahy mezi prvky. Tyto atributy jsou „vyplněny“ během transformace a tak jsou vytvořeny vztahy v modelu. Ostatní atributy (například jméno prvku) nejsou transformací měněny.

Pokud definujeme všechny předpisy transformací mezi koncepty v celé metodě, tak potřebujeme informace o všech konceptech metody. Ty získáme z metamodelu metody.

Definice předpisu transformace bude obsahovat :

- **název** – (pokud možno) výstižné pojmenování
- **označení** – signatura předpisu transformace
- **popis a použití** – popis a použití této transformace
- **diagram** – diagram přechodu mezi koncepty dané transformace.
- **podmínky** – musí být splněny před provedením transformace, jinak transformaci nelze provést
- **vstupy** – typy vstupů do transformace a jejich násobnost

- **výstup** – typ prvku[*] který se po provedení předpisu transformace přidá do modelu.
- **realizaci transformace** (algoritmus) – vlastní algoritmus transformace
- **poznámka** – obsahuje doplňující informace.

Transformace používané v metodě postupných transformací prvků jsou obecně velmi jednoduché a jsou typicky dvou typů:

- buďto přidáváme prvek typu objekt[†] a pak se vlastní algoritmus transformace skládá z vytvoření nového prvku, jeho vložení do modelu a vložení záznamu o transformaci do modelu (případně záznamu o předchůdcích prvku).
- nebo přidáváme prvek typu vztah a transformace se skládá z vytvoření nového prvků, vložení do modelu, vytvoření a vložení do modelu záznamu o transformaci a zanesení informace o vztahu do modelu. To typicky děje vyplněním atributů vztahu odkazujících na prvky ve vztahu.

Je vidět, že vlastní transformace jsou velmi jednoduché. Složitější je to s vstupními podmínkami. Ty nám omezují, kdy lze takovou transformaci vykonat. Častokrát je v nich podmínka, že prvky vstupující do transformace musí být „vhodné“. Zajištění této podmínky je na tvůrci modelu. On odpovídá za to, že vybere správné prvky. Na základě těchto vybraných prvků a ostatních vstupních podmínek lze mu nabídnout transformace, které lze z danými prvky provést.

Výběr vhodného způsobu zápisu algoritmu transformace závisí na okolnostech, zda chceme vytvořené předpisy transformací implementovat (pak patrně zvolíme jazyk modelovacího nástroje), jaký (meta)modelovací jazyk daná metoda používá (např. pro modely založené na MOF použiji jazyk QVT), zda má být algoritmus snadno čitelný (použijeme snadno čitelný pseudokód), apod.

7.6.1 Příklad předpisu transformace

Na Obr. 38 je uveden příklad předpisu transformace. Zde předepsaná transformace vytváří z 1..N funkcí jeden scénář. U podmínek provedení transformace nejsou uvedeny takové, které vyplývají z jiných částí definice. Například ze vstupů vyplývá, že vstupem do transformace bude kolekce

[*] Jako výstup se typ prvku uvádí pro zjednodušení a zpřehlednění. Ve skutečnosti je výstupem transformace nový model obohacený o tento prvek.
[†] viz část 6.2 a 6.4.

prvků typu Funkce obsahující minimálně jeden prvek. Vstupní podmínka k tomu doplňuje, že tyto prvky musí být vhodné k této transformaci.

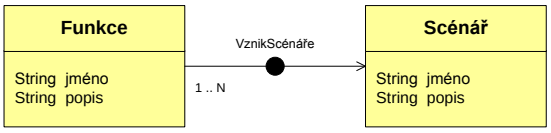
Název: Vznik scénáře

Označení: VznikScenáře(F)

Popis:

Pomocí této transformace vzniká z 1 až N funkcí nový scénář

Diagram:



Vstupy:

Jméno vstupu	Koncept	Násobnost
F	*Funkce*	*1 ... N*

Výstup:

Scénář

Podmínky: `vhodné(F)`

Algoritmus transformace:

```
scenar := Scenar()
model.přidej(scenar)
for fce in F
  scenar.vzniklZ(fce)
```

Poznámka:

Obr. 38 – předpis transformace VznikScénáře

V příkladu na Obr. 38 je použit pseudokód* k popisu algoritmu transformace. Pokud bychom chtěli popsat transformaci například pomocí systému STRIPS, pak je ukázka předpisu transformace uvedena na Obr. 39.

Označení: `VznikScenare(F)`

Podmínky:

`size(F) ≥ 1`

`∀f | f ∈ F , je_typu(f,Funkce) ∧ vhodne(f)`

Algoritmus transformace:

`P: s ∉ M, je_typu(s,Scenar)`

`A: s ∈ M,∀f | f ∈ F, vzniklZ(s,f)`

`D: ∅`

Obr. 39 – předpis transformace zapsaní pomocí STRIPS

7.6.2 Typy transformací podle automatizovatelnosti

V předchozí části jsme řekli, že předpis transformace obsahuje podmínku, při jejímž splnění může být odpovídající transformace vykonána. Na této podmínce (a jejím případném splnění) závisí automatizovatelnost transformace. Ty mohou být:

1. automatické transformace – jejich vstupní podmínky jsou pro daný model splněné (bez dalších vnějších zásahů). Transformace lze tedy provést na modelu automaticky a budou se provádět do té doby, dokud budou vstupní podmínky splněné. Příkladem je přidání třídy, která je generalizací několika dalších tříd. V prvním kroku přidám transformací do modelu novou třídu generalizující jiné třídy, v druhé kroku budu provádět vložení prvku generalizace do modelu. Toto již budu moci provést automaticky, protože u nové třídy vím, z jakých tříd tato třída vznikla.
2. poloautomatické transformace – to jsou transformace takového typu, kde musíme manuálně vybrat vstupní prvky a vybrat typ transformace, která se s následně nimi

* Se syntaxí blízkou jazyku Python

automaticky vykoná. Typickým příkladem je například refaktoring – u moderních vývojářských nástrojů vybereme prvky, které se mají refaktorovat a typ refaktoringu a ten se automaticky vykoná. V metodě postupných transformací jsou poloautomatické takové transformace, u kterých máme jejich předpis (viz Obr. 38) – vybereme vstupní prvky, dále s pomocí vstupní podmínky příslušnou transformaci a ta se provede.

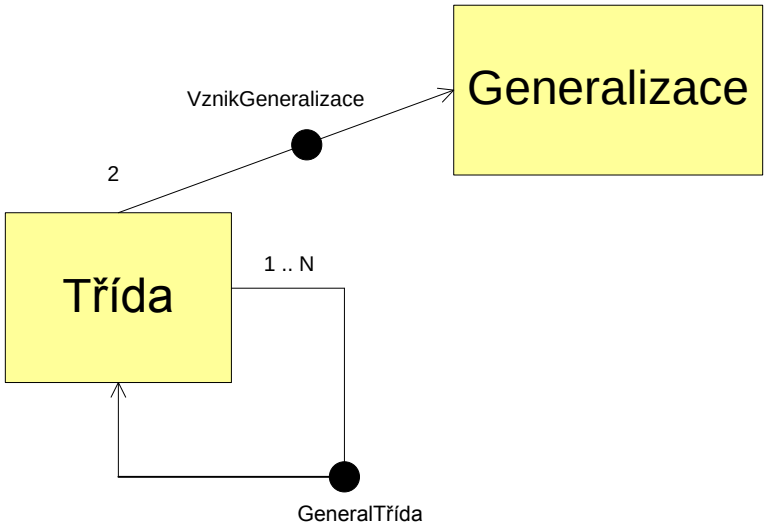
3. manuální transformace – při těchto transformacích neznáme její předpis. Takovým příkladem je použití metody postupných transformací bez použití předpisů transformací (tj. jen podle kapitoly 6) . V tomto případě vložíme nový prvek do modelu, víme o něm že vznikl nějakou transformací[*] a následně v modelu vyhledáme prvky, z kterých tento nový prvek vzniknul. V tomto případě neexistují explicitně vyjádřené vstupní podmínky a za vše (tj. výběr, provedení a zaznamenání transformace) odpovídá tvůrce modelu.

7.7 Složené transformace

V předchozích částech jsme zavedli, že transformace přidává do modelu právě jeden prvek. Takto definovanou transformaci už nelze rozložit na další a je tedy minimální (atomická). Některé transformace však spolu blízce souvisí. Jako příklad lze uvést případ, kdy vytvářím generalizaci několika tříd. V prvním kroku (transformaci) vytvořím novou třídu, která je generalizací několika (vstupních) tříd. Tato nová třída vznikla na základě těchto vstupních tříd. V druhém kroku doplním do modelu prvky typu Generalizace[†]. Tento krok může proběhnou automaticky, protože vím, že generalizovaná třída vznikla pomocí příslušné transformace ze specializovaných tříd. Diagram přechodů mezi pojmy těchto dvou transformací je uveden na Obr. 40.

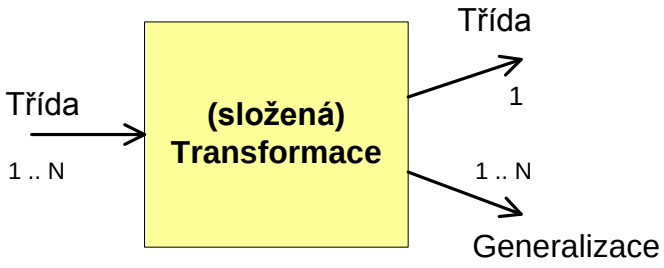
[*] ve smyslu metody postupných transformací

[†] Ty mají (například v UML třídním diagramu) grafickou reprezentaci pomocí (silné) šipky se směrem od předchůdce k následníku.



Obr. 40 – diagram přechodů mezi pojmy pro generalizaci tříd

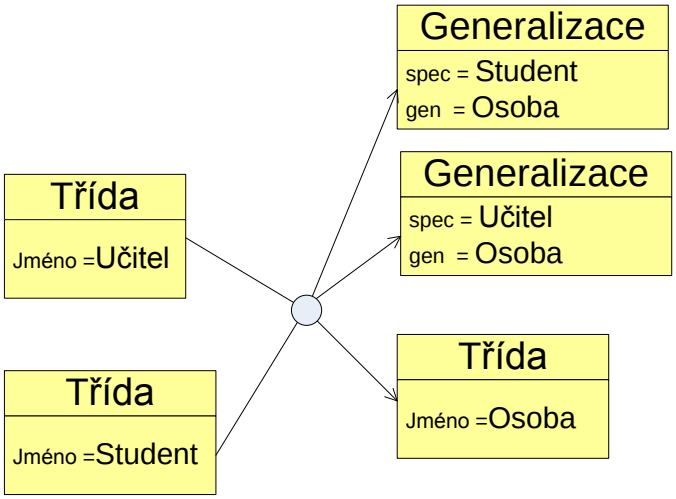
Pokud bych se na tyto dvě transformace díváme jako na jednu složenou transformaci, tak vstupem do ní bude 1 až N tříd a výstupem bude jedna třída, která je generalizací vstupních, a 1 až N generalizací, které reprezentují vztahy mezi vstupním prvkem (třídou) a jeho generalizovaným (zobecněným) prvkem (také třídou). Ukázka zápisu takové složené transformace je na Obr. 41.



Obr. 41 – schematické znázornění předpisu složené transformace

Složená transformace bude přidávat do modelu více prvků. Vstupními prvky složené transformace bude podmnožina všech vstupních prvků (atomických) transformací, přesněji množina všech vstupních prvků jednotlivých transformací bez s množiny všech výstupních prvků jednotlivých

transformací (∪Vst \ ∪Výst)*. Výstupem složené transformace bude nový model s přidanými výstupními prvky jednotlivých transformací (M ∪ ∪Výst).

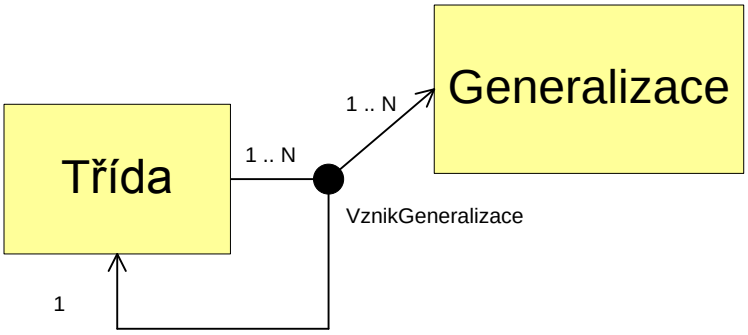


Obr. 42 – příklad vstupů a výstupů složené transformace

Složená transformace je taková, kterou lze rozdělit do více (atomických) transformací†. Jednotlivé transformace na sebe typicky navazují‡, a to tak že výsledek jedné je jedním ze vstupů do další. Předpis složené transformace může mít více vstupů a i samozřejmě více výstupů. Vstupy předpisu transformace budou podmnožinou vstupů jednotlivých předpisů transformací. Výstupy předpisu transformace bude množina všech typů výstupů jednotlivých předpisů transformace.

Příklad grafického znázornění předpisu (tj. diagram přechodu pojmů) generalizace mezi třídami je na Obr. 43 a jedna konkrétní realizace této transformace je na Obr. 42.

* Vst je množina vstupních prvků (atomický) transformací a Výst množina výstupních prvků transformací, které obsahuje složená transformace.
† A to do tolika, kolik prvků do modelu přidává.
‡ Nemusí, transformace obsažené ve složené transformaci mohou být na sobě nezávislé.



Obr. 43 – grafické znázornění složené transformace (tj. diagram přechodů mezi pojmy) pro generalizaci tříd

Složená transformace působí jako „černá skříňka“ do které známe vstupy a její výstupy, ale neznáme její strukturu (tedy jednotlivé atomické transformace).

7.7.1 Metoda jako složená transformace

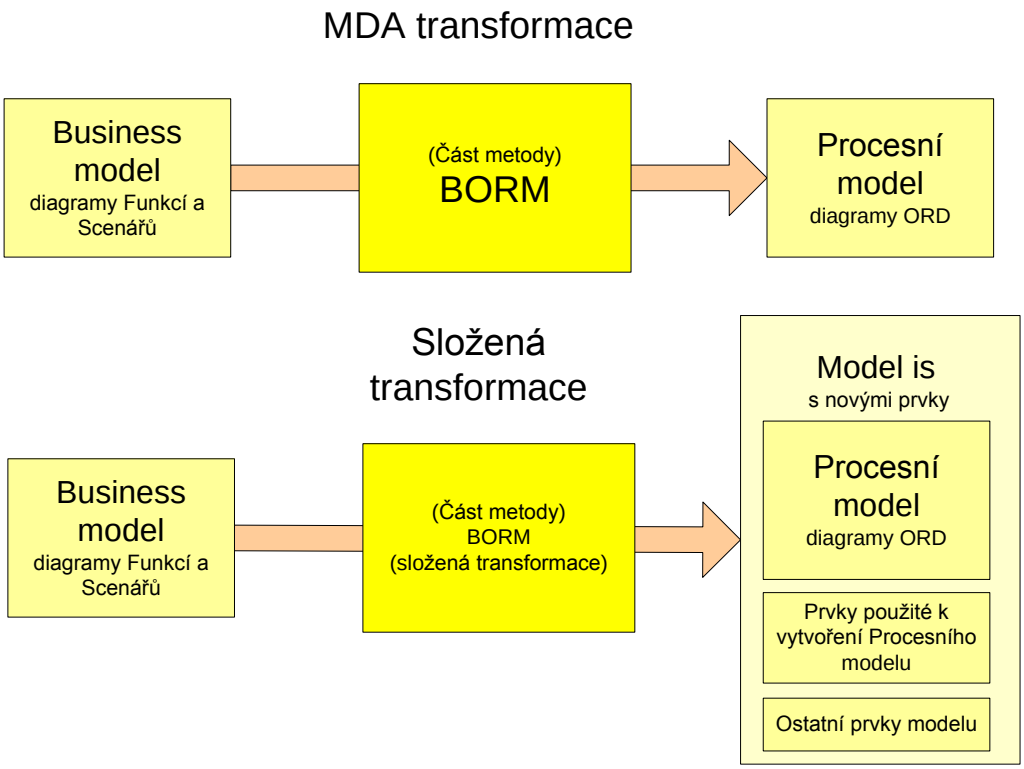
Pokud modelujeme, pomocí metody postupných transformací, část nebo celou metodu*, tak výsledkem bude její model přechodů mezi pojmy. Tento model se bude skládat z předpisů jednotlivých transformací a jejich vzájemných návazností. Tento model můžeme brát jako předpis složené transformace. Pak je pro nás tato metoda „černou skříňkou“ do které známe typy vstupů a po vložení těch „správných“ vstupních prvků získáme hledaný model. Nevýhoda tohoto pohledu je, že nevidíme souvislosti mezi jednotlivými prvky, které se v modelu vytvářejí. Potom je při složitější metodě obtížnější (až skoro nemožné) zvolit hned na začátku ty správné vstupní prvky. Je tedy nutno jako složenou transformaci volit takovou metodu, či její část kde je jasná souvislost mezi jejími vstupy a výstupy a tak jde snadno zvolit správné vstupy. Jako příklad jde zvolit takové (složené) transformace typu refaktoring, kde zvolíme několik vstupních prvků a ty jsou automaticky přetransformovány například v nějaký návrhový vzor.

Pokud se na metodu díváme jako na složenou transformaci , pak je zde jasná souvislost s přístupem k transformacím v MDA (Model Driven Architecture). V MDA je vstupem do transformace nějaký model a výstupem z ní je jiný model. V metodě postupných transformací je vstupem do složené transformace část dosud vytvořeného modelu (je shodná se vstupem do MDA transformace) a

* Příklad pro BORM viz část práce **Chyba! Nenalezen zdroj odkazů.** .

výstupem je model obohacený o nové prvky. Podmnožina* (například vytvářející nějaký diagram) těchto nově vytvořených prvků je shodná s výstupem MDA transformace.

Na obrázku Obr. 44 je ukázána souvislost mezi MDA transformací a složenou transformací.



Obr. 44 – porovnání MDA transformace s složené transformace

7.8 Použití modelu přechodů mezi koncepty

Pokud máme definované všechny předpisy transformací v metodě a tím máme definovaný model přechodů mezi koncepty metody, tak lze tento model s výhodou využít zejména pro:

- Řízení vývojového procesu metodou.
- Kontrolu vytvářeného modelu vůči metodě.
- Automatické, či poloautomatické provádění transformací.

* Ostatní nově vytvořené prvky nám slouží k postupnému přechodu k výstupu MDA transformace.

- Znázornění průběhu metody.
- Kontrolu a vylepšování metod.

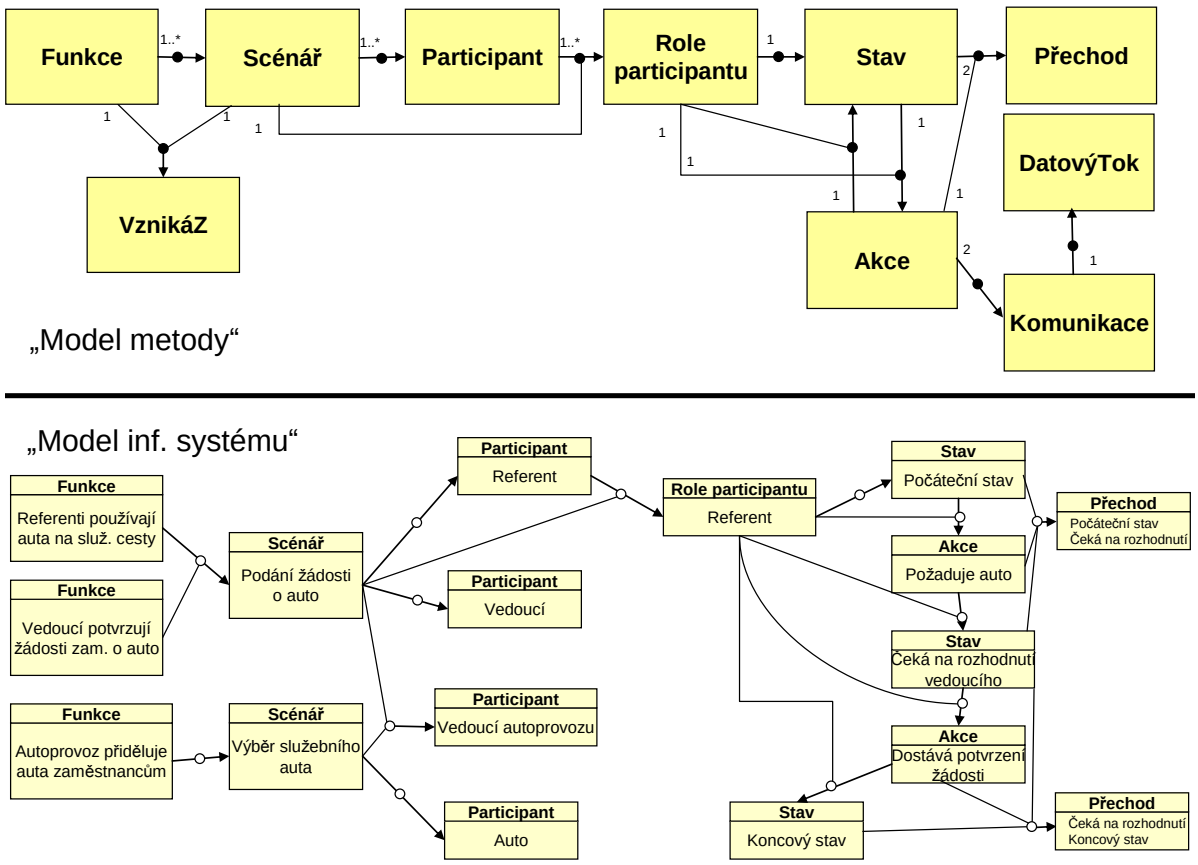
Tyto výhody podrobněji popíšeme v následujícím textu.

7.8.1 Řízení vývojového procesu metodou

Pokud máme definované všechny předpisy transformací v metodě jsme schopni řídit tvorbu modelu vytvářeného informačního systému. V každém momentě jsme schopni určit, jaké transformace jsou v dané chvíli metodou povolené[*]. To zajišťuje, aby vytvářený model informačního systému byl (alespoň formálně – „syntakticky správně“) vytvářen podle dané metody.

Na obrázku Obr. 45 je uveden takový to případ. V jeho horní části znázorněn zjednodušený diagram přechodů mezi koncepty metody (zde BORM) a v dolní části tohoto obrázku je znázorněn diagram postupných transformací mezi prvky informačního systému autoprovozu, který byl vytvořen aplikováním jednotlivých předpisů transformací.

[*] Dovolené transformace jsme schopni určit na základě vybraných vstupních prvků a vstupní podmínce – více viz. část 7.6.



Obr. 45 – Diagram přechodů mezi pojmy metody a jemu odpovídající model is

7.8.2 Kontrola postupu vůči použité metodě

Při použití modelu přechodů mezi koncepty metody jsme schopni zpětně zkontrolovat vytvářený model informačního systému, zda odpovídá předpisům transformacím daných v metodě. Tuto kontrolu provádíme tak, že kontrolujeme zda transformace, kterou prvek vzniknul odpovídá nějakému předpisu transformace.

Tuto kontrolu lze poměrně snadno zabudovat do CASE nástroje a tak může CASE kontrolovat, zda všechny prvky modelu byly vytvořeny v souladu s metodou.

Algoritmus této kontroly je na Výpis 7.

7.8.3 (Polo)automatické provádění transformací

Pokud máme definované všechny předpisy transformací v metodě, tak můžeme všechny transformace, kterými vytváříme model informačního systému provádět automaticky či poloautomaticky (více viz část 7.6).

Automatické transformace se provedou pokud jsou splněny všechny jejich vstupní podmínky. U poloautomatických vybereme vstupní prvky transformace, na základě těchto prvků a vstupní podmínky transformace, jsou nám nabídnuty možné transformace a zvolíme vhodnou transformaci. Ta je provedena a v modelu vznikne nový prvek.

Je také možný opačný přístup – vložíme do modelu nový prvek a na jeho základě jsou nám nabídnuty transformace, pomocí kterých tento prvek může vzniknout[*]. Vybereme vhodnou transformaci a na základě jejího předpisu doplníme vhodné vstupy.

7.8.4 Znázornění metody

Diagram přechodů mezi koncepty[†] metody nám znázorňuje nový pohled na metodu. Ukazuje jakými způsoby (transformacemi) může nový prvek modelu informačního systému vzniknout. Tímto diagramem je metoda ukázána jako posloupnost na sebe navazujících předpisů transformací.

Tento pohled na metodu spolu s jejím metamodelem je vhodný zejména například pro znázornění metody a pro pochopení vztahů v ní. Je velmi vhodný při učení se metodě a také jako reference při jejím používání.

7.8.5 Práce s metodami

Model přechodů mezi koncepty nám pomůže při práci s metodikami. Metodika je typicky složena z více konkrétních metod, které se následně při vlastním vývoji provádějí. Na začátku vývoje informačního systému za pomoci metodiky musíme udělat konkrétní instanci metodiky[‡]. Model přechodů mezi koncepty definuje vstupy a výstupy jednotlivých metod. To určuje, jak a kdy

[*] To jsou ty, které mají daný prvek jako svůj výstup.
[†] Zjednodušený diagram přechodů mezi pojmy metody BORM je Obr. 37 a dopodrobna zpracovaný je v části **Chyba! Nenalezen zdroj odkazů.**.
[‡] Více pro o vytváření instance metodiky je například v [Rumbaugh 1998] .

můžeme jednotlivé metody řadit za sebou[*]. Tak jsme schopni sestrojit konkrétní instanci metodiky – tedy jeden velký model přechodů mezi koncepty.

Další možnost využití modelu přechodů je při kontrole metod. Při vytváření modelu přechodů jsme nuceni „rozebrat“ metodu na nejmenší součásti (tedy koncepty) a následně jí zase „složit“ (pomocí předpisů transformací). Při tom můžeme kontrolovat vnitřní logiku dané metody.

7.9 Závěrečné shrnutí kapitoly

Tato kapitola podává výklad o modelu přechodů mezi pojmy metody. Zavádí základní názvosloví a základní pojmy jako je koncept a předpis transformace. Dále tato kapitola zavádí metamodel diagramu přechodů mezi koncepty a také jeho grafickou podobu. Věnuje se předpisu transformace, který bude definován pomocí těchto bodů:

- název – (pokud možno) výstižné pojmenování
- označení popis a použití – popis a použití této transformace
- diagram – diagram přechodu mezi koncepty dané transformace.
- vstupní podmínky – musí být splněny před provedením transformace, jinak transformaci nelze provést
- vstupy – typy vstupů do transformace a jejich násobnost
- výstup – typ prvku který se po provedení předpisu transformace přidá do modelu.
- realizaci transformace (algoritmus) – vlastní algoritmus transformace
- poznámka – obsahuje doplňující informace.

Věnuje se také pojmu složená transformace její souvislosti s transformacemi v MDA.

Ke konci kapitoly je naznačeno možné použití výše zmíněných principů. Model přechodů mezi koncepty lze zejména využít pro:

- Řízení vývojového procesu metodou.
- Kontrolu vytvářeného modelu vůči metodě.
- Automatické, či poloautomatické provádění transformací.
- Znázornění průběhu metody.

[*] Abych mohl spojit metodu s nějakou jinou, tak potřebuji možnost získat potřebné vstupy z předcházející metody.

- Práci s metodami.

8 Použití metody postupných transformací

V této kapitole jsou uvedeny příklady použití metody postupných transformací. Jsou zde uvedeny algoritmy generující matice dohledatelnosti požadavků na základě modelu informačního systému vytvořeného pomocí metody postupných transformací prvků. Dále jsou zde probrány a navrženy způsoby implementace této metody do CASE nástrojů. Nakonec je zde uvedeno jak postupovat při vytváření modelu přechodů mezi koncepty metody.

8.1 Konstrukce matice dohledatelnosti

Nejobvyklejší způsob zachycení odkazů mezi požadavky a ostatními částmi systému je matice dohledatelnosti požadavků (Requirement Traceability Matrix) – viz část 4.5.4. Nejvíce se používají dva typy zachycení vztahu požadavků s ostatními částmi systému – pomocí tabulky nebo obousměrné matice.

V následujících částech jsou popsány algoritmy generování obou typů matic dohledatelnosti požadavků z modelu vytvořeného pomocí metody postupných transformací

8.1.1 Konstrukce tabulky dohledatelnosti

Tento typ matice (tabulky) dohledatelnosti (Tab. 10) se vytvoří tak, že určíme požadavky jež budeme sledovat, dále určíme u jakých typů prvků modelu budeme evidovat jejich odvození z požadavků. Požadavky budou uvedeny v prvním sloupci tabulky a v dalších sloupcích tabulky budou evidované patřičné odkazy na požadavek. Tabulku vytváříme tak, že postupně, jak vzniká model informačního systému, do řádků vyplňujeme jaké části modelu jsou odvozeny z daného požadavku. Pokud je vztah požadavku k dané části modelu 1:N, tak do jedné řádky dáme několik prvků modelu (viz požadavek UP-29 v Tab. 10). Vztahy mezi požadavkem a prvkem modelu typu M:N budou zachyceny tak, že k danému prvku v tabulce bude více požadavků. Podle [Wiegers 2003] je při ručním vytváření tabulky zachycení vztahů M:N typicky složité a tak v praxi a je těžké je zachytit v plné šíři. Námi navržený algoritmus (Výpis 4) takové problémy nemá a vygeneruje tabulku dohledatelnosi v celé její složitosti.

Uživatelský požadavek	**Funkční požadavek**	**Část návrhu**	**Modul kódu**	**Testovací scénář**
UP-28	katalog.dotaz.třídění	třída Katalog	katalog.setřiď()	hledání 7, 8
UP-29	katalog.dotaz.import	třída Katalog	katalog.importuj() katalog.zkontroluj()	hledání 12,13 hledání 14

Tab. 10 – Příklad tabulky dohledatelnosti požadavků

Vlastní algoritmus vytvoření této tabulky dohledatelnosti požadavků je na Výpis 4. Vstupem do tohoto algoritmu jsou požadavky a seznam typů prvků, které chceme v tabulce sledovat. Algoritmus (pro vygenerování jedné řádky) probíhá tak, že nejdříve najdeme všechny následníky daného požadavku a vybíráme z nich takové prvky, které jsou námi sledovaného typu. Tyto nalezené prvky vložíme do dané řádky. Takto pokračujeme přes všechny zadané požadavky.

```
vytvořMaticiDohledatelnosti(požadavky, typyPrvků)
  matice := Matice()
  for požadavek in požadavky
    řádka := Řádka()
    for následník in najdiNásledníky(požadavek)
      if následník.typ() in typyPrvků
        řádka.přidejPrvek(následník)
    matice.přidejŘádek(řádka)
  return matice


najdiNásledníky(prvek,následek:=Set(),navštívené:=Set())
   if (prvek.naslednik().isEmpty)
      return následek.add(prvek)
   for prv in prvek.naslednik()
     if (not prv in navštívené)
       najdiNásledníky(prv,následek.add(prv),navštívené.add(prv))
```

Výpis 4 – funkce generující matici (tabulku) dohledatelnosti požadavků

8.1.2 Konstrukce obousměrné matice dohledatelnosti

Pomocí tohoto typu (viz Tab. 11) obousměrné matice dohledatelnosti požadavků se dají dobře reprezentovat vztahy mezi dvojicemi požadavků, zejména vztahy typu „specifikuje“, „závisí na“, „je předlohou“ nebo „omezuje/musí brát ohled na“ [Sommerswille 1997]. Vytváří se tak, že zvolíme dvě množiny prvků (jedna z nich je typicky množina požadavků), prvky z první skupiny budou řádky a druhé sloupce a v matici vyplňujeme závislosti mezi nimi.

Funkční požadavek	**Případ užití**			
	PU-1	PU-2	PU-3	PU-4
FP-1	•			
FP-2	•			
FP-3			•	
FP-4			•	
FP-5		•		•
FP-6			•	

Tab. 11 – obousměrná matice dohledatelnosti

Algoritmus vytvářející tento typ matice dohledatelnosti prvků je jednoduchý (viz Výpis 5). Vstupem do funkce `vytvořMaticiDohledatelnosti` jsou dva seznamy prvků. Jeden reprezentující požadavky (ty budou v řádkách) a druhý reprezentující prvky obsažené ve sloupcích. Pro každý požadavek (ze seznamu požadavky) budou nalezeny všechny jeho následné prvky. Ty jsou vyhledány pomocí funkce `najdiNásledníky` – což je funkce, která pomocí rekurzivního (zde do šířky) procházení modelu najde všechny následníky daného prvku. Z vyhledaných následníků jsou následně vybráni ti, kteří jsou také v seznamu prvků ve sloupcích (druhém vstupu)[*]. Následně jsou do matice přidány uspořádané dvojice vztahů požadavek, prvek ze sloupce (zde je matice reprezentována objektem, ale mohl by to být i seznam). Tento algoritmus předpokládá, že požadavky budou předchůdci prvků ve sloupcích (což je logický předpoklad).

[*] Provede se průnik mezi vyhledanými následníky a druhým vstupem.

```
vytvořMaticiDohledatelnosti(požadavky, prvky)
  matice := Matice()
  for požadavek in požadavky
    násl := najdiNásledníky(požadavek)
    for prvek in prvky
      if prvek in násl
        matice.přidejVztah(požadavek, prvek)
  return matice


najdiNásledníky(prvek,následek:=Set(),navštívené:=Set())
   if (prvek.naslednik().isEmpty)
      return následek.add(prvek)
   for prv in prvek.naslednik()
     if (not prv in navštívené)
       najdiNásledníky(prv,následek.add(prv),navštívené.add(prv))
```

Výpis 5 – funkce konstruující oboustrannou matici dohledatelnosti požadavků

Snadnou úpravou získáme takový algoritmus (viz Výpis 6), který vyhledá vztahy typu „prvek je předchůdce nebo následník“ mezi jakýmikoliv dvěma množinami prvků. Zatímco u předchozího algoritmu jsme mohli předpokládat, že prvky v řádcích budou požadavky a tím to budou předchůdci prvků ve sloupcích, tak zde toto nemůžeme předpokládat. Takže vyhledáme k danému prvku všechny jeho předchůdce i náledníky a do matice si poznamenáme oba dva vztahy (tj. „prvek vznikl z“ a „z prvku vznikl“).

```
vytvořMaticiVztahů(prvkyŘadky, prvkySloupce)
  matice := Matice()
  for prvekR in prvkyŘadky
    násl := najdiNásledníky(prvekR)
    pred := najdiPředchůdce(prvekR)
    matice.přidejVztahyPN(prvekR, průnik(násl,prvkySloupce))
    matice.přidejVztahyNP(prvekR, průnik(před,prvkySloupce))
  return matice


najdiPředchůdce(prvek,předchůdce:=Set(),navštívené:=Set())
   if (prvek.předchůdce().isEmpty)
      return předchůdce.add(prvek)
   for prv in prvek.předchůdce()
     if (not prv in navštívenéPrvky)
       najdiPředchůdce(prv,předchůdce.add(prv),navštívené.add(prv))
```

Výpis 6 – funkce konstruující matici vztahů mezi prvky

8.2 Implementace metody postupných transformací prvků

Minimální vlastnost, kterou požadujeme u CASE nástroje, aby mohl využívat některé vlastnosti metody postupných transformací prvků, je schopnost zaznamenat prvky z kterých nový prvek vzniknul, tedy zachytit vztah „prvek vzniknul z“[*]. Tedy jinými slovy potřebujeme schopnost zaznamenat transformaci, její vstupní prvky a transformací vzniklý (výstupní) prvek. Dále samozřejmě potřebujeme, aby CASE nástroj obsahoval programovací jazyk s přístupem k repositáři modelu.

S takto vytvořeným modelem (a s pomocí programovacího jazyku) pak jsme schopni využít (podrobněji zejména část 6.5):

[*] Potřebujeme zachytit předchůdce daného prvku.

- K lepší dokumentaci průběhu tvorby informačního systému.
- Ke kontrole modelu informačního systému, zda vyhovuje metodě postupných transformací.
- K najití požadavků, na základě kterých byl daný prvek do modelu přidán.
- K automatickému vytvoření matic dohledatelnosti požadavků.
- Lokalizovat změny v modelu.
- Kontrolovat postup vytvoření modelu vůči metodě (viz část 7.8.2)

S takovouto „minimální“ implementací je postup tvorby modelu informačního systému takový, že klasicky přidáváme prvky do modelu a předchůdce, z kterých prvek vzniknul explicitně* řešíme až po přidání prvku.

Dalším stupněm implementace („plná“ implementace) je přidání transformací a jejich předpisů. Potom již můžeme nejen zaznamenat transformace, ale i v CASE nástroji bude uložen model používané metody návrhu systému. Pokud máme v CASE nástroji plně implementovánu metodu postupných transformací, tak CASE podpoří (podrobněji část 7.8):

- Řízení vývojového procesu metodou.
- Kontrolu vytvářeného modelu vůči metodě.
- Automatické, či poloautomatické provádění transformací.

S takovouto implementací budeme tedy moci provádět kontroly modelu vůči metodě, nabízet vývojáři vhodné transformace a následně je provést, některé transformace provádět automaticky apod.

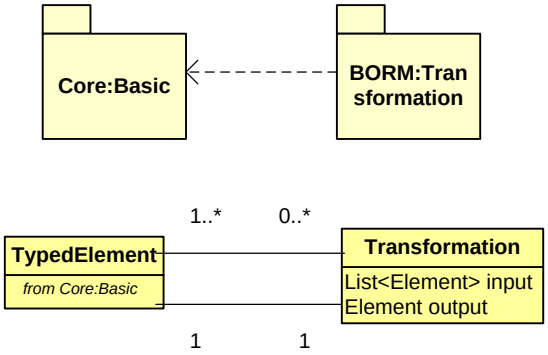
8.2.1 Minimální implementace transformací do CASE nástroje

Pokud chceme provést minimální implementaci metody postupných transformací do CASE nástroje, tak potřebujeme být schopni zaznamenávat transformace – alespoň to, že nějaká transformace proběhla a jaké byly její vstupní a výstupní prvky. V nejjednodušším případě nám stačí, aby si každý prvek pamatoval své předchůdce, tedy z jakých prvků vzniknul. Nejjednodušší způsob jak toho dosáhnout je, aby každý prvek měl jako atribut kolekci, do které by se ukládali jeho předchůdci. Nevýhodou tohoto řešení je, že měním každý prvek modelu a tak již metoda postupných transformací není „přídavná“ vrstva modelu, ale je jeho přímou součástí. Dále budu mít problémy při implementaci, typicky je základní metamodel v CASE nástroji „zadrátovaný“ a

* Tj. je na zodpovědnosti vývojáře přidávání odkazů na (vstupní) prvky a udržování jejich konzistence.

tak nám většina CASE nástrojů nedovolí měnit prvky svého metamodelu*. Tento způsob je tedy vhodný pro případ, kdy implementujeme metodu do nějakého Meta-CASE nástroje, nebo programujeme celý CASE od začátku.

Další způsob minimální implementace je přidáním třídy transformace do metamodelu. Tato třída má odkazy na své vstupy a na svůj výstup. Výhodou tohoto způsobu implementace je, že prvky do metamodelu přidáváme a neměníme stávající. Model vytvořený v CASE nástroji jak s použitím postupných transformací, tak bez ní bude z velké části stejný – k modelu přibude vrstva transformací. Další výhodou tohoto řešení je snadná implementovatelnost – stačí jen přidat jeden prvek do metamodelu a nic jiného se nemusí měnit. Na obrázku Obr. 46 je ukázáno rozšíření (meta)-metamodelu eMOF† o transformace. K metamodelu eMOF přidáváme třídu *Transformation*, která má odkazy na třídu typu *TypedElement* z balíčku *Core:Basic*. Struktura tohoto balíčku metamodelu eMOF je na Obr. 9.



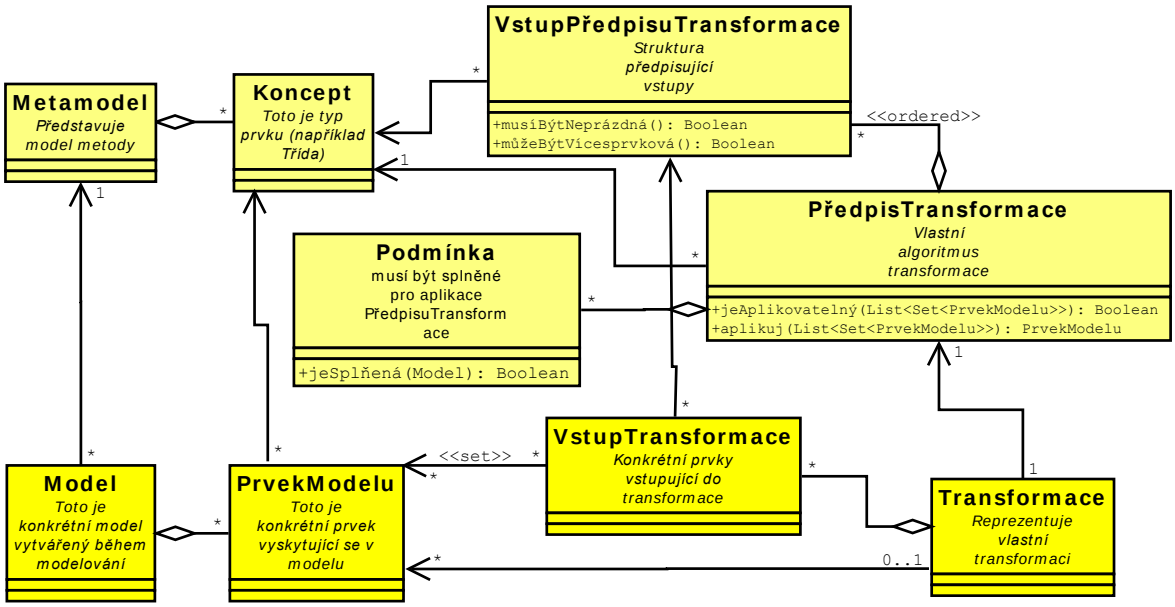
Obr. 46 – implementace metody postupných transformací do eMOF

8.2.2 Plná implementace transformací do CASE nástroje

Pro plnou implementaci metody postupných transformací do CASE nástroje potřebujeme nejen implementovat transformace, ale potřebujeme i implementovat předpis transformace. Na obrázku Obr. 47 je příklad jak obecně implementovat metodu postupných transformací. Barevně jsou odlišeny prvky vlastního modelu a metamodelu (tedy modelu metody). Prvky modelu (*PrvekModelu*, *VstupTransformace* a *Transformace*) budou uloženy v repositáři modelovacího nástroje a prvky modelu metody (*Koncept*, *Podmínka*, *VstupPředpisuTransformace* a

* Ale některé CASE nástroje nám dovolí rozšiřovat svůj metamodel.
† Tedy metamodelu skupiny OMG, na kterém je založeno např. UML.

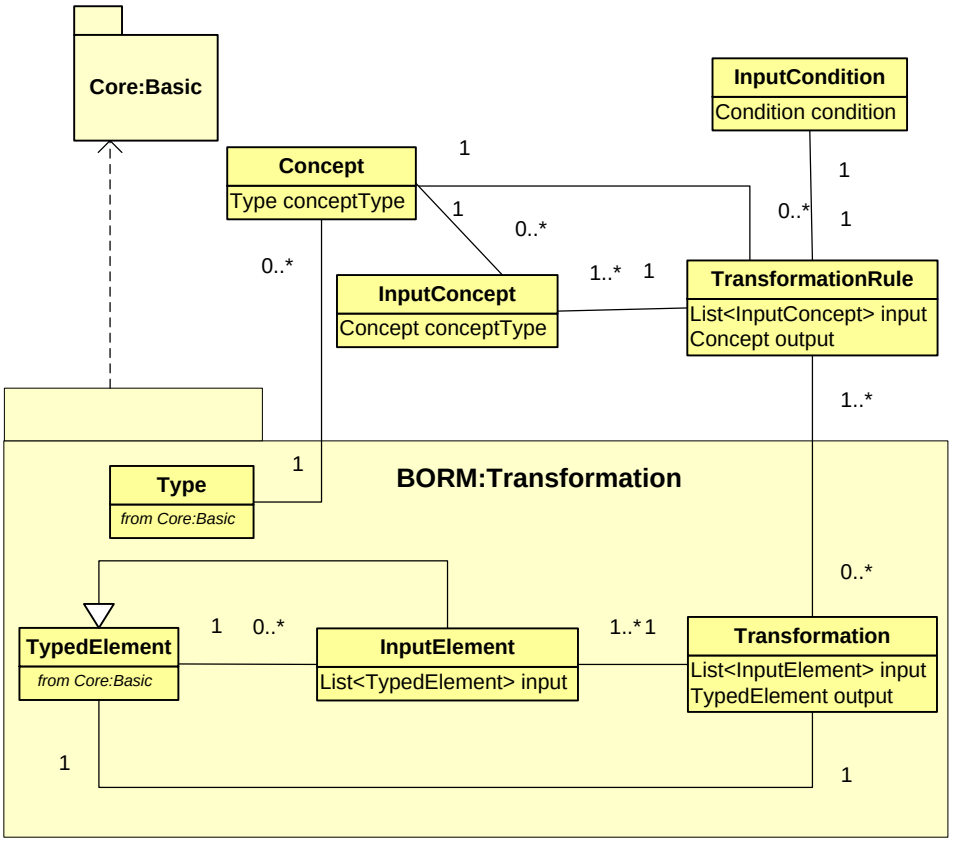
PředpisTransformace) budou uloženy vně repositáře – nejspíše budou zapsány v programovacím jazyce daného CASE nástroje a uloženy mimo v souboru*. To je z toho důvodu, aby v repositáři byl jenom model systému s přidanou vrstvou transformací. Modelovací nástroj bude pracovat tak, že na základě vývojářem vybraných prvků a možných vstupů transformace (třída *VstupPředpisuTransformace*) a vstupní podmínky (implementované v *Podmínka*) vybere vhodný předpis transformace a nabídne ho vývojáři. Po odsouhlasení ho provede a přidá do modelu nový prvek na základě algoritmu v *PředpisTransformace*.



Obr. 47 – Možná implementace metody postupných transformací

Na obrázku Obr. 47 je znázorněna plná implementace postupných transformací založená na metamodelu eMOF.

* Uložení bude záviset na tom, co daný modelovací nástroj dovolí.



Obr. 48 – plná implementace metody postupných transformací do eMOF

8.2.3 *Kontrola modelu vůči metodě*

Pro kontrolu modelu vůči metodě návrhu, jakou byl vytvořen, potřebujeme mít informace o všech typech transformací, které jsou v metodě povoleny. Minimální informace o předpisu transformace, kterou potřebujeme ke kontrole, je informace o povolených typech vstupů do transformace a jejím typu výstupu. Potřebujeme tedy vědět jaké koncepty a jejich násobnosti* jsou na vstupu a koncept na výstupu předpisu transformace. Takto jsme schopni zkontrolovat, zda je model vytvořen formálně správně podle dané metody (tedy zda model dodržuje „syntaxi“ metody).

* Například do transformace vytvářející scénář (viz Obr. 26) může vstoupit 1..N funkcí, tedy násobnost vstupu, který je typu Funkce (neboli koncept Funkce) bude 1..N. Dá se to představit jako proměnný počet parametrů typu Funkce na vstupu do předpisu transformace.

Na Výpis 7 je uveden výpis implementace kontroly modelu informačního systému vůči metodě. Proměnná `MožnéTransformace` obsahuje seznam všech možných typů transformací definovaných metodou. Tyto možné typy transformací jsou popsány třídou `Transformace`. Ta obsahuje seznam povolených typů vstupů (v instanční proměnné `možnéVstupy`) a výstupu (`možnýVýstup`). transformace. Funkce `zkontrolujModel` testuje každý prvek modelu, zda pro něj existuje nějaká možná transformace s danými vstupy a výstupem. Pokud neexistuje, tak se chyba zaznamená. Tato kontrola se děje voláním funkce `zkontrolujTransformaci`. Vlastní kontrola vstupů a výstupu se děje pomocí metod třídy `Transformace testVstupy` s `testVýstup`.

Uvedený způsob kontroly je vhodný i pro minimální implementaci metody postupných transformací prvků. V modelu máme uloženy informace o vstupech a výstupech jednotlivých transformaci. Vlastní možné typy transformací jsou zapsány jako data programu, který provádí kontrolu.

Pokud máme úplnou implementaci metody postupných transformací, tak teoreticky nemusíme žádnou kontrolu – každá provedená transformace byla provedena podle předpisu a tak by měla být „syntakticky“ správná.

```
MožnéTransformace:=… #seznam všech povolených typů trans. v metodě

zkontrolujModel(model)
  for prvek in model.dejPrvky()
    if (zkontrolujTransformaci(prvek) = CHYBA)
      zaznamenejChybu(prvek)

zkontrolujTransformaci(prvek)
  for tr in MožnéTransformace
    if tr.testVystup(prvek)
      if tr.testVstupy(prvek.predchudce())
        return OK
  return CHYBA

class Transformace()
  možneVstupy:=… #seznam možných typů vstupů a jejich možný počet
  možnýVýstup:=… #typ možného výstupu

  def testVstupy(vstupy)
    tmpTVstupy := map(na_typ(vstupy))
    zbyvajiciVstupy := tmpVstupy
    for typ in možneTypy
      tmpVstupy:=filter(not je_typu(typ), zbyvajiciVstupy)
        if not typ.nasobnost(size(filter(je_typu(typ),tmpTVstupy))
          return false
    if size(zbyvajiciVstupy) = 0
      return true
    return false

  def testVystup(výstup)
   if možnýVýstup.typ(je_typu(výstup))
     return true
   else
     return false
```

Výpis 7 – funkce kontrolující model vůči metodě

8.3 Vytvoření modelu přechodů mezi koncepty metody

Pro vytvoření modelu přechodů mezi koncepty metody budeme potřebovat co nejvíce znalostí o popisované metodě. Budeme zejména potřebovat:

- Metamodel metody – v něm jsou formálně zaznamenány jednotlivé elementy s kterými metoda pracuje a vztahy mezi nimi.
- Popis metody – ne vše se dá zachytit metamodelem. V popisu metody je zaznamenáno, jak se metoda používá, jak jednotlivé prvky vznikají a používají se. Čím bude popis metody formálnější, tím konstrukce jejího modelu bude snazší. Bohužel popis metody je častokrát dán pouze neformální, textovou formou. Popis můžeme např. získat z oficiální dokumentace (pokud existuje) k metodě, z různých učebnic, atd. Dobrý zdroj informací jsou CASE nástroje implementující danou metodu.
- Zkušenost s metodou – potřebujeme co nejvíce (nejlépe vlastních) zkušeností s danou metodou.

Vlastní postup bude tento:

1. Danou metodu se pokusíme rozdělit na menší části, které tvoří logický celek. Vhodná místa na rozdělení metody jsou například ty, které se věnují konstrukci jednoho diagramu. S metodou rozdělenou na menší části se lépe pracuje.
2. Z metamodelů takto získaných částí metody získáme kandidáty na koncepty. Kandidáti na koncepty budou takové elementy metamodelu, s kterými návrhář IS pracuje (například jsou použity v nějakém diagramu). Získané kandidáty rozdělíme[*] na ty, jejichž instance budeme používat jako prvky a na ty, které budeme používat jako atributy. Koncepty budou takové elementy metamodelu u kterých nás zajímá jejich vznik a to jak ovlivňují ostatní koncepty.
3. Vybereme vstupní koncepty – to jsou takové, které jsou vstupem do této části metody. Pokud budeme potřebovat přidáme nové koncepty. častokrát potřebujeme doplnit vstupní koncepty.
4. Z popisu metody získáme jednotlivé přechody (neboli předpisy transformací) mezi koncepty – tj. jaké jiné koncepty jsou potřeba k vzniku jednotlivých konceptů, dále určíme vstupy jaké budou vstupy do jednotlivých předpisů transformací.

[*] Toto rozdělení na koncepty a atributy bude různé, podle toho na co daný model budeme používat.

5. Pokusíme se minimalizovat počet předpisů transformací pomocí vytvoření hierarchií generalizovaných konceptů (viz část 7.4.1). V této chvíli můžeme vytvořit diagram přechodů mezi koncepty metody.
6. Popíšeme jednotlivé předpisy transformací (viz část 7.6) – tj. popíšeme vstupy a výstup transformace, vstupní podmínky a vlastní algoritmus transformace.

Tento postup je samozřejmě iterativní a můžeme se vracet k jakémukoliv předchozímu bodu a postupně vylepšovat tento model přechodů mezi koncepty metody, až bude vyhovovat našim požadavkům.

Příklad takto vytvořeného modelu přechodů mezi koncepty metody a popisu jednotlivých předpisů transformací je pro metodu BORM v části Příloha C**Chyba! Nenalezen zdroj odkazů.**.

8.4 Závěrečné shrnutí kapitoly

V této kapitole jsou podrobněji probrány některé zajímavé příklady použití metody postupných transformací:

- Automatická konstrukce matic dohledatelnosti z modelu informačního systému vytvořeného pomocí metody postupných transformací
- Možnosti implementace metody postupných transformací do CASE nástrojů.
- Návod pro vytvoření modelu přechodů mezi koncepty zkoumané metody.

9 Závěr

9.1 Shrnutí a diskuze

V rešerši jsem se zabýval problematikou analýzy a návrhu softwarových projektů. Zabýval jsem se zejména způsoby, jakými se snažíme o zlepšení úspěšnosti jejich tvorby. A to jak dnes již klasickými částmi softwarového inženýrství, jako jsou životní cykly vývoje softwaru, různými paradigmaty programování a metodikami, tak i moderními jako je například metamodelování či transformace modelů. Věnoval jsem se také problematice softwarových požadavků.

Na základě takto získaných znalostí bylo zřejmé, že práce s požadavky je jednou z nejdůležitějších částí procesu vývoje informačních systémů. Zaměřil jsem se zejména na sledování a dohledatelnost požadavků, což nám pomáhá k dodržování a dokumentaci životního cyklu tvorby softwaru.

Na základě toho, jak pracuje vývojář při tvorbě modelu informačního systému a toho, že jsem chtěl zajistit sledování a dohledatelnost všech prvků modelu informačního systému navrhl jsem tyto základní postuláty metody, kterou jsem nazval metoda postupných transformací prvků. Tyto postuláty jsou (viz část 6.3):

1. Každý nový prvek, přidávaný do modelu informačního systému, musí mít smysl.
2. Nový prvek, mimo vstupních, vzniká na základě již v modelu existujících prvků pomocí transformace.
3. Transformace mění původní model na nový přidáním právě jednoho nového prvku do stávajícího modelu.
4. Do modelu lze zvnějšku přidat nový (tzv. vstupní) prvek pomocí (vstupní) transformace. Tento prvek nevzniká na základě prvků v modelu již existujících a je dodán zvnějšku.

Následně jsem v části 6.3.3 zkoumal, jaké vlastnosti bude mít model systému vytvořený na základě těchto postulátů (kapitola 6). Pro zachycení proběhnuvších transformací v modelu jsem navrhnul metamodel a grafickou prezentaci těchto transformací – diagram transformací mezi prvky (část 6.4).

Model vytvořený pomocí metody postupných transformací nám umožňuje (část 6.5):

- Lepší dokumentaci průběhu tvorby informačního systému.
- Zprůhlednit vztahy v modelu a tím lépe zajistit jeho konzistenci.

- Provádět některé kontroly modelu informačního systému.
- Najít požadavky, na základě kterých je daný prvek v modelu.
- Automaticky vytvořit matice dohledatelnosti požadavků.
- Lokalizovat změny v modelu

Už toto jednoduché rozšíření o postupné transformace standardního modelu nám umožňuje některé unikátní věci – zejména podrobnou dokumentaci o vlastním procesu tvorby informačního systému, automatické generování matic dohledatelností, či hledání požadavků, díky kterým je daný prvek v systému. Cenou, kterou za to musí vývojář platit je však složitější vytváření vlastního modelu, kdy musí vývojář u každého nového prvku určit, na základě jakých prvků v modelu vznikl. Toto se snaží odstranit druhá část práce, která se snaží nalézt všechny předpisy transformací obsažené v metodě návrhu informačního systému a tím vývojáři pomoci s hledáním prvků, z kterých nový prvek vzniknul.

V této části práce se stává důležitým pojem koncept. Prvek je instancí konceptu (tedy je to „datový“ typ). Koncepty určují typy prvků, které vstupují do předpisu transformace. Předpis transformace ještě obsahuje podmínku. Předpis se může provést až v tom okamžiku, kdy existují (jsou vybrány) prvky vhodných typů a je splněna podmínka. Během provedení předpis přidá jeden prvek do modelu pomocí transformace. Koncepty, spolu s předpisy transformací tvoří model přechodů mezi koncepty. Pokud jsou v tomto modelu přechodů všechny koncepty a všechny předpisy transformací nějaké metody, můžeme mluvit o modelu přechodů v metodě.

Tento model přechodů mezi koncepty se může použít k (viz 7.8):

- Řízení vývojového procesu metodou.
- Kontrole vytvářeného modelu vůči metodě.
- Automatickému, či poloautomatickému provádění transformací.
- Znázornění průběhu metody.
- Práci s metodami.

S implementovaným modelem přechodů mezi koncepty se mění i práce vývojáře tak, že vybere vstupní prvky a CASE nástroj mu nabídne možné transformace k vykonání. Vývojář si jednu vybere a ta se provede a přidá jeden prvek do modelu systému. Pokud jde vykonat nějaká transformace automaticky, tak se vykoná. Druhý způsob práce by mohl být opačný – vývojář vloží jeden prvek do modelu systému a CASE mu nabídne prvky, z kterých musí vybrat nějaké vstupní. Takový způsob práce je to daní za to, že jednak je při vývoji (formálně) dodržovaná metoda návrhu

a může následně použít výhodné vlastnosti modelu vytvořeného pomocí metody postupných transformací prvků (viz části 6.5 a 7.8)

Metoda postupných transformací může být implementována do CASE nástroje buď „minimalisticky“, kdy vývojář může jen ručně k novému prvku přidávat jeho předchůdce (část 8.2.1) a „plně“ kdy jsou mu nabízeny vhodné transformace (viz část 8.2.2).

Metodu postupných transformací jsem aplikoval na metodu BORM a navrhnul jsem její model přechodů mezi pojmy.

9.2 Rekapitulace cílů práce

Hlavním cílem této práce bylo navrhnout metodu návrhu informačních systémů, která zajistí vzájemnou dohledalnost jejích prvků. Tento cíl splnila metoda postupných transformací mezi prvky Ta byla navržena v kapitolách 6 a 7. Její aplikace byly popsány v kapitole 8.

K tomuto cíli přispěly tyto postupné výsledky :

1. Metoda postupných transformací která se skládá z:
    - Z terminologie (část 6.2 a 7.3), postulátů (část 6.3) a z nich vyplývajících vlastností (6.3.3) a z návrhů využití této metody (část 6.5 a 7.8).
    - Modelu postupných transformací prvků a jeho diagramu (část 6.4).
    - Modelu přechodů mezi koncepty a jeho diagramu (část 7.4).
    - Postupu pro vytvoření modelu postupných transformací (část 8.3).
2. Navržené způsoby implementace metody postupných transformací do CASE nástrojů (část 8.2).
3. Algoritmy generující matice dohledatelnosti požadavků (část 8.1)), hledající požadavky k zadanému prvku (část 6.5.4), hledající prvky zasažené změnou (část 6.5.6) a kontrolující formální správnost modelu vůči použité metodě (část 8.2.3).
4. Model přechodů mezi koncepty metody BORM a definice jejích předpisů transformací (viz Příloha C).

9.3 Náměty do budoucna

Výzkum týkající se metody postupných transformací prvků je teprve na začátku a otevírá do budoucna mnohé zajímavé praktické i teoretické oblasti výzkumu:

1. Asi nejdůležitější pro rozšíření a používání metody postupných transformací prvků je CASE nástroj. V současné době začíná práce na jeho návrhu a implementaci. Některé náměty týkající se implementace a požadované vlastnosti jsou publikovány v článku „CASE Tools: Is something still missing?“ [Pícka 2009b].
2. Další důležitou prací bude aplikovat metodu postupných transformací prvků na další metody a metodiky návrhu softwaru.
3. Více bude potřeba prozkoumat složené transformace, protože by mohli znamenat zjednodušení a zpřehlednění modelu přechodů mezi koncepty metody.
4. Bude potřeba prozkoumat souvislosti této metody s MDA a případně využít nástroje MDA zavedené (QVT, transformace v eMOF atd.)
5. Bude potřeba prozkoumat souvislosti mezi modelem přechodů mezi koncepty a složtostí metod. Počet možných typů transformací mezi koncepty metody by mohl odpovídat míře složitosti metody.

10 Literatura

10.1 Vlastní publikace autora

10.1.1 Publikace v časopisech

[Pícka 2005a] Picka. M.: Metamodeling and Development of Information Systems. In Zemedelska ekonomika (Agrarien economics) no 50 (2004/2). Ustav zemedelskych a potravinářských informací. Praha. 2004. ISSN 0139-570X

[Pícka 2008a] Pícka, M.: Development of Informations System Using Method of Gradual Transformation. In: Scientia Agriculturae Bohemica, 39, 2008, p. 61-66. ISSN 1211-3174

10.1.2 Skripta

[Merunka et al. 2004] Merunka V., Pergl R., Pícka M.: Objektově orientovaná tvorba software, ČZU Praha, 2004, ISBN 80-213-1159-2

[Merunka et al. 2005] Merunka V., Pergl R., Pícka M.: Objektově orientovaný přístup v projektování informačních systémů, skriptum ČZU PEF Praha 2005, ISBN 80-213-1352-8

10.1.3 Příspěvky ve sbornících konferencí

[Pícka 2003a] Pícka M.: Classification of BORM metodology. In Sborník příspěvků z doktorandského semináře, PEF ČZU, Praha 2003, ISBN 80-213-1016-2

[Pícka 2003b] Pícka M.: Metamodelování v praxi, In Sborník z mezinárodní vědecké konference Agrární perspektivy XII, PEF Česká zemědělská univerzita, Praha 2003, ISBN 80-213-1056-1

[Pícka 2003c] Pícka M.: Definice omezení v metamodelu BORMu pomocí jazyka OCL, MendelNET 2003 - Sborník příspěvků z konference studentů doktorského studia, MZLU, Brno

[Pícka 2003d] Pícka M.: Metamodel BORMu jako rozšíření metamodelu UML, Objekty 2003 - sborník příspěvků osmého ročníku konference, Fakulta elektrotechniky a informatiky, VŠB - Technická univerzita Ostrava, Ostrava 2003

[Pícka 2004a] Pícka M.: Záznam procesu tvorby informačního systému, In Sborník příspěvků z doktorandského semináře, PEF ČZU, Praha 2004, ISBN 80-213-1150-9

[Pícka 2004b] Pícka M.: Aspektově orientované programování. In Tvorba softwaru 2004 – sborník přednášek, Tanger sro, Ostrava 2004, ISBN 80-85988-96-8

[Pícka 2004c] Pícka M.:: Postupný vývoj informačního systému. In Sborník z mezinárodní vědecké konference Agrární perspektivy XIII, Česká zemědělská univerzita, Praha 2004, ISBN 80-213-1190-8

[Pícka 2004d] Pícka M.: Řízený vývoj informačního systému. , Objekty 2004 - sborník příspěvků devátého ročníku konference, PEF Česká zemědělská univerzita, Praha 2004.

[Pícka 2004e] Pícka M.: Pokus o sondu studijních předpokladů pro informatiku. Sborník příspěvků Informatika XV. PEF MZLU.2004

[Pícka 2005a] Pícka M. Temporální datový sklad jako integrační prvek znalostního systému organizace. Firma a konkurenční prostředí 2005 - Firm and competitive environment. Konvoj s.r.o. 2005. ISBN 80-7302-097-1

[Pícka 2005b] Pícka M Spojení business a informačního inženýrství pomocí metody BORM. In: Modelování a optimalizace podnikových procesů - sborník 8, ročníku mezinárodního semináře. 2005.ISBN 80-7043-352-3

[Pícka 2005c] Pícka M. Použití transformací v metodě BORM v nástroji Craft.CASE 2.x. Objekty 2005 - Sborník příspěvků desátého ročník konference. 2005. ISBN 80-248-0595-2

[Pícka 2005d] Pícka M. Metamodelling Utilisation for Biometric Data Applications. Agrázní perspektivy. 2005. ISBN 80-213-1372-2.

[Pícka 2005e] Pícka M. BORM z pohledu transformací pojmů. Sbormík prací z mezinárodní vědecké konference Agrární perspektivy XIV. 2005. ISBN 80-213-1372-2

[Pícka 2006a] Pícka M. Modelling of Information Systems using gradual transformation. Sborník prací z mezinárodní konference - Agrární perspektivy XV. 2006. ISBN 80-213-1531-8

[Pícka 2006b] Pícka M. LambdaTalk. Objekty 2006, Sborník příspěvků XI. ročníku konference. 2006 ISBN 80-213-1568-7

[Pícka 2006c] Pícka M. Gradual Modeling of Information System - Model of Method Expressed as Transitions Between Concepts. In: Proceedingd of the 8th International Conference on Enterprise Information Systems - ICEIS 2006. ISBN 972-8865-43-0

[Pícka 2007a] Pícka M. Relationships of methods in Model of Metodology Concept Transormations. Sborník prací z mezinárodní konference - Agrární perspektivy XVI. 2007. ISBN 978-80-213-1675-1

[Pícka 2007b] Pícka M. Formalizace transformace procesního modelu. Sborník prací z mezinárodní konference - Agrární perspektivy XVI. 2007. ISBN 978-80-213-1675-1

[Pícka 2007c] Pícka M. Aspektové programování v jazyku Python. In: Tvorba softwaru 2007. ISBN 978-80-248-1427-8

[Pícka 2008b] Pícka M., Papík M.: Transformace procesů BORMu do Petriho sítě. In: Objekty 2008 - zborník príspevkov 13. ročníka konferencie. Žilinská univerzita v Žilině v EDIS. ISBN 978-80-870-927-3

[Pícka 2008c] Pícka M.: Implementing of method of gradual transformation into CASE tool. Sborník prací z mezinárodní vědecké konference Agrární perspektivy XVII. ISBN 978-80-213-1813-7

[Pícka 2009a] Pícka M.: Vytvoření matice dohledatelnosti požadavků pomocí metody postupných transformací. Sborník prací z mezinárodní vědecké konference Agrární perspektivy XVIII. ISBN 978-80-213-1965-3

[Pícka 2009b] Pícka M., Pergl R.: CASE Tools: Is something still missing?. In Objekty 2009 - sborník příspěvků čtrnáctého ročníku konference. ISBN 978-80-7435-0090-2.

10.2 Ostatní literatura

[Agile 2001] Agile Manifesto. [online] http://agilemanifesto.org

[Agrawal 2003] Agrawal A. GReAT: A Metamodel Based Model Transformation Language, Automated Software Engineering, 18th IEEE International Conference, 2003

[Ambler 1998] Ambler S.,W.: Process Patterns : Building Large-Scale Systems Using Object Technology, Cambridge University Press 1998, ISBN 0-521-64568-9

[Ambler 1999] Ambler S.,W.: More Process Patterns : Delivering Large-Scale Systems Using Object Technology, Cambridge University Press 1999, ISBN 0-521-65262-6

[Ambler 2005] Ambler, S. W.: The Object Primer – Agile Model-driven Development with UML 2.0, Cambridge University Press 2005, ISBN 978-0-521-54918-6

[Ambler et al. 2005] Ambler S. W., Nalbone J., Vizdos M. J.: The Enterprise Unified Process, Prentice Hall 2005, ISBN 0131914510

[Ambler 2006] Ambler S.: Agile Unified Process, dostupný na http://www.ambysoft.com/unifiedprocess/agileUP.html

[Antoniol et. al 2002]	Antoniol, G., Cancory G., Casazza G., De Lucia A. Merlo E.:Recovering Traceability Links Between Code and Documentation. IEEE Transaction on Software Engineering vol 28 no. 10. pp 970-983. 2002
[Baldwin 2001]	Baldwin, D.: Software Development Life Cycles: Outline for Developing a Traceability Matrix,
[Beck 2002a]	Beck K.: Extrémní programování (český překlad) Grada 2002, ISBN 80-247-0300-9
[Beck 2002b]	Beck K.: Test Driven Development: By Example, Addison-Wesley 2002, ISBN 0321146530
[Beck 2004]	Andres C., Beck K.: Extreme programming Explained -- Embrace Change, Addison-Wesley 2004, ISBN 0321278658
[Beck at al. 2001]	Beck K. at al.: Manifest agilního vývoje, http://agilealliance.org
[Bělohlávek et al. 2001]	Bělohlávek F., Košťan P., Šuleř O.: Management, Rubico 2001, ISBN 80-85839-45-8
[Bezivin 2005]	Bézivin J. On the Unification Power of Models. Software and System Modeling. (SoSym) 4(2):171--188. Springer 2006. ISSN 1619-1366
[Blaha et al. 1995]	Blaha M., Premerlani M.: Object-Oriented Modeling and Design for Database Applications, Prentice Hall 1995, ISBN 0-13-123829-9
[Boehm 1988]	Boehm, Barry: A Spiral Model of Software Development and Enhancement, Computer, IEEE, 21(5):61-72, May 1988
[Boese 2008]	Boese W.: Reports Fault U.S. Anti-Missile Approach. Arms Control Today, November 2008
[Booch 1994]	Booch G.: Object-Oriented Analysis and Design with Applications, Second Edition, Benjamin Cummings 1994, ISBN 0-8053-5340-2
[Brooks 1988]	Brooks F. P.: No Silver Bullet> Essence and Accidents of Softwatr Engineering. Computer 20(4) 1988, p.10 - 19
[Buchalcevová 2002]	Buchalcevová A.: Agilní metodiky, sborník konference OBJEKTY 2002, ČZU Praha 2002, ISBN 80-213-0947-4
[Buchalcevová 2005a]	Buchalcevová A.: Metodiky vývoje a údržby informačních systémů, Grada 2005, ISBN 80-247-1075-7
[Buchalcevová 2005b]	Buchalcevová, A.: Metodika feature-driven development neopouští modelování a procesy, a přesto přináší výhody agilního vývoje, In: Tvorba softwaru 2005, ČSSI Ostrava 2005
[Bulthuis et al. 1998]	Henderson-Sellers, B.; Bulthuis, A.: Object-Oriented Metamethods, Springer-Verlag New York, 1998, ISBN 0-387-98257-4
[Carda 2007]	Carda A. Systémová integrace: SOS In:Sborník príspevkov z konferencie Systémová integracia 2007, SSSI 2007
[CMS 2005]	CMS: Selecting a Development Approach. http://www.cms.hhs.gov/SystemLifecycleFramework/
[Coburn 2004]	Coburn A.: Crystal Clear, Addison-Wesley 2004, ISBN 0201699478

[Cortex]	http://www.it-cortex.com/Stat_Failure_Rate.htm [2010-4-5]
[CraftCASE 2009]	Analyticý a modelovací nástroj Craft.CASE – manuál k verzi 2.1. 2009
[DeGrace 1990]	DeGrace P., Stahl L.H.: Wicked Prolems, Righteous Solutions: Catalogue of Modern Software Engineering paradigms, Yourdon Press 1990, ISBN 013590126X
[Dijkstra 1972]	Dijkstra E.: The Humble Programmer, Communications of ACM 15(10) 1972 p. 859-866, 1972. online dostupné na http:// www.cs.utexas.edu/~EWD/ewd03xx/EWD340.PDF
[Drabick 1999]	Drabick, R.: On Track Requirements. Software Testing and Quality Engineering. 1(3) 1999
[Drbal 1996]	Drbal P.: Proudy v objektově orientovaných metodikách, sborník konference Tvorba softwaru 1996, Ostrava
[DSDM 2003]	DSDM Consortium: DSDM: Business Focused Development, Pearson Education 2003, ISBN 0321112245
[Ewusi 2003]	Ewusi-Mensah K., Software Development Failure, MIT Press, 2003, ISBN 0-262-05072-2
[Felsing 2002]	Felsing J. M., Palmer S. R.: A Practical Guide to Feature-Driven Development, Prentice Hall 2002, ISBN 0130676152
[Fowler et al. 1999]	Fowler M. et al.: Refactoring: Improving the Design of Existing Code, Addison-Wesley 1999, ISBN 0201485672
[Fujaba]	Fujaba Tool Suite. [online] http://www.fujaba.de/
[Gamma et al. 1995]	Gamma, E.; Helm, R.; Johnson, E.R; Vlissides, J.: Design Patterns - Elements of Reusable Object Oriented Software, Addison-Wesley, New York, USA, 1995. ISBN 0201633612
[Gerber et al. 2002]	Gerber, A., Lawley, M., Raymond, K., Steel, J., Wood, A.: Transformation: Themissing link of MDA. In: Graph Transformation. Volume 2505 of Lecture Notes. In Computer Science., Springer-Verlag (2002) 90–105 Proc. 1st Int'l Conf. Graph Transformation 2002
[Gibbs 1994]	Gibbs W. W.: Software's Chronic Crisis, Scientific American 9, 1994
[Gogolla et al 2002]	Gogolla M., Richters M.. Expressing UML class diagrams properties with OCL. Lecture Notes in Computer Science, Vol.2263, 2002, 85-114
[Gogolla et al 2003]	Godolla, M., Lindow, A.: Transforming Data Model with UML. In Knowledge Transformation for Semantic Web. IOS Press. Amsterdam. 2003. p. 18-33
[Gogolla et al 2007]	Gogolla M., Büttner F., Richters M.: USE: A UML-based specification environment for validating UML and OCL. Sci. Comput. Program. 69(1-3): 27-34 (2007)
[Gotel 1994]	Gotel, O., Finkelstein, I.: An Analysis of Requirements Traceability Problem. In Proceedind of the First International Conference on Requirements Engineering. 94-101. 1994

[Grady 1999]	Grady R. B.: An Economic Release Decision Model: Insight into Software Project Management. In Proceeding of the Aplications of Software Measurement Conference, 227-239. 1999
[Hecksel 2007]	Hecksel D.: Methodology Evaluation and Selection, dostupné na http://www.davidhecksel.com/projectcontext/whitepaper.html
[Henderson-Sellers1994]	Henderson-Sellers B., Edwards J.M.: MOSES - A Second Generation Object-Oriented Methodology, pp 68-73, Object Magazine 4(3) 1994
[Hibbert et al. 2007]	[Hibbert et al. 2007] Hibberd M.,Lawley M., and Raymond K. Forensic debugging of model transformations. In Model Driven Engineering Languages and Systems, 2007
[Highsmith 1999]	Highsmith J.: Adaptive Software Development: A Collaborative Approach to Managing Complex Systems, Dorset House Publishing Company, Inc. 1999, ISBN 0932633404
[Holland 1996]	Holland J. H., Hidden Order: How Adaptation Builds Complexity, Addison-Wesley Publishing Co., 1996, ISBN 0201442302
[IEEE 610.12]	IEEE Std 610.12-1990 IEEE Standard Glossary of Software Engineering Terminology. 1990
[IEEE 830]	IEEE Std. 830-1998 IEEE Recommended Practice fo Software Requirements Specifications. 1999
[Charette 2005]	Charette R. N.: Why Software Fails. IEEE Spectrum 9 2005, online na http://www.spectrum.ieee.org/sep05/1685
[Chen 1975]	Chen P.: The Entity-Relationship Model: Toward a Unified View of Data, Proceedings of the International Conference on Very Large Data Bases, September 22-24, 1975, Framingham, Massachusetts, USA. ACM
[ISO 9000]	ČSN EN ISO 9000: Systémy managementu jakosti – základy, zásady a slovník, Český normalizační institut, Praha, 2002
[ISO 90003]	ISO/IEC 90003: Software Engineering – Guidelines for the application of ISO 9001:2000 to computer software, First edition 2004
[ISO 9126-1]	ISO/IEC IS 9126-1 Information Technology - Software product Quality – Part 1: Quality model
[ISO 9126-2]	ISO/IEC TR 9126-2 Information Technology - Software product Quality – Part 2: External metrics
[ISO 9126-3]	ISO/IEC TR 9126-3 Information Technology - Software product Quality – Part 3: Internal metrics
[ISO 9126-4]	ISO/IEC TR 9126-4 Information Technology - Software product Quality – Part 4: Quality in use metrics
[IT-UK 2003]	The statictic of IT project in the UK 2002-2003, online na http://www.computerweekly.com/pmsurveyresults/surveyresults.pdf
[Jacobson et al. 1992]	Jacobson, I., Christerson, M., Jonsson, P., Övergaard, G., Object-Oriented Software Engineering - A Use Case Driven Approach, 1992, Addison Wesley ISBN 0-201-54435-0

[Jacobson et al. 1999]	Jacobson I., Booch G., Rumbaugh J.: The Unified Software Developement Process, Addison Wesley, 1999, ISBN 0-2015-7169-2
[Jeffries 2000]	Jeffries R.: Extreme Programming Installed (The XP Series), Addison-Wesley 2000, ISBN 0201708426
[Jarke 1998]	Jarke, M.: Requirements tracing, Communications of ACM 41(12). p. 92-94
[Kadlec 2004]	Kadlec V.: Agilní programování. Metodiky efektivního vývoje softwaru, Computer Press 2004, ISBN 80-251-0342-0
[Keller 2004]	Keller E.: Technology Paradise Lost - Why Companies Will Spend Less to Get More from Information Technology, Manning Publications 2004, ISBN: 1932394133
[Kelly 1997]	Kelly S.: Towards a Comprehensive MetaCASE and CAME Environment: Conceptual, Architectural, Functional and Usability Advances in MetaEdit+. Ph.D. Thesis, University of Jyvaskyla, 1997
[Kelly et al. 2008]	Kelly S., Tolvanen J-P. : Domain-Specific modeling: Enabling full code generation. Wiley. 2008
[Keyes 2002]	Keyes J.: Software Engineering Handbook, AUERBACH, 2002, ISBN 0849314798
[Koch ea al. 2006]	Koch N., Zhang G. and Escalona M.J. Model Transformations from Requirements to Web System Design. In Proc. of 6th International Conference on Web Engineering (ICWE 2006), ACM, 281–288, Palo Alto, USA, July 2006.
[Kruchten 2000]	Kruchten P.: The Rational Unified Process – An Introduction, Addison-Wesley, 2000, ISBN 0201707101
[Kuske 2000]	Kuske, S.: Transformation Units - A Structuring Principle for Graph Transformation Systems. PhD thesis, University of Bremen 2000
[Lawrence 1997]	Lawrence, B.: Requirements Happens ..., American Programmer 10(4) 1997
[Leffingwell 1997]	Leffingwell, D.: Calculating the Returns on Investment from More Effective Requirements Management, American Programmer 10(14) 1997
[Leffingwell 2000]	Leffingwell, D., Widrig D.: Managing Software Requirements> An Unified Approach. Addison-Wesley 2000
[Liskov 1994]	Liskov B, Wing J.: A behavioral notion of subtyping, ACM Transactions on Programming Languages and Systems (TOPLAS), Volume 16, Issue 6 (November 1994), pp. 1811 - 1841
[Martin 1990]	Martin J.: RAD, Rapid Application Development, MacMillan Publishing Co., New York, 1990.
[Martin et al. 1995]	Martin J., Odell J.: Object-Oriented Methods - A Foundation, Prentice Hall 1995, ISBN 0-13-63856-2
[Mellor. 1994]	Mellor P.: CAD: Computer Aided Disaster. High Integrity Systems, Vol. 1, No. 2. (1994), pp. 101-156

[Mellor at al. 2004]	Mellor, S. a Scott, K. a Uhl, A. a Weise, D. MDA Distilled: Principles of Model-Driven Architecture. Addison Wesley.2004. ISBN 0-201-78891-8
[Merunka et al. 2003]	Carda A., Merunka V., Polák J.: Umění systémového návrhu, Grada Publishing 2003, ISBN 80-247-0424-2
[Merunka 2006]	Merunka, V.: Knovledge modelling using CraftCASE tool.Agricultural Economy. 52, 2006(4), ISSN 0139-570X
[Merunka et al 2010]	Merunka V, Merunkova I., Brozek J.: Organization Modeling and Simulation using BORM Approach. In EOMAS 2010
[Mens et al. 2005]	Mens T., Czarnecki K., Van Gorp P.: 04101 Discussion -- A Taxonomy of Model Transformations. In: Language Engineering for Model-Driven Software Development, Dagstuhl Seminar Proceedings 04101. 2000. ISSN 1862-4405
[Meyer 1997]	Meyer B.: Object Oriented Software Construction, Prentice Hall, 1997, ISBN 013629155
[Mily et al. 1995]	Mili, H., Pachet, F., Benyahia, I., and Eddy, F. Metamodeling in OO. OOPSLA '95 Workshop summary. 1995
[Moore 1965]	Moore, G. E.: Cramming more components into integrated circuits. Online: ftp://download.intel.com/museum/Moores_Law/Articles-Press_Releases/Gordon_Moore_1965_Article.pdf
[Molhanec 2007]	Molhanec, M.: Modely, modelování a MDA. In Sborník z konference Objekty. 2007
[Nouza 2009]	Nouza O. Využití plánovacího systému STRIPS při automatizaci transformací v objektově orientovaných modelech. In Sborník konference Objekty. 2009
[OASIG 1996]	Clegg et al.: The performance of information technology and the role of human and organizational factors, OASIG Study, University of Sheffield, UK 1996
[OMG 2000]	Model Driven Architecture. Draft 3.2. [online] http://www.omg.org/cgi-bin/doc?omg/00-11-05
[OMG 2003]	MDA Guide Version 1.0.1 . [online] http://www.omg.org/cgi-bin/doc?omg/03-06-01
[OMG 2006]	MOF Version 2.0. [online] http://www.omg.org/spec/MOF/
[OMG 2007]	MOF 2.0/XMI Mapping, v2.1.1. [online] http://www.omg.org/spec/XMI/
[OMG 2008]	Meta Object Facility (MOF) 2.0 – Query/View/Transformation. Version 1.0. [online] http://www.omg.org/spec/QVT/
[OMG 2009]	UML Version 2.2. [online] http://www.omg.org/spec/UML/
[OMG 2010]	Object Constraint language. Version 2.2. [online] http://www.omg.org/spec/OCL/
[OpenUP]	OpenUP wiki. Online na http://epf.eclipse.org/wikis/openup/

[Page-Jones 2001]	Page-Jones M.: Základy objektově orientovaného návrhu v UML.Grada Publishing. Praha 2001. ISBN 80-247-0210-X
[Pigoski 1997]	Pigoski, T.M.. Practical Software Maintenance: Best Practices for Managing Your Software Investment . New York: John Wiley and Sons, Inc, 1997
[Pinheiro 2000]	Pinheiro F.: Formal and Informal Aspects of Requirements Tracing, III Workshop de Engenharia de Requisitos, WER‘2000, Rio de \janeiro, Brasil 2000
[Pergl 2008]	Pergl, R.: Metody řízení softwarových projektů využívající moderní paradigmata. Disertační práce PEF ČZU v Praze. 2008
[Poppendieck 2003]	Poppendieck M., Poppendieck T.: Lean Software Development: An Agile Toolkit for Software Development Managers, Addison-Wesley 2003, ISBN 0321150783
[Rossi at al. 1992]	Rossi M., Gustafsson M., Smolander K., Johansson L.-A. and Lyytinen K.: Metamodeling editor as a front end tool for a CASE shell. In: Advanced Information Systems Engineering. Springer. 1992. ISBN 978-3-540-55481-3
[Royce 1970]	Royce W.: Managing the Development of Large Software Systems, IEEE Proceedings 1970
[Richta 2006]	Richta, K.: Quo vadis, SI? aneb Lesk a bída softwarového inženýrství. konference Europen 2006
[Rumbaugh et al. 1991]	Rumbaugh J., Blaha M., Premerlani W., Eddy F., Lorensen W.: Object-Oriented Modelling and Design, Prentice Hall 1991, ISBN 0-13-630054-5
[Rumbaugh et al. 1998]	Rumbaugh, I. Jacobson, G. Booch: The Unified Modeling Language Reference Manual,Addison-Wesley, 1998, ISBN 0-2013-0998-X
[Scott et al. 2004]	Scott K. et al.: MDA Distilled, Addison-Wesley 2004, ISBN 0201788918
[Schmuller 2001]	Schmuller J.: Myslíme v jazyku UML. Grada Publishing. Praha 2001. ISBN 80-247-0029-8
[Spector 1986]	Spector A., Gifford D.: A computer science perspective of bridge design. Communications of ACM 29(4) p. 267-283, 1986
[Smith 1993]	Smith T. J. READS: A requirements engineerin g tool. In Stephen Fickas and Anthony Finkelstein, editors, Proceedings, IEEE International Symposium on Requirements Engineering, pages 94-97, California, January 1993.
[Sommerswille 1997]	Sommerswille I, Sawyer, P.:Requirements Engineering. A Good Practice Guide. Chichester, England. John Wiley&Sons 1997
[SSADM]	SSADM Version 4.2 Structural Standards. http://www.ogcio.gov.hk/eng/prodev/es3.htm
[Standish 1995]	Standish Group International: The Chaos Report, dostupné na http://net.educause.edu/ir/library/pdf/NCP08083B.pdf

[Surynek 2009] Surynek, P.: Constraint Programming in Planning. MFF UK Praha, 2009

[Vaníček 2004] Vaníček, J.: Měření a hodnocení jakosti informačních systémů. ČZU, PEF 2004, druhé přepracované vydání vysokoškolských skript, 2004.

[Wegner 1997] Wegner, P.: Why Interaction is More Powerful Than Algorithms, Communications of the ACM 40(5), p. 80-91, 1997

[Weiss 1991] Weiss, E.H., How To Write Usable User Documentation. Phoenix, Arizona: The Oryx Press, 1991

[Wilkie 1993] Wilkie G.: Object-Oriented Software Engineering, Addison Wesley 1993, ISBN 0-201-6276-1

[Wiegers 2003] Wiegers K. E.: Software Requirements, Second Edition, Microsoft Press 2003, ISBN 978-0735618794

[Wood 1995] Wood, J., Silver, D.: Joint Application Development, John Willey & Sons Inc., 1995, ISBN 0-47104-299-4

[Yardley 2002] Yardley D.: Succesfull IT Project Delivery: Learning the Lessons of Project Failure. Addison-Wesley 2002, ISBN 0-201-75606-4

[Yourdon 1994] Yourdon E.: Object-Oriented System Design - An Integrated Approach, Prentice Hall 1994, ISBN 0-13-176892-1

Příloha A. BORM

V této kapitole bude popsán soubor objektově orientovaných technik a metod, které je vhodné použít během konstrukce softwarové aplikace. Postupy zde uvedené tvoří metodiku BORM, která je metodikou pro analýzu a návrh informačních systémů. Informace o metodice BORM byl publikován v [Merunka et al. 2000, Carda et al. 2003] . Tato kapitola inspirovaná kapitolou o BORMu publikovanou ve skriptech[*] Merunka, Pergl, Pícka: Objektově orientovaný přístup v projektování informačních systémů [Merunka et al. 2005].

A.1 Metoda BORM

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

BORM lze charakterizovat pomocí následujících tří vlastností:

- 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.
- BORM pro každou jednotlivou fází životního cyklu využívá v diagramech jen omezenou sadu pojmů. Předpokládá se totiž, že během projektování dochází k postupným přeměnám pojmů na jiné. Například ve fází 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

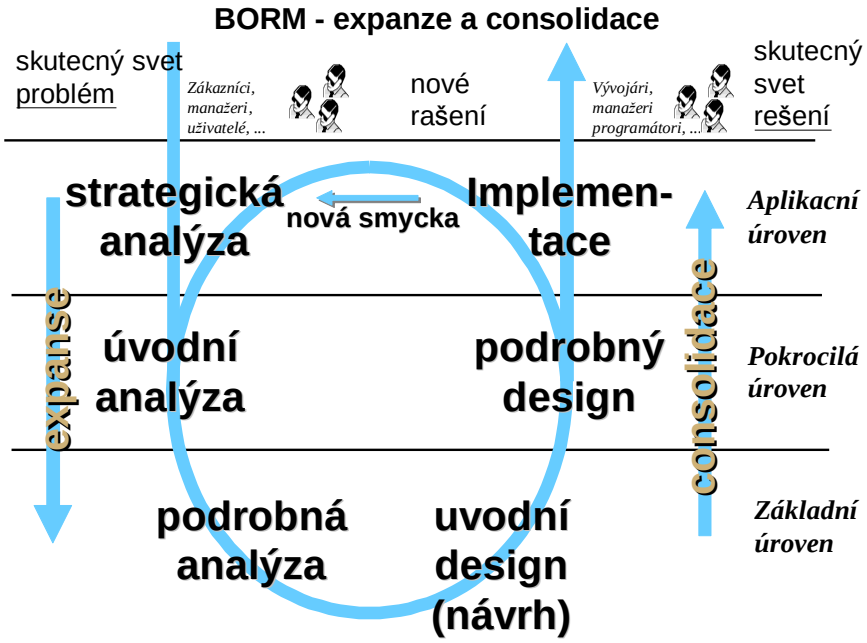
[*] Jejichž je autor této disertace spoluautorem.

s nimi již nepracuje. Nejde jen o postupné zvyšování úrovně detailu ve vytvářeném modelu, ale skutečně o řadu transformací modelu v průběhu životního cyklu.

- 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, které 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 iterační 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 fází dochází k přeměně prvků již existujícího modelu do takové podoby, která je podřízena cílovému implementačními prostředí. 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.



Obr. 49 – 6 fází životního cyklu vývoje systému v BORMu

A.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.

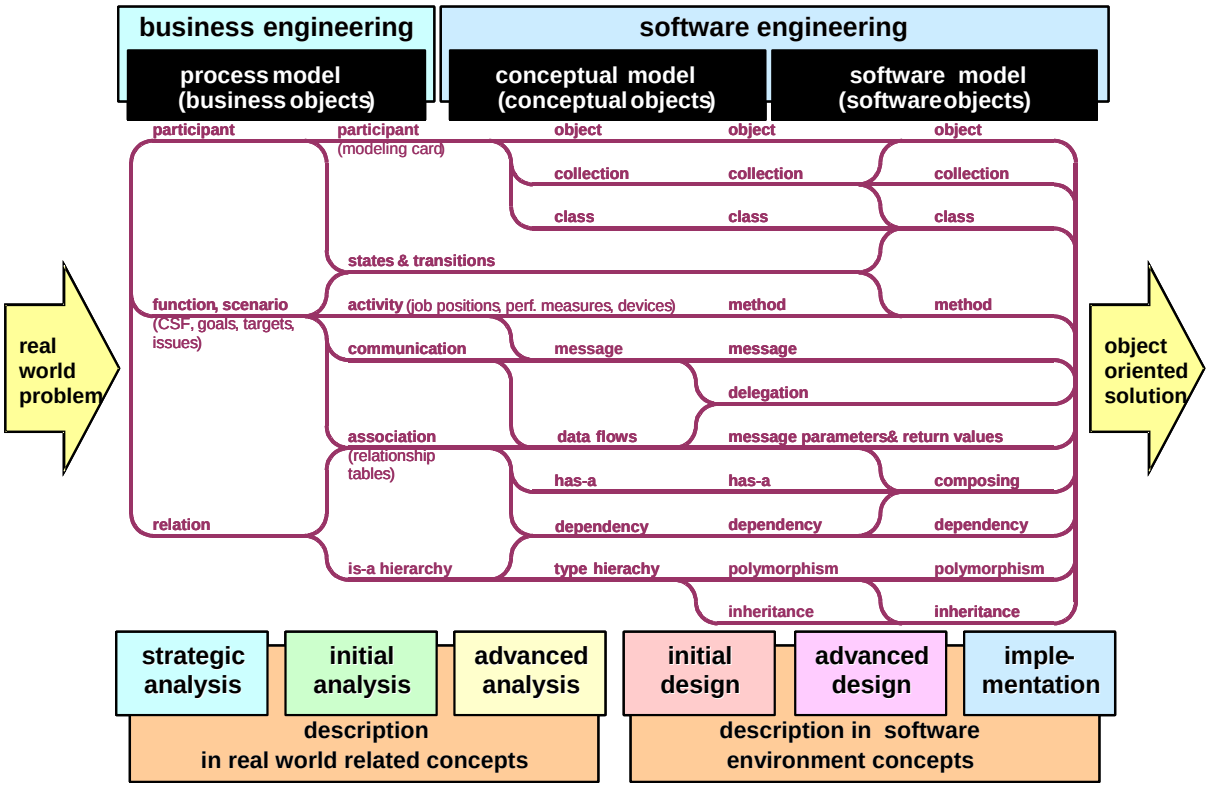
- Aplikační úroveň představuje okruh znalostí a dovedností, se kterými přichází jedna nebo druhá skupina do běžného pracovního styku. Pojmy z aplikační oblasti jsou proto pro příslušníka skupiny známé a srozumitelné. Bohužel jsou však nejvíce vzdálené pojmům aplikační úrovně skupiny druhé a informační systémy proto nelze stavět pouze na této úrovni.
- 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.

- 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:

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



Obr. 50 – Transformace v objektovém modelu

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:

- 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.

- 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.

- 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.

A.3 Fáze expanze a konsolidace

V jednodušším pohledu na 6 fází BORMu, aplikujeme-li iterativní model, můžeme rozlišit dvě hlavní etapy; expanze a konsolidace.

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 konsolidace. 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 konsolidaci je klíčovým faktorem úspěchu softwarových projektů. Samotný iterativní model totiž nezkušeného manažera svádí k neúměrnému počtu iterací s dlouhými expanzemi. Konsolidace se potom odbývá a výsledný produkt nemá potřebnou kvalitu.

A.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:

1 Jako metodu pro rozpoznání zadání pro informační systém a sestavení prvotního modelu tohoto řešení nebo

2 Jako metodu pro modelování, analýzu a reinženýring organizační struktury a business procesů firmy, přičemž následné budování informačního systému zde není podmínkou.

A.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 vnikla 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. [Rubin et al. 1992, Bellin 1997]

- Zaměstnanci (referenti) používají auta pro svoje služební účely
- NĚKTEŘÍ ZAMĚSTNANCI (REFERENTI) DOSTÁVAJÍ DLOUHODOBĚ PŘIDĚLENÁ AUTA PRO SLUŽEBNÍ ÚČELY
- Vedoucí zaměstnanců rozhodují o přidělení služebních aut zaměstnancům na dlouhodobou výpůjčku
- Autoprovoz zajišťuje nákup či pronájem aut a jejich technický servis
- Autoprovoz přiděluje auta zaměstnancům na dlouhodobou výpůjčku
- Zaměstnanci požadují přidělení referentského vozidla na konkrétní služební cestu
- Autoprovoz přiděluje referentská auta zaměstnancům
- Vedoucí zaměstnanců potvrzuje žádost autoprovozu o výpůjčku auta v půjčovně pro služební cestu zaměstnance v případě, že není k dispozici referentské vozidlo pro krátkodobou výpůjčku, nebo rozhodne o dočasném přerušení dlouhodobé výpůjčky

příklad seznamu funkcí

Jedná se o iterativní techniku začínající řízeným interview se zadavateli a pracující 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]. Jednotlivé kroky OBA analýzy, jak jsou použity v BORMu, jsou následující:

1. krok - rozpoznání procesů (plánování scénářů). V tomto kroku se na základě provedeného interview sestaví seznam požadovaných funkcí systému a seznam scénářů systému. Jedná se vesměs o textové popisy, přičemž v nejjednodušší variantě se u každého scénáře rozlišuje původ procesu, vlastní popis procesu, participující objekty a popis výsledku procesu.

2. krok - definování objektů pomocí modelových karet. V tomto kroku se pro každý rozpoznaný 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. Předpokládá se, že pro každý rozpoznaný spolupracující objekt je také vytvářena jeho modelová karta.

3. krok - klasifikace objektů. V tomto kroku dochází k přidání další informace k modelovým kartám jednotlivých objektů. Modelové karty jsou tříděny podle různých kriterií a na základě nalezených podobností se seznam modelových karet upřesňuje a doplňuje.

4. krok - sestavení tabulky vztahů mezi objekty. Tabulka vztahů v nejjednodušší podobě vyjadřuje jaký objekt má vztah s jiným objektem.

5. krok - modelování životních cyklů objektů. V tomto kroku se pro každý rozpoznaný objekt s pomocí informací v tabulce scénářů, modelových kartách a tabulkách vztahů sestaví životní cyklus objektu jako sled jeho stavů a přechodů mezi těmito stavy v podobě procesního diagramu. Tento poslední krok lze v případě první iterace provést ihned po kroku 1. a 2. a teprve poté provést kroky 3. a 4.

Scenario 1	Derived from functions: 2, 3, 4, 5, 8	Phase: as is
Initiation: Rozhodnutí vedoucího	Action: Přidělení služebního auta zaměstnanci na dlouhodobou výpůjčku	Result: Zaměstnanec má přiděleno služební auto na dlouhodobou výpůjčku
Auto (default)		
Referent (performs)		
Vedoucí (approves)		

Vedoucí autoprovozu (cooperates)		
Scenario 2	Derived from functions: 2, 3, 4, 6	Phase: as is
Initiation: Rozhodnutí vedoucího	Action: Zrušení dlouhodobé výpůjčky služebního auta	Result: Zaměstnanec vrací dlouhodobě vypůjčené auto do autoparku, autopark po převzetí auta zajistí technickou prohlídku
Auto (default)		
Referent (is informed)		
Vedoucí (is responsible)		
Vedoucí autoprovozu (cooperates)		

příklad 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 of: **Referent** in diagram 'výpůjčka auta':	Auto	Vedoucí	Vedoucí autoprovozu
start: požaduje auto		✓	
čeká na rozhodnutí vedoucího: dostává potvrzení žádosti		✓	✓
čeká na rozhodnutí vedoucího: dostává zamítnutí žádosti		✓	

čeká na přidělení auta: je informován o přidělení auta			✓
čeká na přidělení auta: je informován o ukončení rezervace		✓	
dostal přidělené auto: vyzvedává auto z autoparku	✓		
má auto ve službě: vrací auto do autoparku	✓		

Příklad modelové karty (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.

A.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:

Pojem	**Symbol**	**Popis**
Objekt = Participant	Obdélník se jménem zobrazeným uvnitř v levém horním rohu.	Objekt (je označován jako "Participant") představuje účastníka modelovaného procesů.
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 čárou s participantem nebo jeho stavem. Ovály mohou být kresleny také dovnitř k nim příslušných objektů.	Aktivity 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	Podmínkou se vyjadřuje omezená platnost komunikace nebo aktivity.

	objektu.	

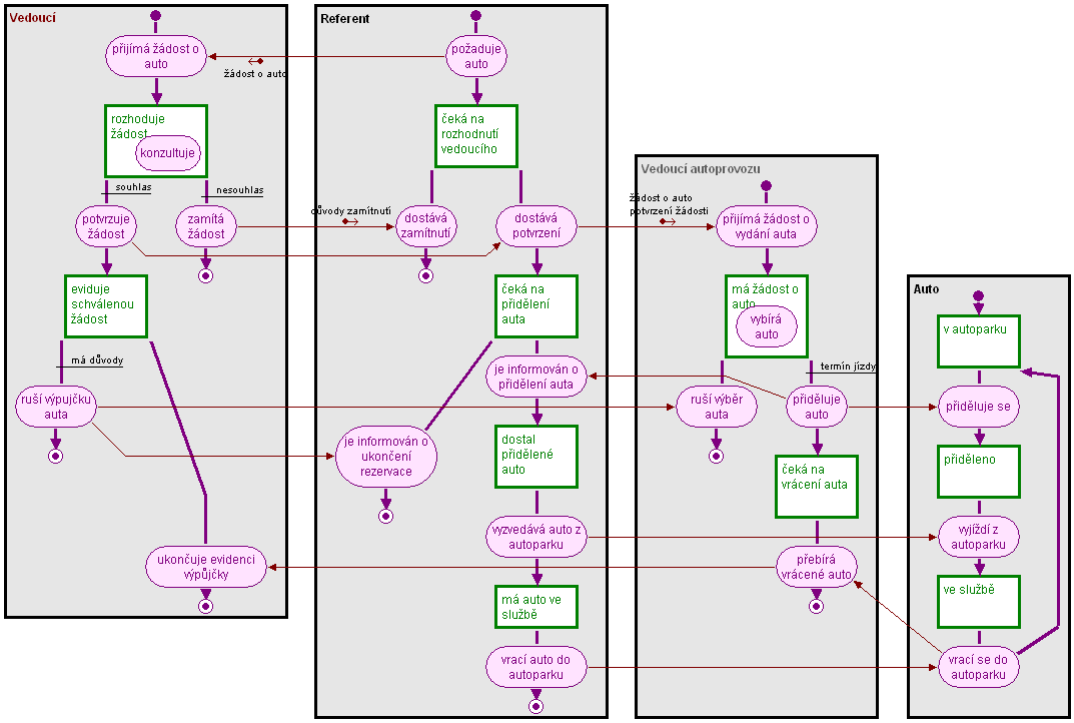
Tab. 12 - Pojmy diagramu ORD

ORD dovoluje modelovat jednotlivé procesy současně dvojím způsobem:

- 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.
- 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 (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[*], jsou velmi dobře využitelné v interview, ve kterých se diagram sestavuje nebo verifikuje.

[*] ORD je diagram, kde každý objekt (participant) je modelován jako Mealyho automat. Při simulaci se využívá synchronizace pomocí Petriho sítě.



Obr. 51 – Příklad diagramu popisujícího proces

A.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.
- 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.

- 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.

A.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.

A.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 použít k upřesnění modelu zadání, ale zároveň model tvořený konceptuálními objekty slouží pro zahájení návrhu, kdy již považujeme problém za rozpoznaný a začínáme jej konsolidovat do počítačově realizovatelné podoby.

A.5.1 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.

Pro tyto kroky jsou k dispozici rozhodovací tabulky obsahující sadu transformačních pravidel. Kromě pravidel, která říkají, jaké transformace pojmů a vazeb jsou a nejsou možné, se používají i návrhové vzory.

object relationship (from A to B) / **behavioral constraints**	B is dynamicaly accessible from A	HAS-A hierarchy		IS-A hierarchy		B is an instance of a class A	B is dependent on A
		normal	aggregation	poly--morphism	inheritance		
A needs B to perform anything	**Yes**	**Yes**	**Yes**	**No**	**No**	**No**	**Yes**
A needs to pass data to B	**Yes**	**Yes**	**Yes**	**No**	**No**	**No**	**Yes**
A needs to get data from B	**Yes**	**Yes**	**Yes**	**No**	**No**	**No**	**No**
B shares the same behaviour as A	**No**	**No**	**No**	**Yes**	**Yes**	**No**	**No**
B uses the methods of A	**No**	**No**	**No**	**No**	**Yes**	**No**	**No**
values of A have influence to values or behav. of B	**No**	**No**	**No**	**No**	**No**	**Yes**	**No**
behav. (methods) of A have influence to values or behav. of B	**No**	**No**	**No**	**No**	**Yes**	**No**	**No**
values or behav. of B have influence to values or behav. ofA	**No**	**No**	**Yes**	**No**	**No**	**No**	**No**

Rozhodovací tabulka pro přechod od business ke konceptuálnímu modelu

A.5.2 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:

1. UML nerozlišuje mezi polymorfismem, děděním, hierarchií typů a hierarchií „je jako“. V našem přístupu tyto vazby graficky rozlišujeme (Vysvětlení rozdílu je v kapitole 5.3.8).
2. 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.
3. 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.
4. 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.

A.6 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:

- Přeměna hierarchie typů na hierarchii dědění mezi třídami a případná transformace vícenásobné dědičnosti na jednoduchou.
- Doplnění tříd o atributy a metody, které umožní realizovat stavy a přechody (současné objektové jazyky se stavy a přechody přímo nepracují).

- Využití návrhových vzorů pro optimalizaci a pro kontrolu proveditelnosti modelu v daném programovacím jazyce.
- Pokud to cílové prostředí vyžaduje, tak nahradit vazby závislosti.
- Pokud to cílové prostředí vyžaduje, tak nahradit vazby delegování.
- Napojení modelu na struktury v existujících systémech.

A.7 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.

A.8 Evoluce hierarchií objektů

Nové typy se v objektových systémech realizují většinou pomocí tříd, přičemž ale progamátoři 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

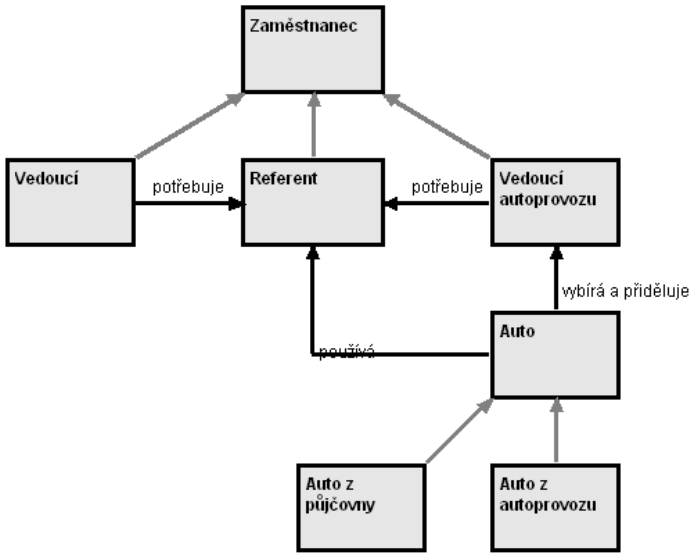
    a. Z pohledu polymorfismu – objekty na nižších úrovních hierarchie potom musejí být schopny dostávat stejné zprávy, jako objekty vyšších úrovní. Právě tato hierarchie je hierarchie typů. Její místo je ve fázi konceptuálního modelování.

    b. 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.

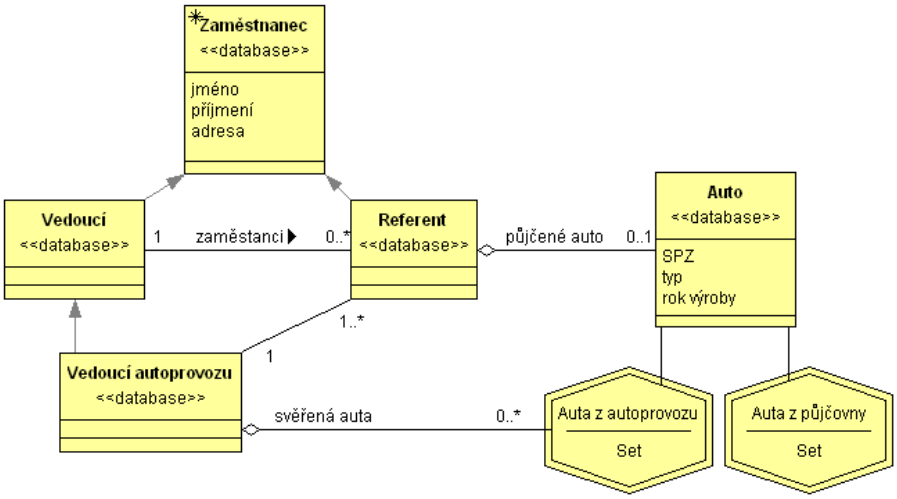
V BORMu se nejprve pracuje pouze s hierarchií „je jako“, z ní se později odvodí hierarchie typů a nakonec se na základě typů navrhne hierarchie dědičnosti. Tento přístup, kdy se s hierarchiemi objektů pracuje v různých fázích vývoje různě, zabraňuje nadměrnému a nesprávnému použití dědičnosti v průběhu 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. 52 – Příklad proměny hierarchií objektů – fáze business modelování

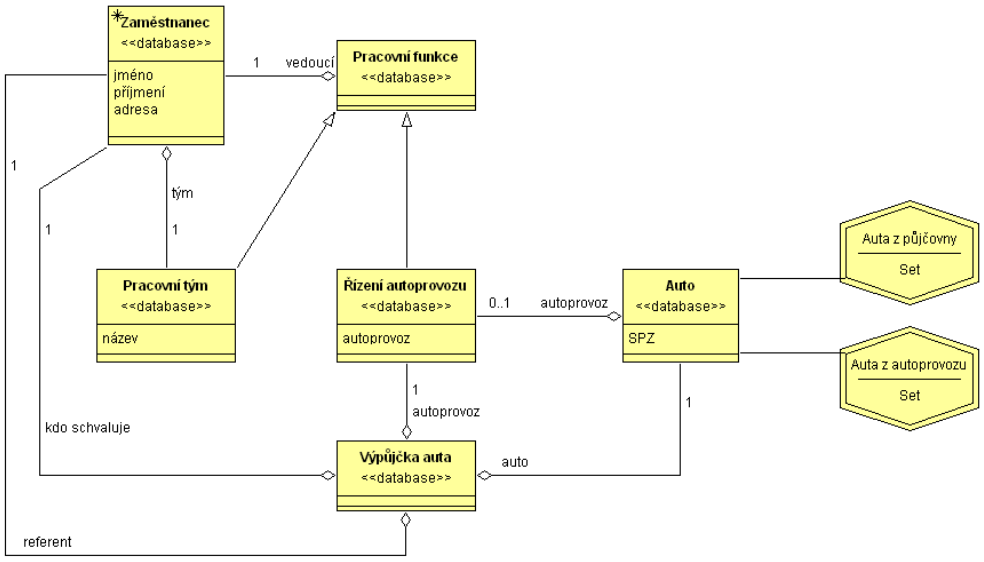
V následné fázi konceptuálního modelování došlo nejen ke zpřesnění modelu, ale také ke změně hierarchie. Kritériem už totiž není příslušnost do domény, ale chování objektů. Proto je teď vedoucí autoprovozu „podtypem“ vedoucího. (vedoucí autoprovozů se chovají stejně jako vedoucí a k tomu mají ještě další 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. 53 – Příklad proměny hierarchií objektů – 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 zaměstanců (v diagramu třída označená hvězdičkou). Nebylo proto možné z předchozí fáze rozpoznané podtypy implementovat jako různé třídy. Proto se musel použít návrhový vzor „Player-Role“, který různé typy a podtypy objektů dovoluje implementovat bez dědění jen pomocí skládání (v příkladu to je skládání k objektu pracovní funkce).



Obr. 54 – Příklad proměny hierarchií objektů – 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.

A.9 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

* BORM používá standard UML, ale rozšiřuje ho o nové symboly, které v kombinaci se stávajícími dovolují vytvářet další druhy diagramů.

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 závislost 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 sestrojit samostatný zjednodušený „object-class“ diagram definující data a vazby daného stavu. Právě odlišnosti 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.

A.10 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, Larrson 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.

[*] UML zná jen diagram stavů a přechodů, který se týká jen jedné třídy objektů. Zde ale hovoříme o celém modelu se všemi objekty, jak na sebe vzájemně působí.

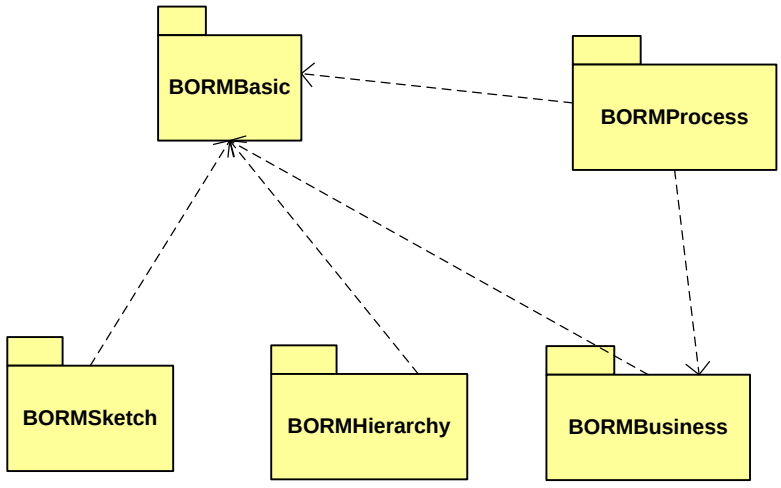
- **Podcenění možností metod**, které vede k jejich omezení na pouhou manipulaci se složkami objektů (metody pouze typu "ukaž" a "nastav"). Vzhledem k provázanosti datové a funkční stránky v OOP je velmi vhodné s některými metodami počítat přímo v návrhu datové struktury a tím ušetřit na objektových vazbách a datových atributech objektů.
- **Zjednodušení modelu výpočtu směrem k vN** 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 množina objektů v konceptuálním modelu a z toho vyplývající nevhodná implementace typů pomocí tříd.

Příloha B. Metamodel metody BORM

Tento metamodel byl vytvořen na základě [Merunka et. al 2003], [Pícka 2005b],[CraftCASE 2009] a dalších. Název jednotlivých prvků je převzat z názvosloví, které používá CASE nástroj implementující metodu BORM – Craft.CASE [CraftCASE 2009].

B.1 Metamodel BORM – struktura

Název prvku	Popis
BORMBasic	Balíček (package) představující obsahující strukturální prvky a vztahy BORMu.
BORMSketch	Balíček obsahující prvky týkající se úvodních fází BORMovského projektu – sketch diagramu.
BORMHierarchy	Balíček obsahující prvky týkající se tvorby hierarchií.
BORMBusiness	Balíček obsahující prvky týkající se business diagramů, neboli vztahů mezi funkcemi a scénáři.
BormProcess	Balíček obsahující prvky týkající se procesních diagramů BORMu (neboli ORD diagramů)

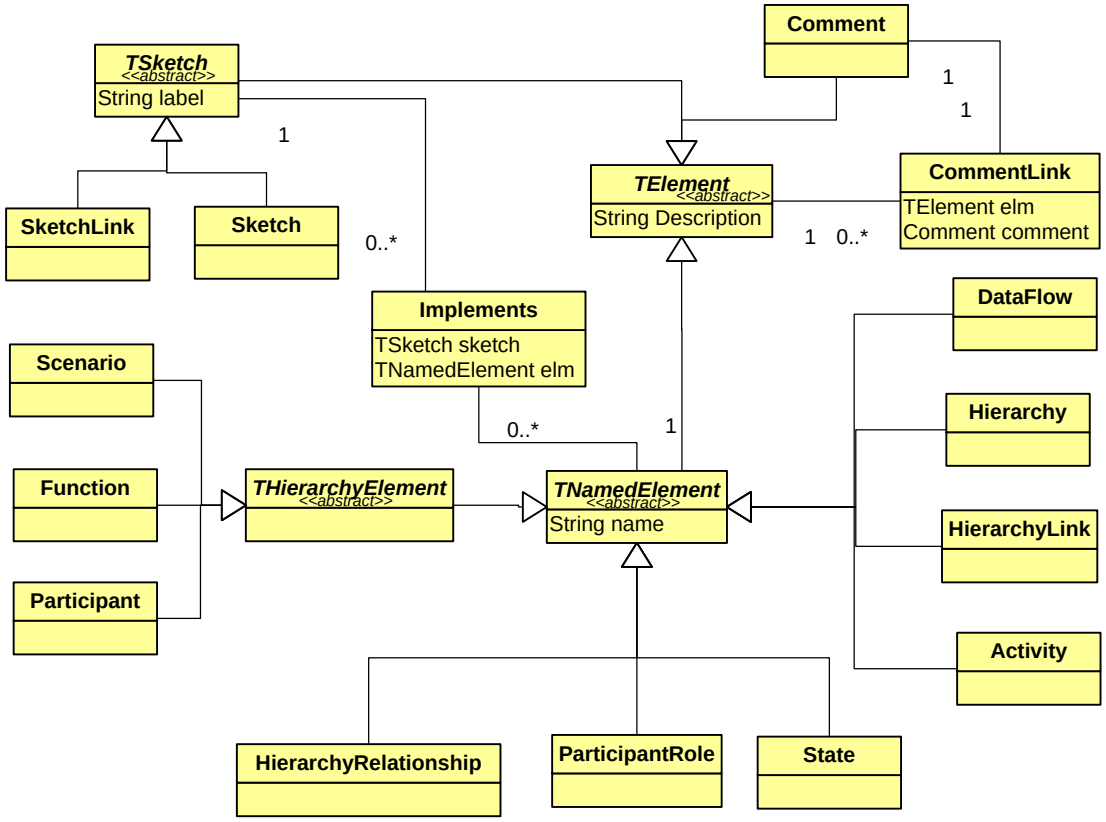


Obr. 55 – Základní struktura metamodelu BORMu

B.2 Metamodel BORM – základní prvky

Tato část metamodelu zachycuje základní prvky (typicky abstraktní) struktury metamodelu a další pomocné prvky (např. komentáře).

Název prvku	Popis
TElement	Základní prvek metamodelu (abstraktní), jsou z něj odvozeny všechny ostatní prvky.
TNamedElement	Pojmenovaný abstraktní prvek .
TSketch	Abstraktní prvek sketch diagramu..
THierarchyElement	Abstraktní prvek, z kterého jsou odvozeny prvky tvořící hierarchie.
Implements	Spojuje prvky sketch diagramu s jejich implementací.
Comment	Komentář k prvku modelu.
CommentLink	Spojuje prvek s komentářem.

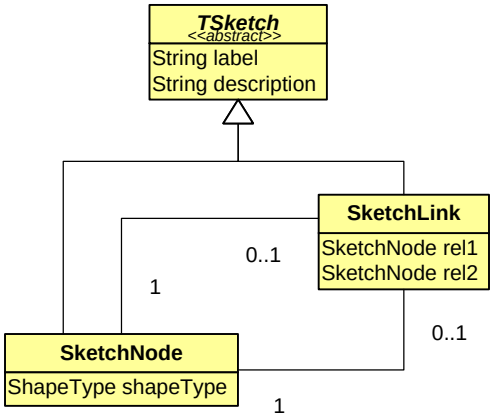


Obr. 56 – Metamodel BORM – základní prvky

B.3 Metamodel BORM – sketch

Tato část metamodelu obsahuje prvky týkající se úvodních částí projektu v BORMu – vytvoření sketch diagramu. Tento diagram zachycuje požadavky na systém, jeho základní strukturu a pod. získané během interview se zadavatelem projektu.

Název prvku	Popis
SketchNode	Balíček (package) představující obsahující strukturální prvky a vztahy BORMu.
SketchLink	Balíček obsahující prvky týkající se úvodních fází BORMovského projektu – sketch diagramu.

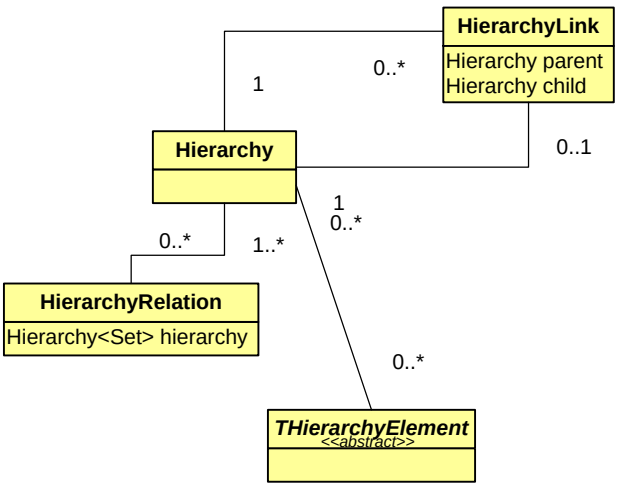


Obr. 57 – Metamodel BORM – sketch

B.4 Metamodel BORM – hierarchie

Tato část metamodelu se věnuje hierarchiím v BORMu (hierarchický diagram). Typickým použitím je např. hierarchická struktura firmy (hierarchie participantů).

Název prvku	Popis
Hierarchy	Prvek vytvářející stromové hierarchie hierarchie
HierarchyLink	Definuje vztahy rodič-potomek v hierarchiích
THierarchyElement	Abstraktní prvek, který se může vyskytovat v hierarchiích. Může to být Scenario, Function nebo Participant.
HierarchyRelationship	Definuje vztahy mezi hierarchiemi.

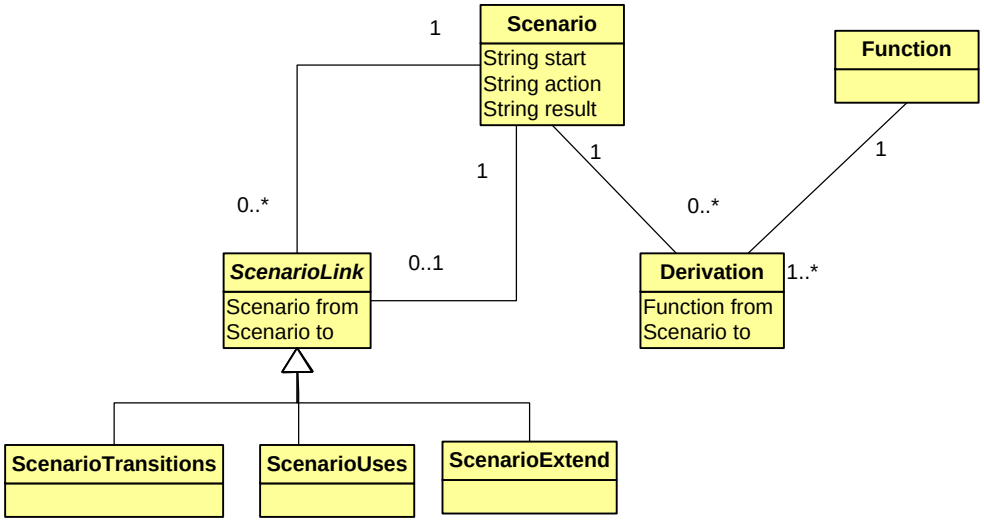


Obr. 58 – Metamodel BORM – hierarchie

B.5 Metamodel BORM – business model

Tato část metamodelu zachycuje prvky vyskytující se v business modelu. V něm jsou znázorněny vztahy mezi funkcemi (Function) a scénáři (Scenario), případně vztahy mezi scénáři.

Název prvku	Popis
Scenario	Scénář zachycuje proces v systému.
ScenarioLink	Abstraktní prvek zachycující vztahy mezi scénáři. Tento vztah může být ScenarioTransition, ScenarioUses nebo ScenarioExtend.
ScenarioTransitions	Vztah zachycující to, že jeden scénář navazuje na jiný.
ScenarioUses	Vztah zachycující to, že scénář používá jiný scénář.
ScenarioExtend	Vztah zachycující to, že scénář rozšiřuje jiný scénář.
Function	Prvek zachycující základní funkce systému.
Derivation	Prvek zachycující vznik scénáře na základě funkce (funkcí).



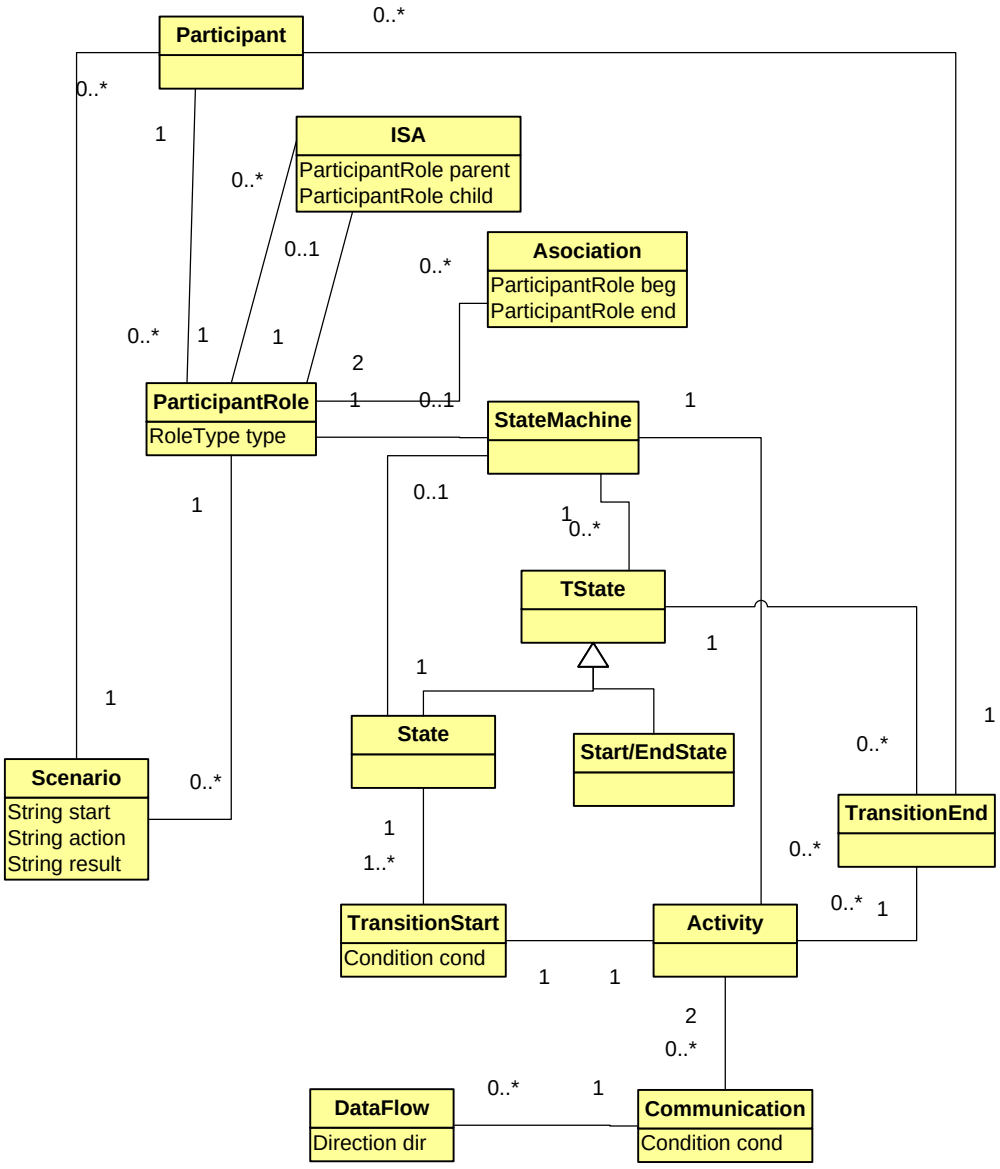
Obr. 59 – Metamodel BORM – business diagram

B.6 Metamodel BORM – procesní model

Tato část metamodelu obsahuje prvky vyskytující se v procesním diagramu BORMu (Object-relationship diagram – ORD).

Název prvku	Popis
Participant	Prvek reprezentující participant.
ParticipantRole	Prvek reprezentující roli participant. Tato role může obsahovat stavový stroj (StateMachine).
ISA	Role participantů můžeme strukturovat pomocí ISA hierarchie.
Asociation	Role participantů mohou mít nějaký vztah (asociaci) s ostatními rolemi.
StateMachine	Role participantu může obsahovat stavový stroj.
TState	Abstraktní prvek, reprezentující stav.
State	Prvek reprezentující stav role participantu. Stav může obsahovat také další stavový stroj (StateMachine).
Start/EndState	Speciální typ stavu – počáteční a koncový.

Název prvku	Popis
TransitionStart	Přechod mezi stavem a aktivitou.
TransitionEnd	Přechod mezi aktivitou a stavem.
Activity	Prvek reprezentující aktivitu, tj. činnost.
Communication	Prvek reprezentující komunikaci mezi dvěma aktivitami.
DataFlow	Během komunikace mezi aktivitami mohou vyměňovat mezi jednotlivými rolemi participantů data.



Obr. 60 – Metamodel BORM – procesní model

Příloha C. BORM z pohledu metody transformací prvků

V této příloze je vytvořen model přechodů mezi koncepty části[*] metody BORM. Byl vytvořrn za použití metamodelu metody BORM (viz Příloha B) a publikací [Merunka et. al 2003], [Pícka 2005b], [CraftCASE 2009] a dalších.

Celkový model přechodů by byl velmi složitý a nepřehledný a tak je rozdělen na jednotlivé, vzájemně na sebe navazující části. Tyto části odpovídají jednomu typu diagramů používaných BORMem. Jména konceptů odpovídají jménům tříd v metamodelu. Každé sekci popisuje část modelu a sekce se skládá z těchto částí:

- model přechodů mezi koncepty
- nově použité koncepty – (ostatní jsou převzaty z metamodelu z přílohy B.)
- nově použité funkce – obsahuje popis funkcí a metod použitých v algoritmu transformace
- jednotlivé předpisy transformací – nejsou zde uvedeny všechny, pouze ty něčím zajímavé, nebo důležité.

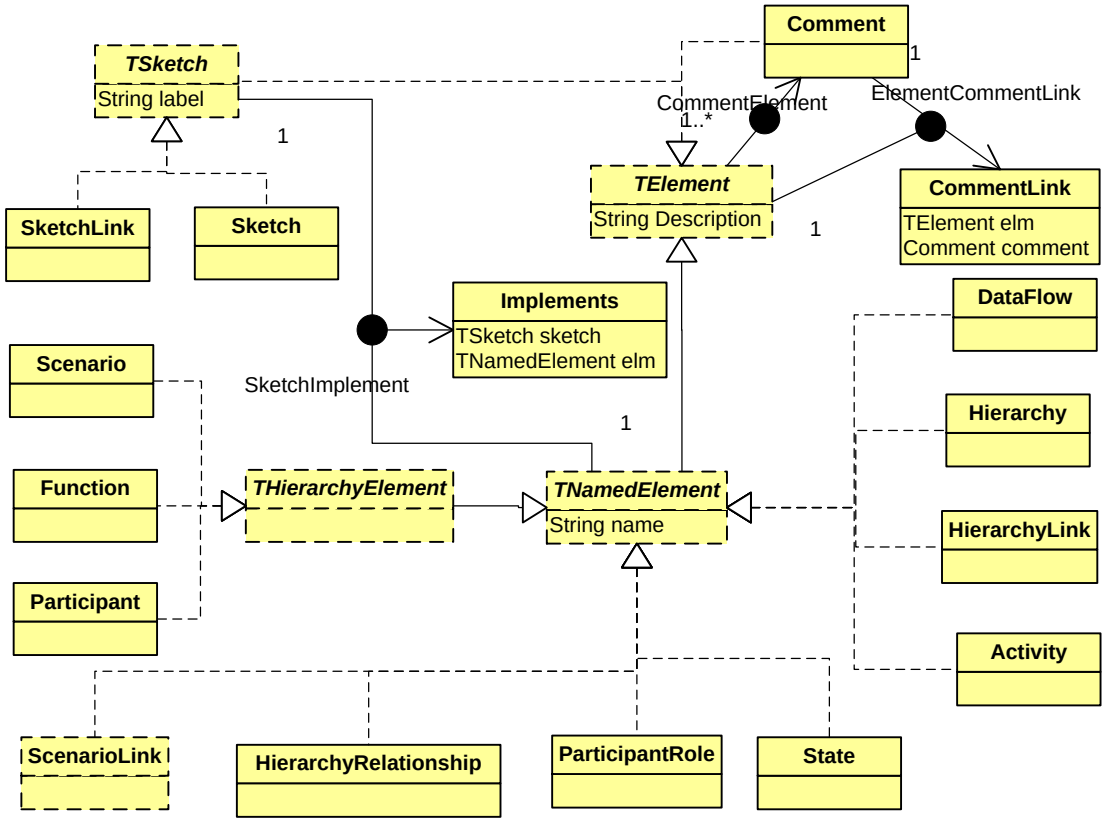
Poznámka:

Generalizované[†] koncepty jsou pro jasné odlišení od přechodů mezi koncepty, znázorněny pomocí přerušované čáry a generalizace mezi nimi také.

[*] Je zde zpracována taková část metody BORM, která se používá v reálných projektech. Není zde zpracována část týkající se přechodu z procesního modelu ke konceptuálnímu (zjednodušený třídní diagram). Konceptuální model se (v BORMovských projektech) používá jen zřídka – typicky se končí u procesních diagramů.

[†] Generalizované koncepty se ve vlastním modelu IS nevyskytují - odpovídají abstraktním třídám.

C.1 Předpisy transformací – Základní struktura



Obr. 61 – Předpis transformací BORM – základní struktura

C.1.1 Nově použité funkce

Název funkce	Popis
vhodné(x)	Test toho, že daný prvek nebo kolekci prvků x lze v daném kontextu použít. Toto nelze provést automaticky a je zde na vývojáři, který provádí danou transformaci aby takový prvek vybral.
y.size()	Vrací velikost kolekce y.
y.přidej(x)	Přidávání prvku x do kolekce y.
y.přidejPředchůdce(x)	Zaznamení faktu, že prvek y vznikl na základě prvky x, neboli přidání
y.jeKonceptu(x)	Test, zda pevek x je konceptu y.

Název funkce	Popis
y.jePredchůdce(x)	test, zda pevek y je předchůdcem prvku x.
y.dejPrvky(x)	vrátí všechny prvky kolekce y, které jsou konceptu k.
Implements()	Vytvářím nový prvek typu Implenents.
Comment()	Vytvářím nový prvek typu Comment.
CommentLink()	Vytvářím nový prvek typu CommentLink.

C.1.2 Implementace Sketche

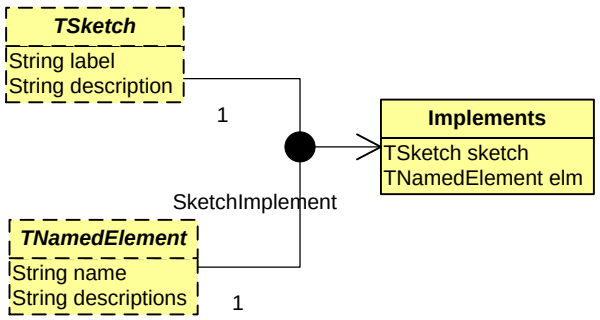
Název: Sketch_Implements

Označení: Sketch_Implements (ts , tne)

Popis:

Zaznamenávám si, že TSketch implementuje TNamedElement

Diagram:



Vstupy:

Jméno vstupu	Koncept	Násobnost
ts	*TSketch*	*1*
tne	*TNamedElement*	*1*

Výstup:

Implements

Podmínky: `vhodné(ts), vhodné(tne), ts ∈ tne.predchudce()`

Algoritmus transformace:

```
impl := Implements()      #nový objekt třídy Implements
model.přidej(impl)        #přidání do modelu
impl.sketch := ts
impl.elm := tne
impl.přidejPředchůdce(ts,tne)  #do impl si poznačím, že vznikl z ts
a ne
```

Poznámka:

TNamedElement může implementovat daný TSketch, jen když z něj vznikl.

C.1.3 Nový komentář

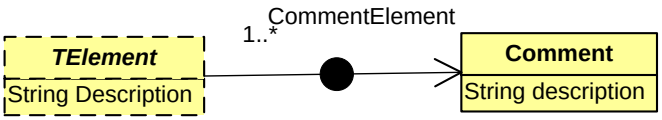
Název: Comment_Element

Označení: Comment_Element (E)

Popis:

Vytvářím komentář k prvkům.

Diagram:



Vstupy:

Jméno vstupu	Koncept	Násobnost
E	*SketchNode*	*1..N*

Výstup:

Comment

Podmínky: `vhodné(E)`

Algoritmus transformace:

```
comm := Comment()
model.přidej(comm)
comm.přidejPředchůdce(E)
```

Poznámka:

C.1.4 Linky ke komentáři

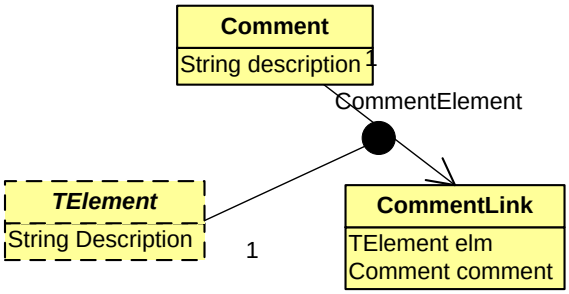
Název: Element_CommentLink

Označení: Element_CommentLink (elm, com)

Popis:

Spojuje komentář s komentovaným prvkem.

Diagram:



Vstupy:

Jméno vstupu	Koncept	Násobnost
elm	*TElement*	*1*
comm	*Comment*	*1*

Výstup:

CommentLink

Podmínky:

```
∃ elm  ∃ com, elm.jeKonceptu(TElement), com.jeKonceptu(Comment),
elm.jePredchudce(com): ¬∃ clink, clink.jeKonceptu(CommentLink):
clink.jePredchudce(elm) ∧ clink.jePredchudce(com)
```

Algoritmus transformace:

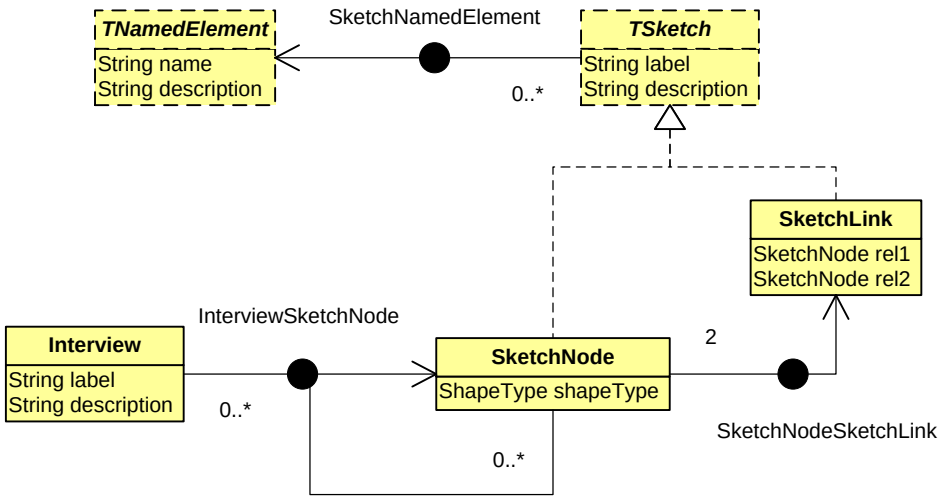
```
nalezeno := false
```

```
for clink in model.dejPrvky(CommentLink)
        if (clink.elm = elm} and (clink.comment = comm)
                    nalezeno = true;
                    break
if nalezeno = false
        commentLink := CommentLink()
        commentLink.elm := elm
        commentLink.comment := comm
        model.pridej(commentLink)
```

Poznámka:

Velmi zajímavý předpis transformace. Může probíhat automaticky. Proběhne v tom okamžiku, pokud v modelu proběhla transformace Element_CommentLink (tj. byl do modelu přidán nový komentář) a doplní potřebné CommentLinky.

C.2 Předpisy transformací – Sketch



Obr. 62 – Předpis transformací BORM - Sketch

C.2.1 Nově použité koncepty

Název konceptu	Popis
Interview	Interview zachycuje požadavky zákazníka vzniklé na základě rozhovoru s ním. Je to vstupní prvek

C.2.2 Nově použité funkce

Název funkce	Popis
TNamedElement()	Vytvářím nový prvek typu TNamedElement
SketchNode()	Vytvářím nový prvek typu SketchNode().
SketchLink().	Vytvářím nový prvek SketchLink().
x.jeGeneralizací(y)	Koncept x je generalizací konceptu y. Zde určuji typ konceptu výstupního prvku.
x.getInstance()	Vzniká nová instance konceptu x.

C.2.3 Nový obecný element

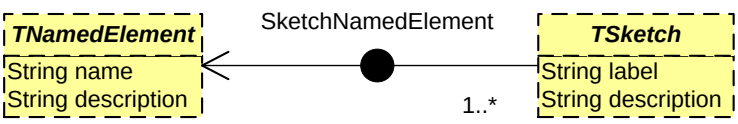
Název: Sketch_NamedElement

Označení: Sketch_NamedElement (S, Typ)

Popis:

Pomocí této transformace může vzniknout jakýkoliv prvek typu TNamedElement ze jednoho či více Sketchů.

Diagram:



Vstupy:

Jméno vstupu	Koncept	Násobnost
S	*TSketch*	*1 ... N*
Vystup_Typ	*Koncept*	*1*

Výstup:

TNamedElement

Podmínky: `vhodné(S), Vystup_Typ.jeGeneraliací(TNamedElement)`

Algoritmus transformace:

```
elm := Vystup_Typ.getInstance()
model.přidej(elm)
for pred in S
  sketchNode.přidejPředchůdce(pred)
```

Poznámka:

Na tomto předpisu je zajímavé, že vstup i výstup nejsou koncepty, ale generalizované koncepty. Takže tento předpis je vlastně šablonou. Ze základní struktury (C.1) lze zjistit, že za gener. konceptem TSketch se mohou dosadit 2 různé koncepty a za TNamedElement 10 konceptů. Tento předpis nám nahrazuje 20 předpisů transformací s konkrétními koncepty. Tato transformace je svou podstatou velice nebezpečná a dovoluje nám přidávat téměř každý prvek do modelu. Pro zachycení vztahu, že nějaký prvek z úvodního zadání (tj. TSketch) je implementován nějakým dalším prvkem (v tomto případě TNamedElement) je nutno použít transformaci SketchImplements.

C.2.4 Vytvoření nového SketchNodu

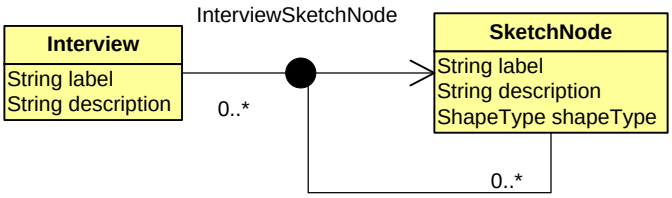
Název: Interview_SketchNode

Označení: Interview_SketchNode (I , S)

Popis:

Pomocí této transformace vzniká SketchNode, ten vzniká na základě Interview (0..N) a SketchNode (0..N).

Diagram:



Vstupy:

Jméno vstupu	Koncept	Násobnost
I	*Interview*	*0 ... N*
S	*SketchNode*	*0 ... N*

Výstup:

SketchNode

Podmínky: `vhodné(I), vhodné(S), I.size() + S.size() >= 1`

Algoritmus transformace:

```
sketchNode := SketchNode()
model.přidej(sketchNode)
for pred in I,S
  sketchNode.přidejPředchůdce(pred)
```

Poznámka:

SketchNode není vstupní prvek (to je Interview) a tak musí vzniknout alespoň z jednoho jiného prvku.

C.2.5 Vztahy mezi Sketchi

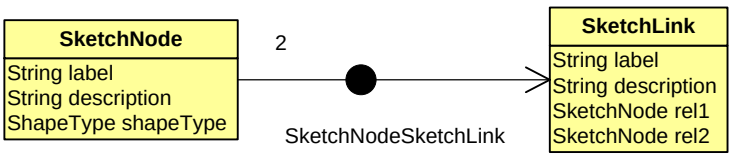
Název: SketchNode_SketchLink

Označení: SketchNode_SketchLink (node1 , node2)

Popis:

Pomocí této transformace vzniká SketchNode, ten vzniká na základě Interview (0..N) a SketchNode (0..N).

Diagram:



Vstupy:

Jméno vstupu	Koncept	Násobnost
node1	*SketchNode*	*1*
node2	*SketchNode*	*1*

Výstup:

SketchLink

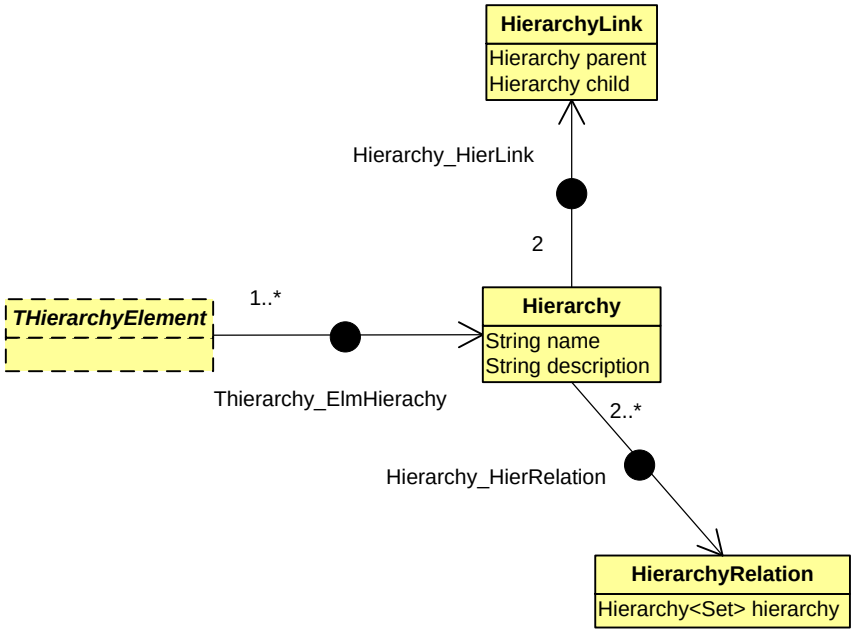
Podmínky: `vhodné(node1), vhodné(node2)`

Algoritmus transformace:

```
sketchLink := SketchLink()
model.přidej(sketchLink)
sketchLink.rel1 := node1
sketchLink.rel2 := node2
sketchLink.přidejPředchůdce(node1,node2)
```

Poznámka:

C.3 Předpisy transformací – Hierarchie



Obr. 63 – Předpis transformací BORM - hierarchie

C.3.1 Nově použité funkce

Název funkce	Popis
Hierarchy()	Vytvářím nový prvek typu Hierarchy.
HierarchyLink()	Vytvářím nový prvek typu HierarchyLink. Pomocí něho se zachycuje stromová struktura prvků typu Hierarchy.
HierarchyRelation()	Vytvářím nový prvek typu HierarchyRelation .

C.3.2 Vznik nového prvku tvořícího hierarchie

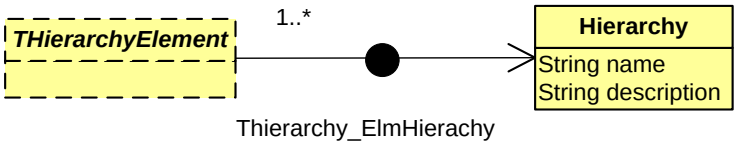
Název: THierarchy_ElmHierarchy

Označení: THierarchy_ElmHierarchy (HElm)

Popis:

Pomocí této transformace vzniká nový prvek tvořící hierarchie..

Diagram:



Vstupy:

Jméno vstupu	Koncept	Násobnost
HElm	*THierarchyElement*	*1 ... N*

Výstup:

Hierarchy

Podmínky: `vhodné(S)`

Algoritmus transformace:

```
elm := THierarchyElement()
model.přidej(elm)
for pred in HElm
  elm.přidejPředchůdce(pred)
```

Poznámka:

Další možnost vytvoření Hierarchy je pomocí transformace Sketch_NamedElemant.

C.3.3 Vytvoření hierarchie mezi prvky

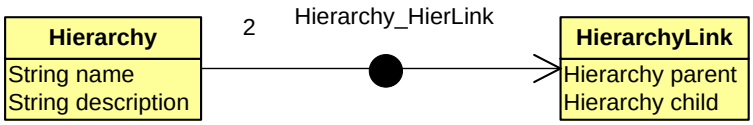
Název: Hier_HierarchyLink

Označení: Hier_HierarchyLink (pred, nasl)

Popis:

Tato transformace vytváří stromovou strukturu mezi prvky Hierarchy.

Diagram:



Vstupy:

Jméno vstupu	Koncept	Násobnost
pred	*Hierarchy*	*1*
násl	*Hierarchy*	*1*

Výstup:

HierarchyLink

Podmínky: `vhodné(pred), vhodne(nasl)`

Algoritmus transformace:

```
elm := HierarchyLink()
model.přidej(elm)
elm.parent := pred
elm.child := nasl
elm.přidejPředchůdce(pred, nasl)
```

Poznámka:

C.3.4 Vztah mezi hierarchiemi

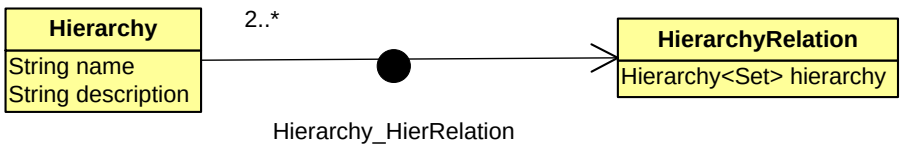
Název: Hierarchy_HierRelation

Označení: Hierarchy_HierRelation(H)

Popis:

Pomocí této transformace vznikají vztahy mezi hierarchiemi.

Diagram:



Vstupy:

Jméno vstupu	Koncept	Násobnost
H	Hierarchy	*2 ... N*

Výstup:

HierarchyRelation

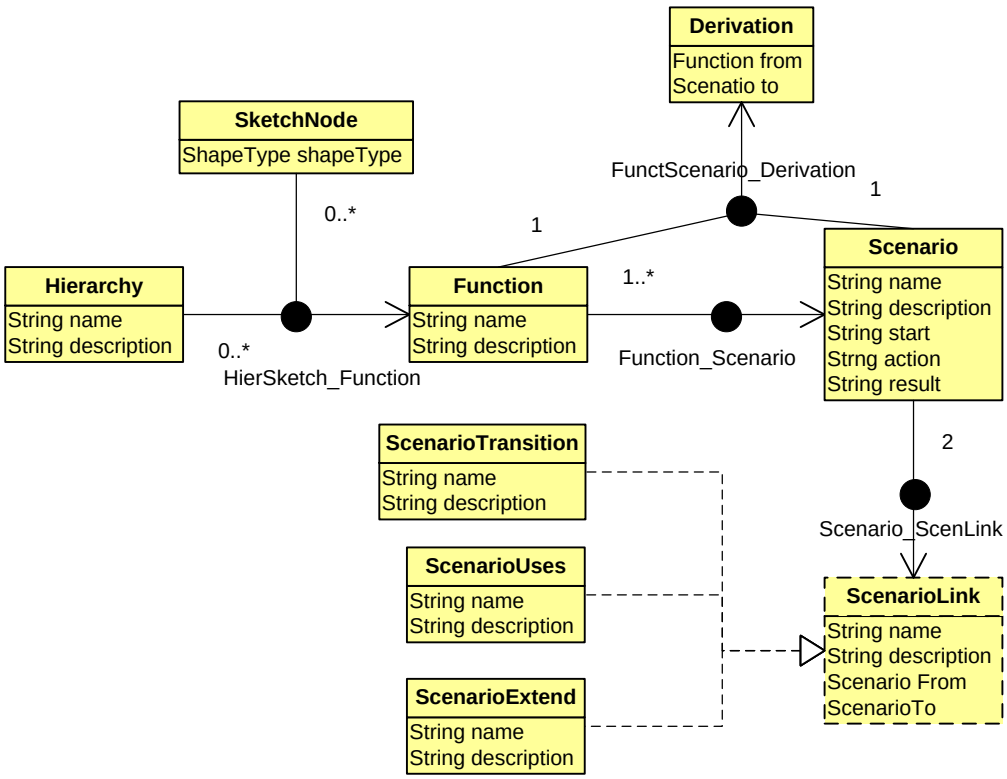
Podmínky: `vhodné(H)`

Algoritmus transformace:

```
elm := HierarchyRelation()
model.přidej(elm)
for pred in H
  elm.přidejPředchůdce(pred)
```

Poznámka:

C.4 Předpisy transformací – Business model



Obr. 64 – Předpis transformací BORM – business model

C.4.1 Nově použité funkce

Název funkce	Popis
Function()	Vytvářím nový prvek typu Function.
Derivation()	Vytvářím nový prvek typu Derivation – tj. odvození scénáře z funkci.
Scenario()	Vytváří nový scénář.

C.4.2 Vznik nové funkce

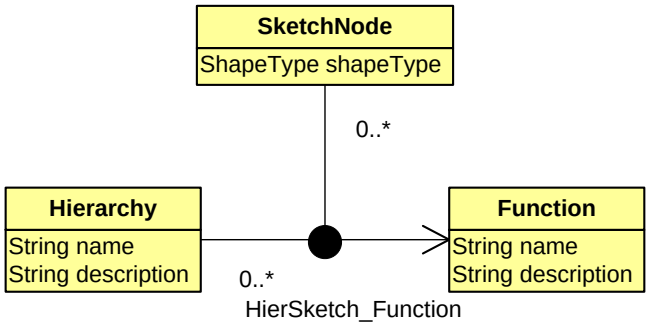
Název: HierSketch_Function

Označení: HierSketch_Function (H, S)

Popis:

Pomocí této transformace vzniká nová funkce (typu Function).

Diagram:



Vstupy:

Jméno vstupu	Koncept	Násobnost
H	*Hierarchy*	*0 ... N*
S	*SketchNode*	*0 ... N*

Výstup:

Function

Podmínky: `vhodné(S), vhodné(S), H.size() + S.size() >= 1`

Algoritmus transformace:

```
elm := HierSketch_Function()
model.přidej(elm)
for pred in H,S
  elm.přidejPředchůdce(pred)
```

Poznámka:

C.4.3 Vznik scénáře

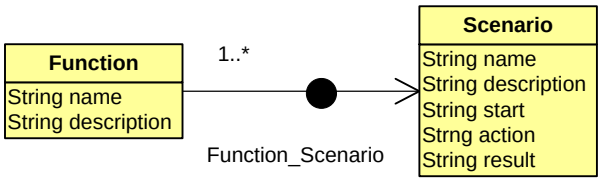
Název: Function_Scenario

Označení: Function_Scenario(F)

Popis:

Pomocí této transformace vzniká z 1 až N funkcí nový scénář

Diagram:



Vstupy:

Jméno vstupu	Koncept	Násobnost
F	*Function*	*1 ... N*

Výstup:

Scenario

Podmínky: `vhodné(F)`

Algoritmus transformace:

```
scenar := Scenar()
model.přidej(scenar)
for fce in F
  scenar.přidejPředchůdce(fce)
```

Poznámka:

C.4.4 Vytvoření linku mezi funkcemi scénářem

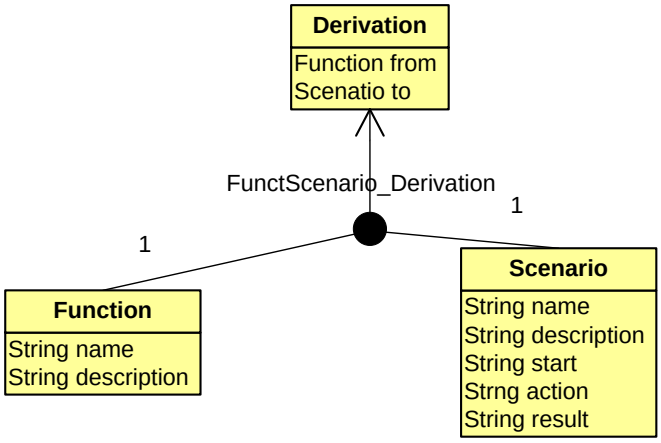
Název: FunctScenario_Derivation

Označení: FunctScenario_Derivation (from, to)

Popis:

Pomocí této transformace se znázorní fakt vytvoření scénáře z funkce.

Diagram:



Vstupy:

Jméno vstupu	Koncept	Násobnost
from	*Function*	*1*
to	*Scenario*	*1*

Výstup:

Derivation

Podmínky:

```
∃ func  ∃ scen, func.jeKonceptu(Function),
scen.jeKonceptu(Scenario), fce.jePredchudce(scen): ¬∃ der,
der.jeKonceptu(Derivarion): der.jePredchudce(fce) ∧
der.jePredchudce(scen)
```

Algoritmus transformace:

```
nalezeno := false
for der in model.dejPrvky(Derivace)
        if (der.from = from} and (der.to = to)
                    nalezeno = true;
                    break
if nalezeno = false
        derivation := Derivation()
        derivation.from := fce
        derivation.to := s
        model.pridej(derivation)
```

Poznámka:

Tato transformace probíhá automaticky (nepotřebuje žádný lisský vstup – neboli ve vstupní podmínce není `vhodné()`). Je podobného typu jako Element_CommentLink.

C.4.5 Vznik scénáře

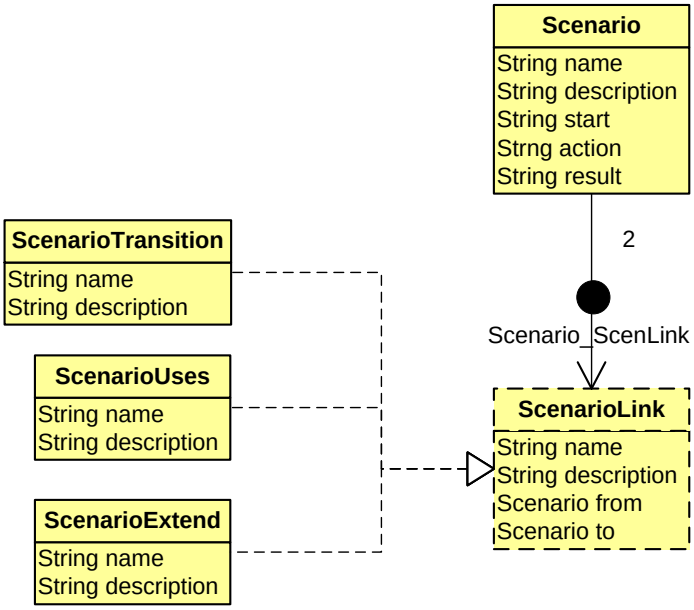
Název: Scenario_ScenLink

Označení: Scenario_ScenLink (from,to,outputConcept)

Popis:

Zachycuje vztahy mezi scénáři. Scénáře mohou na sebe navazovat (ScenarioTransition), jeden může rozšiřovat další (ScenarioExtend) nebo hop může používat (ScenarioUses).

Diagram:



Vstupy:

Jméno vstupu	Koncept	Násobnost
from	*Scenario*	*1*
to	*Scenario*	*1*
Vystup_Typ	*Concept*	*1*

Výstup:

ScenarioTransition, ScenarioUses, ScenarioExtend

Podmínky: `vhodné(from), vhodné(to)`

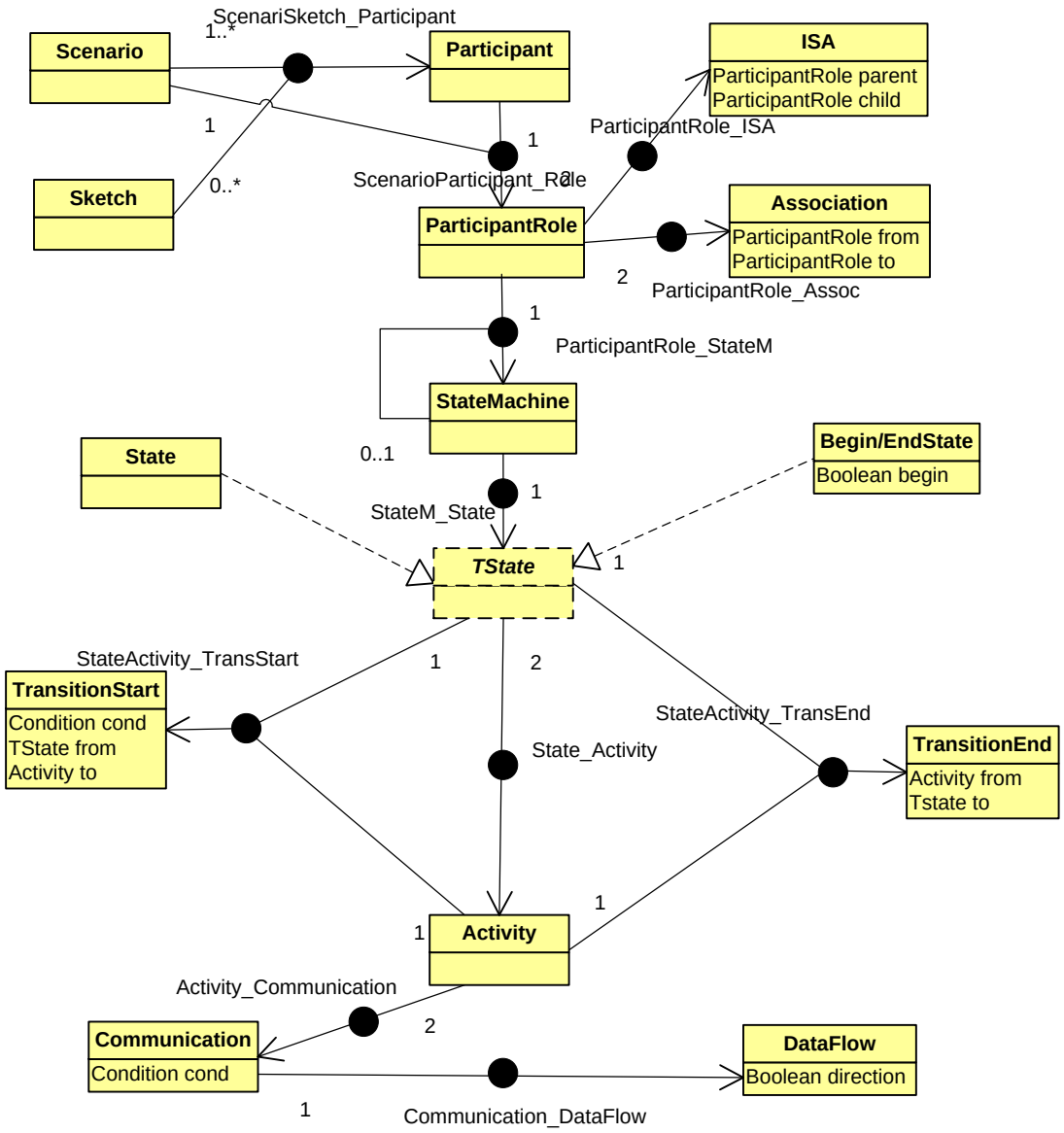
`Vystup_Typ.jeGeneraliací(ScenarioLink)`

Algoritmus transformace:

```
elm := Vystup_Typ.getInstance()
model.přidej(elm)
elm.from := from
elm.to := to
elm.přidejPředchůdce(from, to)
```

Poznámka:

C.5 Předpisy transformací – Procesní model



Obr. 65 – Předpis transformací BORM – procesní model

C.5.1 Nově použité funkce

Název funkce	Popis
Participant()	Vytváří nový participant.

ParticipantRole()	Vytváří novou roli participantu.
StateMachine()	Vytváří nový stavový stroj.
Activity()	Vytváří novou aktivitu.
TransitionStart()	Vytváří novou úvodní část přechodu mezi stavy a to konkrétně část přechodu mezi stavem a aktivitou.
TransitionEnd()	Vytváří novou závěrečnou část přechodu mezi stavy a to konkrétně část přechodu mezi aktivitou a stavem.
Communication()	Vytváří novou komunikaci mezi aktivitami.
DataFlow()	Vytváří nový datový tok.

C.5.2 Nový participant

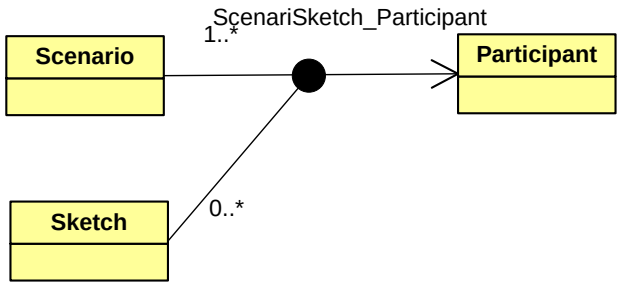
Název: ScenarioSketch_Participant

Označení: ScenarioSketch_Participant (Sc, S)

Popis:

Pomocí této transformace vzniká nový participant.

Diagram:



Vstupy:

Jméno vstupu	Koncept	Násobnost
Sc	*Scenario*	*1 ... N*
S	*Sketch*	*0 ... N*

Výstup:

Participant

Podmínky: `vhodné(Sc), vhodné(S)`

Algoritmus transformace:

```
elm := Scenario()
model.přidej(elm)
for pred in S, Sc
  elm.přidejPředchůdce(pred)
```

Poznámka:

C.5.3 Nová role participantu

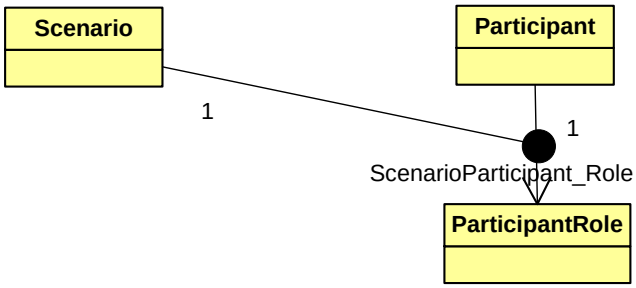
Název: ScenarioParticipant_Role

Označení: ScenarioParticipant_Role (Sc, P)

Popis:

Pomocí této transformace vzniká nova role participantu.

Diagram:



Vstupy:

Jméno vstupu	Koncept	Násobnost
Sc	*Scenario*	*1*
P	*Participant*	*1*

Výstup:

ParticipantRole

Podmínky: `vhodné(Sc), vhodné(P)`

Algoritmus transformace:

```
elm := ParticipantRole()
model.přidej(elm)
for pred in Sc,P
  elm.přidejPředchůdce(pred)
```

Poznámka:

C.5.4 Nový stavový stroj

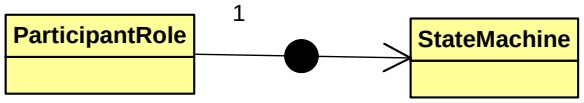
Název: ParticipantRole_StateM

Označení: ParticipantRole_StateM (role)

Popis:

Pomocí této transformace vzniká nový stavový stroj.

Diagram:



ParticipantRole_StateM

Vstupy:

Jméno vstupu	Koncept	Násobnost
role	*ParticipantRole*	*1*

Výstup:

StateMachine

Podmínky: `vhodné(role)`

Algoritmus transformace:

```
elm := StateMachine()
model.přidej(elm)
elm.přidejPředchůdce(role)
```

Poznámka:

C.5.5 Nový stav

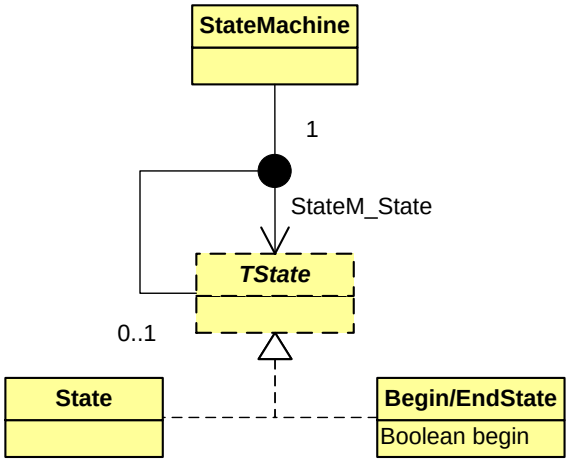
Název: StateM_State

Označení: StateM_State (state, stM)

Popis:

Pomocí této transformace vzniká nový stav.

Diagram:



Vstupy:

Jméno vstupu	Koncept	Násobnost
state	*TState*	*1 ... N*
stM	*StateMachine*	*1 ... N*
Vystup_Typ	*Concept*	*1*

Výstup:

TState

Podmínky: `vhodné(state), vhodné(stM), Vystup_Typ.jeGeneraliací(TState)`

Algoritmus transformace:

```
elm := TState()
model.přidej(elm)
for pred in state, stM
```

```
elm.přidejPředchůdce(pred)
```

Poznámka:

C.5.6 Nová aktivita

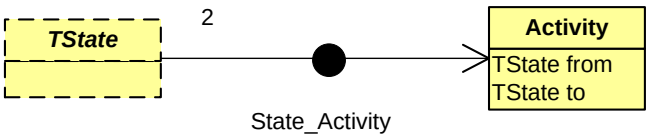
Název: State_Activity

Označení: State_Activity (from, to)

Popis:

Pomocí této transformace vzniká nová aktivita na základě dvou stavů.

Diagram:



Vstupy:

Jméno vstupu	Koncept	Násobnost
from	*TState*	*1*
to	*TState*	*1*

Výstup:

Activity

Podmínky: `vhodné(from), vhodné(to),(from.predchůdce ∩ to.predchůdce) = sM, size(sM) = 1,sM.jeKonceptu(StateMachine)`

Algoritmus transformace:

```
elm := Activity()
model.přidej(elm)
elm.ftom := from
elm.to := to
elm. přidejPředchůdce(from)
elm. přidejPředchůdce(to)
```

Poznámka:

C.5.7 Nová startovní část přechodu

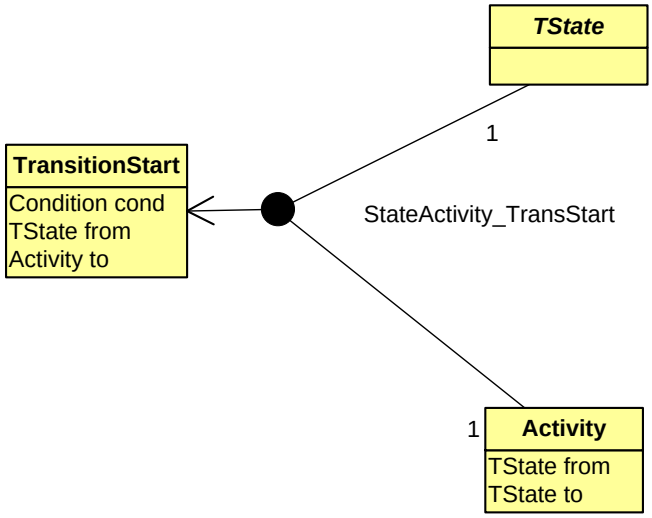
Název: StateActivity_TransStart

Označení: StateActivity_TransStart(state, activity)

Popis:

Pomocí této transformace vzniká nová úvodní část přechodu mezi dvěma stavy.

Diagram:



Vstupy:

Jméno vstupu	Koncept	Násobnost
state	*TState*	*1*
activity	*Activity*	*1*

Výstup:

TransitionStart

Podmínky:

```
∃ state  ∃ activity, state.jeKonceptu(TState),
activity.jeKonceptu(Activity), state.jePredchudce(activity): ¬∃
tranStart, tranStart.jeKonceptu(Derivarion):
tranStart.jePredchudce(state) ∧ tranStart.jePredchudce(activity)
```

Algoritmus transformace:

```
nalezeno := false
```

```
for ts in model.dejPrvky(TransitionStart)
        if (ts.from = state} and (ts.to = activity)
                    nalezeno = true;
                    break
if nalezeno = false
        tranStart := TransitionStart()
        tranStart.from := state
        tranStart.to := activity
        model.pridej(tranStart)
```

Poznámka:

Tato transformace je prováděná automaticky.

C.5.8 Nová startovní část přechodu

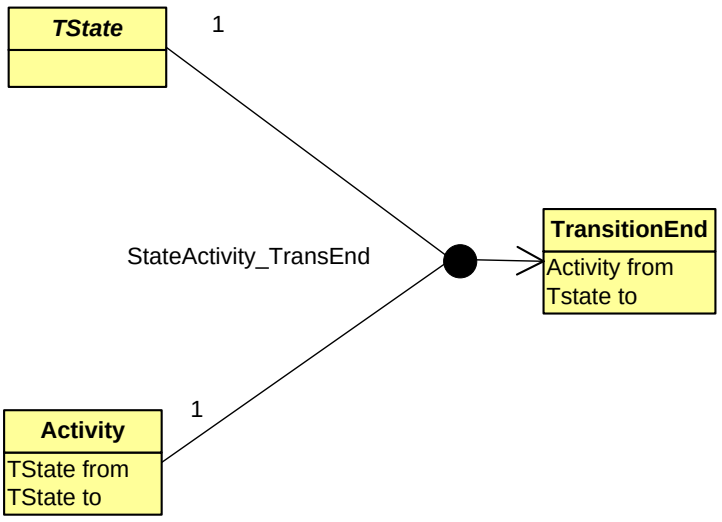
Název: StateActivity_TransEnd

Označení: StateActivity_TransEnd (activity, state)

Popis:

Pomocí této transformace vzniká nová koncová část přechodu mezi dvěma stavy.

Diagram:



Vstupy:

Jméno vstupu	Koncept	Násobnost

activity	*Activity*	*1*
state	*TState*	*1*

Výstup:

TransitionEnd

Podmínky:

```
∃ state  ∃ activity, state.jeKonceptu(TState),
activity.jeKonceptu(Activity), state.jePredchudce(activity): ¬∃
tranStart, tranStart.jeKonceptu(Derivarion):
tranStart.jePredchudce(state) ∧ tranStart.jePredchudce(activity)
```

Algoritmus transformace:

```
nalezeno := false
for te in model.dejPrvky(TransitionEnd)
        if (te.from = activity} and (te.to = state)
                    nalezeno = true;
                    break
if nalezeno = false
        tranEnd := TransitionEnd()
        tranEnd.from := activity
        tranEnd.to := ctate
        model.pridej(tranStart)
```

Poznámka:

Tato transformace je prováděná automaticky. Provádí se ve stejnou chvíli jako StateActivity_TransStart.

C.5.9 Nová komunikace

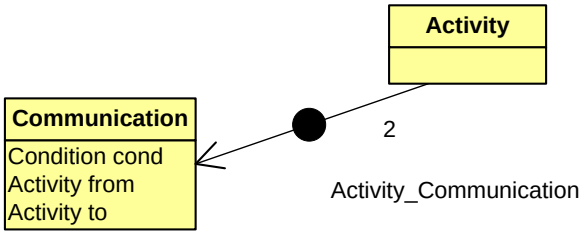
Název: Activity_Communication

Označení: Activity_Communication (from, to)

Popis:

Pomocí této transformace vzniká nová komunikace mezi dvěma aktivitami.

Diagram:



Vstupy:

Jméno vstupu	Koncept	Násobnost
from	*Activity*	*1*
to	*Activity*	*1*

Výstup:

Communication

Podmínky: `vhodné(from), vhodné(to)`

Algoritmus transformace:

```
elm := Communication()
model.přidej(elm)
elm.from := from
elm.to := to
elm.přidejPředchůdce(from,to)
```

Poznámka:

C.5.10 Nový datový tok

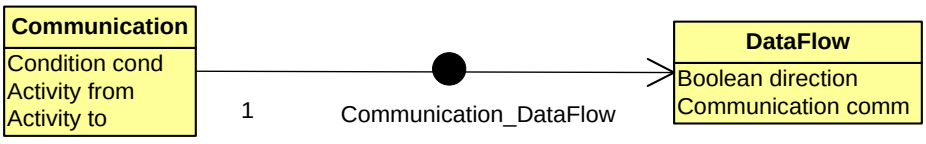
Název: Communication_DataFlow

Označení: Communication_DataFlow (comm)

Popis:

Pomocí této transformace vzniká nový datový tok.

Diagram:



Vstupy:

Jméno vstupu	Koncept	Násobnost
comm	*Communication*	*1*

Výstup:

DataFlow

Podmínky: `vhodné(comm)`

Algoritmus transformace:

```
elm := DataFlow()
model.přidej(elm)
elm.comm := elm
elm.přidejPředchůdce(comm)
```

Poznámka: