Extrémně Efektivní Modelování s použitím UML

(skripta k řadě produktů EFEM samostatně neprodejná, první vydání)

RNDr. Ilja Kraval, autor, mailto:objects@objects.cz

K uvedené problematice lze objednat školení in-house pro firmy, blíže viz adresa SERVER OBJEKTOVÝCH TECHNOLOGIÍ

http://www.objects.cz

Důležité upozornění

Technologie zvaná „Extrémně Efektivní Modelování“ (dále také EFEM) obsahuje specifické myšlenky a postupy chráněné autorským zákonem a jsou také chráněnými značkami.

K použití a nasazení těchto specifických postupů pouze doklad o zakoupení licencovaného produktu podle podmínek licencování anebo výslovný písemný souhlas autora.

Tato skripta jsou součástí produktu a jsou samostatně neprodejná. Jejich prodej nebo šíření jinými osobami bez svolení autora je považováno za trestný čin porušení autorských práv.

Obsah

1. Úvod co je EFEM? 8
1.1 Tvorba softwaru metodou „Tunel“ 8
1.2 Tvorba softwaru byrokratickým způsobem....9
1.3 Extrémně Efektivní Modelování jako technologie tvorby SW 9
1.4 Produkty řady EFEM....11
2. Úrovně abstrakce informačního systému 12
2.1 Zavedení pojmu abstraktní úrovně....12
2.1.1. Nejnižší úroveň abstrakce - kódování....12
2.1.2. Nejvyšší úroveň abstrakce - analytické modelování 13
2.1.3. Střední úroveň abstrakce modelování návrhu 14
2.2 Úplnost dokumentace v EFEM....15
2.2.1. Mapování z úrovně AM do D, fázování vývoje....16
2.2.1.1 Dokumenty typu AM+D+C a dokumenty typu přechodu mezi úrovněmi 16
2.2.1.2 Fázování prací na SW....16
2.2.2. Analytické modelování jako abstraktní programování....17
2.2.3. Problém podrobnosti a úplnosti analytického modelování 17
2.2.4. Maximální efektivita přechodu z analytického modelování do designu a do kódu 18
2.2.5. Modely podniku (BUSINESS MODELING) a jejich vztah k AM 20
2.2.6. Analýza jako zdroj informací....22
2.2.7. Podcenění analytického modelování a časované bomby v projektech....22
2.2.8. Analytické modelování a přechod na nové technologie....23
2.2.9. Role v projektu: analytik, designér a programátor 24
2.2.9.1 Analytik....24
2.2.9.2 Designér....24
2.2.9.3 Programátor 25
3. Princip maximální opětovné použitelnosti (re-use) v EFEM a identita prvku v modelu 27
3.1 Zavedení principu maximálního re-use při tvorbě IS....27

3.2 Úplnost dokumentace z hlediska opětovné použitelnosti 29
3.3 Objektově orientovaný přístup a jeho použití v EFEM 30
3.3.1. Chyba ztráty identity prvku 30
3.3.2. Vnější a vnitřní pohled na prvek modelu......................................31
3.3.3. Principy objektově orientovaného přístupu 32
3.3.3.1 Objektově orientované programování 34
3.3.3.2 Analytické modelování 34
3.3.3.3 Modelování podniku 34
3.3.3.4 Syntaxe UML...34
3.3.3.5 Objektově orientovaná lidská mysl a abstrakce....................35
3.4 Praktické důsledky použití OOAP pro řízení projektů a obecná doporučení pro vedoucího projektu...35
3.4.1.1 Princip řízení projektu tvorbou dokumentů 35
3.4.1.2 Princip zavedení rolí...36
3.4.1.3 Princip jedné hlavy 36
3.4.1.4 Princip opětovné použitelnosti pro pravidla tvorby dokumentů36
3.4.1.5 Princip opětovné použitelnosti mezi dokumenty...................37
4. Použití a syntaxe CLASS MODELU v EFEM ve fázi analytického modelování38
4.1 Principiální úvahy tvorby modelu tříd ve fázi analytického modelování...38
4.2 Úroveň informace jako typ a výskyt, úroveň meta a pojem třída v AM....38
4.3 Syntaxe modelu tříd analytického modelování.......................................40
4.3.1. Zavedení třídy v analytickém modelu tříd 41
4.3.2. Kompozice..42
4.3.2.1 Multiplicita..42
4.3.2.2 Jednosměrné a obousměrné vztahy 43
4.3.2.3 Role...43
4.3.3. Atribut 44
4.3.4. Běžná asociace 45
4.3.4.1 Běžná asociace jako provázání nezávislých entit.................45
4.3.4.2 Vlastnost „*isNavigable*“...47
4.3.4.3 Kvalifikace vazby 48

4.3.4.4 Běžná asociace jako vztah k parentovi v kompozici ku N 49
4.3.5. Sdílená neboli slabá agregace 50
4.3.6. Asociativní třída 52
4.3.6.1 Běžná asociace ku N...54
4.3.7. Násobná asociativní třída 55
4.3.8. Vztah generalizace specializace...56
4.3.8.1 Abstraktní třída 57
4.3.9. EFEM a syntaxe UML v CLASS MODELU 58
5. Přehled modelů UML použitých v EFEM a jejich postavení v projektu............59
5.1 USE CASE MODEL (pouze AM)...59
5.2 CLASS MODEL (AM, D) 60
5.3 SEQUENCE MODEL (AM, D)...60
5.4 COLLABORATION MODEL (AM, D) 61
5.5 STATE CHART MODEL (AM, D, BM)...61
5.6 ACTIVITY MODEL (AM, D, BPM) 62
5.7 COMPONENT MODEL (pouze D) 62
5.8 DEPLOYMENT MODEL (pouze D)..62
5.9 Rozdělení modelů podle nezbytnosti dokumentace.................................63
5.9.1. Modely nezbytné pro projekt..63
5.9.2. Princip minimální, jednoznačné a přitom úplné dokumentace 65
5.9.3. Modely podpůrné pro projekt 66
6. Syntaxe UML používaná v EFEM..67
6.1 Společná syntaxe ve všech modelech UML 67
6.1.1. TAGGED VALUE...67
6.1.2. STEREOTYPE 67
6.1.3. NOTE..68
6.1.4. CONSTRAINT 68
6.1.5. DEPENDENCY..68
6.1.5.1 Odvoditelné vztahy DEPENDENCY 69
6.1.6. PACKAGE 71
6.1.6.1 PACKAGE a NESTING 71

6.1.6.2 PACKAGE a DEPENDENCY 72
6.1.6.3 PACKAGE a EFEM MODEL MANAGMENT 74
6.1.6.3.1 CONTROL PACKAGE LIBRARY MODEL.........................76
6.1.6.3.2 Porovnání použití dokumentu MANIFEST a použití CONTROL PACKAGE LIBRARY MODEL 77
6.1.6.3.3 BATCH IMPORT a BATCH EXPORT pro CONTROL PACKAGE 78
6.1.6.3.4 Prvky CONTROL PACKAGE a jejich NESTING................78
6.1.6.4 PACKAGE a CLASS MODEL AM 79
7. Použití USE CASE MODELU v EFEM 80
7.1.1. Prvky PROFITABLE USE CASE a CO-USED USE CASE...........80
7.1.2. Dodržení zásady „případ užití je popisem budoucího programu“ .81
7.2 Technika vyhledávání všech prvků USE CASE pomocí rozkladu procesů podniku (BPM) 83
7.2.1. Podstata rozkladu BPM 84
7.2.2. Syntaxe dekompozice procesů podniku 85
7.2.2.1 Prvek BUSINESS PROCESS...85
7.2.3. Rozklad procesů a kapitoly uživatelské příručky 87
7.2.4. Syntaxe pro vztah prvků USE CASE a procesů podniku 87
7.2.5. Modely chodu procesů podniku 90
7.2.6. Syntaxe chodu procesů v EFEM 92
7.3 USE CASE MODEL 93
7.3.1. Prvek USE CASE v USE CASE MODELU 94
7.3.1.1 Název prvku USE CASE 94
7.3.1.2 USE CASE SCENARIO a jeho syntaktická pravidla.............95
7.3.1.3 Vzory pro scénáře prvků USE CASE, SCENARIO PATTERNS 95
7.3.1.3.1 Vzor pro spojený pojem..96
7.3.1.3.2 Vzor pro scénář naplnění běžné asociace (číselníkové vazby) obsluhou výběrem ze seznamu 97
7.3.1.3.3 Vzor pro scénář naplnění běžné asociace zadáním identifikátoru a nalezením prvku..98

7.3.1.3.4 Vzor pro scénář přidání resp. odebrání prvku agregace ku N v USE CASE SCENARIO 98
7.3.1.3.5 Vzor pro scénář editace v USE CASE SCENARIO 99
7.3.1.3.6 Vzor pro scénář zadání prvku v kompozici ku jedné v USE CASE SCENARIO..99
7.3.1.3.7 Vzor pro scénář hromadného zpracování v USE CASE SCENARIO 100
7.3.1.3.8 Vzor pro scénář vyhledání instance v seznamu 100
7.3.1.3.9 Specifické scénáře v USE CASE SCENARIO.................100
7.3.1.4 Použití BASIC PATH a ALTERNATE PATH pro EXCEPTION FLOW..101
7.3.2. Interakce mezi prvky USE CASE v USE CASE MODELU..........104
7.3.2.1 Vnější a vnitřní pohled na prvek USE CASE v interakci 104
7.3.2.2 Princip maximální opětovné použitelnosti v USE CASE MODELU 105
7.3.2.3 Vztah interakce případů užití mezi projekty........................105
7.3.2.4 Interakce INCLUDE v USE CASE MODELU......................105
7.3.2.4.1 Instance prvku USE CASE a interakce INCLUDE...........106
7.3.2.5 Interakce EXTEND v USE CASE MODELU.......................108
7.3.2.6 Interakce GENERALIZATION - SPECIALIZATION v USE CASE MODELU 109
7.3.2.7 PRECONDITION a POSTCONDITION 111
7.3.3. Prvek ACTOR v USE CASE MODELU.....................................113
7.3.4. Význam prvku ACTOR pro EFEM a časté chyby při jeho zavedení116
7.3.4.1 Chyba „Neplodné diskuse nad kontextem prvku ACTOR“ .116
7.3.4.2 Chyba „Příliš mnoho štěňat u misky“.................................118
7.4 Kontrola úplnosti dokumentu UC MODELING 120
7.5 Použití dokumentu UC MODELING v projektu.....................................122
7.5.1. Použití dokumentu pro vývoj...122
7.5.2. Použití dokumentu pro vedoucího projektu................................122
7.5.3. Použití dokumentu pro tvorbu dokumentů strategického modelování 123
7.5.4. Použití dokumentu pro smlouvy s odběratelem 123

7.5.5. Použití dokumentu pro testování 124
7.5.6. Použití dokumentu pro obchodní oddělení 124
7.5.7. Použití dokumentu pro uživatelskou dokumentaci......................124

1. Úvod co je EFEM?

1.1 Tvorba softwaru metodou „Tunel“

Normy ISO označují jeden z často používaných postupů tvorby softwaru příznačným názvem „TUNEL“. Je charakterizován tím, že na počátku projektu se vstupuje do černého tunelu, kterým se prochází poslepu od stěny ke stěně, až se nalezne jeho konec.

Tento přístup je charakterizován následujícími vlastnostmi:

- chybí koncepce vývoje a není možná opakovatelnost výsledků
- není možná jakákoliv predikce v projektu
- neexistuje možnost opakovatelných postupů (tj. platí obecná neznalost kdy má co kdo dělat)
- velmi nízká transparence výsledků, nelogičnosti, chyby, nepřehlednost systému
- vysoký stupeň chaosu ve vývoji
- vznik softwarových slepenců, nedodělků
- vysoké nároky na operativní řízení
- vysoká hektičnost prací (předělávky, zbytečná práce apod.) včetně nároků na volný čas
- nízká kvalita softwaru
- nespokojenost zákazníka
- špatné vztahy na pracovišti
- zvýšené náklady a ztráty

Tento způsob tvorby informačních systémů se z uvedených důvodů zásadně nedoporučuje.

1.2 Tvorba softwaru byrokratickým způsobem

Druhým extrémem je snaha zbavit se předešlých nedostatků přehnaně „perfekcionalistickým“ a současně byrokratickým způsobem.

Tento extrémně opačný přístup je charakterizován následujícími body:

- přehnaný optimismus ve změnách metodického řízení
- snaha o zavedení postupů bez zkušeností s nimi
- předjímání a vymýšlení nesmyslných postupů neověřených praxí
- snaha o revoluční skoky ve změnách způsobů práce s následnými krachy při zavádění těchto postupů
- zavádění byrokratických mašinérií do vývoje
- dokumentace pro dokumentaci, tvorba zbytečných anebo zbytečně složitých dokumentů vývoje, zbytečná práce v dokumentaci
- pomalý vývoj a následná netrpělivost zákazníka
- pomalá zpětná vazba ve výsledcích
- nespokojený zákazník
- špatné vztahy na pracovišti
- zvýšené náklady a ztráty

Tento přistup k tvorbě softwaru se také nedoporučuje.

1.3 Extrémně Efektivní Modelování jako technologie tvorby SW

Cílem přístupu Extrémně Efektivního Modelování (EFEM) je odstranit předešlé nedostatky obou zmíněných přístupů. Technologie EFEM na základě praktických zkušeností zavádí metodiky modelování a vývoje vedoucí k efektivní tvorbě softwaru. Tyto metodiky se vyznačují zejména tím, že odstraňují uvedené nedostatky obou chybných postupů. Navíc oproti jiným technologiím se EFEM vyznačuje těmito body:

- extrémně vysoká efektivita vývoje při zachování požadavků na požadovanou dokumentaci při použití syntaxe jazyka UML
- mnohem vyšší jednoduchost, vyšší stupeň logiky a tedy vyšší transparence, a to jak artefaktů vývoje, tak postupů prací

- přenositelnost a implementační kompatibilita nejenom výsledků vývoje, ale i samotného EFEM mezi CASE nástroji, tj. implementační přenositelnost technologie
- aplikovatelnost principů EFEM na sebe sama, tj. na samotné zavedení EFEM ve firmě. Principy EFEM se nevztahují pouze na způsob tvorby SW, ale jsou aplikovány i na způsob zavádění EFEM. Tato skutečnost výrazně zjednodušuje zavedení této technologie do firmy
- jako důsledek předešlého bodu snadnost zavedení EFEM ve firmách

Požadavek aplikovatelnosti EFEM sama na sebe má principielní charakter. Jinak složitý proces zavádění technologie podléhá v tomto případě stejným již jednou přijatým principům EFEM. Stačí jednou tyto principy přijmout obecně. Postup zavádění těchto principů pro tvorbu SW se tak stává jednoduchým, logicky transparentním a kvalitně řízeným procesem.

1.4 Produkty řady EFEM

Metodologie EFEM má svůj praktický výstup v řadě produktů. Tato řada produktů zavádí nástroje pro použití EFEM použitelné pro modelovací nástroj ENTERPRISE ARCHITECT (dále také EA).

Řada produktů se dále dělí podle jednotlivých vydání na produkty se zaměřením na určité oblasti modelování, tyto produkty jsou postupně uvolňovány k prodeji. Fyzicky se jedná se o souhrn souborů v elektronické podobě obsahující artefakty:

- Tato skripta „Extrémně Efektivní Modelování s použitím UML". S vydáním nového produktu řady OC dochází také k vydání nové verze těchto skript. Skripta mají za úkol seznámit uživatele technologie EFEM s jejími principy.
- Postupky se zaměřením na EA v PDF formátu pro zavedení firemní metodologie
- Pokyny jsou určeny pro vedoucí pracovníky vývojových týmů, seznamují uživatele EFEM s nasazením EFEM a s možnými postupy v EFEM
- Šablony a doplňující prvky jako jsou vzory určené přímo pro nástroj EA, které umožňují efektivně a snadno použít EA v rámci EFEM technologie.

Důležité upozornění: Oprávnění k použití specifických postupů technologie EFEM vyžaduje podle autorského zákona zakoupení licence daného produktu řady EFEM (viz úvodní upozornění na počátku dokumentu).

2. Úrovně abstrakce informačního systému

2.1 Zavedení pojmu abstraktní úrovně

Jedním z hlavních pilířů technologie Extrémně Efektivního Modelování (EFEM) je důsledné používání tzv. úrovní abstrakce softwaru. Pojem úroveň abstrakce je zaveden již v normách ISO a pro modelování informačního systému se považuje za velmi důležitý. Z toho důvodu je zde pojmu abstraktní úroveň věnována náležitá pozornost. Správné pochopení úrovní abstrakce usnadňuje nejenom další práce v modelování pomocí UML, ale vede také ke správným postupům ve fázování vývoje informačního systému.

Pokud se vyvíjí informační systém, musí se o něm tvůrci vyjadřovat přesně a také jednoznačně. V postupech prací a následně v dokumentaci se musí dodržovat určitá úroveň abstrakce popisu, tj. úroveň abstrakce při náhledu na systém. Z hlediska praxe a efektivity technologie EFEM se doporučuje dodržovat minimálně tři úrovně abstrakce:

- kódování
- modelování designu
- analytické modelování

2.1.1. Nejnižší úroveň abstrakce - kódování

Na nejnižší úrovni abstrakce je na informační systém nahlíženo pouze jako na kód, který lze zkompilovat do „chodících“ souborů. Jedná se o pohled programátora, který tento kód tvoří. Míní se tím nejenom veškerý kód v daném jazyce, ale i SQL příkazy, skripty, zkompilované komponenty apod. Pojem nejnižší úroveň zde nemá význam hanlivý, je tím myšleno pouze „technologicky nejníže“. Pokud se tedy prohlíží zdrojový kód, jedná se o nejnižší úroveň abstrakce daného informačního systému.

Tato nejnižší úroveň abstrakce se také nazývá „implementační úroveň“, „realizační úroveň“, nebo prostě ***kódování (dále také zkratka C jako „code“)***. Výslednými dokumenty, které ztvárňují myšlenky z této úrovně abstrakce, jsou zdrojové kódy, zkompilované balíky souborů, SQL skripty, apod. Spadá sem vše, co v konečném důsledku fyzicky realizuje a fyzicky instaluje informační systém.

2.1.2. Nejvyšší úroveň abstrakce - analytické modelování

Protipólem oproti nejnižší úrovni abstrakce informačního systému je nejvyšší úroveň abstrakce nazývaná ***analytické modelování (dále také AM jako „Analytical Modeling“)***. Výsledkem analytického modelování jsou dokumenty ve formě modelů, které popisují podrobně informační systém v nejvyšší možné abstrakci pouze pomocí pojmů bez implementačních podrobností.

Na této úrovni se pohybují všichni účastníci projektu, kteří nazírají na informační systém jako na souhrn evidovaných pojmů a jako na souhrn chování výskytů těchto pojmů. Z pohledu této úrovně abstrakce systém „něco provádí“, „něco umí“, „něco eviduje“ apod. Aniž by se na této úrovni zaváděly jednotlivé elementy z programovacího jazyka, popisuje se podrobně, co systém provádí, jaké informace se evidují a jakou mají tyto informace skladbu. Používají se pouze pojmy jako synonymum pro „evidované informace“ (například faktura, běžný účet apod.). Popisuje se také chování výskytů z těchto pojmů.

Znamená to, že popis systému na abstraktní úrovni AM je oproštěn od prvků a syntaxe daného programovacího jazyka a je ve svém vyjádření implementačně nezávislý. Myšlenky ztvárněné na této úrovni mají povahu modelů informačního systému. Efektivním a standardním jazykem pro jejich zápis je modelovací jazyk UML.

Důležité je nyní upozornit na skutečnost, že předmětem AM je informační systém. Znamená to, že se jedná o nejvyšší abstrakci napsaného programu, přičemž samotný program je realizován na nejnižší úrovni - kódování. Modely AM nejsou nějakými „obecnými“ větami, ale konkrétním popisem programu.

Pomocí modelů v UML se na abstraktní úrovni AM odpovídá na základní otázku „CO?“, tj. odpovídá se na otázku „o co v informačním systému jde, co se eviduje a jak se výskyty informací chovají“. Na této úrovni se pohybují všichni účastníci projektu, včetně těch, kteří neumějí programovat, ale také těch, kteří programovat umějí. V okamžiku, kdy se vyžaduje takzvaně „vysvětlit“ k čemu systém slouží, co eviduje a jak eviduje apod., automaticky se pohybujeme na této nejvyšší úrovni abstrakce.

Úroveň abstrakce AM je používána všemi účastníky projektu včetně uživatelů resp. expertů na danou problémovou doménu, tj. z hlediska tvorby informačních systémů „laiků“. Výstupy z této úrovně jsou po určitých drobných úpravách srozumitelné každému. Díky této skutečnosti se mohou k informačnímu systému vyjadřovat další účastníci projektu, kteří jinak netvoří technologický tým, což jsou například externí konzultanti, uživatelé, obchodníci apod.

2.1.3. Střední úroveň abstrakce modelování návrhu

Mezi abstraktní úrovní AM a abstraktní úrovní kódování C existuje ještě jedna abstraktní úroveň, která se nazývá ***modelování designu (dále také zkratka D)***. Synonymem pro tuto úroveň je také název „modelování návrhu“ nebo také pouze „návrh“ resp. „design“. Na této abstraktní úrovni, tj. z úrovně tohoto pohledu, vzniká technologický návrh. Jsou to modely navrhující (zadávající) realizaci v daném vývojovém prostředí. Zjednodušeně se dá říci, že modely návrhu „přetavují“ modely z abstraktní úrovně AM, která je implementačně nezávislá, do konkrétního návrhu (modelu), který je již implementačně závislý, ale jedná se stále ještě o návrh, tj. o model. Prvky designu tedy ještě nejsou konkrétními již realizovanými prvky (kódované apod.), ale pouze jejich technologickým návrhem. Modelování návrhu se stává zadáním pro nejnižší úroveň abstrakce, tj. pro kódování.

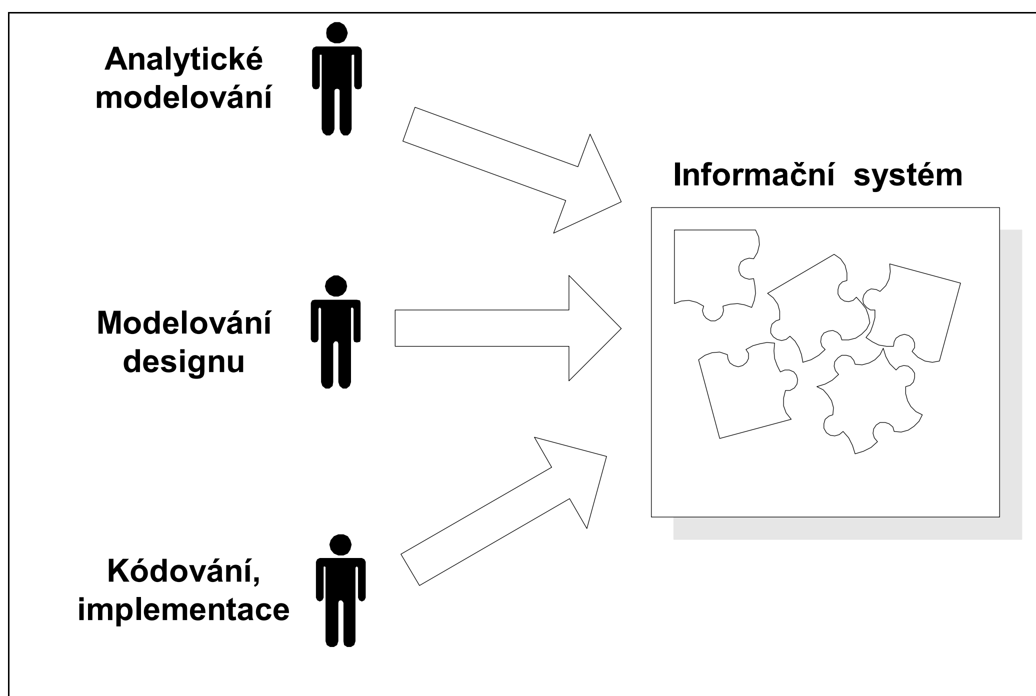
Pro lepší pochopení významu úrovně abstrakce modelování návrhu D si lze představit, že artefakty abstraktní úrovně AM jsou úplným zadáním pro tvorbu modelů návrhu D a následně modely návrhu jsou úplným zadáním pro kódování C. Artefakty návrhu jsou stále ještě modely, ale v daném vývojovém prostředí již konkrétně zadávajícími, jak se bude systém realizovat.

Modelování návrhu D je na rozdíl od AM již implementačně závislé a poplatné danému prostředí. Na rozdíl od abstraktní úrovně AM se na úrovni abstrakce modelování návrhu D nepoužívá výhradně UML. Ukazuje se jako efektivní použít pro tvorbu výstupů z této úrovně ve větší míře nástroje daného vývojového prostředí. Výsledné dokumenty tak mohou podléhat jiným pravidlům, než je syntaxe UML. V některých případech bývá tvorba některých prvků kódování zjednodušena automatickými procesy (generace kódu apod.).

Zatímco úroveň abstrakce AM odpovídá na otázku „CO?“, modelování návrhu D odpovídá na otázku „JAK?“ ve smyslu „jak má být to, co je ztvárněno v analytickém modelování, naprogramováno“.

Vznikají tak dokumenty designu jako zadání pro tvorbu kódu.

Uvedené tři základní úrovně abstrakce zobrazuje následující obrázek:



obrázek 1 Abstraktní úrovně informačního systému

2.2 Úplnost dokumentace v EFEM

Je zřejmé, že pokud se požaduje zdokumentovat vývoj ve své úplnosti, musí být dostatečně zdokumentován na všech třech úrovních AM + D + C. Mnoho firem se dopouští té chyby, že nemá dostatečně zdokumentovánu úroveň abstrakce analytického modelování a také modelování designu je mnohdy velmi slabé.

Z hlediska technologie EFEM je dokumentace AM + D + C natolik úplná, že:

- chodící kód (C) je úplný, je vyhovující ve funkčnosti (bezchybný), transparentní a lze jej opakovaně kontrolovat a upravovat
- modely designu (D) jsou dostatečné pro tvorbu předešlého bodu (kódování) a lze je opakovaně kontrolovat a upravovat
- modely úrovně analytického modelování (AM) jsou dostatečné pro tvorbu předešlého bodu (designu) a lze je opakovaně kontrolovat a upravovat

Jedním z hlavních cílů technologie EFEM je právě stanovení postupů pro maximálně efektivní tvorbu artefaktů na úrovních abstrakce AM+D+C tak, aby tento postup nepřinášel zpoždění v realizaci projektu.

2.2.1. Mapování z úrovně AM do D, fázování vývoje

2.2.1.1 Dokumenty typu AM+D+C a dokumenty typu přechodu mezi úrovněmi

Ukazuje se, že pro celkovou dokumentaci vývoje nestačí pouze vytvořit artefakty na těchto třech úrovních AM+D+C. Pokud se jedna úroveň abstrakce opouští a přechází se na jinou úroveň abstrakce, musí se tento přechod také náležitě popsat a zdokumentovat. Jednou z nejčastějších chyb při vývoji informačního systému je právě nedodržování tohoto principu. Tvůrce při popisu systému používá současně několik abstraktních úrovní a provádí jejich „promíchání“, což je z hlediska dokumentace nepřípustné. Každý pohled na informační systém, tj. model informačního systému, je vymezen svou úrovní abstrakce, tj. buď patří do AM nebo do D nebo do C. Tím je také ohraničen ve své abstrakci. Vyžaduje se proto přesně definovat úroveň abstrakce, kam model spadá.

Na straně druhé je třeba také při fázování vývoje definovat přesně přechod z vyšší úrovně abstrakce do nižší úrovně abstrakce, tj. zdokumentovat použitý proces přechodu z modelu vyšší abstrakce do modelu nižší abstrakce. Požadavek na úplnost dokumentace vede také k nutnosti popsat postupy přechodů mezi úrovněmi abstrakce od vyšší k nižší úrovni. Celý vývoj informačního systému se tak stává tvorbou dokumentů ze dvou oblastí

- dokumentace modelů z jednotlivých úrovní abstrakce AM+D+C
- dokumenty popisujících přechody mezi úrovněmi abstrakce

Teprve až souhrn artefaktů z těchto dvou typů dokumentace (tj. modely systému na abstraktních úrovních a přechody mezi úrovněmi) dává pode EFEM ucelenou dokumentaci vývoje.

2.2.1.2 Fázování prací na SW

Tři různé pohledy na IS z hlediska abstraktních úrovní AM+D+C implikují posloupnost prací na vyvíjeném IS, tj. ***fázování projektu***. Postupuje se vždy logicky od nejvyšší úrovně abstrakce AM směrem k nižší úrovni abstrakce, tedy od AM do D a z D do C. Časová posloupnost tvorby však neznamená, že musí být hotovy všechny prvky AM v úplnosti k tomu, aby se mohlo přistoupit k tvorbě modelování designu D. Avšak vždy na základě modelů analytického modelování vznikají artefakty modelování návrhu. Na základě výsledku modelování designu vznikají následně artefakty zdrojového kódu jako je samotný zdrojový kód, SQL příkazy apod.

Výsledek vyšší úrovně abstrakce se stává zadáním pro tvorbu nižší úrovně abstrakce. Přechod z analytického modelování do modelování designu nazýváme ***mapování z analýzy do designu***. Postup mapování z AM do D je také nutnou součástí dokumentace. Přechodu z designu do kódu se v technologii EFEM říká ***kódování***, nebo ekvivalentně ***realizace***.

2.2.2. Analytické modelování jako abstraktní programování

Tvorba na úrovni abstrakce AM je velmi obtížná a vyžaduje zkušené pracovníky - analytiky. Tvorba dokumentů AM podléhá svým přesným pravidlům. Syntaxe AM umožňuje velmi přesně a velmi podrobně popsat systém, aniž by se přitom používaly prvky z dané konkrétní (vývojářské) technologie, tj. prvky z D. K vyjadřování se používají abstraktnější pojmy v notaci UML. Celý systém se takto popisuje ve své úplnosti na abstraktnější úrovni. Výsledkem je abstraktní obraz informačního systému, který se poté realizuje v dalších nižších úrovních abstrakce D a následně C.

Tvorbu modelů AM lze přirovnat k ***abstraktnímu programování***. Výsledkem je „implementačně nezávislý abstraktní program informačního systému“. Modely z AM lze proto považovat za ekvivalent abstraktního obrazu budoucího kódu. Pokud se v technologii EFEM používá pojem „model informačního systému na úrovni AM“, má tím autor na mysli de facto „abstraktní model konkrétního napsaného programu“.

2.2.3. Problém podrobnosti a úplnosti analytického modelování

Velmi častou chybou při chápání abstraktní úrovně analytického modelování je nedocenění její poměrně hluboké podrobnosti. Skutečnost, že tato úroveň abstrakce je nejvyšší, neznamená, že se jedná o jakýsi mlhavý, nebo dokonce neurčitý popis. Modely na této úrovni jsou velmi podrobné a obsahují velmi mnoho informací, tj. popisují velmi podrobně informační systém.

Modely AM obsahují vše, co je třeba k naprogramování systému, pouze obsah těchto modelů je implementačně nezávislý. Popis systému neobsahuje konkrétní prvky z konkrétního vývojového prostředí. Neobsahuje konkrétní rezervovaná slova z programovacích jazyků a prostředí (jako je například `unit`, `record`, `label`), neobsahují konkrétní prvky obrazovek z prostředí (jako je například `CommandButton`, `TextBox`, `Listbox`), neobsahují syntaxi z konkrétních relačních databází (jako je například `"SELECT * FROM TOSOBA"`) apod.

Namísto takovéhoto konkrétního vyjádření v daném prostředí se používají abstraktnější vyjádření, jako jsou věty například: „Obsluze se zobrazí seznam faktur“,

„Obsluha vybere..., obsluha zadá...“ apod. Přitom tento popis se chápe jako úplný, takže pokud se něco v modelech analytického modelování nenachází, nebude to naprogramováno.

2.2.4. Maximální efektivita přechodu z analytického modelování do designu a do kódu

Při zavádění moderního způsobu tvorby informačního systému při respektování úrovní abstrakce je mnohdy jednou z často kladených otázek, zda tvorba modelů na úrovních AM+D+C příliš nezpomalí celkový vývoj. Je třeba podotknout, že při nesprávném přístupu opravdu takové riziko hrozí. Firma se může utopit v horních úrovních abstrakce, aniž by byl přitom viděn cíl, tj. naprogramovaný systém. Pracovníci neustále malují zbytečné obrázky a efekt z hlediska programování není buď žádný anebo minimální. Nakonec se paradoxně programuje podle jiných dokumentů resp. podle slovního zadání, které vůbec neodpovídá dokumentaci.

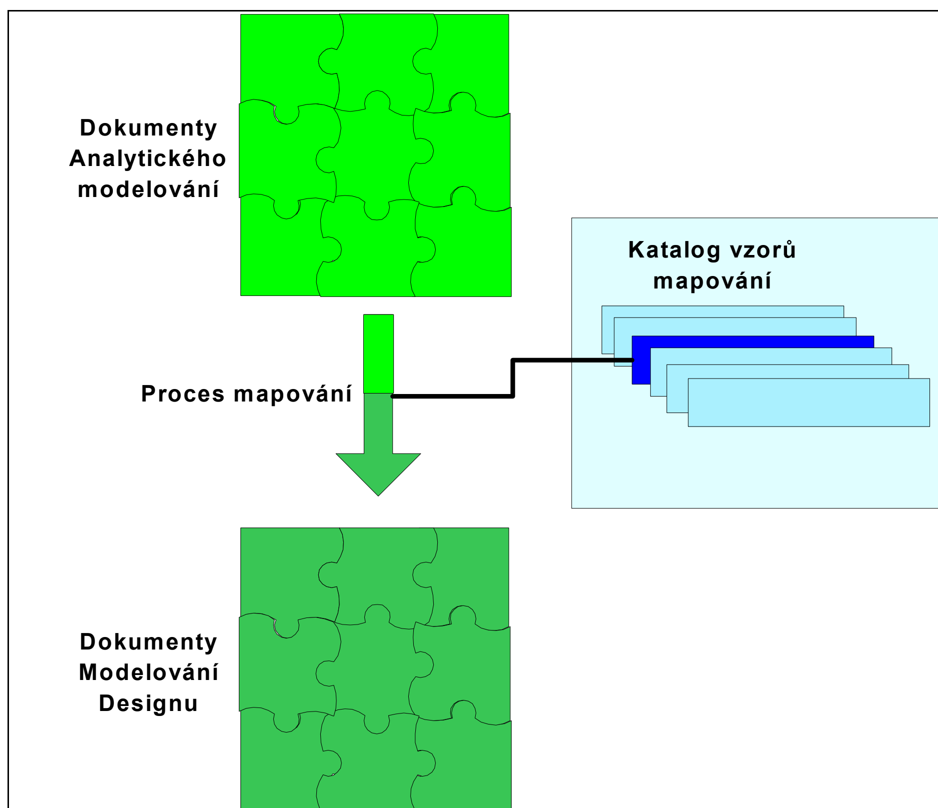
Jedním z přínosů technologie EFEM jsou pracovní postupy, které odstraňují toto riziko a vedou skutečně k extrémně rychlé tvorbě informačního systému při úplnosti dokumentace na všech úrovních abstrakce. Řešení v EFEM spočívá ve dvou vzájemně souvisejících bodech:

- zavedení správné, přesné a efektivní syntaxe modelování na horní úrovni abstrakce AM (syntaxe abstraktního programování v AM využívající UML)
- přesně definované a opakující se postupy mapování z vyšší úrovně abstrakce do nižší, tj. zavedení vzorů mapování z AM do D

Díky těmto dvěma bodům lze velmi rychle přecházet z úrovně AM do D. První bod zabezpečuje jednoznačnost, jednotnost a přesnost dokumentace AM, čímž vzniká přesné a jednoznačné zadání pro design. Druhý bod zefektivňuje přechod z AM do D opakujícími se postupy mapování, takž mnohdy dokumentace designu bývá tvořena pouhým odkazem na zvolený postup.

Co se týče dalšího přechodu z úrovně abstrakce modelování designu do kódování (tj. sama realizace z D do C), tak ta již tolik nepodléhá technologii EFEM. Jeho efektivita je dána efektivitou použití nástrojů daného konkrétního vývojového prostředí.

Postup vedoucí k efektivitě a rychlosti tvorby informačního systému při přechodu od AM do D ukazuje následující obrázek:



obrázek 2 Postup mapování v EFEM je dán katalogem vzorů

Pracovník tvořící modely designu D (tmavě zelená oblast) vychází jednak z modelů analytického modelování AM (světle zelená oblast) a současně vybere a použije již zdokumentovaný vzor mapování do designu „odkazem“ do číselníku“ všech vzorů mapování (modrá oblast). Dokumentace se tak skládá ze tří částí:

- dokumenty analytického modelování (AM)
- dokumenty designu (D)
- použitý výskyt vzoru mapování (AM =>D)

Tohoto přístupu se použije vždy a bez výjimky, a to i kdyby se daný vzor mapování použil pouze jednou. I v tomto případě je třeba tento jinak unikátní vzor umístit do katalogu a odkázat se na něj (i když na něj bude pouze jeden odkaz).

Seznam možných vzorů mapování se tak v životě firmy rozšiřuje a firma tak stále více efektivněji přechází z AM do D.

2.2.5. Modely podniku (BUSINESS MODELING) a jejich vztah k AM

Pro úroveň abstrakce AM se někdy používá také zkrácený název „analýza“. Pracovník, který tvoří modely z této úrovně, se nazývá analytik. Protože se mnohdy pojem analýza používá v jiných souvislostech a v jiném významu, dochází tak často ke kolizi pojmů.

Někdy se pod pojmem analýza skrývá tzv. modelování podniku (BUSINESS MODELING - dále také zkratka BM). I v tomto případě se jedná o modelování, ale liší se od analytického modelování předmětem modelu, tj. liší se tím, co se vlastně modeluje.

Zatímco v analytickém modelování AM je předmětem modelu zkoumaný informační systém, tj. jedná se vlastně o abstraktní obraz programu (viz obrázek 1 Abstraktní úrovně informačního systému), u BM je předmětem samo prostředí, tj. podnik, do kterého má být informační systém dosazen. Protože dané prostředí podniku je také objektově orientované, lze pro toto modelování použít také syntaxi UML. Objekty v modelech BM jsou zde objekty daného prostředí (pracovník, vedoucí, oddělení, oběžník, obsluha, knihovník apod.). Z tohoto modelování má významné postavení tzv. BUSINESS PROCESS MODELING, zkráceně BPM. Nosným prvkem tohoto modelu jsou aktivity podniku neboli procesy podniku.

Modelování podniku se provádí hlavně z těchto důvodů:

- Je třeba vyvinout informační systém a nejsou dobře známy procesy podniku, které tento informační systém bude podporovat. V tomto případě se jedná pohled na modely podniku z hlediska softwarových vývojářů, konkrétně analytiků, kteří „pátrají“ po funkcionalitách budoucího informačního systému. Modely podniku napomáhají nalézt tyto funkcionality systému. Zvláštní postavení má toto modelování jako technika vyhledání všech případů užití (prvky USE CASE) informačního systému (viz odpovídající kapitola tvorby dokumentu UC MODELING).
- Je třeba navrhnout nové procesy podniku (například nová služba banky, nová služba operátora mobilních telefonů apod.) a současně navrhnout nové funkcionality informačního sytému. V tomto případě se problém také týká softwarových pracovníků.
- Je třeba optimalizovat procesy podniku, například navrhnout nový způsob koordinace přepravy, optimalizace chodu podniku, chodu skladu apod. Artefakty z modelů podniku jsou přehledné modely v grafické podobě, které srozumitelně a jasně popisují chod jak původních, tak optimalizovaných procesů podniku.

Pro EFEM jsou důležité body 1 a 2, zatímco bod 3 spadá pod kompetenci konzultačních firem optimalizujících chod podniků.

Pokud dodavatel softwaru nezná procesy podniku, tj. jak to má v daném prostředí správně chodit, nemůže navrhnout dobrý informační systém. Z toho důvodu některé softwarové firmy dodávají modelování podniku BM jako součást vývojových prací na informačním systému. Na základě praxe v několika desítkách firem je třeba upozornit na některá úskalí tohoto modelování.

V prvé řadě je třeba upozornit na skutečnost, že nelze vytvořit pouze model podniku BM a nezohledňovat přitom navrhovaný informační systém. Je nezbytné tvořit oba modely BM a AM souběžně a neustále přecházet od modelu podniku BM k modelům AM a zpět. V opačném případě hrozí, že bude vykonána spousta zbytečné práce a projekt tvorby informačního systému bude ohrožen. V případě, že firma bude nejprve modelovat podnik a poté informační systém, hrozí tato rizika:

- Existuje zřejmá zpětná vazba mezi navrhovaným modelem informačního systému a modelem podniku. Podle toho, jaké se navrhne řešení informačního systému, může se zpětně návrhem IS ovlivnit chod podniku. To je nejvážnější riziko postupu modelování podniku, které nezohlední návrh IS.
- Samotné modelování podniku vyvíjené bez ohledu na funkcionalitu informačního systému nemusí mít pevně stanovené hranice. Pod jeho působnost může spadat spousta jinak nezajímavých věcí, například lze do modelů podniku zahrnout nejenom již dále neřešené oblasti, ale je možné také zbytečně „rozpitvávat“ nezajímavé detaily. Nejenom, že to stojí čas a peníze, ale navíc zbytečná práce má nepříjemný dopad na tvůrčí atmosféru v týmu.
- Mnohdy nastává nepříjemná situace, že model podniku BM je zaměněn s modelem informačního systému resp. naopak. Jeden pracovník hovoří o podniku a druhý pracovník tento model chápe jako model informačního systému resp. obráceně. Obzvlášť nepříjemná je tato záměna při vztahu mezi dodavatelem a odběratelem softwarového řešení, kdy dodavatel odevzdává model podniku a odběratel jej chápe jako model informačního systému. To může vést k vážným kolizím při dohodě nad funkcionalitami systému. Je třeba upozornit na to, že tyto záměny bývají velmi záludné, protože jsou dobře maskovány shodou názvů pojmů v obou oblastech modelování (například pojem směnka v bance a evidovaná směnka v IS mají stejný název, ale jsou to dva oddělené pojmy ze dvou různých modelů). Záměny tohoto typu jsou velmi snadno přehlédnutelné. Z praxe se ukazuje, že uvedená chyba je charakteristická pro modelování stavových modelů u evidenčních systémů, u čistě technologických systémů je toto riziko minimální.

Z důvodu efektivního modelování se doporučuje, aby se modely BM tvořily souběžně s modely AM a nikoliv nějak extra dopředu jako zvláštní úvodní část projektu. Model podniku BM je sice mnohdy velmi důležitý a pro projekt dokonce velmi často i nezbytný, ale z hlediska vývoje SW je třeba jej chápat pouze v rovině sekundární informace, tj. „proč je model informačního systému takový, jaký je“. Cílem EFEM jako technologie tvorby SW je získat dobrý model AM. Designér se opírá pouze o artefakty AM samotného informačního systému a z nich dostává úplné a postačující

informace pro svou tvorbu v oblasti D. Designéra modely podniku BM vůbec nezajímají, protože jsou mimo jeho rozsah řešení (považuje BM za „zbytečnou omáčku“). Častou chybou analytika bývá dodání dokumentace modelů BM a prohlášení těchto modelů za modely AM. Designér tak dostává spoustu jinak zbytečných informací o podniku, ze kterých lze teprve dedukcí vyčíst, co má vlastně program dělat. Designér tak musí tyto informace přebrat, vyhodnotit a vytvořit z nich pro sebe analytický model programu, což měl původně dostat jako zadání. Teprve poté přistupuje k designu.

Je třeba zdůraznit, že modely podniku BM mají své důležité místo v jiných částech projektu, například při popisu chodu služby podporované informačním systém, což si odběratel systému určitě rád přečte a posoudí správnost modelu podniku.

Modelování podniku je věnována dostatečná pozornost v dalším výkladu o EFEM v přímé vazbě na efektivní tvorbu analytických modelů. Postupky pro modelování podniku je také součástí dodávaného produktu

2.2.6. Analýza jako zdroj informací

Pod pojmem analýza se někdy také skrývá tvorba souhrnu dokumentů, které nějak pojednávají o dané problémové doméně a mají k řešení nějaký relevantní vztah. V tomto případě má pojem analýza význam „tvorba zdroje informací“. V žádném případě se nejedná o tvorbu dokumentace AM.

Na rozdíl od AM je zdroj informací pouze souhrnem dokumentů, který slouží k dobré orientaci v dané problémové doméně. Mezi tyto dokumenty patří: souhrn všech zákonů a předpisů z dané oblasti, výsledky konzultací se zákazníkem, požadavky na systém apod.

Na rozdíl od toho model z AM pojednává již konkrétně o samotném programu, jaké informace eviduje, jak se chovají, co dělá obsluha se systémem apod.

2.2.7. Podcenění analytického modelování a časované bomby v projektech

Mnoho firem podceňuje fázi tvorby AM. Mnohdy je tato fáze zdokumentována jen slabě anebo vůbec. Na první pohled se zdá, že práce na analytickém modelování mohou zpomalit tvorbu softwaru, ale praxe ukazuje, že opak je pravdou.

V prvé řadě je třeba si připomenout skutečnost, že každý vyvíjený software vždy prochází fází tvorby v úrovni abstrakce analytického modelování. Nelze si představit situaci, že by byl nějaký software napsán bez toho, že by se autor vůbec nezamyslel nad tím „CO vlastně programuje“. Přitom ona otázka „CO?“ je výstižnou zkratkou pro práci analytického modelování. Takže výsledky této fáze existují vždy a je jenom otázkou, zda tyto výsledky zůstanou v hlavách autorů anebo se pokud možno

efektivně zdokumentují. Chybějící dokumentace analytického modelování je nutně nahrazována sekundární tvorbou „ústních vysvětlení“ tvořených přímo na místě. Stabilní artefakt dokumentace je takto nahrazen nestabilními dočasnými artefakty rozhovorů se všemi zápornými důsledky. V tom lze také spatřovat nejnepříjemnější efekt chybějící dokumentace analytického modelování: Artefakty vývoje nejsou samostatně stojící, nelze je považovat za jeden zapouzdřený „balík“.

Důsledky neúplné dokumentace, v tomto případě chybějící dokumentace analytického modelování, jsou nasnadě. V podstatě vedou k již uvedeným efektům metody řízení projektů s názvem „TUNEL“. Projekt se stává nepřehledným a je patrná nízká transparence výsledků. Vznikají nelogičnosti, chyby, nepřehlednosti, tj. artefakty vývoje jsou nečitelné resp. čitelné velmi obtížně. Pracovníci se stávají nezastupitelnými, musí být neustále po ruce, nelze je odvolat na nové projekty, i když „starý“ projekt je již hotov. Velmi pracně se zavádějí opakovatelné postupy. Je znatelný vysoký stupeň chaosu ve vývoji. Vznikají softwarové slepence a nedodělky. V důsledku těchto jevů rostou stále větší nároky na operativní řízení. Díky nízké transparenci, nečitelnosti, díky opakování prací a tvorbě zbytečné práce se zvyšuje hektičnost prací, včetně nároků na volný čas apod. Paradoxně mnohdy tuto skutečnost zvýšeného zbytečného úsilí firmy udávají jako svou přednost. Jevy zde popsané se stávají časovanou bombou a mají pochopitelně negativní vliv na celou firmu.

2.2.8. Analytické modelování a přechod na nové technologie

Existuje ještě jedna velmi často se vyskytující situace, která očividně ukazuje na nutnost analytického modelování. Stává se, že firma potřebuje přejít z jedné technologie na novou technologii, například ze zastaralé dvojvrstvové technologie na novou třívrstvovou s využitím WEB technologie při takřka stejných požadavcích na analytické úrovni. Vzniká problém, jak tento přechod na novou technologii uskutečnit.

Nelze jen tak přejít z jedné technologie do druhé pouze na úrovni designu. To by znamenalo napsat nějaký „automat – konverter“, který by vzal jeden kód a zkonvertoval by jej do druhého kódu (z jednoho D do druhého D), podobně jako kompilátor tvoří ze zdroje spustitelné soubory. Protože takto postupovat nelze, tak je zřejmé, že autoři systému nové technologie musejí znát, co vlastně programují a tedy musí se opřít o abstraktní úroveň analytického modelování. Jednoduše řečeno musí si říct „o co tam vlastně jde“, což je podstata AM.

Pokud existují artefakty AM, pak by se v podstatě jednalo o jednoduchý projekt nového mapování z AM do jiného nového designu D podle jiných vzorů. Pokud tato dokumentace AM neexistuje, opět vzniká stejná situace uvedená již v předešlé kapitole: Pracovníci jsou nuceni předávat si své poznatky z úrovně analytického modelování pouze ústně a opět se objevuje celá škála problémů metody „TUNEL“.

2.2.9. Role v projektu: analytik, designér a programátor

Rozdělení prací do třech úrovní abstrakce s sebou přináší také rozdělení rolí ve vývojovém týmu.

Podle těchto úrovní abstrakce lze rozlišit tyto tři role:

- analytik (tvoří modely analytického modelování, příp. také BPM)
- designér (tvoří modely designu, mapuje z analýzy do designu)
- programátor (tvoří samotný chodící kód)

Následuje seznam nutných znalostí, které by pracovníci v těchto rolích měli ovládat. Pomocí tohoto výčtu dovedností lze také lépe pochopit význam odpovídajících abstraktních úrovní.

2.2.9.1 Analytik

Analytik potřebuje zejména zvýšenou až dokonalou znalost UML, tj. oproti ostatním rolím je guru na UML, zná syntaxi UML do takřka všech detailů včetně příslušného CASE nástroje. Je to jeho hlavní „abstraktní programovací jazyk“. Současně se znalostí UML by měl mít velmi dobře rozvinuté abstraktní a kreativní myšlení s vysokou představivostí. Analytik je schopen „vidět budoucí systém před sebou“. K tomu, aby byl schopen rychle předat ostatním členům týmu tuto představu, musí umět dobře a rychle modelovat, tj. musí mít schopnost rychle a přesně formulovat myšlenky a zapsat je do modelů.

Je vcelku pochopitelné, že analytik má dobré znalosti z problémové domény, kam informační systém spadá. Pokud existuje odběratel, analytik spolupracuje s konzultantem formou tzv. interview (pohovorů). Nelze opominut ještě jednu důležitou vlastnost, která vyplývá z toho, že analytik se kontaktuje se zákazníkem při určování funkcionalit systému a pomocí takovýchto „interview určuje budoucí chování systému´: Analytik musí mít také odpovídající schopnost komunikace s uživatelem nad systémem. Jsou na něj kladeny zvýšené nároky ohledně schopnosti asertivního jednání se zákazníkem. Velmi schopný analytik je schopen „dotlačit“ uživatele ke správnému řešení a to dokonce bez konfliktů a kolizí.

2.2.9.2 Designér

Designér se na systém již dívá trochu jinýma očima. Pro něj je tvorba systému již technologickým úkolem: Má zadání od analytika, umí a zná dané vývojové prostředí,

takže vyvstává otázka designéra, „jak tuto analýzu navrhne v tomto prostředí“? Pro něj jako experta na technologii softwaru je sama podstata problému v analýze v podstatě nezajímavá a slouží mu jako zdroj (zadání) pro jeho tvorbu. Jako dobrý technolog je schopen navrhnout jakýkoliv systém, když má dobré zadání. Oním dobrým zadáním jsou právě výsledky práce analytika. Synonymem pro roli designéra je někdy používaný název role „technolog“ anebo „architekt“. Designér je hlavní guru na danou vývojovou technologii. V některých případech je problematika vyvíjeného softwaru technologicky natolik složitá, že se na projektu musí podílet hned několik designérů. Každý z nich je specialistou na určitou oblast technologie (např. databáze, bezpečnost, operační systémy, WEB apod.). Znalosti designů technologů vystihuje tento seznam:

- znalost UML zejména jako pasivní čtenář
- databázová teorie, návrh ERD resp. jiné databáze (stromová apod.)
- vlastnosti používané databáze (administrace apod.)
- principy návrhu GUI daného prostředí
- problematika bezpečnosti daného prostředí
- operační systém
- sítě
- HW pro návrh nasazení SW

... a jiné speciální znalosti z dané technologie.

V neposlední řadě je třeba zmínit také znalost již známých vzorů mapování, tj. schopnost použít již jednou použitých postupů mapování.

2.2.9.3 Programátor

Pro programátora je tvorba systému již „pouhé“ napsání a rozchození daného kódu. Z toho důvodu musí mít tyto dovednosti:

- znalost UML zejména jako pasivní čtenář
- velmi dobrá znalost vývojového prostředí, dobrá orientace a znalost všech možností tohoto prostředí atd.
- velmi dobrá znalost syntaxe daného jazyka (např. Java, C# apod.) a schopnost rychle psát a samostatně testovat kód v daném prostředí
- znalosti syntaxe spolupráce s danou databází (např. SQL syntaxe apod.)
- základní znalosti designéra (nikoliv detailní)

Povaha prací na všech třech úrovních se podstatně liší: Zatímco analytik je spíše povahou „spisovatel“, který se pohybuje na vyšší úrovni abstrakce, designér jako technolog je typický představitel technokrata, který si libuje ve finesách daného prostředí. Programátor zde vystupuje již jako realizátor programu.

Je vhodné, aby se v různých projektech programátoři účastnili fáze designu, například veřejnou oponenturou resp. diskusemi apod. Práce programátora se totiž může stát v tomto rozdělení prací nezajímavá, protože není příliš kreativní.

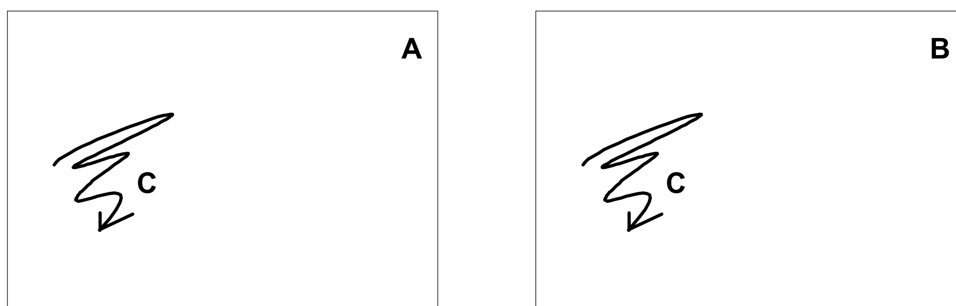
Rozdílná povaha prací mnohdy způsobuje velmi velké problémy u těch vývojářů, kteří musí pracovat v několika rolích současně. Jedná se o ten nešťastný případ, kdy pracovník je povinen odevzdávat výsledky podle pokynu „sám analyzuj, sám navrhni v technologii a sám programuj“. Takovéto pendlování mezi rolemi vyžaduje, aby se daný pracovník v určité chvíli choval jako analytik, poté se změnil na designéra a poté na programátora. Přitom však povahy těchto rolí jsou diametrálně odlišné. Nejenom, že se tímto pendlováním zabraňuje specializaci, kdy každý umí všechno (například firma má experta na relační databázi ORACLE, který současně velmi dobře ovládá problematiku zahraničních akreditivů), ale navíc rozdílná povaha prací vede i k vnitřním psychologickým konfliktům. Každý pracovník většinou tíhne k určitému typu práce, například více analytické a méně technologické anebo naopak. Pendlování mezi těmito rolemi může způsobit určité kolize při práci v jiné roli, než v té, která danému pracovníkovi lépe vyhovuje. Klasický technolog se raději vyhýbá fázi analytického modelování, klasický analytik se zase velmi nerad řeší podrobnosti dané technologie, protože při odhalení funkcionality by se rád zabýval dalšími otázkami.

3. Princip maximální opětovné použitelnosti (re-use) v EFEM a identita prvku v modelu

3.1 Zavedení principu maximálního re-use při tvorbě IS

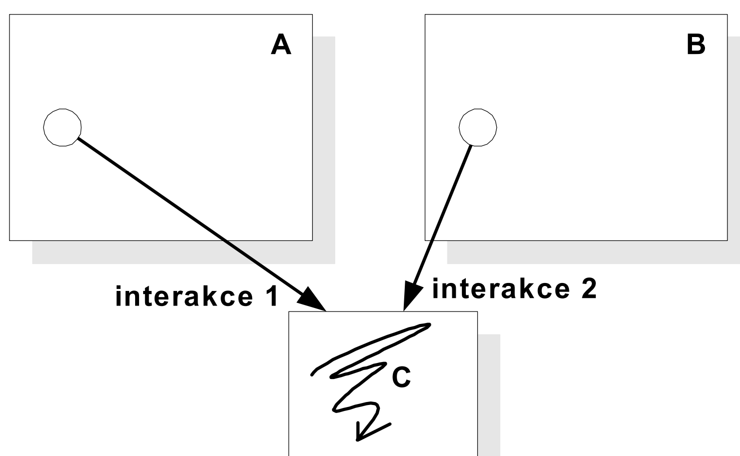
Jedná se o jednoduchý základní axiom jehož důsledky se linou celou metodologií EFEM. Pod opětovnou použitelností, kterou budeme nazývat také re-use, máme na mysli následující situaci:

Představme si, že existuje výskyt nějaké (libovolné) informace *A* a existuje výskyt nějaké (libovolné) informace *B*. Zjistíme, že ve výskytu informaci *A* se vyskytuje určitá část z A , která se opakuje i v *B*, označme tuto opakující se část jako *C* (viz následující obrázek, část C označena klikyhákem):



obrázek 3 Situace vedoucí k opětovné použitelnosti (re-use)

Při aplikaci opětovné použitelnosti se vytvoří nový prvek jako interakce mezi prvky *A, B, C*. Opakující se část *C* se „vytkne“ mimo oba prvky *A, B* a touto interakcí se prováže zpět do bodů, ze kterých byl vytknut (vznikne obdoba odkazu). Výsledkem je následující situace (viz obrázek 4):



obrázek 4 Situace zavádějící opětovnou použitelnost výskyty interakcí

Zavedení opětovné použitelnosti není v žádném případě novinkou a je obecným a dobře známým jevem v programování a obecněji v modelování SW. Můžeme namátkou jmenovat následující příklady:

- volání funkcí ve strukturálním programování,
- normalizace databáze,
- dědění a jiné interakce mezi třídami (např. asociace),
- obecně interakce mezi prvky modelu v UML, kdy jeden prvek používá druhý prvek (include a extends mezi případy užití apod.) aj.

Princip maximální opětovné použitelnosti zní v tomto případě tak, že je třeba prioritně vždy zavádět opětovnou použitelnost. Prvky *A*, *B*, *C* mohou být cokoli, čeho se může náš problém týkat. Nemusí to být pouze vývoj SW, ale například metodiky firmy, tvorba dokumentace apod. Jedná se o „návod" aplikovatelný na vše, co je třeba řešit, včetně zavádění technologie EFEM do firmy, tvorby dokumentace apod.

Princip maximální opětovné použitelnosti doporučuje následující: „Obrázek 1 je považován za indikaci chyby. Části, které by se opakovaly, se nesmí tvořit dvakrát, ale musí se vytknout jako jeden prvek a poté je třeba se na něj z několika míst odkazovat (ukazovat si na něj interakcí). To platí pro libovolnou situaci, která se řeší."

Rozdíl mezi známou opětovnou použitelností a tímto principem maximální opětovné použitelnosti je v tom, že uvedený princip nejenom že umožňuje, ale vyžaduje použití re-use. Tímto požadavkem se zavádějí postupy a vlastně se ukazuje na to, co není provedeno a co se má provést, tj. povést proces vytknutí.

Samotná opětovná použitelnost nemusí být obecně vyžadována a dokonce někdy bývá úmyslně porušována (kopírování prvků, optimalizace návrhu, rozpouštění prvků v designu apod.).

Prvky v interakci opětovné použitelnosti mohou být opravdu „cokoliv“, tj. vše, co vyžaduje opětovnou použitelnost. Mohou jimi být (a měly by být) samozřejmě prvky modelu SW, tj. prvky vyvíjeného SW, a to až po kód. Navíc však tomuto principu maximální opětovné použitelnosti podléhají například také postupy ve firmě, dokumentace projektu, zavádění vzorů, opakování postupů při designu apod. Princip maximálního re-use přikazuje opakující se postupy nějak vytknout a opětovně je použít, což vede k zavedení postupek, k tvorbě návodek, k zavedení návrhových vzorů apod. Současně se vyžaduje, aby tyto dokumenty (metodiky, vzory apod.) samy mezi sebou dodržovaly princip maximální opětovné použitelnosti, tj. aby se mezi sebou odkazovaly a propojovaly se tak, aby nedocházelo k opakování částí dokumentů.

Nedodržení principu maximální opětovné použitelnosti vede k následujícím nepříjemným efektům:

- opakování prací (při vývoji, při tvorbě dokumentace, při tvorbě metodik apod.), jako důsledek následuje ztráta efektivity
- opakování oprav při změnách (což je podobné jako předešlý bod, ale v jiné fázi tvorby a údržby SW)
- ztráta transparence řešení, což je nejnepříjemnější důsledek. Díky nedodržení principu maximální opětovné použitelnosti se některé věci opakují několikrát a to „bůhví kde“. Věci nejsou na svých logických místech, kde by se měly nacházet. Dokumentace je obtížnější, je nelogická a navíc mnohem více pracná. Hledání všech opakujících se změn je neúnosně zdlouhavé, systém se stává zbytečně složitý a nepřehledný, v důsledku více chybový. K největší ztrátě efektivity dochází právě díky ztrátě transparence, která vzniká jako důsledek nedodržení principu maximálního re-use.

3.2 Úplnost dokumentace z hlediska opětovné použitelnosti

Pro použití principu re-use se nejenom ztransparentní samotný informační systém, ale také se velmi zjednodušuje systém dokumentace. Technologie EFEM doporučuje použití tohoto principu na samotné zavádění technologie EFEM.

Mezi dokumenty projektu musí existovat sjednocující princip re-use a žádný artefakt zavedených postupů se nesmí opakovat. Všechny opakující se situace musejí být vystiženy tak, aby byly dány odkazem a nikoliv opakováním „stejných elementů“ v jednotlivých částech dokumentace znovu a znovu. Použije se metoda vytknutí a

odkazu vždy, když se něco (cokoliv) má opakovat. Důsledkem toho je například zavádění metodik (opakující se postupy), vzory apod. Například „obrázek 2 Postup mapování v EFEM je dán katalogem vzorů“ ukazuje nejenom princip mapování z analýzy do designu, ale vyjadřuje také opětovnou použitelnost při použití katalogu mapování odkazem jako obdoby číselníku všech mapování.

Protože se jedny prvky dokumentace vždy odkazují na určité jiné prvky dokumentace, tak se celkový systém dokumentace pro tvůrce velmi zjednoduší. Například u zmíněného obrázku (obrázek 2 Postup mapování v EFEM je dán katalogem vzorů) autor nějakého konkrétního mapování do designu v konkrétním projektu nedokumentuje samotný postup mapování, ale tento postup vybírá jako vzor a odkazuje se něj jako na již hotový postup. Sama dokumentace mapování v konkrétním projektu je tak velmi jednoduchá, protože používá metodu odkazu. Postup mapování je uložen „někde jinde“ a to v katalogu všech postupů mapování. Designér pouze napíše: „mapování podle tohoto vzoru ..., výsledek viz tento model“. Dokumentace projektu se tak výrazně zjednoduší, zpřehlední a zeštíhlí.

3.3 Objektově orientovaný přístup a jeho použití v EFEM

3.3.1. Chyba ztráty identity prvku

Jedna z velmi častých chyb, která vzniká jak při modelování, tak při tvorbě informačního systému, ale také při tvorbě metodik, je chyba ztráty identity prvku.

Podstata této chyby je následující:

Na obrázku „obrázek 3 Situace vedoucí k opětovné použitelnosti (re-use)“ je opakující se část namalována takovým způsobem, aby bylo zřetelné, jak se část C opakuje v obou prvcích A a B. Avšak takto namalovaný obrázek je třeba ze zásady vidět jinak a přesněji.

Protože opakující se část C není identifikována mimo prvky A a B, tak část C jako taková vlastně ani neexistuje a na obrázku C nemá smysl o něm ani takto (jako o C) hovořit. Proto je namalována jako neobvyklý klikyhák. Platí totiž jednoduchá skutečnost, že to, co není v modelu zavedeno jako prvek modelu, tak neexistuje a „nelze o tom hovořit“. Je zřejmé, že se nelze odkázat na něco, co je „pouze nějak uvnitř a není identifikováno jako samostatný prvek“. Dokonce se dá říci, že se jedná o klasický protimluv: Něco neexistuje a přesto se o tom hovoří (viz věta opakující se C). Tato úvaha na první pohled vypadá jako „silně teoretická“, ale ze zkušeností vyplývá, že její nedocenění vede k nejhrubším chybám v modelování.

Velmi podobná a velmi názorná je následující situace:

Každý programátor ví, že nelze zavolat „kus kódu“ uvnitř jiné funkce. Jedna funkce je jako jeden identifikovatelný celek. Jedná se o jeden prvek i se svým celým vnitřkem a svým obalem. Pokud chceme změnit volání na zmíněný „kus vnitřku“, musíme tento vnitřek nějak identifikovat (doslova odtud potud) a tedy odlišit tuto část od jiných částí vnitřku. V případě funkcí se toto oddělení a identifikace učiní tak, že se definuje nová funkce, která bude obsahovat tento kus kódu, vytkne se mimo původní funkci a v bodě vytržení se zavolá přes vnější obal funkce (interface funkce).

Programátorům se může jevit tento postup jako triviální, ale tento princip funguje mnohem více obecněji, jenom namísto funkcí je třeba si představit jakýkoliv obecný prvek a místo volání funkcí obecnou interakci mezi prvky.

Tímto přirovnáním s funkcemi s jejím nevolatelným vnitřkem lze pochopit často opakující se chybu ztráty identity prvku. Prvek, který chceme nějak použít a je o něm řeč, musí být identifikován (tj. zaveden, definován) jako samostatný prvek a musí být následně použit pomocí nějaké interakce s jiným prvkem. Nesmí (a ani nemůže) být schován jako neidentifikovaný a nedefinovaný prvek. Tento až primitivní fakt, který je velmi dobře pochopitelný při volání funkcí, si mnohdy tvůrci informačního systému neuvědomují při tvorbě IS v rovině analytického modelování. Obranou proti této chybě je vyčleňovat všechny prvky modelu, které mají svůj nárok na život (tj. o kterých je řeč) a zavádět je do modelů a poté je interakcemi provazovat s jinými prvky. Tím vznikají požadované struktury prvků modelů.

3.3.2. Vnější a vnitřní pohled na prvek modelu

S otázkou opětovné použitelnosti a následně s pojmem interakce souvisí další velmi důležitý pojem pro modelování a tím je vnější a vnitřní pohled na prvek modelu.

Příklad použití interakce ukazuje „obrázek 4 Situace zavádějící opětovnou použitelnost“. Prvek A interaguje s prvkem C, podobně také prvek B interaguje s prvkem C. Pokud někdo hovoří o prvku A jako takovém, má tím automaticky na mysli tento prvek i s jeho vnitřní interakcí na prvek C. Tomu odpovídá jakýsi „sumarizující“ vnější pohled na prvek A. V tomto vnějším pohledu je interakce zkoumaného prvku A s prvkem C nezajímavá, hovoří se pouze o prvku A.

Stejně tak je tomu u funkcí, což je zvláštní případ těchto našich obecnějších úvah. Pokud programátor volá funkci A a tato funkce uvnitř sebe volá funkci C, tak z hlediska vnějšího pohledu na funkci A je toto vnitřní volání pro použití vnější funkce absolutně nezajímavé. Vnější pohled na funkci znamená identifikace její existence mezi ostatními funkcemi s možností ji zavolat (interface funkce) a nic víc.

Podobně je tomu s libovolným prvkem modelu. Existuje vnější pohled na libovolný prvek modelu. Tento pohled vyjadřuje jedinou skutečnost: „To je tento identifikovaný

prvek, na který lze ukázat, tj. který lze použít“. V tomto pohledu vůbec není obsažena vnitřní struktura prvku, která je reprezentovaná vnitřními interakcemi tohoto prvku.

Druhý pohled na prvek modelu je vnitřní pohled, což je pohled na interakce tohoto prvku s jinými prvky. Tento pohled je však až za hranicí vnějšího pohledu na prvek, tj. je „uvnitř“ tohoto prvku. To je druhý, vnitřní pohled na prvek modelu, na jeho skladbu.

Přitom platí jakýsi rekurzivní mechanismus: Z hlediska interakce tohoto prvku s dalšími prvky nahlíží daný prvek na tyto další prvky vnějším pohledem a interakce jeho vnitřních prvků je pro něj opět skryta.

Platí velmi důležitá a praktická rada pro modelování v UML vyplývající z praktických zkušeností: Je třeba vždy tyto dva pohledy na prvek modelu, tj. vnější a vnitřní pohled, vždy od sebe důsledně oddělovat. Buďto se hovoří o prvku A z hlediska vnějšího pohledu, tj. jako o prvku ve smyslu celku, anebo se hovoří o prvku A z hlediska jeho vnitřní struktury, nikdy však současně.

3.3.3. Principy objektově orientovaného přístupu

Programátoři se většinou seznamují s principy tak zvaného *objektově orientovaného programování*“, tj. OOP. Jedná se o principy zavedení objektů v programování, například v jazycích JAVA, C#, DELPHI apod. V programování se používají základní pojmy objektového programování jako jsou objekt, instance, třída, interface, dědění apod.

Avšak existuje obecnější pojetí objektů než je objektově orientované programování, a tím je *objektově orientovaný přístup* (dále také OOAP jako *Object Oriented Approach*). Tyto dva pojmy (objektově orientované programování a objektově orientovaný přístup) je třeba od sebe odlišit, i když spolu úzce souvisejí. Jejich vztah je takový, že objektově orientované programování spadá pod obecnější a pod rozsáhlejší pojem objektově orientovaný přístup a chápe se jako jeho zvláštní případ aplikace tohoto přístupu na programování.

Existuje několikero možných prostředí, kterým se říká *objektově orientované*. V každém z takových prostředí má pojem „objekt“ jiný význam. Lze ale vypozorovat určité společné rysy tohoto pojmu „objekt“. Z hlediska objektově orientovaného přístupu lze vysledovat společné vlastnosti objektů ve všech OOAP prostředích:

- je zaveden vnější a vnitřní pohled na prvky. Toto rozdělení má povahu zapouzdření s těmito dvěmi základními vlastnostmi, které činí objekt objektem:
    - Vnější pohled na objekt (tj. užití objektu zvně) nemá přímo zpřístupněnu vnitřní strukturu objektu. Ukázání na objekt v některé z interakcí je ukázání na reprezentaci vnějšího pohledu, tj. na reprezentaci obalu tohoto prvku, tj. veřejného rozhraní. Obecný objekt může být požádán zvně o nějakou službu tohoto rozhraní, přičemž

implementace prvku je při vnějším pohledu, tj. při této žádosti o službu, skryta. Současně s ukázáním na veřejné rozhraní konkrétního objektu , tj. na vnější reprezentaci služeb konkrétního prvku, je tento prvek pochopitelně jednoznačně identifikován oproti jiným prvkům.

  - Prvek definovaný ve své vnitřní struktuře neví „kdo, kdy a jak“ jej použije, tj. kam a do jaké interakce bude prvek dosazen a kým bude o službu požádán. Vnitřní pohled neví nic o konkrétním vnějším použití (pohledu zvně), tj. o použití prvku v interakci. Říká se, že použití prvku je z hlediska vnitřního pohledu anonymní. Pro uživatele prvku se zavádí obecný název „klient prvku“ (to je ten, který vstupuje do interakce s prvkem zvně a používá rozhraní). Klient zůstává z hlediska vnitřního pohledu anonymní. Hovoříme takto ekvivalentně o anonymitě klienta objektu.

- existuje tzv. dichotomie (rozdělení prvků do dvou skupin) na druh a instance druhu. Vlastnosti prvků nejsou definovány přímo ve výskytech prvků, ale pomocí druhu prvků, kam prvek patří. Daný výskyt z druhu nemá definovány vlastnosti sám o sobě, ale má své vlastnosti dány díky příslušnosti k druhu, kde jsou vlastnosti definovány podobně jako v biologii. Druh se nazývá třída, výskyt se nazývá instance třídy. Je zajímavé, že samy druhy podléhají opět principům OOAP, tj. jsou chápány jako objekty. Lze tedy opět rozlišit vnější a vnitřní pohled na druh jako objekt (první dvě vlastnosti objektů) a také lze zavést dichotomii u druhu, tj. druh druhu. Jinak řečeno platí vlastnost dichotomie o existenci druhu na samotný druh. Znamená to, že zkoumaný druh výskytů lze chápat opět jako výskyt nějakého vyššího druhu (vztah meta). Rekurzivní přechod „meta“ nahoru k druhu druhu, dále k druhu druhu druhu atd. je sice teoreticky neuzavřený, ale zastavuje se na té úrovni, kdy je přechod k dalšímu druhu z hlediska řešení konkrétního problému dále nezajímavý a je tedy prakticky zbytečný.

Uvedené vlastnosti jsou v přímém vztahu k principu maximální opětovné použitelnosti. První vlastnost zavádí možnost, aby N klientů mohlo bez kolizí použít tentýž výskyt (ukázat si na něj). Navíc se umožňuje, aby použitý prvek rekurzivně používal jiné výskyty jako klient dalších výskytů (použití vnitřní struktury podléhá opět principům OOAP). Třetí vlastnost umožňuje neopakovaně definovat výskyty stejných vlastností v jednom místě, tj. ve třídě. Současně je třeba upozornit na jednoznačnou identifikaci prvku jako identifikaci jeho vnějšího obalu a nikoliv jeho vnitřní struktury.

Následuje omezený výčet těch objektově orientovaných prostředí, která splňují uvedené vlastnosti OOAP a přitom jsou pro modelování a programování v EFEM důležitými. Současně je stručně vysvětleno, co je v tomto prostředí chápáno jako objekt a co jako třída, co je vnější a vnitřní pohled.

3.3.3.1 Objektově orientované programování

Zde jsou objekty přímo naprogramované struktury v programu jako instance tříd s interfacem. Vnější a vnitřní pohled je realizován pomocí zapouzdření se základními pojmy „interface objektu a implementace objektu“. Existuje klientský pohled na objekt (klient tj. uživatel objektu, vidí pouze interface) a existuje implementační pohled na vnitřní strukturu objektu (existuje konkrétní vyplnění kódu za interfacem). Narušení zapouzdření znamená, že klient může pracovat s vnitřní strukturou objektu, aniž by použil interface (například public atribut apod.). Pravidla tohoto OOAP prostředí jsou dána syntaxí daného objektově orientovaného jazyka.

3.3.3.2 Analytické modelování

Zde jsou objekty chápány jako výskyty evidovaných informací v informačním systému. Hranice výskytů informací jsou přesně dány pojmenováním těchto výskytů (tato evidovaná fyzická osoba, tato faktura). Vnitřní struktura odpovídá interakcím mezi informacemi (rodné číslo fyzické osoby jako její atribut, dodavatel faktury jako běžná asociace apod.). Vlastnosti těchto výskytů se definují ve třídách informací, čímž se zavádějí pojmy v AM. Pravidla pro práci v tomto OOAP prostředí jsou dána syntaxí UML.

3.3.3.3 Modelování podniku

Zde jsou objekty chápány jako prvky modelu podniku (pracovník, oběžník, oddělení, služba, proces zpracování XY apod.). Pravidla pro modelování tohoto OOAP prostředí jsou dána syntaxí UML (ale je možné použít i jiný modelovací jazyk).

3.3.3.4 Syntaxe UML

Sama pravidla modelování, tj. prostředí práce s prvky modelu v UML, lze také chápat jako objektově orientované prostředí, protože zavedené prvky modelu mají samy o sobě charakter objektů. Lze u nich určit taková pravidla pro práci s objekty, jako jsou například: jak je možné prvky modelu instanciovat a z jakých možných tříd, jaké mají prvky mezi sebou povolené vztahy, jakou kardinalitu, kdo koho může použít, kdo je čí atribut, jakou mají asociaci apod. Pravidla pro práci v tomto OOAP prostředí (pozn.: toto prostředí se nazývá meta-model UML) jsou dána opět syntaxí UML, tj. UML je schopno díky této vlastnosti popsat svá syntaktická pravidla pomocí UML.

3.3.3.5 Objektově orientovaná lidská mysl a abstrakce

Není bez zajímavosti, že naše úvahy a myšlení podléhají těmto principům, aniž bychom si to uvědomovali. Vymezení pojmů a jejich instancí splňují logicky tyto principy. Pokud pojmenujeme nějakou konkrétní věc, tak o této věci hovoříme z hlediska vnějšího pohledu automaticky jako o „celku". Například ukážeme: „Tady stojí moje auto..." Máme tím na mysli pod jedním „prstem - ukazatelem" jakýsi „sumarizující" pohled, protože při tomto ukázání na tuto věc máme na mysli automaticky „moje auto se vším všudy..." Jedním ukázáním na tuto věc jsme vymezili všechny hranice tohoto pojmu a současně jej tak i „pojmově zapouzdřili" (moje auto se vším všudy). Na druhou stranu pokud začneme zkoumat „vnitřní interakce " tohoto auta, můžeme nalézt spoustu dalších interních věcí, jako jsou jeho vnitřní součástky, jeho zaplacená pojistka a jiné konkrétní pojmy. Navíc tyto konkrétní výskyty jsou instancemi tříd, tj. obecných pojmů (auto jako pojem a toto konkrétní auto).

3.4 Praktické důsledky použití OOAP pro řízení projektů a obecná doporučení pro vedoucího projektu

Pro zavádění EFEM mají uvedené víceméně teoretické skutečnosti ohledně OOAP tyto praktické důsledky:

3.4.1.1 Princip řízení projektu tvorbou dokumentů

Projektové řízení musí podléhat principu *„driven by documents".* Znamená to, že pracovník je vždy řízen tak, že se po něm nevyžaduje vykonání nějaké činnost, ale odevzdání určitého dokumentu. Činnost je teprve odvozena od tohoto požadavku a pro samotné řízení není zadání činnosti rozhodující. Je zřejmé, že pro vyslovení takového požadavku ve smyslu tvorby a vydání dokumentu musí být definována podoba tohoto dokumentu, jeho forma a jeho předpokládaný obsah. Pracovník je takto řízen jako „objekt vydávající dokument".

Dokumenty mají povahu existujících objektů: zapouzdřují znalosti a výsledky práce na tvorbě SW do konkrétní podoby. Vyžadování dokumentů a kontrolu jejich chodu má v rámci tvorby SW na starosti pracovník v roli vedoucího projektu.

3.4.1.2 Princip zavedení rolí

U každého dokumentu se podle jeho povahy určuje ***role***, která je odpovědna za jeho vyhotovení. Určený pracovník vstupuje do této role a plní tak službu této role. Jeden pracovník tak může vystupovat v několika rolích. Vždy však v konkrétní situaci plní jednu službu vydání dokumentu, tj. hraje právě jednu roli a žádnou jinou. Znamená to, že pracovník v dané konkrétní roli vydává konkrétní dokument bez ohledu na jeho další pozice v jiných rolích. Pokud pracovník figuruje v několika rolích, tak se doslova vyžaduje jeho „schizofrenní" přepínání mezi rolemi („výměna interfaců"). Velmi častou chybou při řízení projektu je zanedbání tohoto jednoduchého principu. Pokud pracovník pracuje současně v několika rolích, může provést optimalizaci chodu jako vnitřní předávku dokumentů uvnitř sám sebe, tj. uvnitř svého prostoru objektu, a nikoliv přes rozhraní služby vydávající dokument ven. Velmi častou situací je případ, kdy daný pracovník sám problém analyzuje, sám jej navrhne v designu a sám jej naprogramuje. Výsledkem je řízení projektu metodou „TUNEL" a následně také velmi slabá dokumentace projektu. Dodržování principu „v dané chvíli jedna role" má na starosti pracovník v roli vedoucího projektu.

3.4.1.3 Princip jedné hlavy

Může existovat pouze jeden objekt, tj. jeden pracovník, jedna role jako instance, která v konečném důsledku vydává požadovaný dokument. Principielně neexistuje možnost vydat dokument společně, tj. týmově. Pokud se vzhledem k rozsahu projektu vyžaduje týmová práce, tak musí být stanovena role „vedoucí v dané roli" (například hlavní analytik apod.), který nakonec dokument vydává. Používá k tomu pracovníky v podřízených rolích. Je třeba zdůraznit, že tento pracovník v řídící roli uplatňuje vůči svým pracovníkům v podřízených rolích všechny principy řízení zde již uvedené, tj. musí mít možnost tyto principy vůči svým dočasným podřízeným aplikovat. Vedoucí tedy po svých podřízených opět vyžaduje dokumenty a kompletuje odevzdávaný dokument.

3.4.1.4 Princip opětovné použitelnosti pro pravidla tvorby dokumentů

Při vyžadování dokumentu od pracovníka v dané roli se společné rysy tohoto požadavku nespecifikují pokaždé znovu, ale centralizují se. Na tyto centralizované požadavky se lze při vyslovení žádosti o tvorbu dokumentu odkázat a následně použít. Těmito společnými požadavky jsou vlastně postupky a návodky pro tvorbu dokumentů určitého typu. Těmito postupkami jsou dána formální pravidla a je stanovena jednotná syntaxe pro tvorbu dokumentů. Tímto postupem se dosahuje jednak zkrácení vyslovení požadavku na dokument (při zadání práce stačí pouhý odkaz na postupku), současně je dosaženo požadované jednotnosti dokumentace.

Tvůrce dokumentu se při znalosti těchto postupek soustřeďuje pouze na obsah dokumentů, formální podoba a formální nekreativní postup je dán postupkami. Součástí produktů řady EFEM jsou postupky pro efektivní modelování pomocí syntaxe UML pro CASE nástroj EA.

3.4.1.5 Princip opětovné použitelnosti mezi dokumenty

Existují vztahy vzájemné použitelnosti a tedy odvoditelnosti mezi vzniklými dokumenty. Tato zásada určuje povinnost náhledu na vztahy mezi vznikajícími dokumenty z hlediska jejich vzájemných použití. Každý výskyt informace v těchto dokumentech se objeví pouze jednou a neodporuje tak principu maximální opětovné použitelnosti.

4. Použití a syntaxe CLASS MODELU v EFEM ve fázi analytického modelování

4.1 Principiální úvahy tvorby modelu tříd ve fázi analytického modelování

Základní myšlenkou tvorby modelu tříd ve fázi analytického modelování je skutečnost, že věci, které se nakonec programují, lze na úrovni AM pojmenovat. Současně je možné na úrovni AM vysvětlit struktury a interakce těchto pojmů. Takovými pojmy mohou být například faktura, osoba, běžný účet, směnka atd., které se „evidují“, tj. se kterými se v informačním systému pracuje. Díky tomuto přesnému popisu poté designér navrhuje prvky SW (např. model ERD a třídy v OOP). Popis tříd ve fázi AM je přesný díky přesné syntaxi UML v CLASS MODELU.

4.2 Úroveň informace jako typ a výskyt, úroveň meta a pojem třída v AM

Z hlediska popisu informace se při modelování analytik může pohybovat ve dvou možných rovinách:

- buď se v AM popisuje informace (pojem) v obecnosti a udává vlastnosti takto zavedených informací, tj. druhů,
- anebo se v AM popisuje konkrétní výskyt informace.

V prvém případě se popisuje typ informace a v druhém případě se popisuje výskyt tohoto typu. Podobně se vyjadřuje nejenom analytik při modelování, ale každý, kdo se podílí na analytickém modelování, protože tento přístup odpovídá normálnímu lidskému uvažování (viz objektově orientovaný přístup).

Typ informace budeme nadále nazývat třídou AM resp. třídou informace nebo typem informace. Nelze tuto třídu zaměňovat s pojmem třída v OOP, té odpovídá prvek třída v designu, tj. konkrétní třída v objektově orientovaném jazyce. Třída v AM je pojem, se kterým se pracuje v IS, třída v OOP je prvkem kódu v daném jazyce (doslova `class` jako rezervované slovo).

Výskyt z třídy AM budeme nazývat instance informace nebo výskyt informace, někdy také objekt (opět nelze zaměňovat s naprogramovaným objektem v OOP). Z hlediska předešlého výkladu můžeme typ informace neboli třídu v AM považovat za obdobu zavedeného pojmu a výskyt za evidované výskyty těchto pojmů.

Například část rozhovoru analytika se zákazníkem - odběratelem informačního systému v bance může vypadat takto:

Zákazník: „Budou se evidovat běžné účty. Každý účet má tyto vlastnosti...(následuje výčet vlastností)“

Analytik: „Kolik tak řádově předpokládáte, že by tak asi mělo těchto účtů v evidenci být, samozřejmě plus minus...“

Zákazník: „Řádově tak okolo 800 000, možná víc, možná méně...“

Všimněme si, že se jedná se o klasický rozhovor na úrovni AM. Nehovoří se o tabulkách resp. konkrétních prvcích programu, ale oba se vyjadřují pouze v pojmech tak, jak vyžaduje tato abstraktní úroveň. V první části rozhovoru se zákazník vyjádřil o vlastnostech účtů jako takových a hovořil tedy o třídě běžných účtů jako o vlastnostech druhu evidované informace, ze kterých budou pocházet instance. Ve druhé části rozhovoru, kde se „počítaly“ instance, se hovořilo o výskytech této informace, tj. o počtu výskytů běžných účtů.

Ve většině případů je třída informací tzv. multiinstanční, tj. z daného druhu typu informace lze vytvořit obecně N výskytů informace. Pouze výjimečně existují typy informací, které mají pouze jeden výskyt. Pro tyto situace se použije vzor zvaný SINGLETON (viz přiložená e-kniha Design Patterns v OOP). Jako příklad jedno-instanční informace uveďme případ, kdy se v programu typu desktop, například v kancelářském softwaru, zadávají detaily uživatele tohoto softwaru.

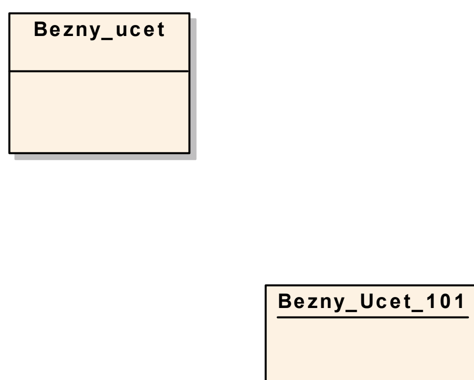
Protože většinou je třída multiinstančí, je vhodné, aby existovala ve firmě dohoda, že pokud není uvedeno jinak, třída informace se chápe automaticky jako multiinstanční.

Syntaxe UML dovoluje modelovat informační systém jak na úrovni tříd informace, tak na úrovni instancí informace. Protože vlastnosti výskytů informace jsou dány vlastnostmi tříd, tak modely, které popisují instance, jsou odvoditelné procesem instanciování od modelů, které popisují třídy.

Z toho důvodu jsou modely tříd cílem modelování a modely instancí lze chápat pouze jako příklady, jaké instance by mohly existovat v evidenci v již běžícím systému.

UML rozlišuje model tříd a model instancí pomocí podtržení názvu daného elementu. V případě, že je název podtržen, vztahuje se k instanci (popisuje instanci), pokud není podtržen, vztahuje se ke třídě (popisuje třídu).

Příklad:



obrázek 5 Třída běžný účet a výskyt konkrétního běžného účtu, u druhého je název podtržen

Jedním z hlavních cílů analytika ve fázi analytického modelování je nalézt odpovídající model tříd, tj. CLASS MODEL AM. Tento analytický model tříd se stává výchozím pro mapování tříd do designu a na základě něj se vytváří odpovídající struktury programu, např. návrh tabulek v relační databázi (dále také RDB) a návrh tříd v OOP. Po zhotovení programu tyto struktury odpovídají přesně takto navrženým typům informací analytického modelu tříd. Existuje několik možných cest mapování, jak realizovat daný analytický model. Z nich jsou vybírány konkrétně určitá řešení mapování.

4.3 Syntaxe modelu tříd analytického modelování

Analytik má pro tvorbu CLASS MODELU AM díky syntaxi UML k dispozici několik málo pravidel. Díky této skutečnosti se modely tříd AM stávají přehlednými, jednoznačnými a syntakticky přesnými.

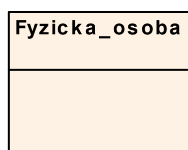
Následuje výčet pravidel, která vedou k efektivnímu modelování, tj. jsou nezbytně používaná v EFEM spolu s náležitým vysvětlením a příklady.

Poznámka: Z důvodu přehlednosti vysvětlujeme jednotlivé interakce v určitém „pedagogickém pořadí“ tak, aby byly co nejvíce přístupné efektivnímu analytickému myšlení a tedy aby byly maximálně „schůdné“ při zavedení EFEM. K tomuto pořadí mne vede dlouholetá zkušenost konzultanta. Na konci kapitoly uvedeme probranou látku do celkového souladu rozčleněním z hlediska přesné syntaxe UML.

4.3.1. Zavedení třídy v analytickém modelu tříd

Základní syntaxí modelu tříd je samotné zavedení pojmu tj. třídy v AM. Analytik tímto definuje hranice pojmu tj. typu informace. Základním posláním třídy jako typu informace je zavést předpis (neboli vlastnosti) pro instance, tj. výskyty informace z této třídy.

V daném CASE nástroji, například EA, se zavede nová třída informace s odpovídajícím názvem, například takto:



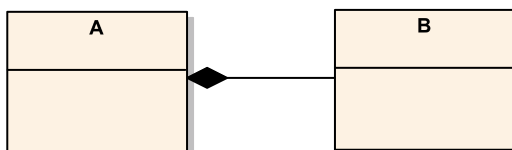
obrázek 6 Zavedení třídy Fyzická osoba

Uveďme nyní několik důležitých pravidel:

- Jedná se o třídu ve smyslu abstrakce jako typ informace. Analytik požaduje, aby na základě této třídy vznikly mapováním do designu (nějaké) struktury programu nesoucí možnosti evidované osoby, mapování do designu může být např. na tabulky v relační databázi a současně třídy v OOP. Tyto struktury v programu budou mít po naprogramování přesně takové vlastnosti, jak je v tomto analytickém modelu určil analytik.
- Protože není řečeno jinak, třída je multiinstanční. Uvnitř systému tedy budou existovat evidované fyzické osoby jako konkrétní výskyty informací osob s vlastnostmi, které budou definovány v této třídě.
- Třída vymezuje hranice pojmu. Kdo použije tuto třídu například ve svém (dalším) modelu (vnější pohled), použije ji se vším, co s ní bude spojeno (vnitřní pohled).
- Zavedená třída se bude mapovat na odpovídající struktury programu v designu podle pravidel mapování.

4.3.2. Kompozice

Kompozice je zvláštní případ interakce mezi třídami. Udává vztah od jedné třídy (např. třídy A) ke druhé třídě (např. třídě B) a značí se spojnicí s černým kosočtvercem takto:



obrázek 7 Znázorněný vztah kompozice

Vztah kompozice se chápe jako vztah celek versus část a udává vztah pro budoucí instance z těchto tříd, na obrázku instance ze třídy A vůči instanci ze třídy B. Instance informace ze třídy A bude v sobě obsahovat instanci informace (resp. několik instancí) ze třídy B jako svou část. Černý kosočtverec označuje celek. Toto obsažení celek versus část je natolik silné, že výskyt informace A ovládá život výskytu informace B jako nadřízený výskyt informace a to vždy a všude. Výskyt z B bude doslova částí výskytu z A. Zrod a zánik výskytu B je pouze jako část výskytu z A a nikdy jinak.

Existuje-li například scénář vymazání evidované instance z A ze systému (byla například omylem zadána), potom instance z A s sebou bere do záhrobí vždy i svou instanci (resp. instance) z B a to vždy bez výjimky. Výskyt z B jako část instance A žije u svého majitele, své instance z A.

4.3.2.1 Multiplicita

Pro kompozici lze udat další informace spojené s tímto vztahem. Prvním z nich je tzv. multiplicita (označovaná také jako násobnost), která se udává na straně třídy B. Multiplicita udává, kolik je možných výskytů z B vůči výskytu A. Vícenásobná multiplicita jako N výskytů se označuje hvězdičkou. Možné hodnoty této multiplicity jsou:

- 0..1 je povolena buď žádná instance nebo jedna instance ze třídy B v majitelství instance ze třídy A
- 1 je povolena právě jedna instance ze třídy B v majitelství instance ze třídy A
- * je povoleno N instancí ze třídy B v majitelství instance ze třídy A

- 0.. * je povolena žádná až N instancí ze třídy B v majitelství instance ze třídy A
- 1..* je povolena jedna až N instancí ze třídy B v majitelství instance ze třídy A
- a jiné kombinace.

Všude, kde se vyskytuje označení pomocí hvězdičky, tj. N výskytů, se instance z B chápou jako seznam. Tento seznam je při kompozici obsažen v instanci z A. Každá z instancí v tomto seznamu je ve svém životě řízena instancí ze třídy A.

4.3.2.2 Jednosměrné a obousměrné vztahy

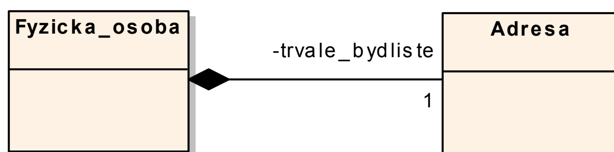
Pro správné modelování je přínosnější, pokud se vztahy mezi prvky chápou jako jednostranné. Pokud je vztah oboustranný, tak se jedná o dva jednostranné vztahy, i když je v syntaxi UML znázorněn v diagramu pomocí jedné spojnice. Vždy se jedná o vztah z jedné strany od jedné instance vůči druhé instanci. Případ obousměrného vztahu je chápán jako dva jednosměrné vztahy, i když v UML se znázorňuje jednou spojnicí.

Technologie EFEM používá důsledně toto rozdělení interakce na dva směry. Zde ve výkladu o kompozici se tedy zatím nehovoří o obráceném vztahu od instance B k instanci A, ale pouze o vztahu od A k B (k obrácenému vztahu se dojde dalším výkladem).

4.3.2.3 Role

Dalším údajem, který lze doplnit ke vztahu kompozice, je tzv. role třídy B vůči třídě A. Zadání role udává, v jaké roli vidí instance z A instanci z B. Je dobré si nyní uvědomit, že jedna a tatáž třída může být použita v různých interakcích a tedy může „hrát" různou roli. Role se značí v diagramu názvem této role u třídy B. Role označuje význam instance resp. instancí z třídy B vůči A. Prakticky nejefektivnějším a nejčastěji používaným postupem je zavést jako roli přímo název této instance (resp. název seznamu těchto instancí v případě N výskytů z B). Název této instance (resp. seznamu instancí) totiž logicky vystihuje tuto roli.

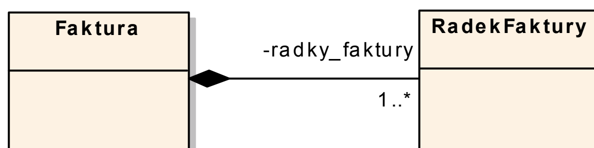
Význam role lze vysvětlit na příkladu s bydlištěm. Zavede se třída adresa (tato třída vytváří instance obsahující ulici, město a PSČ). Potom se k fyzické osobě může přiřadit adresa trvalého bydliště. Zvolí se jako řešení kompozice, tj. výskyt fyzické osoby bude obsahovat jako svoji kompozici výskyt adresy v roli trvalého bydliště. Tato věta se napíše graficky v UML takto:



obrázek 8 Příklad na roli

V tomto případě instance trvalého bydliště žije v řízení fyzické osoby, tj. pokud se například nějaká fyzická osoba z IS vymaže, potom nemůže její adresa trvalého bydliště zůstat „viset“ v systému a odchází také s danou fyzickou osobou.

Podobně u faktury obsahující řádky faktury se může zavést role takto:



obrázek 9 Role radky_faktury označuje význam třídy - seznam řádků faktury

V tomto případě je role názvem celého seznamu řádků obsaženého ve faktuře.

4.3.3. Atribut

Existuje zvláštní případ vztahu mezi informacemi, který sice svou povahou odpovídá kompozici s multiplicitou ku 1, ale značí se v UML jinak. Jedná se o tzv. atribut. Je to takový vztah informace, ve kterém instance ze třídy A obsahuje instanci z typu B stejně jako u kompozice ku 1, ale typ B je považován za tzv. ***obecně známý primitivní datový typ***. V tom případě se B zavádí nikoliv jako kompozice tříd, ale jako atribut ve třídě A. Jinak vše ostatní, co bylo řečeno o kompozici, platí i pro atribut. Z tohoto hlediska lze atribut chápat jako jednodušší zápis syntaxe kompozice ku 1 u jednoduchých typů informace.

Technologie EFEM doporučuje zavést v analytickém modelování minimálně tyto datové typy atributů : *krátké číslo, dlouhé číslo, řetězec a boolean*

Je možné tento seznam ve firmě ještě rozšířit, případně upřesnit. Je však třeba dodržet určitá pravidla. Pokud se zavádí informace jako atribut, tak:

- daný typ informace nesmí být poplatný danému řešení problémové domény, typ atributu je opravdu obecně známým, všude použitelným datovým typem

- násobnost vazby k atributu je 1. Pokud je násobnost ku N, tak v tom případě má kompozit právo na svou vlastní třídu. UML sice syntaxi „násobných atributů“ dovoluje, ale z hlediska analytického modelování a následného mapování do designu by násobné atributy vedly ke kolizím. Tato možnost syntaxe násobných atributů v UML existuje pro možnost modelování obdoby datových struktur polí v designu a v analytickém modelování se nepoužívá.
- typ atributu nesmí reprezentovat informaci složenou z dalších informací (viz například třída Adresa). Technologie EFEM přísně vyžaduje zavést pro takovouto informaci třídu a nikoliv datový typ atribut.

Atribut musí být jednoduchým datovým typem (skalár).

Při dodržení těchto pravidel je mapování do designu rychlé a efektivní s možností použití vzorů. Uvedené kolize při mapování do designu by vznikly díky tomu, že v případě použití složených atributů resp. násobných atributů by tyto atributy vyžadovaly ještě navíc další krok normalizace informace.

4.3.4. Běžná asociace

Vztah běžné asociace je opět směrový vztah mezi třídami, který vede ke vztahu mezi instancemi. Většinou (více než z 99 %) se jedná o vztah od A k B v multiplicitě „ku 1“. Pokud se nalezne vztah běžné asociace ku N, tento vztah se s určitými analytickými výhodami převede na jiný výhodnější vztah, který bude blíže vysvětlen později.

Běžná asociace je vztahem, kdy instance ze třídy A může používat instanci ze třídy B, má ji k dispozici, přičemž tento vztah od instance ze třídy A k instanci ze třídy B není vztahem celek versus část jako kompozice. Naopak, instance ze třídy A dostane k dispozici již existující instanci ze třídy B, tj. instance ze třídy B není jejím majetkem. Instance ze třídy A sice může instanci ze třídy B používat, ale nemá právo s ní vůbec nějak nakládat ve smyslu ovládání (jako je tomu u kompozice), dokonce to má vysloveně zakázáno. Instance ze třídy B je pro instanci ze třídy A pouze „zapůjčena“.

Z hlediska analytického modelování existují dvě základní situace, kde se běžná asociace vyskytuje:

- vztah provázání nezávislých entit
- vztah „parent v kompozici ku N“

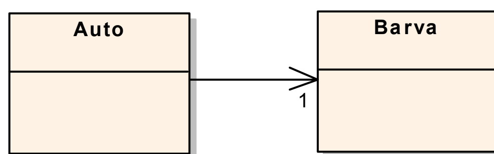
4.3.4.1 Běžná asociace jako provázání nezávislých entit

Vztah provázání nezávislých entit je také slangově (a nepřesně) nazýván „číselníková vazba“ nebo také „vazba do číselníku“. V tomto případě existuje nějaký

seznam výskytů (například číselník), ze kterého se vybere jeden výskyt a ten se provåže na aktuální výskyt, který tento výskyt z číselníku potřebuje.

Jako příklad lze uvést informační systém evidence na dopravním inspektorátu. Pro evidenci aut se zavede seznam barev. Existuje a je k dispozici nějaký (v dané chvíli konečný) seznam barev nazvaný „číselník barev“. Při editaci auta se vybere jedna z existujících barev a ta se provåže s daným výskytem auta.

V modelu tříd se tato vazba zobrazí následujícím způsobem:



obrázek 10 Běžná asociace v použití „číselníkové vazby" auto má barvu

Uvedený vztah se interpretuje tak, že výskyt ze třídy auto bude mít k dispozici jeden výskyt ze třídy barva. Výskyt auta může tento výskyt barvy používat, nesmí jej však ani zrušit, ani založit. Pokud se vymaže ze systému daný výskyt auta, potom odpovídající výskyt barvy zůstane v systému zachován. V případě vztahu kompozice je tomu přesně naopak.

Povaha vazby (běžná asociace versus kompozice) vždy ukazuje, jak se výskyty informací dynamicky chovají. Při scénáři naplnění vazby běžné asociace musí daný výskyt barvy již existovat, poté musí být nějak vybrán ze všech výskytů barvy a následně musí být v dané technologii nějak provázán na výskyt auta. Pokud daný výskyt barvy neexistuje, potom by se ve scénáři výběru muselo odskočit do jiného scénáře, což je nějaký jiný proces nad číselníkem, a tam založit nový výskyt barvy. Teprve poté jej lze provázat v běžné asociaci k autu.

Kdyby se nezvolila vazba auta na barvy jako běžná asociace, ale jako kompozice, obsluha by při zadávání nového auta mohla editovat barvu jako vždy nový „vlastní řetězec“, tj. nevybíral by ze seznamu barev, ale doslova by ji „ťukal“ znovu a znovu. Zadával by nové barvy pro každé auto, přitom by nebyl vázán žádným pravidlem výběru, pouze by barvu znovu zadal jako řetězec. Díky této libovůli by se mohly objevit barvy jako „červeno-červeno-hnědá“ nebo „fialkově-zelená“, jak si obsluha smyslí. Právě proto, aby se zamezilo této nežádoucí kreativitě a získala se jednoznačnost, použije se vztah běžné asociace slangově nazývaný „číselníková vazba“.

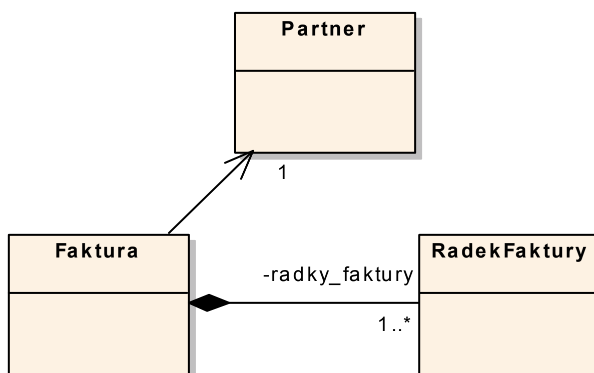
Samozřejmě je v kompetenci analytika, jaký typ vztahu bude zaveden. Tím analytik určuje chování aplikace. Od volby vztahu se odvíjí další práce designéra, který již

díky této povaze vazby takříkajíc nemůže vytvořit jiný vztah. Designér musí pouze tento typ vazby realizovat v konkrétní technologii a k tomu mu pomáhá omezený výčet možných realizací.

Uvedený příklad na barvy aut je názorný. Ukazuje, jak vlastně analytik zjistí, zda se jedná o kompozici nebo o běžnou asociaci: Položí se dotaz: Plní se vazba nějakým výběrem (například obsluhou) anebo se zadává (edituje) napřímo? Barva se vybírá, a to je signál pro zavedení běžné asociace a nikoliv kompozice.

Uvedenou „číselníkovou vazbu“ lze zobecnit pro libovolné informace, které se potřebují pouze provázat a tento vztah nemá přitom povahu kompozice. Zde se již nejedná o číselníkovou vazbu jako takovou, ale obecné provázání nezávislých entit.

Například daná faktura má svého partnera (dodavatel - odběratel). Instance partnera musí být faktuře známa a získává se výběrem, tj. není jeho kompozitem. V modelu lze tedy číst následující vztah:



obrázek 11 Běžná asociace na partnera a kompozice řádků faktury

Vztah instancí ze třídy faktura vůči partnerovi je jiné povahy než vztah vůči řádkům faktury. Svou povahou je vztah k partnerovi stejný jako vazba do číselníku. Pro daný výskyt faktury se musí výskyt partnera vybrat z již existujících výskytů. Daná faktura si na něj pouze „ukáže“ (někdy se říká pouze „vidí“), ale neobsahuje jej jako kompozici. Při vymazání faktury partner v systému zůstává. Povahou je tato vazba stejná jako „číselníková vazba“, pouze partner nemá povahu číselníku, tj. nemá povahu jednoduché entity typu vlastnost.

4.3.4.2 Vlastnost „*isNavigable*“

Vazba běžné asociace v obrázku k používané instanci je v modelu označena šipkou. Tato šipka souvisí s vlastností „*isNavigable*“ konce vztahu, tj. s již zmíněnou směrovostí, někdy se nazývá také jako propustnost. Pokud je konec vztahu označen

touto šipkou, znamená to, že ve směru šipky je možnost viditelnosti od jedné instance k druhé, avšak nikoliv naopak. Pokud není v modelu zobrazena šipka, chápe se to jako možnost viditelnosti v obou směrech.

V případě běžné asociace typu „číselníková vazba“ šipka vyjadřuje tu zřejmou skutečnost, že instance ze třídy auto má sice k dispozici instanci ze třídy barva, může ji používat a „ví o ní“, ale obráceně to neplatí. Pokud se podíváme na vnitřní strukturu informace barvy, tak tato informace neví nic o nějakých autech. V důsledku se tato vlastnost jednostranného vztahu projeví přímo v tom, že naprogramované auto potřebuje barvy, auto ve své vnitřní struktuře potřebuje třídu barev, a nikoliv naopak. Číselník barev je samostatně stojící entita.

Například při mapování do OOP se tato skutečnost projeví přímo prakticky tak, že třída `CAuto` (ve smyslu OOP) potřebuje třídu `CBarva`, avšak třída `CBarva` nepotřebuje třídu `CAuto`. Znamená to, že například třída `CBarva` může být implementována v komponentě, kterou si komponenta s třídou `CAuto` přilinkuje.

Pokud je třeba pracovat s auty pouze určité barvy, není to problém barvy jako takové. Musí se vybrat ze všech výskytů ta auta, která tuto barvu mají. Tento dotaz na auta je však problém aut a nikoliv problém barvy. Analytik tuto skutečnost „filtru“ vyjádří na úrovni výskytů aut. Poté se toto mapuje do designu. Například v technologii s relační databází se může pro takový filtr nad vazbou v designu využít nějaký SELECT nad tabulkou aut apod.

4.3.4.3 Kvalifikace vazby

V souvislosti s touto problematikou technologie EFEM zavádí jeden důležitý pojem, který se nazývá kvalifikace vazby.

Nechť instance ze třídy A používají instance ze třídy B, tj. v modelu tříd se objevila odpovídající vazba asociace mezi třídami A a B (viz „obrázek 10 Běžná asociace v použití „číselníkové vazby" auto má barvu“). Kvalifikace vazby znamená nasazení výběrové podmínky na množinu instancí ze třídy A, přičemž tato výběrová podmínka používá vazbu k instancím ze třídy B. Nejčastěji je potřebná výběrová podmínka vyjádřená formulací „všechny instance ze třídy A, které mají vztah k jedné instanci B“. Například v případě evidence aut se jedná o podmínku „všechna auta jedné dané barvy“. Nasazení takovéto výběrové podmínky využívající vazbu může být složitější a může pokračovat dále v dalších vazbách dalších instancí, jako například „všechny faktury dodavatelů z jednoho města apod.“

Analytik si (na rozdíl od designéra) může představit, že kvalifikace vazby probíhá tak, že se sekvenčně projdou všechny instance ze třídy A, posoudí se podmínka pro danou instanci a určí se výsledek „platí - neplatí“. Designér uvedenou podmínku může vyřešit například filtrem nad relacemi v RDB.

4.3.4.4 Běžná asociace jako vztah k parentovi v kompozici ku N

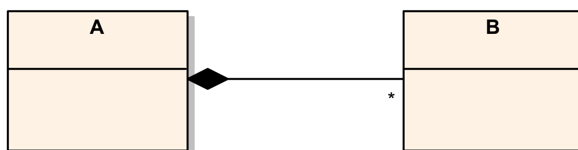
Další situace, která vede k použití běžné asociace, je zpětný (obrácený) vztah v kompozici jedna ku N. Tento vztah se programátorské hantýrce nazývá „vztah k parentovi". Jako příklad lze uvést fakturu a její řádky, viz *„obrázek 11 Běžná asociace na partnera a kompozice řádků faktury"*. V tomto případě vztah k parentovi vyjadřuje, v jakém vztahu je instance řádku faktury vůči svému majiteli, zde k instanci faktury. V tomto případě se vyžaduje, aby také instance řádku znala svou instanci faktury, protože pokud se pracuje s instancí řádku, potřebuje se používat odpovídající instance faktury.

Ukazuje se, že tento vztah k parentovi má úplně stejnou povahu a vlastnosti, jako má již uvedená vazba do číselníku a také se úplně stejně mapuje do SW technologie. Daná instance řádku faktury dostane již existující instanci svého majitele k dispozici a prováže se na ni. Výsledkem je, že instance řádku faktury „vidí" svého majitele (instanci faktury) stejně, jako když nějaká instance „vidí" prvek z číselníku.

Dá se říci, že instance faktury drží své instance řádků jako kompozici a obráceně instance řádku „vidí" svou instanci faktury jako parenta v běžné asociaci. Vazba na parenta se naplní v okamžiku zrodu instance řádku faktury a v životě instance řádku se již nemění (instance řádek neputuje nikam k jinému majiteli).

Běžná asociace se jako vazba na parenta ve vztahu kompozice ku N znázorní v CLASS MODELU pomocí syntaxe směrové vazby kompozice. Důležité je, zda se při kompozici ku N použije nebo nepoužije šipka.

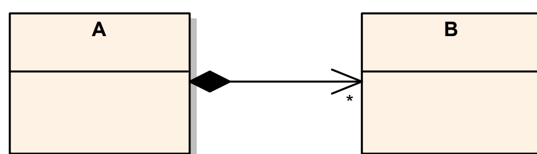
Pokud se nepoužije šipka, tj. diagram se namaluje takto:



obrázek 12 Kompozice spolu s běžnou asociaci zpět

potom autor vyjádřil skutečnost, že vztah od B k A je běžnou asociací a tedy instance z B vidí svého parenta a potřebuje jej.

Pokud se použije směrovost vazby s šipkou takto:

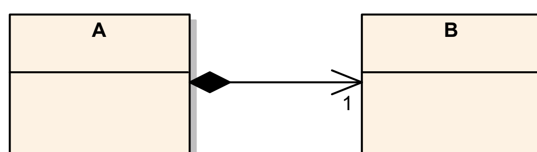


obrázek 13 Jednosměrná kompozice ku N

potom instance ze třídy B „nevidí“ instanci ze třídy A, tj. instance ze třídy B o instanci ze třídy A (o parentovi) nic neví.

Většina informačních systémů vyžaduje v kompozici existenci vztahu ve směru na parenta, tj. používá většinou model podle *„obrázek 12 Kompozice spolu s běžnou asociaci zpět“*. Výjimku tvoří technologické systémy, kdy například instance ze třídy A ovládá „shora“ svoje výskyty ze třídy B, ale přitom nikdy není třeba přecházet ve zpracování zpět k instanci ze třídy A, tj. sekvence zpracování je vždy shora dolů (rozeslání zpráv typu broadcast apod.). Klasickým příkladem takovéto situace může být implementace vzoru OBSERVER (viz odpovídající kniha Design Patterns v OOP).

Podobně u kompozice ku jedné je správnější v EFEM malovat vztah vždy takto:

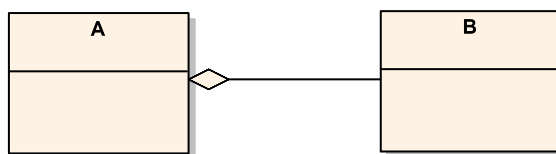


obrázek 14 Kompozice ku 1 je vždy směrová

protože kompozice ku 1 je automaticky směrován od A k B. Použitý element z B jako vložený do instance ze třídy A neví nic o třídě A.

4.3.5. Sdílená neboli slabá agregace

Vztah sdílené neboli slabé agregace (shared aggregation) se značí takto:



obrázek 15 Sdílená agregace

Vztah je podobný jako kompozice v tom smyslu, že se opět jedná o vztah celek versus část, kde kosočtverec označuje celek. Na rozdíl od kompozice však není tento vztah silný. Již neexistuje jednoznačné majitelství instance ze třídy A vůči instanci ze třídy B a mohou se vyskytovat i jiní majitelé instance ze třídy B. Důsledkem toho je, že mohou existovat takové scénáře, ve kterých se sice s instancí ze třídy A něco děje (například vymazání), ale instance ze třídy B nemusí v tomto scénáři toto pocítit. Na druhou stranu existují scénáře, kdy se daná instance ze třídy A chová vůči instanci ze třídy B jako majitel stejně jako v kompozici. Vztah implikuje chování mezi instancemi „v některém scénáři jako kompozice a v některém scénáři nikoliv“.

Dá se z toho usoudit, že rozdíl sdílené agregace oproti kompozici spočívá ve větší volnosti majitelství. Instance ze třídy A v některých scénářích působí jako majitel, v některých scénářích nikoliv (majitelem je někdo jiný).

V kompozici je identifikace vnitřního prvku dána kontextem jeho majitele a to vždy a bez výjimky. Ve sdílené neboli slabě agregaci toto pravidlo jednoznačnosti neplatí.

Z praktického hlediska se rozdíl mezi kompozicí a sdílenou agregací v určité fázi analytických prací příliš nerozlišuje, protože nemohou být známy všechny možné scénáře a tudíž všechna možná majitelství. Tento vztah je upřesňován později. Nutno ale podotknout, že kompozice bývá identifikována z povahy věci většinou mnohem dříve než sdílená agregace.

V UML spadají vztahy typu kompozice a sdílená agregace do obecnějšího pojmu agregace, tj. v UML se vztah agregace dělí na kompozici a sdílenou agregaci.

V technologii EFEM je pro modelování v AM důležitá skutečnost, že pokud se zavede v modelu AM vztah agregace mezi prvky (tj. buď sdílená agregace nebo kompozice), tak analytik dává designérovi pokyn, že instance celku (označená kosočtvercem) v programu ovládá své vnitřní členy, tj. používá je přímo svým chováním.

Důsledkem zavedení agregace je proto viditelnost prvků od majitele ke svým částem. Tato viditelnost se mapuje až do napsaného kódu, což je prakticky hmatatelný důsledek. Zavedení agregace v AM tak vede k přímému důsledku, kterým je to, že třída majitele musí bezpodmínečně mít ve viditelnosti třídu svých částí. Například vztah agregace vede při jednom určitém zvoleném mapování do databáze ke kaskádovitým operacím.

U běžné asociace typu číselníková vazba takováto viditelnost neexistuje. Pokud by s určitou mírou nepřesností byla agregace obráceně zavedena, tj. item číselníku by vlastnil ty prvky, které jej používají, vedlo by to ke změně vztahu viditelnosti a tedy ke změnám ve vztahu závislosti. V předešlém příkladu s číselníkem třída barva nepotřebuje v dosahu viditelnosti třídu aut. Třída barva není na třídě aut závislá, tj. z hlediska syntaxe UML není vůči ní ve vztahu DEPENDENCY. Pokud by však analytik navrhl, aby existovala obráceně sdílená agregace, tj. barva by obsahovala svoje auta, tak by analytik designérovi zadal úkol provázat viditelnost zpětně od barvy k „jejím autům“, tj. zavedlo by se „ovládání od barvy k autům“. Daná barva by poté znala svoje auta a chápalo by se to jako slabá agregace, tj. barva má svoje auta jako vnitřní seznam. Slabá agregace je zvolena proto, protože instanci auta lze ovládat i od jiných instancí, než je instance barvy. Důsledkem viditelnosti od barvy k autům by byl ten nepříjemný fakt, že by instance barvy v sobě obsahovaly seznam instancí aut a to až do kódu. U takovéto konstrukce je třeba být obezřetný. Pokud se jedná o vztah do číselníku, je třeba používat zásadně běžnou asociaci.

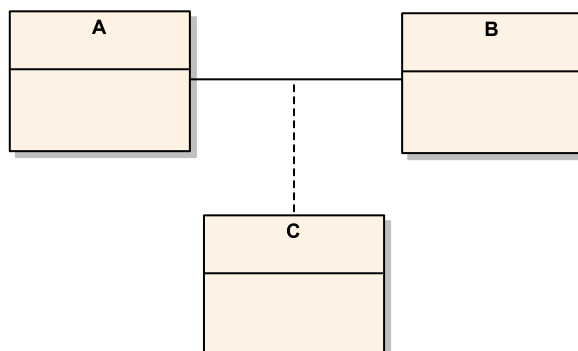
4.3.6. Asociativní třída

Při modelování v AM je mnohdy zapotřebí zavést takový typ informace, který má jednak charakter typu informace, tj. odpovídá třídě v AM podle předešlých kapitol, a přitom výskyty z této informace zprostředkovávají vztah mezi jinými výskyty informací.

Takovýto typ informace se nazývá asociativní třída (ASSOCIATION CLASS). Technologie EFEM doporučuje zavést takovýto typ informace, tj. asociativní třídu vždy, když se vyskytne alespoň jeden z těchto případů:

- Nalezne se vztah mezi výskyty M:N. Tento vztah je chápán jako „nehotový“ a podle EFEM se povinně doplní asociativní třídou.
- Nalezne se informace, která by měla být přiřazena ke každému výskytu vazby mezi informacemi, tj. existuje informace potřebující být „pověšena“ na výskyty vztahu mezi dvěma informacemi.

Na následujícím obrázku je znázorněna asociativní třída C v syntaxi UML:



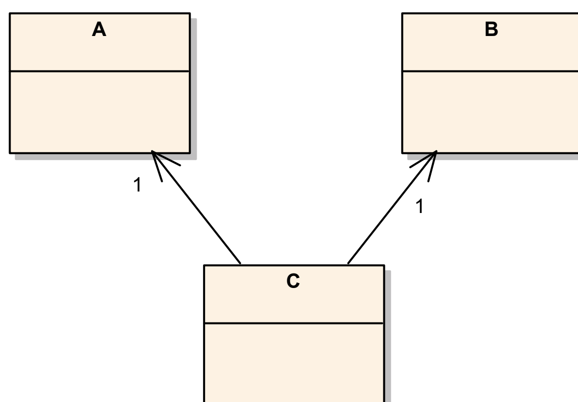
obrázek 16 Asociativní třída C

Asociativní třída implikuje vztah mezi výskyty typů A, B, C. Třída C zprostředkovává vztah mezi A a B a to tak, že:

- Každý výskyt z C používá jeden výskyt z A a jeden výskyt z B (stejně jako v číselníkové vazbě).
- Třída C je multiinstanční. Seznam výskytů z C musí mít tu vlastnost, že lze od tohoto seznamu zadáním instance A získat ze seznamu pouze ty instance z C, které tuto instanci používají a naopak symetricky, zadáním instance B tomuto seznamu z výskytů C lze získat pouze ty instance C, které tuto instanci B používají (je zavedena povinná kvalifikace vazby z obou stran).

Těmito dvěma vlastnostmi umožňují výskyty z C zprostředkovat vztah mezi A a B a to tak, že klient pro obsluhu tohoto vztahu mezi A a B používá seznam z výskytů C. Při zadání vstupní podmínky (viz kvalifikace vazby) do tohoto seznamu se vstupním parametrem rovným „instance A“ mu tento seznam poskytne ta C, která tuto instanci používají. Protože instance C používá instanci B, lze získat jak informace v C, tak informace z příslušných instancí B viděných z C.

Pokud se zavede v modelu namísto asociativní třídy taková třída, která má pouze dvě běžné asociace, tj. takto:



obrázek 17 Dvě běžné asociace versus asociativní třída

tak z hlediska struktury se jedná o ekvivalentně stejný vztah jako u asociativní třídy. Rozdíl je v dynamice chování instancí: V modelu na předešlém obrázku není řečeno, že existuje nějaký vztah od instancí A k instancím B a že se lze přes C k těmto instancím dostat. Předešlý model se dvěma běžnými asociacemi ukazuje vlastnosti instance C a nic víc. Pokud je vztah mezi A a B nalezen, potom je syntakticky přesnější převést vztah dvou běžných asociací na asociativní třídu, struktura instance typu C se přitom nezmění. Mnohdy je signálem pro nalezení asociativní třídy právě předešlý obrázek se dvěma běžnými asociacemi.

Příklad na asociativní třídu: V dopravním inspektorátu se evidují auta a majitelé. Jeden majitel může mít N aut a jedno auto má v čase N majitelů. Je splněn první signál pro zavedení asociativní třídy a tím je vztah M:N. Druhý signál je splněn také, protože údaj časového intervalu (od, do) nepatří ani do majitele a ani do auta, ale pro každý výskyt vazby (tzv. link) mezi auto a majitele. Zavede se asociativní třída Majitelství, která obsahuje atributy od a do (datumy). Ve struktuře instance Majitelství je patrná jedna instance Auta a jedna instance Majitele. Množina všech instancí „umí“ výběrové podmínky Majitelství pro dané Auto resp. Majitelství pro daného Majitele (resp. složitější podmínky) a tím zprostředkovává vazbu mezi Autem a Majitelem.

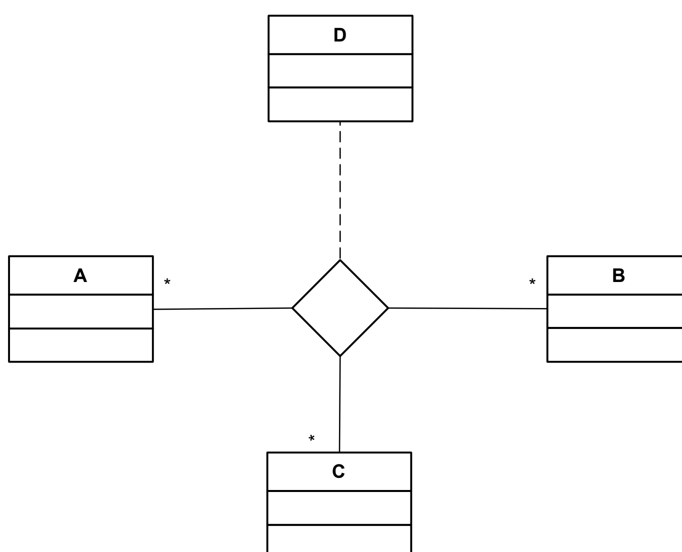
4.3.6.1 Běžná asociace ku N

V kapitole pojednávající o běžné asociaci (číselníkové vazbě) bylo uvedeno, že vztah běžné asociace se ve valné většině případů vyskytuje ve vztahu ku 1.

Pokud se nalezne vztah násobné asociace od jedné instance k N instancím (a nejedná se o agregaci), technologie EFEM doporučuje, aby se tento vztah automaticky převedl na vztah přes asociativní třídu s omezením na jedné straně ku 1. Návrh je tak otevřen k možnému a častému požadavku na rozšíření obecnějšího vztahu M ku N, což většinou bývá později požadováno.

4.3.7. Násobná asociativní třída

Násobnou asociativní třídu lze chápat jako zobecnění asociativní třídy pro více než dvě instance informace, které je třeba provázat pomocí jiných instancí informace. Zatímco asociativní třída C dala do vztahu dvě informace A a B, násobná asociace dává do vztahu vícero informací než dvě. V syntaxi se používá v asociaci zavedený kosočtverec:



obrázek 18 Zavedení násobné asociativní třídy

Třída D na předešlém obrázku je násobnou asociativní třídou, avšak nespojuje dvě třídy, ale tři (A,B, C). Význam násobné asociativní třídy je úplně stejný jako u předešle zavedené asociativní třídy, pouze počet konců asociace je vyšší. Výskyty ze třídy D mají tu vlastnost, že každý z těchto výskytů používá jednu instanci z A, jednu instanci z B a jednu instanci z C. Naplnění instance nastává výběrem ze seznamů stejně jako u běžných asociací. Vztah je opět velmi podobný jako tři běžné asociace z D do tří entit, ale navíc lze přecházet různým způsobem od jedněch instancí z jedné třídy k jiným instancím přes funkcionalitu filtrů seznamu z výskytů D. Výskyty D mohou nést také svou nějakou informaci.

Protože strukturou je výskyt násobné asociativní třídy shodný se strukturou N běžných asociací, zavádí se někdy N běžných asociací namísto násobné asociativní třídy. Pokud je popis scénářů v pořádku, nejde o závažnou chybu, ale jedná se pouze o sémantickou chybu UML.

Příklad na násobnou asociativní třídu z evidence informačního systému vysoké školy: Zkouška na vysoké škole se koná v určité místnosti, z určitého předmětu a

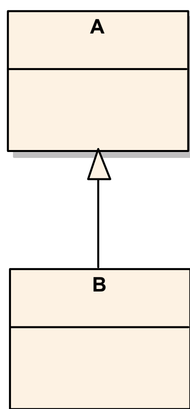
provede ji určitý učitel. Zkouška se koná k určitému datum a v určité hodině. Pro založení výskytu zkoušky je třeba vybrat existující místnost ze seznamu místností, je třeba vybrat existující předmět ze seznamu předmětů a je třeba vybrat existujícího učitele ze seznamu učitelů. Navíc je třeba zadat datum a čas. Vznikla třída Zkouška jako násobná asociativní třída spojující tři třídy: třída Místnost, třída Předmět, třída Učitel (viz A, B,C v předešlém diagramu). Zkouška má navíc atributy datum a čas.

4.3.8. Vztah generalizace specializace

Vztah generalizace a specializace (dále také GEN-SPEC) se podstatně liší od předešlých vztahů. To je také v syntaxi UML zdůrazněno tím, že předešlé vztahy se vyjadřovaly pomocí vztahu asociace mezi třídami, kdežto vztah GEN-SPEC pod tuto kategorii vztahu nespadá a stojí samostatně a zvlášť.

Vztah GEN-SPEC je opět směrovým vztahem pro opětovnou použitelnost. Na rozdíl od předešlých vztahů se nepromítá do vztahu instancí, ale stojí pouze jako interakce mezi třídami, tj. jedná se o opětovné použití pouze mezi třídami.

Jedna třída může použít druhou třídu a tak provést tak zvanou specializaci. Použitá třída se nazývá generalizující (obecnější) a ta, která používá tuto třídu, se nazývá specializací. Interakce se vyjadřuje spojnicí s trojúhelníkem s vrcholem ukazujícím směr interakce použití ke generalizující třídě:



obrázek 19 Vztah GEN-SPEC

Vztah GEN-SPEC je vztah použití mezi druhy a druhy jsou vlastnosti dávané instancím. Z toho vyplývají tyto závěry:

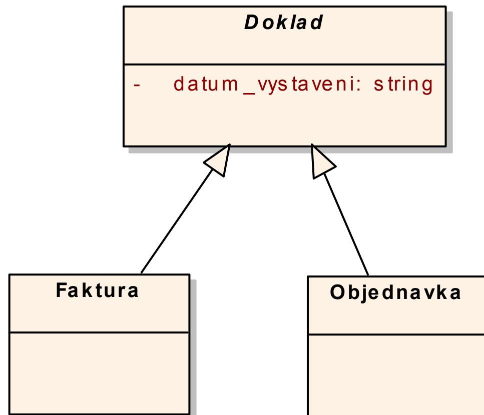
- Instance ze třídy B bude mít ty vlastnosti, které jsou definované ve třídě B a ty vlastnosti, které jsou definované také v A (a které není zakázáno interakcí „zespodu“ v GEN-SEC použít). Tato vlastnost je evidentně transitivní, tj. pokud je A specializací jiné třídy, třída B přebírá transitivně také tyto vlastnosti ještě

obecnější třídy, která by byla nad A (na obrázku není tato obecnější třída znázorněna). Hovoří se také o dědění vlastností ve vztahu GEN-SPEC, což je pojem původně speciálně zaveden v OOP ve fázi D a je převzat i do AM.

- Protože instance ze třídy B získá děděním v GEN-SPEC vlastnosti ze třídy A, tak všude, kde hraje roli třída A, může hrát tuto roli také instance ze třídy B. Hovoříme o tzv. zástupnosti rolí zespodu nahoru. Po přechodu do designu se v OOP ve statických jazycích tato vlastnost projevuje tzv. „kompatibilitou typu zespodu nahoru“.

4.3.8.1 Abstraktní třída

Protože třída může vystupovat v interakci GEN-SPEC jako použitá na straně generalizující, nemusí být povinně chápána pouze jako vzor pro tvorbu instancí. Může být také pouze vzorem pro vztah GEN-SPEC, přičemž instance informace z ní tvořené by neměly smysl. Mohou tak vznikat třídy v AM, které nebudou tvořit instance a jsou pouze zdrojem pro vztah GEN-SPEC jako třídy generalizující. Takovéto třídy se nazývají abstraktní a jejich označení jako abstraktní vede v modelu UML již ve fázi AM k označení kurzivou. Příklad:



obrázek 20 Příklad na abstraktní třídu

Vztah GEN-SPEC je čistě druhový a doporučuje se, aby byl tzv. druhově disjunktní. Daná instance spadá buď do jednoho druhu nebo do jiného druhu. Nedodržení této zásady vede k tzv. *„explosion of subclassing“*, tj. k prudkému rozmnožení počtu tříd vzniklých kombinatorickým křížením druhů, případně k nepříjemnému problému putování instance ze třídy do třídy (vyžaduje se, aby instance do určité doby byla z jedné třídy a od určité doby z jiné třídy). Řešením tohoto problému, pokud vznikne, je použití jiné interakce než GEN-SPEC, což vede k použití obdoby vzoru BRIDGE.

Tento problém a jeho řešení je podrobně rozebrán v přiložené e-knize Design Patterns v OOP v kapitole BRIDGE.

4.3.9. EFEM a syntaxe UML v CLASS MODELU

Tato kapitola dává do vzájemných vztahů názvosloví CLASS MODELU v UML a v EFEM.

V UML se v CLASS MODELU zavádějí následující možné interakce s tímto členěním:

- Interakce mezi třídami je buď generalizace-specializace anebo asociace. U asociace lze navíc zavést další vlastnosti u konců asociace (role, směrovost, agregace atd.)
- Asociace má buď právě jeden konec označen jako agregace nebo nemá žádný konec označen jako agregace. Pokud je konec označen jako agregace, považuje se tento konec za celek.
- Agregace je buď kompozice nebo sdílená agregace podle „síly“ agregace
- Lze zavést asociativní třídu, což je současně jak třída, tak asociace

Technologie EFEM vychází zásadně z této syntaxe, navíc ji však doplňuje o praktickou extenzi a to tak, že zavádí při tvorbě CLASS MODELU v AM speciální názvy odpovídající směrovosti vazby. V UML tato směrovost spadá pod syntaxi vlastnosti konce asociace „*isNavigable*“. V EFEM se některé z těchto vazeb v určitém směru navíc ještě pojmenovávají. Tímto vztahy asociace definované v UML (viz předešlý seznam) v určitém směru a určitých vlastností dostávají v EFEM svoje nová analytická označení, což se projeví jako velmi praktické a efektivní zejména pro mapování do designu. Tyto vztahy se totiž stejně chovají a poté se také stejně programují.

Technologie EFEM takto zavádí následující členění vztahů:

- Kompozice nebo sdílená agregace od majitele ke svým částem (vztah buď ku 1 nebo ku N), což odpovídá v syntaxi UML kompozice nebo sdílené agregace pouze v daném směru od celku směrem k částem (neuvažuje se v té chvíli opačný směr).
- Běžná asociace jako nový pojem směrového vztahu. To odpovídá v UML dvěma možnostem: Buď se jedná o vztah asociace bez označení agregace ve směru od jednoho prvku k jednomu druhému (viz např. „číselníková vazba“ resp. „propojení nezávislých entit“) anebo se jedná o „vztah k parentovi“, což

je obrácený vztah od jednoho prvku z N částí k celku. Všechny tyto zdánlivě nesouvisející vztahy se v EFEM nazývají stejně a to „běžná asociace“.

- Ostatní syntaxe, tj. asociativní třída a GEN-SPEC, odpovídají členění názvů podle UML.

Důvodem nového doplňujícího názvosloví v členění směrových vztahů jsou stejné technologické postupy mapování těchto vztahů do designu. U každého vztahu běžné asociace totiž dochází k naplnění vazby stejně a to propojením vazby na existující prvek. Naopak u kompozice nebo sdílené agregace ve směru k částem dochází k tvorbě nových prvků.

5. Přehled modelů UML použitých v EFEM a jejich postavení v projektu

Model v UML se chápe jako určitý pohled na informační systém podle určitých kritérií a pomocí určitých typů prvků modelu. Kromě toho daný model vyjadřuje popis systému na určité úrovni abstrakce.

V dalších kapitolách je uveden seznam všech modelů používaných v UML. Tyto modely přebírá technologie EFEM do svých procesů efektivní tvorby informačních systémů. Kromě vztahů zavedených v těchto modelech lze pomocí extensivních mechanismů UML zavést další modelové vztahy mezi prvky, které napomáhají efektivní dokumentaci.

U každého modelu je stručně vysvětlen jeho význam, jeho pozice v abstraktních úrovních a také je stručně ukázáno použití tohoto modelu v kontextu efektivní tvorby informačního systému. V nadpisech je hned za názvem modelu uvedeno jeho možné použití v úrovních abstrakce (AM resp. D).

Z hlediska překladu do češtiny je prioritní vždy název prvku UML v angličtině (pozn.: češtináři prosím nechť prominou jinak nepřípustné skloňování těchto anglických výrazů). Důvodem použití anglických výrazů je jednoznačnost syntaxe jazyka UML. Jazyk UML má svou syntaxi stejně jako každý jiný jazyk, například jako programovací jazyk JAVA nebo C#. Prvky pocházející ze syntaxe UML budou proto psány velkými písmeny, aby byla zdůrazněna jejich syntaktická odlišnost jako „rezervovaná slova jazyka UML“.

5.1 USE CASE MODEL (pouze AM)

Model případů užití. Používá se výhradně ve fázi AM. Ukazuje systém z pohledu všeho možného použití informačního systému. Podává velmi sofistikovaný pohled na funkcionality a chování systému, vymezuje hranice systému a vymezuje obsah řešení. V jednotlivých případech užití (tj. v prvcích USE CASE) jsou obsaženy tzv.

scénáře těchto užití jako dynamická složka modelu (USE CASE SCENARIO). Tyto scénáře patří pro další zpracování k nosným artefaktům AM a jsou pro designéra a programátora velmi potřebné.

Model je z hlediska EFEM pro informační systémy nezastupitelný a používá se vždy. Jeho obrovskou výhodou je to, že po malých úpravách je výstup z tohoto modelu srozumitelný všem účastníkům projektu. Lze jej proto opětovně použít jako výstup pro jiné velmi důležité dokumenty, jimiž jsou uživatelská dokumentace, plány testů, obchodní nabídky, přílohy smluv o dodávce softwaru apod.

5.2 CLASS MODEL (AM, D)

Model tříd. Používá se jak v AM, tak v D. V AM vyjadřuje vlastnosti a vztahy tříd informací, které se budou evidovat ve svých výskytech (viz předešlá kapitola). U hybridních systémů v designu CLASS MODEL vyjadřuje třídy v OOP a tabulky v relační databázi (ERD), což jsou jednak jako mapované obrazy tříd pocházejících z AM, ale může také zavádět další nové tzv. služební třídy z daného prostředí (GUI, ODBC, ADO NET, XML PARSER apod.).

Tvorba CLASS MODELU je jak v AM, tak v D nezastupitelná a artefakty patří k nosným dokumentům v EFEM. Používá se vždy, v D nemusí být z hlediska syntaxe UML úplný a některé části modelu se nahrazují efektivnějšími prostředky vývojového prostředí. Například návrh GUI v D se neprovádí v UML, ale v daném prostředí přímým návrhem.

V AM se oproti tomu vyžaduje konkrétní úplnost CLASS MODELU bez výjimek.

5.3 SEQUENCE MODEL (AM, D)

Sekvenční model. Používá se jak v AM, tak v D. Na rozdíl od modelu tříd je tzv. instanční, tj. pracuje s instancemi a nikoliv s třídami, třídy je možné zavést až sekundárně k daným instancím. Vyjadřuje sekvence zpráv mezi instancemi, které si v časové posloupnosti (tj. v dané sekvenci) předávají požadavky na zpracování, tzv. zprávy. Tímto způsobem model přesně vyjadřuje dynamiku v časové posloupnosti posílání zpráv mezi objekty. Ve fázi AM se jedná o zasílání požadavků v sekvenci mezi instancemi informací. V D se jedná o zprávy mezi objekty designu (objekty z daného jazyka, instance komponent atd.).

Je sice možné teoreticky namodelovat celý systém až do posledního kódu ve všech sekvencích, ale není to praktické. Tato cesta se kdysi doporučovala (například mapování USE CASE MODELU do SEQUENCE MDOELU), ale již se nedoporučuje pro velmi nízkou efektivitu.

Sekvenční model se používá jednak ve vzorech a pro velmi specifické scénáře zpracování, kde je žádoucí zobrazit graficky některé složité scénáře zpracování.

Opakující se stereotypy v dynamice systému (výběr obsluhou, zadání atributů atd.) se v D navrhnou pomocí scénářů USE CASE MODELU a zvolených vzorů mapování, což je efektivnější, než provádět rozpis každého scénáře zvlášť do specifického SEQUENCE MODELU.

5.4 COLLABORATION MODEL (AM, D)

Model spolupráce. Používá se jak v AM, tak v D. Je plně zastupitelný SEQUENCE MODELEM. Pokud se modeluje určitá situace, volí se buď SEQUENCE MODEL anebo COLLABORATION MODEL, tj. není třeba tutéž situaci modelovat oběma modely. COLLABORATION MODEL stejně jako SEQUENCE MODEL zobrazuje objekty a zprávy mezi nimi. Nezdůrazňuje však časovou sekvenci zpráv, ale viditelnost jednotlivých instancí mezi sebou, tj. soustřeďuje se na objektové struktury. Podél spojnic viditelnosti (tzv. prvky LINKS) se zobrazují zprávy zasílané mezi objekty. Pokud je žádoucí zdůraznit struktury objektů, použije se tento model. Pro vyjádření sekvencí v časové posloupnosti je vhodnější použít SEQUENCE MODEL. Pro COLLABORATION MODEL platí stejné postavení v projektu jako pro SEQUENCE MODEL.

V literatuře se zavádí také tzv. INSTANCE neboli OBJECT MODEL (model instancí resp. objektový model). Z hlediska syntaxe UML se jedná o zvláštní případ COLLABORATION MODELU, ve kterém jsou pouze instance ve strukturách a nejsou zobrazeny žádné zprávy.

5.5 STATE CHART MODEL (AM, D, BM)

Stavový model. Model je plně převzat z předešlých škol modelování. Používá se v AM i v D, ale také v BM (modelování podniku). V AM vyjadřuje stavy instancí informace v průběhu jejich života. V D vyjadřuje stavy objektů OOP resp. instancí komponent, částí systému apod. V modelech BM vyjadřuje stavové mechanismy objektů podniku.

Nejčastěji se používá při tvorbě informačních systémů v modelování podniku, kdy vyjadřuje stavy objektů v podniku a přechody mezi těmito stavy. Pomocí tohoto modelu v modelování podniku se efektivně určují případy užití informačního systému, které mají tyto přechody podporovat.

Druhou oblastí, kde se výrazně projevuje použití stavového modelu, jsou technologické systémy pracující v reálném čase. U těchto systémů existuje přímá návaznost mezi přechody stavového modelu a aktivitami objektů, tj.mezi přechody stavů a operacemi objektů.

5.6 ACTIVITY MODEL (AM, D, BPM)

Model aktivit. Používá se jak v AM, tak v D. Zavádí se s výhodou v těch situacích, kdy jsou velmi dobře známy činnosti neboli aktivity systému, ale nejsou známy kompetence a scénáře spolupráce výskytů. Z hlediska syntaxe se jedná o extenzi modelu stavového, protože aktivita je chápána jako stav, po který trvá určitá činnost.

Po určité syntaktické extenzi se ACTIVITY MODEL používá také jako model pro BPM. V tom případě aktivity v modelu nejsou aktivity informačního systému, ale aktivity podniku, tj. procesy podniku.

5.7 COMPONENT MODEL (pouze D)

Komponentní (také komponentový) model. Vyskytuje se pouze ve fázi designu. UML chápe komponentu jako uzavřenou a ohraničenou část implementace systému, která je opětovně použitelná a vyměnitelná. Návrh komponent je tedy již poplatný danému prostředí. Hlavním posláním COMPONENT MODELU je určit, ve kterých komponentách budou implementovány a kódovány jednotlivé třídy a interfacy a jaké jsou vztahy mezi komponentami. Z tohoto se určuje důležitá informace, která komponenta si potřebuje přilinkovat kterou jinou komponentu. K tomu vyjádření se používá vztah DEPENDENCY.

UML rozlišuje jednak tzv. source komponentu a tzv. binary komponentu. Moderní technologie zvaná „komponentní technologie“ chápe pod komponentou pouze binary komponentu. Source komponenta je v UML zavedena pro možnosti modelovat systémy v technologiích nepodporující moderní komponentní technologie a linkují implementaci pouze na úrovni zdrojových textů (např. Pascal 7.0 apod.) .

5.8 DEPLOYMENT MODEL (pouze D)

Model rozmístění (také model nasazení). Model, který je nejdále vzdálen od abstraktní úrovně analytického modelování. Zobrazuje rozmístění zdrojů, stroje, síť, disky, instalace komponent, kde budou žít instance komponent atd. Stručně řečeno, model zobrazuje rozmístění HW a SW informačního systému.

5.9 Rozdělení modelů podle nezbytnosti dokumentace

Technologie EFEM rozděluje modely pro tvorbu informačních systémů do dvou skupin:

- Modely nezbytné pro projekt
- Modely podpůrné pro projekt

5.9.1. Modely nezbytné pro projekt

Pojem „nezbytný“ je zde chápán jako „technologicky nezbytný“. Modely nezbytné pro projekt se vyznačují tím, že v každém projektu tvorby evidenčního informačního systému budou vždy samostatně tvořeny. Pokud nebudou samostatně tvořeny, resp. neponesou veškerou minimálně nutnou informaci dokumentace (neúplnost modelu), považuje se projekt za nedostatečně zdokumentován. Nutný obsah těchto modelů je v produktu EFEM přesně definován požadovaným tvarem dokumentu a postupkami jeho tvorby (šablony a jejich plnění).

Logickou úvahou lze odvodit seznam modelů, které jsou pro tvorbu IS nenahraditelné. Logika úvahy spočívá v té skutečnosti, že vyšší úroveň abstrakce musí být úplným zadáním pro tvorbu artefaktů nižší úrovně abstrakce. Pojem úplnosti je vysvětlen v odpovídající kapitole „Úplnost dokumentace v EFEM“.

V úvaze se postupuje zpětně v posloupnosti fázování vývoje:

- Informační systém (pozn.: může se jednat i o jednoduchý SW) je nasazen u zákazníka. Pro toto nasazení muselo existovat zadaní, čemuž v předešlém výčtu odpovídá DEPLOYMENT MODEL. Může se jednat i o velmi jednoduchý model, například pouze údaj o minimální konfiguraci stroje, na kterém bude nainstalován nějaký velmi jednoduchý software, anebo se jedná o zobrazení poměrně dost složité architektury několika strojů (web server, aplikační servery, databázové servery, zálohové servery, autentikační server apod.) a zobrazení umístění komponent na těchto strojích. DEPLOYMENT MODEL je důležitý, protože bez něj neexistuje zadání pro nasazení SW (instalace).
- Jednotlivé naprogramované a nainstalované prvky softwaru musely být programátory vytvořeny, takže tito pracovníci museli vědět, do jakých implementačních celků museli kód umístit a následně zkompilovat resp.

zahrnout do řešení. Těmto prvkům odpovídají komponenty. Dalším nezbytným modelem, který zadává implementační celky, je COMPONENT MODEL.

- V jednotlivých komponentách je před kompilací napsán zdrojový kód, tj. v tomto zdrojovém kódu jsou umístěny napsané třídy pocházející jak z AM, tak z D. Pokud by byl proveden tzv. reverse engeneering, tj. zpětné vytvoření CLASS MODELU z kódu v nějakém CASE nástroji, bylo by možné získat celkový obraz (model) napsaného kódu v CLASS MODELU designu. V tomto modelu by se vyskytovaly všechny naprogramované třídy, jak třídy z AM, tak třídy z D (například GUI prvky apod.). Je zřejmé, že opačným procesem, tj. generací kódu z tohoto takto získaného modelu, by se obráceně opět získal původní kód. Logicky z toho vyplývá, že by teoreticky bylo možné považovat za nutný celý CLASS MODEL se všemi naprogramovanými třídami, který je v UML obrazem zdrojového kódu. Avšak ukazuje se, že z hlediska efektivity tvorby IS není výhodné prvky SW, které vznikají v designu (např. návrh obrazovek apod.), modelovat pomocí UML. Jednalo by se o zbytečně pracný a zdlouhavý proces. Většinou vývojová prostředí velmi dobře podporují tvorbu těchto prvků a to velmi efektivně. V technologii EFEM se doporučuje nahradit neefektivní tvorbu celého CLASS MODELU včetně služebních tříd tímto postupem:
    - Vytvoří se USE CASE MODEL, který obsahuje tzv. scénáře. Na jedné straně se získá výčet a popis všech funkcionalit systému (což je přínos pro projekt), na straně druhé je z těchto scénářů patrné, jaké budou existovat prvky v CLASS MODELU, se kterými scénáře pracují, a jak okolí spolupracuje s těmito vnitřními prvky.
    - Vytvoří se CLASS MODEL AM v celkové úplnosti, ten je „unikátní“ a musí být vytvořen. USE CASE MODEL a CLASS MODEL AM musí být spolu konzistentní, protože prvky USE CASE pracují s prvky instancí z CLASS MODELU AM.
    - Ze scénářů v prvcích USE CASE a s odpovídajícího CLASS MODELU AM získává designér velmi přesné a dostatečné zadání pro tvorbu všech prvků v designu. Na základě těchto dvou modelů je schopen vytvořit design systému.

Namísto tvorby celého CLASS MODELU se tedy tvoří USE CASE MODEL a CLASS MODEL AM (který neobsahuje prvky D). Tato kombinace efektivněji nahrazuje tvorbu celkového CLASS MODELU i se služebními třídami.

Z vedeného vyplývá, že technologie EFEM zavádí z hlediska technologie tvorby SW tyto modely jako vždy nezbytné:

- USE CASE MODEL
- CLASS MODEL
- COMPONENT MODEL

- DEPLOYMENT MODEL

Použití těchto modelů v rámci své nezbytnosti je následující:

- Pomocí USE CASE MODELU a na základě popisu jejich scénářů vzniká zadání pro dynamiku spolupráce vnitřních prvků systému s okolím. Pomocí scénářů se zavádějí služební třídy v daném prostředí, které zprostředkují tuto interakci objektů pocházejících z AM s okolím. Znamená to, že návrh obrazovek a podobných objektů se neprovádí v modelu tříd, ale návrh vzniká na základě správných a přesných popisů scénářů případů užití.
- Vytváří se CLASS MODEL AM, který je konzistentní s USE CASE MODELEM. V případě kolizí se provádějí případné opravy jak v CLASS MODELU AM, tak v USE CASE MODELU (odstraňování chyb apod.).
- Na základě CLASS MODELU a USE CASE MODELU vzniklých v AM vznikají mapováním třídy a tabulky, včetně služebních tříd formulářů, připojení k databázi apod.
- Na základě COMPONENT MODELU vzniká zadání pro programátora, které určuje rozmístění tříd do komponent. Toto zadání specifikuje jaké typy projektů (přesněji komponent) se mají v kódování založit a jaké třídy a interfacy do nich umístit. Model také určuje, které již hotové komponenty se budou používat a je tedy nutno si je přilinkovat.
- Na základě DEPLOYMENT MODELU se tvoří architektura systému jako rozmístění HW a SW včetně postupu instalace systému.

Všechny tyto modely jsou nezbytné pro naprogramování. V některých případech mohou být modely primitivní. Například COMPONENT MODEL nějakého jednoduchého programu „Kalkulačka“ obsahuje pouze jednu komponentu typu <<*Standard EXE*>> (označuje spustitelný soubor typu EXE pod Windows). DEPLOYMENT MODEL v tomto případě obsahuje informaci o minimální konfiguraci stroje a operační systém.

Oproti tomu známé karetní aplikace pod WINDOWS jako jsou FREECELL, SOLITAIRE apod. již nemají takovýto primitivní komponentní model, protože obrázky karet jsou v samostatném linkovatelném souboru CARDS.DLL.

5.9.2. Princip minimální, jednoznačné a přitom úplné dokumentace

Ne všechny prvky v modelech, které popisují naprogramované prvky informačního systému, musí nutně a bezpodmínečně v modelech existovat. Díky principu opětovné použitelnosti mohou existovat odkazy na jiné prvky dokumentace jiného typu, které usnadňují a zjednodušují dokumentaci modelů. Seznam těchto

použitelných odkazů se chápe jako seznam použitelných vzorů případně tzv. „template“, případně vzor typu „stejně jako...“.

V důsledku tohoto postupu je úplná, jednoznačná a přitom minimální dokumentace naprogramovaných prvků tvořena artefakty dvou typů:

- Prvky modelů, které jsou nezbytné jako unikátní a přitom jsou specifické pro projekt. Tyto prvky jsou specifické pro daný problém. Nejsou odvoditelné z opakujících se situací.
- Odkazy do existujících vzorů, tj. odkazy do opakujících se situací.

Kombinace těchto dvou typů dokumentů dává jednoznačné a přitom minimálně pojaté zadání pro kódování prvků softwaru a tedy i efektivní dokumentaci informačního systému.

5.9.3. Modely podpůrné pro projekt

Mezi modely podpůrné spadají:

- SEQUENCE MODEL
- COLLABORATION MODEL
- ACTIVITY MODEL
- STATE CHART MODEL

Tyto modely se v EFEM považují za podpůrné. Výjimku mohou tvořit technologické systémy (REAL-TIME), kde některé z těchto modelů se stávají mnohem důležitějšími.

V evidenčních informačních systémech není tvorba těchto modelů vždy v projektu nezbytná, i když může nastat případ, že se takový model bude vyžadovat.

Tyto modely se v EFEM zavádějí v těchto situacích:

- použití ve vzoru
- popis specifických a zvláštních situací

Mnohdy je podpůrný model použit nepřímo přes již zavedený vzor. Při modelování dojde k vytvoření odkazu na určitý vzor. Tento vzor je vyjádřen v jednom z podpůrných modelů (například SEQUENCE MODEL). Do vzoru se dosadí za účastníky vzoru vlastní prvky, čímž dojde k naplnění vzoru. Vytvoří se tak nový model, který popisuje daný informační systém pomocí tohoto typu modelu.

V některých případech se tvorba podpůrných modelů stává velmi vhodnou metodou pro získání resp. ověření některého z modelů z kategorie nezbytných modelů. Pomocí podpůrných modelů lze mnohdy získat anebo ověřit nezbytný model efektivněji než bez podpůrného modelu, resp. se může zabránit kolizím u velmi specifických situací. Jako příklad lze uvést složité resp. velmi důležité zpracování

vyjádřené pomocí SEQUENCE MODELU nebo ACTIVITY MODELU. O vhodnosti použití modelu v těchto situacích rozhoduje subjektivní názor autora, který posuzuje nutnost zavedení takového modelu.

6. Syntaxe UML používaná v EFEM

6.1 Společná syntaxe ve všech modelech UML

Existuje několik společných mechanismů ve všech modelech UML. V každém modelu lze vždy zavést následující prvky:

6.1.1. TAGGED VALUE

TAGGED VALUE je extenzivní mechanismus UML. TAGGED VALUE zavádí dvojici „Název + Hodnota“ (podobně jako v HTML resp. XML) a přiřazuje se jako uživatelská vlastnost k nějakému prvku modelu. UML umožňuje k libovolném elementu přiřadit vlastní pojmenovanou hodnotu s vlastním významem a tím elementu přiřadit vlastnost s hodnotou. V některých CASE nástrojích se proto TAGGED VALUE nazývá PROPERTY. Počet a možnosti použití TAGGED VALUE nejsou nijak omezeny.

6.1.2. STEREOTYPE

Jedná se o extensivní mechanismus UML. Zavedení STEREOTYPE umožňuje tvůrci modelu zavést vlastní dodatečnou kategorizaci v typu elementu. Někdy se také hovoří o virtuálním typu elementu. Prvek STEREOTYPE označuje podtyp k danému typu elementu. Tím se vytvoří další jemnější specializace typu elementu. Daný typ, ke kterému se přiřazuje nová kategorizace, se nazývá BASE CLASS. Přiřazením elementu do daného STEREOTYPE je element dodatečně kategorizován. Přiřazení STEREOTYPE k danému prvku je vidět přímo na diagramu jako název umístěný u prvku do závorek <<….>>.

Většina CASE nástrojů (včetně EA) umožňuje pro daný STEREOTYPE zavést také odlišný grafický prvek. Tím lze odlišit nový typ (přesněji podtyp) elementu modelu i v diagramu.

Příklady: Zavede se STEREOTYPE <<*ActiveX DLL*>> pro BASE CLASS s názvem COMPONENT. Ke každé komponentě typu ActiveX DLL přiřadíme tento STEREOTYPE.

Jiný příklad: Zavede se STEREOTYPE <<*Interface*>> pro BASE CLASS typu CLASS. Interface je podtyp třídy, je to tzv. čistá abstraktní třída (viz například e-kniha Design Patterns OOP).

6.1.3. NOTE

Ke každému elementu lze přiřadit NOTE neboli poznámku resp. popis. NOTE obsahuje prostý text popisující prvek. Doporučuje se nešetřit při modelování vysvětleními umístěnými do prvků NOTE, protože jinak se model stává pro čtenáře nepochopitelným. Některé CASE nástroje používají namísto NOTE název DESCRIPTION.

6.1.4. CONSTRAINT

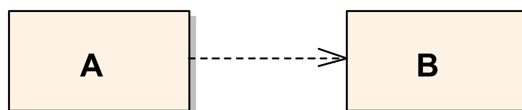
Ke každému elementu anebo k několika elementům lze přiřadit omezující podmínku podobně jako v databázi. Podmínka je typu BOOLEAN a vyjadřuje tu skutečnost, že pokud by mělo nastat FALSE, model nebude konzistentní Model vyžaduje v prvku CONSTRAINT splnění TRUE.

Sémanticky není tato podmínka BOOLEAN nijak omezena, může se použít vlastní pseudo-jazyk. Vyžaduje se srozumitelnost a jednoznačnost. Je možné použít i tzv. OBJECT CONSTRAINT LANGUAGE (tj. OCL), což je specifikace zvláštního jazyka uvedená v příloze UML. Z praktického hlediska není nutné OCL zavádět.

6.1.5. DEPENDENCY

Mezi libovolnými dvěma elementy, které jsou na sobě závislé, lze v modelu zobrazit vztah závislosti. V UML se vztah závislosti nazývá DEPENDENCY a vyjadřuje tu skutečnost, že pokud by se z modelu odstranil prvek, na kterém je druhý prvek závislý, tento závislý prvek by nemohl plnit svou funkci korektně a projevil by se v modelu jako neúplný.

Vztah DEPENDENCY se značí čárkovanou šipkou ve směru závislosti:



obrázek 21 Vztah DEPENDENCY mezi (nějakými) prvky A a B

Vztah DEPENDENCY lze doplnit o STEREOTYPE, který označuje typ závislosti. Tento údaj není z hlediska efektivního modelování podstatný a EFEM jej považuje za doplňkovou informaci.

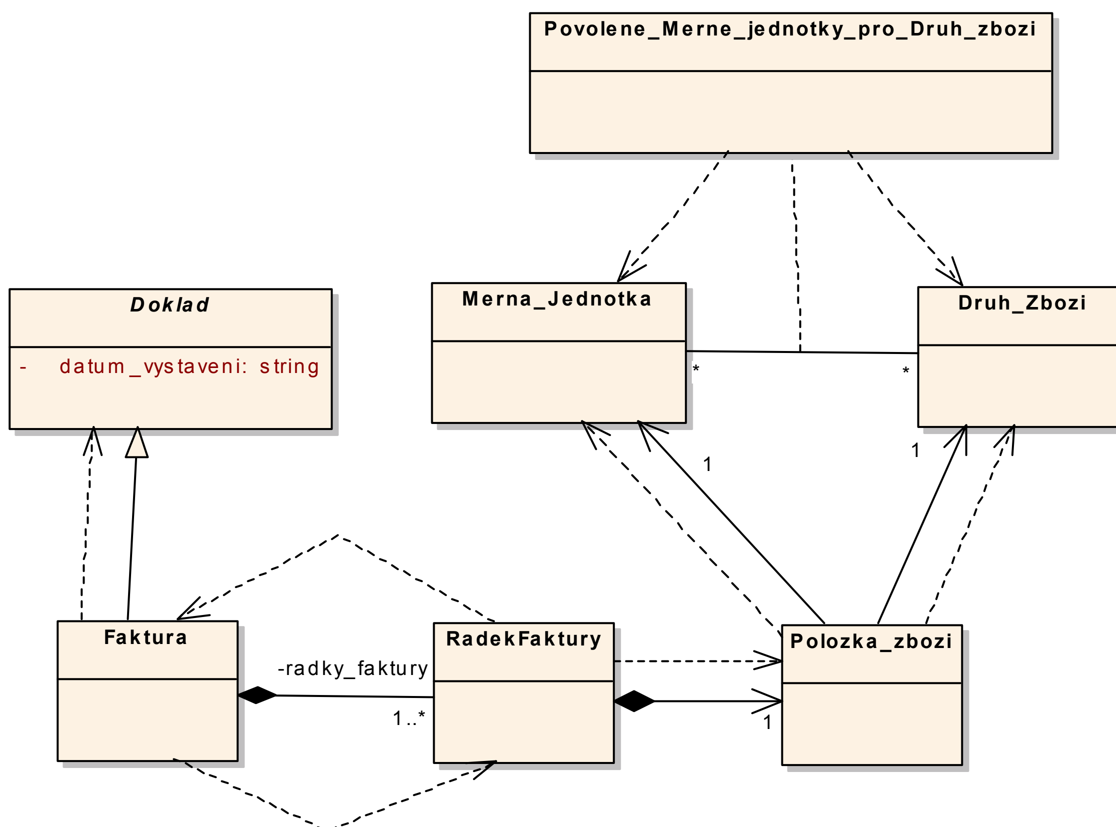
Vztah DEPENDENCY má významné postavení v EFEM v těchto modelech: CLASS MODEL, vztah mezi prvky BPM a prvky z USE CASE MODEL a COMPONENT MODEL.

6.1.5.1 Odvoditelné vztahy DEPENDENCY

Některé vztahy DEPENDENCY vyplývají přímo z logiky jiných vztahů a proto se v modelu neznázorňují. Pro EFEM jsou vztahy DEPENDENCY velmi důležité zejména v CLASS MODELU, protože tam se promítají vysoce prakticky až do kódu. Vztahy DEPENDENCY v CLASS MODELU určují, které třídy musí mít jiná třída v dosahu své viditelnosti (tj. na kterých je v DEPENDENCY), aby mohla být funkční. Tím vztahy DEPENDENCY implikují možné a také nemožné rozdělení systému do vrstev, což vede k praktickému návrhu vrstev systému až do fyzického rozčlenění systému do komponent v COMPONENT MODELU.

V následujícím obrázku jsou přehledně uvedeny všechny vztahy, které jsou potřebné pro tvorbu CLASS MODELU. Obrázek znázorňuje závislosti DEPENDENCY u všech těchto možných vztahů mezi třídami. Vztahy DEPENDENCY jsou znázorněny čárkovanými šipkami a směr šipky určuje, kdo koho potřebuje.

Je třeba podotknout, že zde uvedené vztahy DEPENDENCY se v CLASS MODELU obyčejně neznázorňují. Z hlediska použití technologie EFEM je třeba velmi dobře znát vztahy DEPENDENCY vedené v tomto vzoru:



obrázek 22 Vzor vztahů DEPENDENCY v CLASS MODELU

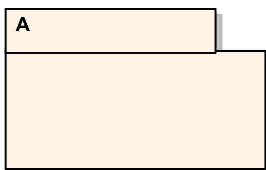
Popis vzoru vztahů DEPENDENCY v CLASS MODELU:

- Ve vztahu GEN-SPEC je dědic ve vztahu DEPENDENCY vůči předkovi a obráceně tento vztah neplatí (viz Faktura versus Doklad).
- V kompozici 1:N je ve vztahu DEPENDENCY majitel vůči svým částem a naopak (viz Faktura versus Řádek Faktury). V případě, že se kompozice určí jako jednosměrná a prvek část nevidí v běžné asociaci svého majitele (znázorňuje se jako kompozice s šipkou vůči částem), potom se jedná pouze o DEPENDENCY od majitele k části a nikoliv naopak. Vlastnosti kompozice ve vztazích DEPENDENCY platí stejně pro sdílenou agregaci.
- Vztah kompozice ku jedné vede k DEPENDENCY od majitele (uživatele) k vlastněné instanci a nikoliv naopak (viz Řádek faktury versus Položka zboží).
- Vztah „číselníkové vazby“ resp. provázání nezávislých entit vede k DEPENDENCY od uživatele k použitému prvku ve směru šipky (viz Položka zboží versus Měrná jednotka resp. Položka zboží versus Druh zboží).

- Asociativní třída se z hlediska DEPENDENCY chová stejně jako dvě běžné asociace. Třídy provázané asociativní třídou nejsou ve vztahu DEPENDENCY a současně asociativní třída je v DEPENDENCY na obou třídách, které propojuje (viz třída Povolené měrné jednotky pro druh zboží versus Měrné jednotky a Druh zboží). V případě násobné asociativní třídy se toto pravidlo zobecňuje pro N tříd. Tyto třídy nejsou mezi sebou ve vztahu DEPENDENCY, ale násobná asociativní třída je ve vztahu DEPENDENCY vůči všem, které propojuje (stejně jako N běžných asociací).

6.1.6. PACKAGE

PACKAGE je typem prvku v UML a slouží k možnosti práce se skupinami elementů, tj. ke „grupování“ elementů. Prvky modelu lze umisťovat do jednotlivých prvků PACKAGE. Pro prvek typu PACKAGE se v UML používá visuální prvek ve tvaru složky:

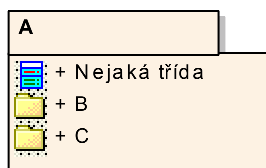


obrázek 23 : Visual prvek PACKAGE má tvar složky

6.1.6.1 PACKAGE a NESTING

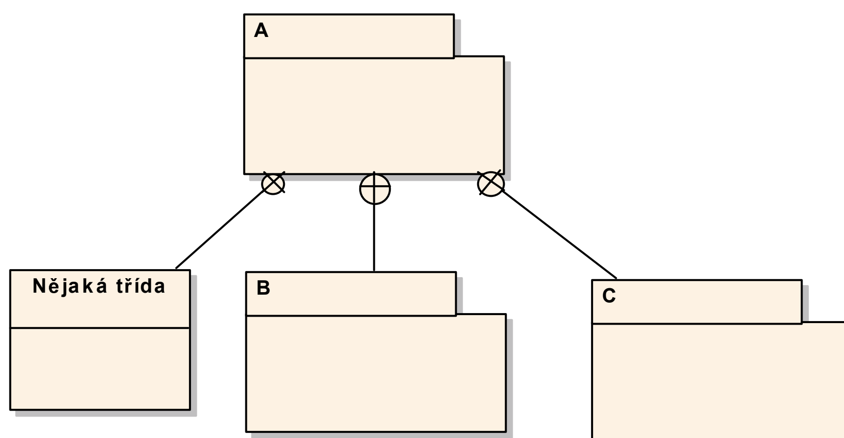
Obsah PACKAGE je důležitý prvek pro práci s daným CASE nástrojem. Obsah PACKAGE je přímo viditelný v okně, které zobrazuje stromovou strukturu projektu daného CASE nástroje (například okno PROJECT VIEW v EA).   K zobrazení obsahu PACKAGE do diagramu, tj. k zobrazení seznamu prvků v PACKAGE, se zavádí interakce zvaná NESTING. UML povoluje dvojí možné zobrazení této interakce v diagramu.

- První možnost zobrazení obsahu PACKAGE používá EA jako implicitní způsob. Spočívá v tom, že elementy obsažené v PACKAGE se v diagramu zobrazí přímo jako seznam elementů uvnitř složky, například takto:



obrázek 24 Zobrazení obsahu PACKAGE přímo, PACKAGE A obsahuje jednu třídu a dva prvky PACKAGE

- Druhou možností notace pro NESTING jako obsahu PACKAGE je použít v diagramu přímo spojnicí označující NESTING kolečkem s plusem u majitele seznamu prvků (kolečko je u PACKAGE):

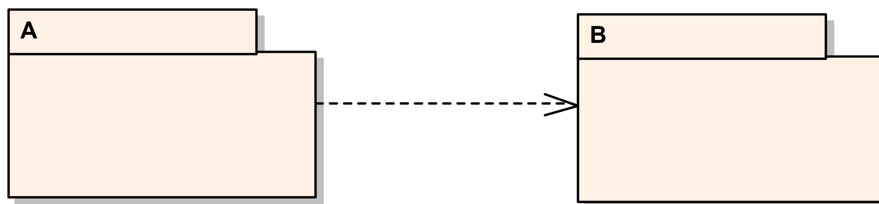


obrázek 25 Jiný možný způsob označení obsahu PACKAGE téže situace jako v předešlém obrázku

Obrázek z diagramu pro NESTING musí souhlasit se stromovou hierarchií. Z toho důvodu se pro EA nedoporučuje syntaxe spojnicí z předešlého obrázku, ale pouze pomocí vnitřního výčtu, který je v EA zaveden implicitně.

6.1.6.2 PACKAGE a DEPENDENCY

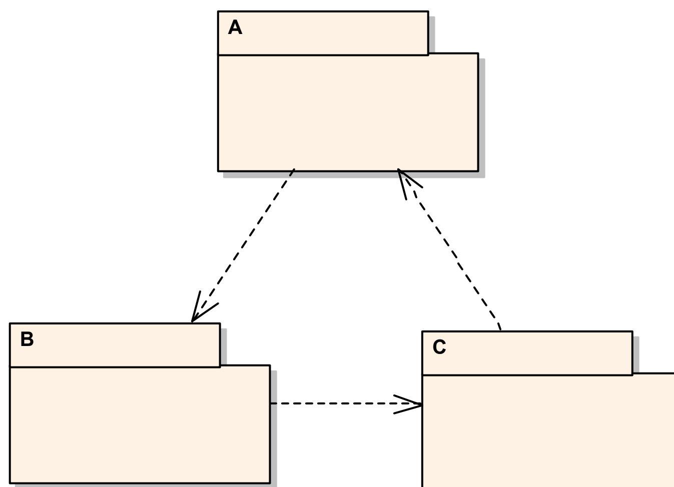
Pokud jeden element z jednoho prvku PACKAGE je ve vztahu DEPENDENCY vůči jinému elementu z jiného prvku PACKAGE, logicky z toho vyplývá také vztah DEPENDENCY mezi těmito dvěma prvky PACKAGE. Tento vztah se nesmí zaměnit s předešle uvedeným vztahem NESTING, kde se jednalo o vnoření do sebe. Zde se jedná o odvozený vztah DEPENDENCY mezi prvky PACKAGE, který vznikl díky interakci mezi prvky umístěnými uvnitř těchto prvků PACKAGE, viz následující obrázek:



obrázek 26 Package A obsahuje nějaký prvek, který potřebuje nějaký prvek z B

Pokud v prvku PACKAGE A neexistuje ani jeden prvek, který je ve vztahu DEPENDENCY vůči nějakému prvku v B, potom vztah DEPENDENCY mezi A a B neexistuje.

Autor modelů zavádí jednotlivé prvky PACKAGE tak, že do nich umisťuje prvky modelu podobně jako soubory do adresářů, což vede k určitému uspořádání modelu do prvků PACKAGE. UML sice dovoluje, ale není doporučováno, aby ve vztahu DEPENDENCY mezi prvky PACKAGE nastal jev zvaný CIRCULAR DEPENDENCY, neboli vztah DEPENDENCY uzavřený do kruhu (kruh ve smyslu teorie grafu):



obrázek 27 Kruhová reference mezi třemi PACKAGE

Vztahu DEPENDENCY do kruhu ve smyslu teorie grafu (obdoba *circular reference*) je třeba se vyhnout. Prvky PACKAGE propojené v DEPENDENCY do kruhu se chovají tak, jako by se jednalo o jeden PACKAGE. V důsledku toho rozložení prvků do kruhově provázaných PACKAGE nepřináší z hlediska modelování informačního systému efektivní přínos a jedná se pouze o pomyslné dělení modelů do fyzicky neoddělitelných celků

Prvek PACKAGE je prvek, který lze použít všude, kde je třeba pracovat se skupinami prvků. Technologie EFEM k tomu navíc zavádí povinné použití PACKAGE a následně DEPENDENCY pro následující oblasti modelování:

- EFEM MODEL MANAGMENT
- Tvorba CLASS MODELU a následně COMPONENT MODELU

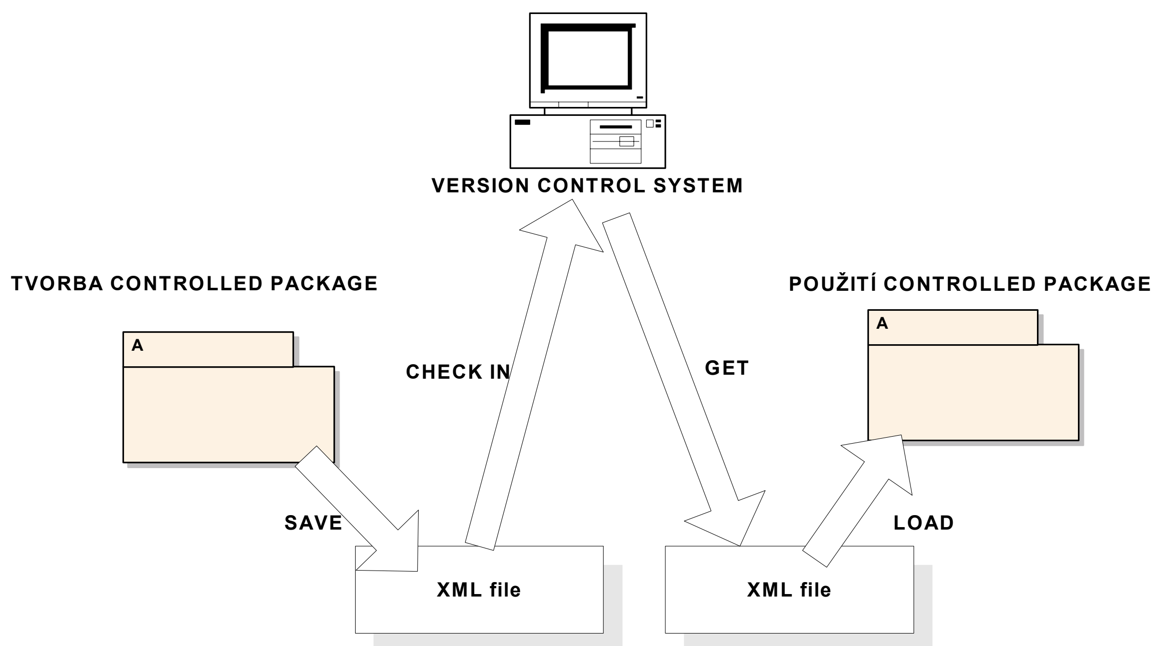
6.1.6.3 PACKAGE a EFEM MODEL MANAGMENT

EFEM MODEL MANAGMENT zahrnuje modelovací techniky a postupy vedoucí k možnosti řízení modelů, jejich spravování, ukládání, řízení verzí, distribuci modelů apod. Základem EFEM MODEL MANAGMENTU je zavedení necirkulárních PACKAGE, které se nazývají CONTROL PACKAGE (viz EA).

CONTROL PACKAGE je PACKAGE, který je provázán se svým souborem typu XML. Do tohoto souboru se PACKAGE vždy ukládá a z něj se vždy načítá. Přiřazený soubor XML je fyzickým nositelem daného PACKAGE, tj. nese implementaci daného PACKAGE. Uložení daného PACKAGE do XML souboru znamená uložení celé části modelu (obsah PACKAGE) do souboru. Naopak načtení z tohoto souboru znamená obnovení části modelu ze souboru. Jeden CONTROL PACKAGE se tak stává obdobou komponenty, ale nikoliv pro program, ale pro modely.

V EFEM technologii všechny modely v UML, které jsou publikovány jako dokumentace projektů ve firmě, jsou spravovány přes prvky CONTROL PACKAGE a jejich XML soubory. Jiné typy souborů, které mohou být ukládány z daných CASE nástrojů (tj. nejsou XML soubory CONTROL PACKAGE) se považují pouze za pomocné. Například v EA všechny soubory s příponou `EAP` jsou pro modely pouze pracovní a jsou použity na lokálním stroji pro vizualizaci modelů načtených pomocí mechanismu CONTROL PACKAGE. Prvek CONTROL PACKAGE se tak stává základní implementační jednotkou všech modelů podobně jako například komponenta v komponentní technologii.

Princip práce s CONTROL PACKAGE a jeho přiřazeným XML souborem ukazuje následující obrázek:



obrázek 28 Princip práce s CONTROL PACKAGE v EFEM MODEL MANAGMENT

Práce s CONTROL PACKAGE a jeho XML soubory je v principu stejná jako práce se zdrojovými texty v daném systému pro řízení verzí, například řízení zdrojových souborů pod Visual Source Safe nebo CVS, resp. v jednoduché knihovně souborového systému. Zdrojové kódy zde zastupují texty v XML souborech a vývojové prostředí je reprezentováno daným modelovacím CASE nástrojem. Modelování se takto podřizuje stejným a již známým principům práce s nástrojem pro řízení verzí. Tyto postupy jsou tvůrcům softwaru velmi dobře známy na úrovni zdrojových textů. Technologie EFEM tyto postupy pouze přebírá a aplikuje je na XML soubory přiřazené k prvkům CONTROL PACKAGE. Postupy pro konfiguraci projektů, verzování apod. jsou v modelování tímto automaticky přeneseny do úrovně práce s daným nástrojem pro řízení verzí.

Pro další výklad je použita konkrétně technika práce s Visual Source Safe, avšak stejné principy platí i pro CVS.

Pro práci s CONTROL PACKAGE a nástrojem pro řízení verzí se zavádí odpovídající názvosloví:

- „check in daného CONTROL PACKAGE " znamená zpracování části modelu, poté jeho uložení do XML souboru jako CONTROL PACKAGE a následné uložení XML souboru do databáze nástroje pro řízení verzí pro další použití.
- „přilinkovat si CONTROL PACKAGE" znamená provést operaci stažení odpovídajícího souboru XML daného CONTROL PACKAGE na lokální stroj (například GET) a zařadit si jej do svých pracovních „stažených" CONTROL

PACKAGE v projektu v daném CASE nástroji. Prvky z tohoto přijatého PACKAGE lze poté používat v modelu, nesmí se však měnit.

- „check out pro daný CONTROL PACKAGE “ znamená vyhradit si jej zamknutím pro zpracování a současně stáhnout z databáze nástroje pro řízení verzí.
- „nová verze“ části modelu CONTROL PACKAGE znamená uložení CONTROL PACKAGE pod novou verzí v databázi nástroje pro řízení verzí
- „label“ znamená určit nálepkou CONTROL PACKAGE v dané verzi. Provádí se pro snazší konfiguraci několika CONTROL PACKAGE se stejnou nálepkou.

Z uvedených postupů práce s CONTROL PACKAGE vyplývá, že postavení daného CASE nástroje, například EA, se použitím CONTROL PACKAGE a VSS výrazně mění. Modely, které jsou tvořeny a tím zviditelněny v těchto nástrojích, se ukládají do CONTROL PACKAGE a tyto do VSS. Daný CASE nástroj se tak stává pouze nástrojem pro zviditelnění daných částí modelu, na kterých se pracuje, podobně jako vývojové prostředí při tvorbě zdrojového kódu.

6.1.6.3.1 CONTROL PACKAGE LIBRARY MODEL

Pro projekty a jejich dokumentaci se tímto v EFEM stává základní knihovnou, tj. zdrojem modelů, databáze nástroje pro řízení verzí (např. VSS nebo CVS). Nástroj EA se dostává do posice výkonného editoru částí modelů uložených v této databázi. EA produkuje jednotlivé XML soubory, které se konfigurují v nástroji pro řízení verzí. Práce s projektem resp. jeho částmi znamená tento postup:

- identifikovat soubory, které tvoří modely projektu, v nástroji pro řízení verzí
- stáhnout si tyto soubory na lokální stroj a otevřít je, případně editovat v EA
- vrátit změněné soubory do databáze pro řízení verzí

Je vcelku pochopitelné, že pokud se má pracovat s určitou částí modelu z projektu, je třeba mít k dispozici nejenom odpovídající prvek CONTROL PACKAGE nesoucí část modelu v XML souboru, ale také všechny ty CONTROL PACKAGE, které tento CONTROL PACKAGE potřebuje ve vztahu DEPENDENCY (tj. přilinkované prvky CONTROL PACKAGE). A nejenom to, je třeba mít k dispozici rekurzivně další CONTROL PACKAGE, které potřebují tyto přilinkované prvky CONTROL PACKAGE jako přilinkované a tak rekurzivně dále. To je vlastně obsahem prvního bodu předešlého postupu „identifikovat všechny soubory, které tvoří model projektu“.

Podoba s komponentní technologií, kde je třeba také přilinkovat potřebné komponenty, je zřejmá.

Existují dvě možnosti, jak přiřadit k danému prvku CONTROL PACKAGE informace o prvcích PACKAGE, které jsou k němu ve vztahu v DEPENDENCY a je třeba je přilinkovat:

- Použije se MANIFEST. Jedná se o dokument textového tvaru (může být prostý text resp. XML), který je přiřazen k danému prvku CONTROL PACKAGE. V obsahu MANIFESTU se lze dočíst, které další CONTROL PACKAGE je třeba stáhnout spolu s daným CONTROL PACKAGE. Informace obsahuje názvy potřebných CONTROLE PACKAGE a jejich pozice v knihovně. Jedná se o nejjednodušší způsob evidence, který má své výhody a nevýhody: Výhodou je, že informace se nachází pouze u daného sledovaného prvku CONTROL PACKAGE, tj. lze jej „vyexportovat“ jako sebepopisný prvek a není třeba tuto informaci číst někde jinde. Nevýhody: Stahování všech prvků z knihovny je poměrně složité, protože po přilinkování potřebných prvků se musí přistoupit k dokumentům MANIFEST těchto prvků a pokračovat rekurzivně dále. Navíc také při jakémkoliv přeskupení prvků v knihovně (CONTROL PACKAGE REFACTORING) nastávají problémy kvůli vyhledání všech prvků, které tyto změny pocítí.
- Druhou doporučenou možností je zavést CONTROL PACKAGE LIBRARY MODEL. Jak sám název napovídá, jedná se o „centrální“ model samotné knihovny VSS resp. CVS. Tento model obsahuje prvky typu PACKAGE. Názvy těchto prvků odpovídají přesně názvům prvků PACKAGE, které tvoří modely projektů (nejedná se však přímo o tyto prvky z modelů, ale o prvky se shodným názvem). U každého prvku PACKAGE je uvedeno umístění tohoto prvku v knihovně. Model zobrazuje (v několika diagramech), které prvky PACKAGE jsou na kterých prvcích PACKAGE v DEPENDENCY. Ukázkovým příkladem takového diagramu by mohl být například *„obrázek 26 Package A obsahuje nějaký prvek, který potřebuje nějaký prvek z B“*. Uvedený model CONTROL PACKAGE LIBRARY MODEL je tedy centrální evidencí všech potřebných vztahů DEPENDENCY mezi prvky CONTROL PACKAGE.

6.1.6.3.2 Porovnání použití dokumentu MANIFEST a použití CONTROL PACKAGE LIBRARY MODEL

Výhoda použití modelu CONTROL PACKAGE LIBRARY MODEL je zřejmá: Při změnách typu CONTROL PACKAGE REFACTORING je možné pracovat s celým modelem současně a tedy vysoce přehledně. Stahování probíhá také díky modelu cíleně a přehledně.

Nevýhodou je nutnost spravovat centrální „sekundární“ model a zpřístupnit jej všem pracovníkům. Každý pracovník tedy potřebuje k danému PACKAGE získat informace ještě navíc z tohoto modelu, sám PACKAGE již není sebepopisný, tj. informačně samonosný. Je možné tuto nevýhodu zmírnit tak, že bude vždy přístupná HTML dokumentace tohoto modelu. Postup pro práci s dokumentem LIBRARY MODEL je popsán v dokumentu Postupky.

6.1.6.3.3 BATCH IMPORT a BATCH EXPORT pro CONTROL PACKAGE

Nástroj EA umožňuje optimalizovat práci se skupinami prvků CONTROL PACKAGE pomocí tzv. BATCH zpracování. Výběrem jednotlivých prvků CONTROL PACKAGE do skupin pro BATCH zpracování lze uložit anebo načíst prvky v seznamu pro BATCH zpracování najednou.

6.1.6.3.4 Prvky CONTROL PACKAGE a jejich NESTING

Pokud se zavedou dva prvky PACKAGE, které jsou ve vztahu NESTING (jeden prvek je umístěn pod druhý ve stromové hierarchii), a tyto dva prvky PACKAGE se změní na typ CONTROL PACKAGE, může dojít při práci s těmito vnořenými prvky ke kolizi následujícím způsobem:

Nechť vrchní prvek PACKAGE je označen jako A a spodní prvek PACKAGE jako B (tj. prvek B je umístěn ve stromu pod prvek A a oba jsou typu CONTROL PACKAGE). Pokud se prvek PACKAGE A uloží do XML souboru (označen jako `A.xml`), tak se uloží všechna jeho data včetně dat z vnořeného prvku PACKAGE B. Protože prvek PACKAGE B je také typu CONTROL PACKAGE, lze jej také uložit do jeho souboru `B.xml`. Vzniknou tak dva soubory `A.xml` a `B.xml`, přičemž v souboru `A.xml` se nacházejí od určité části stejná data, jako v `B.xml`. Pokud nějaký pracovník stáhne soubor `B.xml` a změní tuto část modelu, tak se data obsahující data prvku B v obou souborech `A.xml` a `B.xml` mohou od sebe lišit. Důsledkem toho je, že pokud si jeden pracovník stáhne soubor `A.xml` a importuje si z něj PACKAGE A, dostane prvek PACKAGE B (který je vnořen do A), a tento prvek B je jiný, než prvek PACKAGE B načtený sám o sobě bez prvku A.

Této kolizi se lze vyhnout dvěma způsoby:

- nepoužívat zásadně vnořené prvky CONTROL PACKAGE. To odpovídá obdobě komponentní technologie.
- lze použít sice vnořené prvky typu CONTROL PACKAGE, ale pomocí BATCH operací se musí dosáhnout vždy toho žádoucího stavu, kdy se vnořený prvek také načte sám o sobě a přepíše tak již jednou načtená data od nadřízeného prvku. V předešlém případě to znamená, že pracovník, který načítá soubor `A.xml` je povinen načíst poté i soubor `B.xml` do odpovídajícího prvku B a přepsat tak data prvku B načtená ze souboru `A.xml`. Tento způsob však může vést k častým chybám a také dochází ke zbytečnému zdvojení dat v souborech.

Z praktických důvodů technologie EFEM dává přednost prvému způsobu, tj. vyhýbá se striktně vnořeným prvkům CONTROL PACKAGE.

6.1.6.4 PACKAGE a CLASS MODEL AM

Specifické postavení má prvek typu PACKAGE v CLASS MODELU AM. Nejenom, že zde platí stejné principy práce s prvky CONTROL PACKAGE tak, jak byly uvedeny v předešlé kapitole pro libovolný model, ale navíc však zde přistupuje další hledisko a tím je vrstvení systému do prvků COMPONENT.

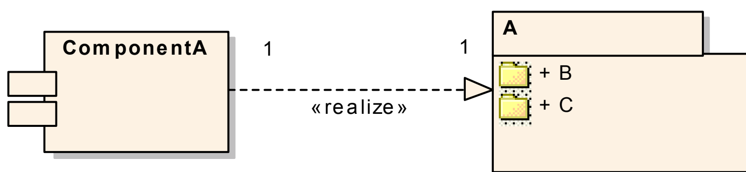
Zavedení necirkulárních prvků PACKAGE v CLASS MODELU AM zavádí prvek CLASS PACKAGE AM. Jedná se o prvek PACKAGE se stereotypem *<<CLASS PACKAGE AM>>*. Tento PACKAGE se vyznačuje tím, že

- jedná se o PACKAGE
- vyskytuje se pouze v CLASS MODELU AM, tj. seskupuje třídy analytické povahy vzniklé v AM
- mezi jednotlivými CLASS PACKAGE AM není zásadně povolena cirkulární reference DEPENDENCY (považováno za chybu modelu)

Rozložení modelu do CLASS PACKAGE AM implikuje možné rozložení tříd do budoucích prvků COMPONENT.

Jeden prvek COMPONENT obsahuje právě jeden prvek CLASS PACKAGE AM, který může obsahovat dalších N prvků CLASS PACKAGE AM pomocí NESTING. Daný prvek COMPONENT je poté přímou realizací D daného nejvyššího prvku CLASS PACKAGE AM.

Situaci znázorňuje následující obrázek:



obrázek 29 Vztah CLASS PACKAGE AM versus COMPONENT.

Prvek CLASS PACKAGE AM, který je realizován prvkem COMPONENT (na předešlém obrázku PACKAGE A), musí být typu CONTROL PACKAGE (pokud tomu tak není, musí se takto dodatečně zavést). Z uvedených pravidel pro prvky typu CONTROL PACKAGE vyplývá, že prvek typu CLASS PACKAGE AM, který je současně typu CONTROL PACKAGE, je takovým prvkem PACKAGE, který realizuje nějaký prvek COMPONENT a je nejvyšším prvkem PACKAGE (root), který nese soubor dané části modelu.

Je zřejmé, že touto konstrukcí se docílí velmi žádaného stavu, kdy určitý prvek COMPONENT realizovaný až do kódu má k sobě přiřazen svůj CLASS MODEL v odpovídajícím XML souboru daného CONTROL PACKAGE. Výsledkem této konstrukce je získání dokumentace dané komponenty od modelu AM až po kód v jednom balíčku.

7. Použití USE CASE MODELU v EFEM

Technologie EFEM zavádí dokument zvaný UC MODELING. Tento dokument používá mimo jiné prvky z USE CASE MODELU podle syntaxe UML.

Cílem dokumentu UC MODELING je získat takové modely a textové popisy, které na úrovni AM popisují úplné chování informačního systému a jeho veškeré funkcionality. Podle syntaxe UML se jedná o nalezení a popis všech případů užití systému. Dokument UC MODELING tak na úrovni AM velmi přesně popisuje to, jak se systém používá a jak se přitom systém chová. Struktura a obsah tohoto dokumentu jsou velmi podobné jako u uživatelské příručky, od čehož je odvozen název tohoto dokumentu (abstraktní příručka uživatele).

Základním prvkem dokumentu UC MODELING je prvek USE CASE odpovídající syntaxi UML. Do češtiny se nejčastěji a také nejpřesněji překládá jako „případ užití". Význam tohoto prvku je přesně takový, jak zní jeho název: Jedná se o jeden případ užití systému. Cílem dokumentu je získat výčet všech případů užití. Tento výčet udává všechna možná užití, tj. všechny případy užití systému a tedy veškerou funkcionalitu systému. Z tohoto hlediska je dokument chápán v popisu jako úplný.

7.1.1. Prvky PROFITABLE USE CASE a CO-USED USE CASE

Technologie EFEM zavádí pro snazší a efektivnější modelování další dodatečnou syntaxi pro prvky USE CASE. Jeden případ užití definovaný v předešlém odstavci se v EFEM nazývá „užitkovým případem užití", ekvivalentně PROFITABLE USE CASE. Nově nalezený případ užití typu PROFITABLE přináší nový užitek systému pro okolí a zvyšuje tak funkcionalitu systému.

Případ užití typu PROFITABLE lze definovat pomocí posloupnosti těchto jevů:

- Vznikne událost v okolí, která vede k neodolatelnému nutkání okolí použít systém (událost vzniká v podniku anebo se jedná o událost času).

- Systém se díky této události použije a proběhne určitý scénář užití (zde se použije případ užití typu PROFITABLE).
- Okolí je uspokojeno daným scénářem (resp. výsledek scénáře spadne do popsané větve nějaké výjimky jako neuskutečnitelný). Případ užití splnilo ve svém hlavním scénáři nějaký užitek.

Spuštění takového případu užití si lze představit jako knoflík na automatu na kávu. Seznam všech případů užití typu PROFITABLE lze chápat jako seznam všech takovýchto knoflíků na automatu.

Kromě případů užití PROFITABLE, které přinášejí přímý užitek pro okolí podle bodů předešlého scénáře, existují podle EFEM ještě „druhé“ případy užití, jejichž nalezení nepřináší přímý užitek a nezvyšuje tak funkcionalitu systému. V technologii EFEM se takovýto případ užití nazývá „vnitřní případ užití“, neboli prvek CO-USED USE CASE. Tyto případy užití jsou důsledkem vzájemného použití uvnitř systému, tj. vznikají sekundárně díky interakcím mezi případy užití. Nalezení a vznik těchto případů užití neznamená zvýšení funkcionality systému, ale vede k lepší vnitřní strukturalizaci mezi případy užití. Využije se přitom opětovná použitelnost (re-use) mezi případy užití.

Rozdíl mezi oběma typy případy užití je velmi podobný jako u funkcí v knihovně. Některé funkce knihovny jsou volatelné „přímo“ uživatelem knihovny a přinášejí tak přímý užitek, jiné funkce (privátní) vznikají sekundárně jako volání uvnitř knihovny a nezvyšují užitek pro uživatele knihovny. Tyto druhé pouze strukturují knihovnu do podoby s vyšším re-use, aby se v knihovně neopakoval kód.

Uvedené rozdělení na PROFITABLE a CO-USED prvky USE CASE je zavedeno v EFEM jako dodatečná syntaxe a slouží k efektivnějšímu postupu vyhledávání prvků USE CASE. V modelu se nemusí tyto prvky odlišovat, jejich rozlišení slouží pouze k efektivnějšímu postupu vyhledávání všech prvků USE CASE.

7.1.2. Dodržení zásady „případ užití je popisem budoucího programu“

Je třeba upozornit na jednu závažnou skutečnost, která se často zanedbává a vede k závažným chybám v modelování s prvky USE CASE.

Protože se při tvorbě USE CASE MODELU navrhuje řešení IS na úrovni abstraktního modelování, tak jeden případ užití, tj. jeden prvek USE CASE, obsahuje popis daného řešení. Prvek USE CASE obsahuje ekvivalentně a doslova v jednom svém scénáři jeden popis programu (a ničeho jiného). Na jeden případ užití lze tedy nahlížet jako na abstraktní slovní popis jednoho budoucího řešení scénáře programu, který se bude řešit na nižších úrovních abstrakce (na úrovních designu a kódování).

Pro autora modelu obsahujícího prvky USE CASE je proto velmi užitečný tento doporučený praktický náhled na prvky USE CASE v technologii EFEM: Jednotlivé prvky USE CASE se chápou jako zadání budoucího řešení pro design a kódování. Co prvky USE CASE obsahují, to se bude programovat, co prvky USE CASE neobsahují, to se programovat nebude.

Samy prvky USE CASE popisují doslova budoucí program. Tento přístup umožňuje jednoduše, logicky a přehledně vymezit, co obsahují prvky USE CASE a co již neobsahují. Tímto se také určuje rozsah řešení v projektu: Bude se programovat právě a jenom to, co obsahují prvky USE CASE daného projektu, nic víc a nic míň. Uvedenou úvahu lze dovést ještě dále: Lze si jednoduše představit, že slovní popis prvku USE CASE ve svém scénáři odpovídá určitému postupu programu při realizaci.

Tento náhled na prvky USE CASE je třeba striktně dodržovat. Mnohdy se autoři modelu obsahujícím prvky USE CASE dopouštějí závažné chyby, která vede k určitému zmatení. Existuje opět dvojí pohled na prvek USE CASE, vnější a vnitřní, tj. opět zde platí základní objektově orientovaný přístup. Vnější pohled na USE CASE reprezentuje užitek prvku USE CASE (tj. co přináší pro venek), vnitřní náhled reprezentuje jeho implementaci ve scénář (jak se USA CASE realizuje). Přitom pohled uživatele (nikoliv programátora) na samotné použití systému se může ve své podstatě jevit z hlediska užitku jako „širší“, než jaký je samotný vnitřní obsah prvku USE CASE. Například při využití externích systémů a jejich funkcionalit může být samotný obsah prvku USE CASE velmi „chudý“, protože daný prvek USE CASE využije hodně funkcionalit externího systému. Z pohledu uživatele se jedná o složitou funkcionalitu, ale uvnitř v popisu prvku USE CASE se nalezne velmi malý scénář (a tedy následně velmi málo kódu) s odvoláním na použití externího systému. Například samotný program pouze převezme údaje od obsluhy, provede nějakou jednoduchou transformaci a zavolá externí systém, předá mu údaje, dostane výsledek a tento vrátí uživateli. V tomto případě se bude programovat „málo“ (scénář je malý), ale systém toho umí „hodně“ díky použití externího systému. Jedná se o klasický příklad vnějšího a vnitřního pohledu.

Je třeba zdůraznit, že v této úvaze lze identifikovat dvojí objektově orientovaný pohled na prvek USE CASE, vnější a vnitřní. Z hlediska vnějšího pohledu (prvek v modelu reprezentuje jeho název jako identifikace) se jedná o celý prvek ve svém vnějším použití „co případ užití umí poskytnout jako jedno užití“, z hlediska vnitřního se jedná o malinký prográmek, který bude naprogramován s použitím interfaců ven k externímu systému. Pro vnější pohled je dostatečný název prvku plus očekávané služby (tj. užitek případu užití), z hlediska vnitřního se jedná o určitý scénář realizace těchto služeb.

7.2 Technika vyhledávání všech prvků USE CASE pomocí rozkladu procesů podniku (BPM)

Základní rozdělení dokumentu UC MODELING vychází z potřeby nalézt takovou techniku, která napomůže systematicky vyhledat všechny případy užití systému. U složitějších informačních systémů nelze totiž nalézt všechny případy užití pouhým výčtem a je třeba použít nějakou systematickou a sofistikovanou techniku vyhledávání všech případů užití. Existuje několik technik vyhledávání všech případů užití.

Vždy se však postupuje tak, že se prvotně vyhledají všechny veřejné případy užití PROFITABLE (tj. užitkové případy užití) a teprve v druhém kroku se vyhledávají vnitřní případy užití jako výsledek vzájemného použití případů užití mezi sebou. V druhém kroku při vyhledávání vnitřních případů užití se již funkcionalita systému nezvyšuje, pouze se jinak vnitřně strukturují případy užití. Při vyhledávání vnitřních případů užití se některé případy užití se doslova vytknou z několika jiných případů užití podobně jako při volání funkcí, ale přitom se funkcionalita, tj. užitkovost systému nezvýší.

Technika rozkladu napomáhá efektivně nalézt všechny užitkové případy užití (všechny knoflíky na automatu na kávu). Nejefektivnější, nejjednodušší a přitom logicky nejlépe vypovídající se jeví technika vyhledávání všech případů užití informačního systému pomocí techniky rozkladu procesů podniku.

Zavádějí se prvky BUSINESS PROCESS v modelu typu BUSINESS PROCESS MODEL (BPM) a přitom se pracuje s dekompozicí těchto procesů. V této technice vyhledávání případů užití informačního systému se jakoby „odbočí“ a využije se „dočasně“ model jiného typu s jiným předmětem modelování, než je informační systém. Je třeba proto ještě jednou zdůraznit, že při tvorbě těchto modelů typu BPM není předmětem modelů informační systém, ale okolí informačního systému, tj. podnik.

Jeden prvek BUSINESS PROCESS je chápán jako stav, po který trvá určitá činnost neboli aktivita podniku, tj. aktivita okolí. Laicky řečeno, přitom však velmi přesně a správně chápáno, prvky BUSINESS PROCESS popisují, co relevantního se děje v okolí, když se používá informační systém. Technika vyhledávání USE CASE v EFEM využívá tuto techniku. Zavedou se diagramy modelu podniku a z nich se odvozuje, jaké budou existovat případy užití informačního systému. Je třeba poznamenat, že pro designéra a programátora jsou tyto modely BPM při přechodu do nižších úrovní abstrakce absolutně nezajímavé a považuje je za „omáčku“. Modely BPM určují kontext použití systému od okolí a pro vývojáře designéra a programátora jsou zajímavé až samotné případy užití systému (prvky USE CASE), které popisují, co se bude konkrétně programovat.

Může se však stát, že se modely podniku BPM použijí v dokumentech jiného typu, například v obchodních dokumentech, v nabídce zákazníkovi apod. Jejich vypovídající schopnost o kontextu použití informačním systému je natolik silná, že je mnohdy vhodné tyto dokumenty takto použít také pro tyto účely

Technologie EFEM používá techniku BPM a upravuje ji tak, aby směřovala co nejrychleji k výsledku pro vývojáře nejdůležitější, tj. k nalezení všech případů užití informačního systému.

Poznámka: Existují návody a metodiky, které používají jiné prvky pro modelování podniku než BUSINESS PROCESS. Nejznámější z nich je zavedení prvku BUSINESS USE CASE (prvek případu užití podniku) namísto prvku BUSINESS PROCESS. Praktické zkušenosti s použitím prvků BUSINESS USE CASE v několika projektech však přivedly při zavádění technologie EFEM k doporučení raději používat prvky BUSINESS PROCESS než prvky BUSINESS USE CASE. Použití prvků BUSINESS USE CASE vede velmi často k velkému zmatení v identifikaci prvků USE CASE podniku a prvků USE CASE informačního systému. Navíc vznikají často omyly ohledně identifikace prvků okolí systému resp. okolí podniku. Z toho důvodu je v EFEM zvolen postup využívající raději prvky BUSINESS PROCESS, čímž je jasně a striktně oddělena logika chodu podniku a logika chodu informačního systému. Tato logika je v tomto případě jasná a transparentní: Pokud hovoříme o prvku typu BUSINESS PROCESS, hovoříme o logice podniku, pokud hovoříme o prvku typu USE CASE, hovoříme o informačním systému, tj. o případu užití programu.

7.2.1. Podstata rozkladu BPM

Myšlenka postupu rozkladu BPM vedoucí k nalezení všech případů užití je následující:

Naleznou se všechny procesy podniku BUSINESS PROCESS (pozor, nikoliv informačního systému), které budou podporovány informačním systémem. Většinou zákazník resp. konzultant je schopen tyto procesy určovat poměrně přesně anebo je schopen o nich relevantně diskutovat. Po nalezení všech těchto procesů podniku lze určit, kterými případy užití informačního systému budou jednotlivé procesy podniku podporovány. Tím jsou nalezeny odpovídající případy užití informačního systému. Pokud se naleznou všechny procesy, které mají být podporovány systémem a k nim se určí odpovídající jejich případy užití systému, logicky by měly být nalezeny všechny případy užití systému.

Technika rozkladu využívá té skutečnosti, že procesy a tedy i procesy podniku lze rozkládat shora dolů dekompozicí, protože kompozice (a opačně dekompozice) procesů je z hlediska modelování v UML „povolená operace“. Horní proces podniku je tedy chápán jako souhrn všech procesů podniku, které budou podporovány informačním systémem, tj. celým programem. Název tohoto souhrnného procesu můžeme ztotožnit s názvem informačního systému a hovoříme o něm jako o všech procesech podniku, které systém podporuje. Provede se logický rozklad této velké

skupiny procesů na procesy jako menší „podprocesy“. Vzniknou tak menší skupiny procesů podniku jako menší procesy a získá se tak první úroveň rozkladu procesů podniku. Takto se postupuje shora dolů až do úrovně, kdy se jednoduchým výčtem začnou specifikovat případy užití, které podporují daný proces podniku. Je třeba zdůraznit, že tento rozklad je čistě pomyslný a pouze logický. Rozklad procesů podniku vůbec nesouvisí s rozložením systému na komponenty informačního systému apod. Rozložení dekompozice není proto jednoznačné, jiný autor by mohl dospět k jinému rozkladu. Nalezený seznam všech případů užití však musí být pro tentýž informační systém stejný

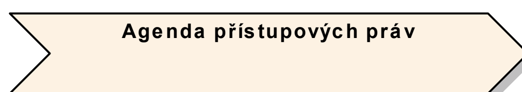
7.2.2. Syntaxe dekompozice procesů podniku

Zde je popsána pouze syntaxe podle UML. Konkrétní postup pro EA je uveden v odpovídajících postupkách.

7.2.2.1 Prvek BUSINESS PROCESS

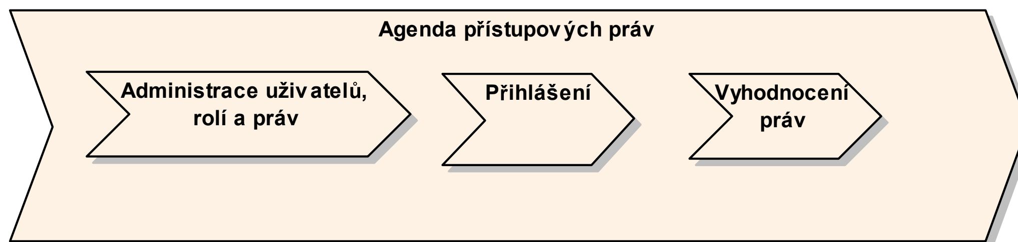
Jeden prvek proces podniku, tj. BUSINESS PROCESS, je chápán jako stav, po který trvá určitá činnost podniku. Syntaxe použití prvku je v UML odvozena od prvku ACTIVITY. Prvek BUSINESS PROCESS je podle syntaxe UML zaveden jako prvek ACTIVITY se STEREOTYPEM <<*business process*>>. Protože se vlastně jedná o prvek typu ACTIVITY, platí pro něj stejná syntaktická pravidla jako pro typ prvku ACTIVITY.

Pro prvek BUSINESS PROCESS lze zvolit jiný prezentační element, než používá „klasické“ UML pro prvek typu ACTIVITY (původně ovál). Jeví se jako vhodné použít všeobecně používaný grafický element „šipky“, například takto:



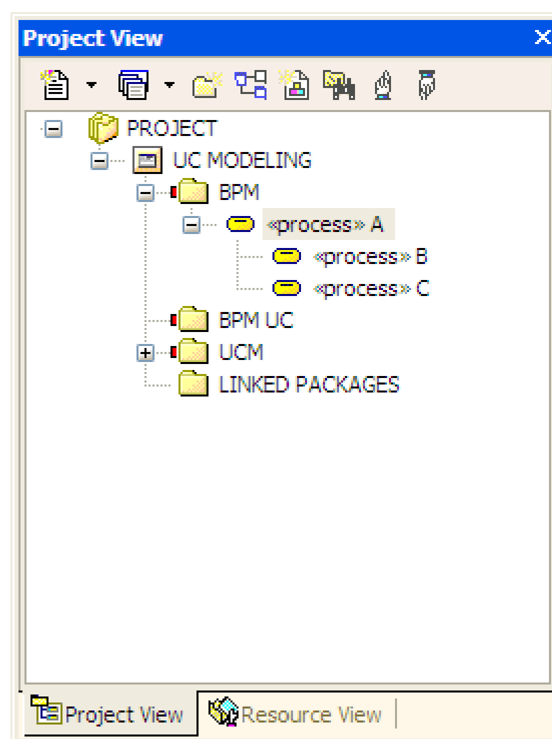
obrázek 30 Prvek ACTIVITY se stereotypem PROCESS se chápe jako aktivita podniku

Diagram rozkladu procesu pak vypadá takto:



obrázek 31 Příklad rozkladu procesů podniku

Přitom ve vztahu mezi procesy existuje vztah nadřízenosti prvků COMPOUND ACTIVITY a SUBACTIVITY (proces znázorněný jako větší je COMPOUND ACTIVITY a obsahuje svoje SUB-ACTIVITY. To se znázorní jednak podle vzoru předešlého diagramu vložením prvků do sebe, ale také vložením prvků do sebe ve stromu PROJECT VIEW:



obrázek 32 Rozklad procesů v PROJECT VIEW

7.2.3. Rozklad procesů a kapitoly uživatelské příručky

Jako dobrou pomůcku při rozkladu procesů podniku lze použít jednoduchou a názornou představu, že se tímto postupem tvoří kapitoly budoucí uživatelské příručky. Skladba kapitol této příručky odpovídá jednotlivým úrovním rozkladu procesů podniku.

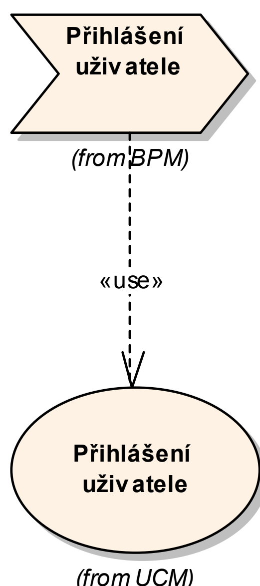
Představa uživatelské příručky není vůbec vzdálená od postupu rozkladu procesů: Uživatelská příručka není nic jiného, než návod „co dělat se systémem“, tj. obsahuje popis procesů podniku z hlediska uživatele. Kapitoly člení tyto popisy procesů do logicky uspořádaných skupin, což lze ztotožnit s rozkladem procesů.

Na předešlém obrázku tedy lze rozpoznat odpovídající první kapitoly uživatelské příručky: „A“, podkapitoly „B“ a „C“.

7.2.4. Syntaxe pro vztah prvků USE CASE a procesů podniku

Cílem rozkladu procesů podniku je nalézt všechny případy užití informačního systému. Dekompozice procesů stále blíže specifikuje jemnější procesy. V určité úrovni se získá taková úroveň dekompozice, že lze vyjmenovat případy užití podporující daný proces.

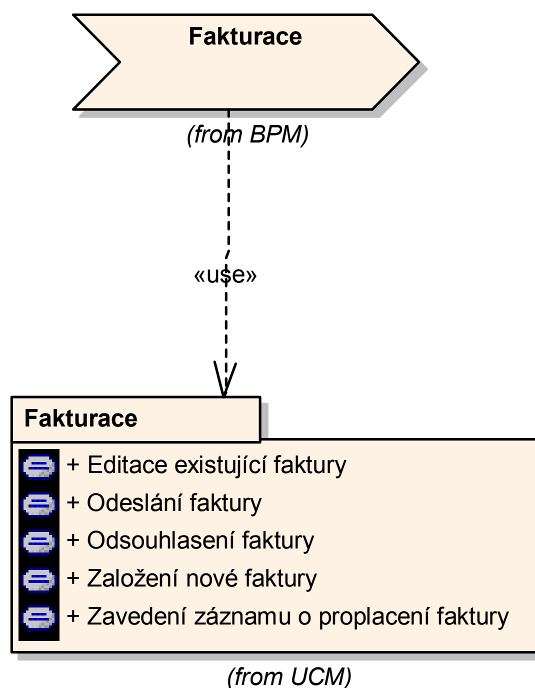
V případě, že se nalezne pouze jeden případ užití podporující daný proces, provede se vazba DEPENDENCY od procesu k prvku USE CASE přímo mezi jedním prvkem proces a prvkem USE CASE. Technologie EFEM doporučuje použít vztah DEPENDENCY ve směru od procesu směrem k PACKAGE prvků případů užití a použít STEREOTYPE <<*use*>>:



obrázek 33 Podpora IS v procesu podniku

Uvedený obrázek je grafickým vyjádřením věty: Proces podniku „Přihlášení uživatele“ používá jeden případ užití nazvaný také „Přihlášení uživatele“. Doporučuje se, aby oba názvy byly buď úplně shodné anebo velmi blízké. Prvky proces a případ užití mají však přes shodu názvů jiný význam: Zatímco proces znázorňuje chování a aktivitu podniku (tj. daného prostředí), případ užití znázorňuje užitek a chování samotného informačního systému, na který se daný proces v určitém okamžiku obrací. Z toho je patrná i možná shoda názvů obou prvků.

Může se stát, že zobrazí nikoliv jeden, ale několik případů užití, které podporují daný proces. Zavede se jeden prvek PACKAGE obsahující odpovídající prvky USE CASE, které podporují daný proces. Tato skutečnost se vyjádří tak, že se tento jeden prvek PACKAGE se v modelu prováže s daným procesem. Obrázek podpory procesu daným informačním systémem potom vypadá například takto:



obrázek 34 Několik prvků USE CASE podporuje jeden souhrnný proces

Uvedený obrázek je vyjádřením věty: Proces fakturace je podporován několika případy užití, vyjmenovanými v daném prvku PACKAGE.

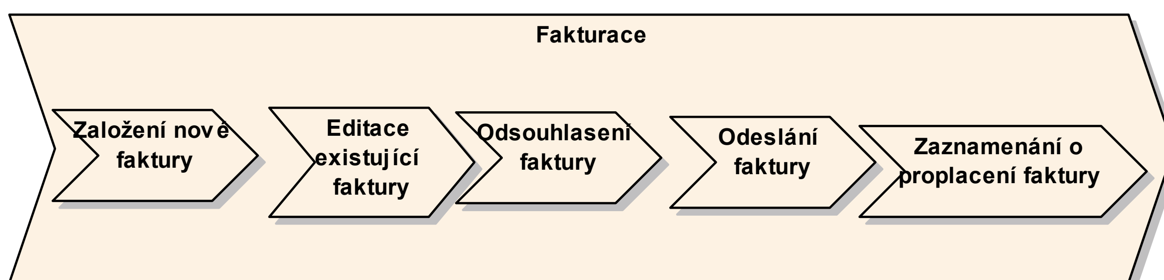
V modelování je třeba od sebe striktně oddělovat problematiku „co se děje v podniku" (proces Fakturace) a co „umí systém" (skupina případů užití). Vazba DEPENDENCY v tomto případě ukazuje použití ve směru od procesu podniku k systému.

Vždy lze pokračováním dekompozice procesů dojít k takovým procesům, kdy jeden proces je podporován jedním prvkem USE CASE a nepoužít tak PACKAGE jako v předešlém příkladu. V předešlém případě by bylo možné pokračovat v rozkladu dále na další procesy podniku, které odpovídají daným případům užití.

7.2.5. Modely chodu procesů podniku

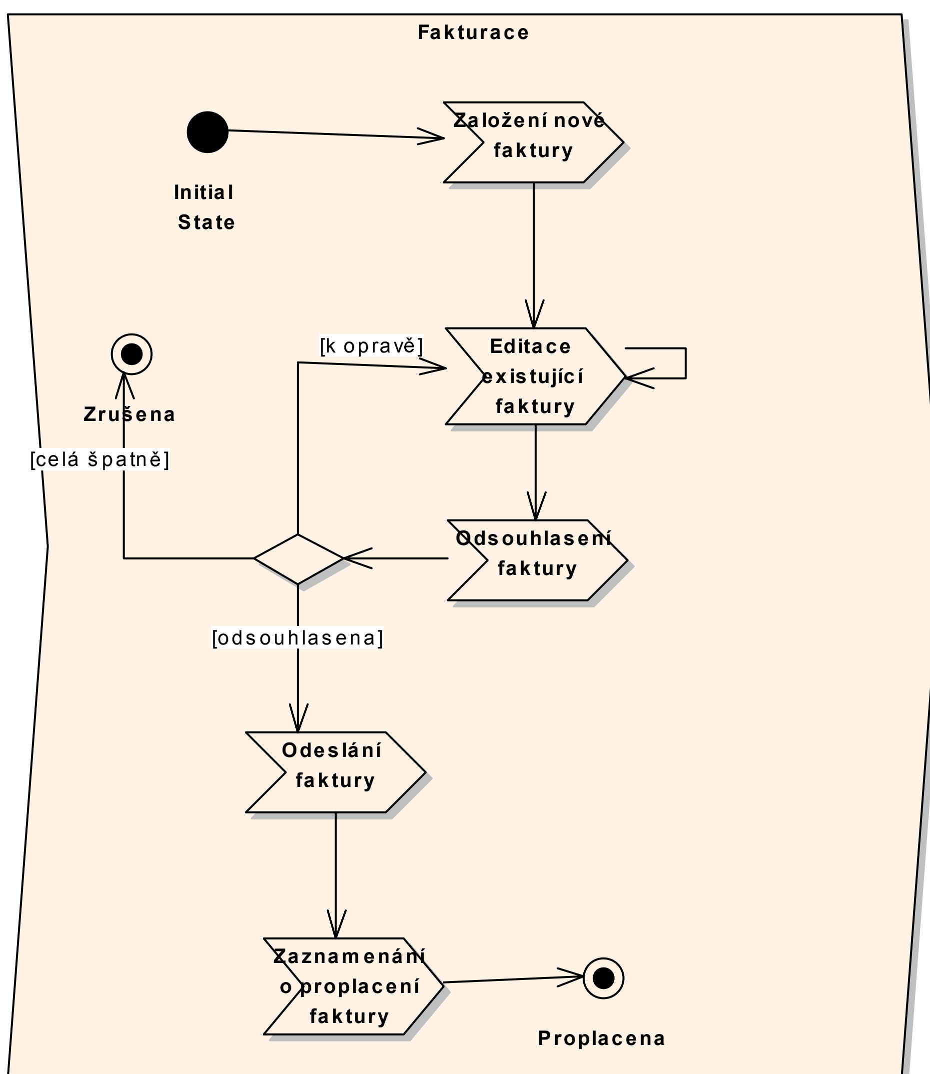
Uvedenou situaci s fakturací by bylo možné namodelovat v BPM ještě podrobněji s dalším rozkladem na procesy „Založení nové faktury“, „Odsouhlasení faktury“ apod.

Pokud jsou tyto procesy známy z povahy věci, stačí znázornit rozklad takto:



V některých případech nelze nalézt procesy podniku pouhým výčtem rozkladu nadřízeného procesu jako v předešlém příkladu. Ne vždy je totiž tato znalost dostatečná a ne vždy je situace jasná a přehledná. V tom případě je vhodné pro nalezení další úrovně dekompozice modelovat také navíc tzv. „chod procesů podniku“.

Modelování chodu procesů je vhodné tam, kde se jedná o chod složitých procesů, například složité účetní procesy, složité služby v bance (práce se směnkou, práce se zahraničním akreditivem apod.), nebo například chod služby operátora mobilních telefonů (předplacení lístků do kina a vydání lístků přes mobil a WEB) apod. Odpovídající diagram by pak vypadal takto:



obrázek 35 Příklad na rozklad s chodem procesu

Oba diagramy na dvou předešlých obrázcích jsou v podstatě shodné (oba totiž vyjadřují dekompozici), pouze druhý podrobněji rozepisuje chod daného procesu a je bohatší o další prvky, které první diagram s pouhým rozkladem nemá.

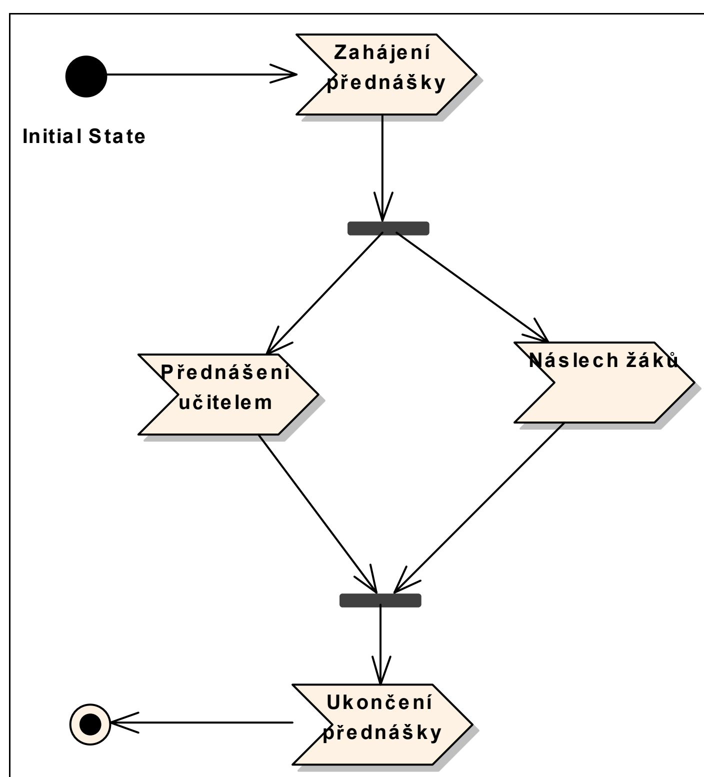
Praxe ukazuje, že druhý podrobnější popis i s chodem procesů se efektivně používá v těchto situacích:

- cílem je (mimo jiné) ukázat chod procesů služeb jako produkt (například služba apod.)
- na přímou žádost zákazníka (odběratele IS)
- u velmi složitých procesů, kdy nalezení chodu napomáhá najít „podprocesy“ složeného procesu
- jako pěkný okrasný doplněk obchodní dokumentace

7.2.6. Syntaxe chodu procesů v EFEM

Pro vyjádření chodu procesů jsou z hlediska EFEM dostatečné tyto syntaktické prvky:

- BUSINESS PROCESS, stav činnosti neboli aktivity podniku (viz předešlé kapitoly)
- COMPOUND PROCESS (složený proces) a jeho SUB-ACTIVITY, k vyjádření prvku je třeba použít jednak vložení jednoho prvku do druhého v diagramu a jednak vložení prvku ve stromu PROJECT VIEW.
- FLOW (také TRANSITION) je přechod z jednoho stavu aktivity do druhého stavu aktivity, značí se šipkou. Prvek FLOW obsahuje EVENT (nepovinné), což je událost, vyvolávající snahu k opuštění stavu (trigger) aktivity, obsahuje GUARD, což je omezující podmínka typu BOOLEAN. Tato podmínka musí být splněna, aby došlo k přechodu. Pokud není splněna, k danému FLOW (přechodu) nedojde. GUARD je viditelný u FLOW jako text (BOOLEAN) v hranatých závorkách. Většinou se používá pro větvení. Příklady na FLOW s GUARDEM viz obrázek chodu fakturace.
- DECISION (kosočtverec) neboli rozhodnutí je větvení pro FLOW. Vchází do něj jeden prvek FLOW a vychází z něj několik prvků FLOW. Pomocí prvků GUARD se u těchto FLOW identifikuje, který vycházející prvek FLOW se v dané situaci realizuje. Jednotlivé FLOW jsou ve svých prvcích GUARD vzájemně disjunktní, tj. z daných prvků FLOW vycházejících z DECISION se realizuje vždy maximálně jeden. Jako příklad na DECISION viz obrázek BPM pro fakturaci.
- SYNCH(H) resp. SYNCH(V) v UML také označováno jako FORK (vidlička). Prvek slouží v BPM k vyjádření spolupráce u paralelních procesů. Pokud do prvku vchází jeden prvek FLOW a vychází z něj několik prvků FLOW, pak procesy na koncích vycházejících FLOW běží paralelně a naopak, pokud několik FLOW vcházejí do tohoto prvku, znamená to synchronizaci procesů. Příklad použití prvku FORK (SYNCH(H)):



obrázek 36 Paralelní procesy a použití syntaxe FORK

- Jeden počáteční a několik koncových bodů (černý kruh a černé kruhy s kružnicí)
- Uvedený výčet je dostatečný pro popisy chodů, které mají za úkol najít rozklad procesů a následně prvky USE CASE. Pokud však je cílem získat podrobné modely BPM, kdy se popisují podrobnosti (události, objekty podniku apod.) je možné rozšířit syntaxi o další prvky, které jsou uvedeny v dokumentu: http://www.sparxsystems.com.au/WhitePapers/The_Business_Process_Model.pdf

7.3 USE CASE MODEL

Další částí dokumentu UC MODELING je část zvaná USE CASE MODEL. Tato část obsahuje modely typu USE CASE MODEL podle syntaxe UML včetně interakcí mezi prvky USE CASE a vůči prvkům ACTOR.

7.3.1. Prvek USE CASE v USE CASE MODELU

Případ užití USE CASE je cílovým elementem dokumentu UC MODELING. Jeden případ užití USE CASE lze chápat přesně tak, jak jej vyjadřuje jeho překlad: případ užití systému.

Je třeba znovu zdůraznit dvojí pohled na prvek USE CASE: Zvně se prvek USE CASE chápe jako užití, jinak řečeno jako prvek chování systému přinášející užitek. Platí tyto dvě rovnice odpovídající pohledu z OOAP:

- ve vnějším pohledu platí „jeden USE CASE = jeden užitek systému“
- ve vnitřním pohledu platí „jeden USE CASE = jeden scénář“

Chování celého informačního systému (tj. celého programu) je popsáno pomocí scénářů všech prvků USE CASE, tzv. USE CASE SCENARIO. Díky prvkům případů užití se získává velmi přesný a přitom abstraktní popis informačního systému srozumitelný všem účastníkům projektu.

Prvek USE CASE se označuje pomocí elipsy (vnější pohled na prvek):

obrázek 37 Prvek USE CASE

Technologie EFEM u každého prvku USE CASE specifikuje tyto údaje:

- jednoznačný název (povinné)
- scénář tvořený pojmy z problémové oblasti (povinné) (USE CASE SCENARIO). Scénář realizuje plnění případu užití konkrétním chováním systému, implementuje případ užití.
- podmínky typu constraint zvané PRECONDITION a POSTCONDITION (nepovinné).

7.3.1.1 Název prvku USE CASE

Název je povinný a je jednoznačný. Volí se výstižný vzhledem k případu užití a nemusí být povinně velmi krátký. Někdy je výhodnější zvolit i delší název. Zadává se

jako podstatné jméno slovesné s několika přídavnými jmény a předměty resp. jako podstatné jméno vyjadřující nějakou činnost. V názvu se musí vyskytovat pojmy z problémové domény, tj. nesmí se v něm vyskytovat jakékoliv technologické názvy určující způsob řešení (zakázané pojmy jsou tabulka, soubor apod.).

7.3.1.2 USE CASE SCENARIO a jeho syntaktická pravidla

Každý případ užití je ve svém chování realizován určitým scénářem, tzv. USE CASE SCENARIO. Jedná se o slovní popis chování, které vede ke splnění potřeby u případu užití. Je třeba zdůraznit, že popis obsažený v USE CASE SCENARIO je v zásadě popisem části programu, který bude realizován v nižší úrovni abstrakce a ničeho jiného. Tento popis podléhá určitým syntaktickým pravidlům, která je třeba dodržovat.

Technologie EFEM používá pro tato pravidla vzory, tzv. SCENARIO PATTERNS.

7.3.1.3 Vzory pro scénáře prvků USE CASE, SCENARIO PATTERNS

Technologie EFEM zavádí pro vyjádření některých opakujících se scénářů vzory scénářů (SCENARIO PATTERNS). Tyto vzory vyjadřují ustálené části scénářů, které se díky povahám vazeb mezi informacemi vždy opakují.

Vzor SCENARIO PATTERN lze použít těmito způsoby:

- text vzoru se přenese přes schránku do textu scénáře a vyplní se odpovídající části proměnných vzoru. Proměnné vzoru jsou označeny pomocí závorek <> a musí být nahrazeny konkrétními účastníky vzoru. Použití přes kopii není z hlediska opětovné použitelnosti úplně korektní, protože se nejedná o odkaz, ale o kopii. Ukazuje se však z praxe, že se jedná o efektivnější způsob použití prvků SCENARIO PATTERNS v USE CASE SCENARIO než použití odkazu.
- druhý způsob použití SCENARIO PATTERNS spočívá ve znalosti vzoru „zpaměti“ a jeho následném použití bez přenesení přes schránku. Používá se tedy přímá editace textu popaměti stejným způsobem, jako by byl kopírován. U většiny vzorů lze doporučit právě tento způsob použití vzorů.

V syntaxi vzoru v následujících kapitolách platí následující označení: Text vzoru, který má být použit, je napsán proloženým tiskem a ohraničen rámečkem. Prvky, které je třeba nahradit konkrétními hodnotami, se označují závorkami < >. Volitelný text, který se za určitých okolností vynechává, je označen závorkami [ ].

7.3.1.3.1 Vzor pro spojený pojem

Text vzoru

<B> koho čeho <A>

Poznámky k použití

Výraz „koho čeho“ vyjadřuje vztah druhého pádu a v použití vzoru se vynechává. Pro použití vzoru se volí výhradně postup v předešlém odstavci uvedený jako 2, tj. vzor je nutno si pamatovat, je neefektivní jej používat přenosem přes schránku. Vzor se používá při zavádění tzv. spojených pojmů, které vyjadřují vztahy pojmů (použitelné vztahy pojmů viz kapitola CLASS MODEL V AM).

Vztah složeného pojmu *<B> koho čeho <A>* může být jeden z následujících vztahů:

- B je atribut A (například *rodné číslo fyzické osoby*)
- B je obsažen v kompozici ku 1 v A (například *adresa trvalého bydliště fyzické osoby*)
- B je ve vztahu běžné asociace (číselníková vazba) vůči A (například *barva auta*)
- B je obsažen v kompozice ku N a vyjadřuje se poté množným číslem B (například *řádky faktury*)
- B je vztah přes asociativní třídu vůči A, v tom případě bývá vztah spojeného výrazu doplněn o vystižení tohoto vztahu (například *povolené typy bankovních služeb pro vybraný typ osoby*)

Vzor pro složený pojem je rekurzivně použitelný a tedy mohou vznikat složené pojmy ze složených pojmů, například *<C> koho čeho <B> koho čeho <A>* apod.

Příklad z editace faktury:

Ve větě *„Obsluze se zobrazí ulice sídla dodavatele vybrané faktury“* existuje složený pojem ve znění *„ulice sídla dodavatele vybrané faktury“*, který vznikl na základě tohoto vzoru a odpovídá v modelu tříd nějakým vztahům, zde například je obrazem vztahu:

- ulice je atribut sídla
- sídlo je v kompozici ku jedné vůči dodavateli
- dodavatel je v běžné asociaci vůči vybrané faktuře

Příklad ze Zadání nového plátna do pletařské výroby (informační systém evidence výroby textilu):

Ve větě: *„Obsluha vybere druh plátna a obsluze se zobrazí seznam povolených barev pro vybraný druh plátna*“ se vyskytuje složený pojem *„povolené barvy*

vybraného druhu plátna". Jedná se o výraz ze vzoru odpovídající asociativní třídě mezi druhem plátna a barvou.

7.3.1.3.2 Vzor pro scénář naplnění běžné asociace (číselníkové vazby) obsluhou výběrem ze seznamu

Text vzoru

Obsluze se zobrazí seznam <B> (<B1>, <B2>, <B3>,...). [Seznam je seřazen podle kritérií <kritéria>]. Obsluha vybere <B>. Vybraný <B> se dosadí do daného <A> jako <role B> .

Poznámky k použití

Tento vzor se používá při naplnění běžné asociace obsluhou výběrem ze seznamu. Při použití vzoru do scénáře se vyplní název pojmu <B>, čímž se určí z jakého seznamu se vybírá. Do závorky se uvádějí prvky, které se mají zobrazit (<B1>, <B2>, <B3>,...), což jsou složené pojmy vůči B. Pokud existují požadavky na seřazení seznamu, uvedou se za závorkou. Vyplní se pojem <A> a dosadí se <role B>, která určuje, proč se vlastně <B> vyhledávalo, tj. v jaké <B> je roli vůči <A>.

Příklad: dosazení ručitele do bankovní půjčky:

Obsluze se zobrazí seznam osob (rodné číslo, příjmení, jméno) seřazené buď podle rodného čísla nebo příjmení podle volby obsluhy. Obsluha vybere osobu a ta se dosadí do dané půjčky jako ručitel.

Příklad dosazení barvy do auta:

Obsluze se zobrazí seznam barev (kód, text). Obsluha vybere barvu a ta se dosadí do daného auta jako barva auta.

7.3.1.3.3 Vzor pro scénář naplnění běžné asociace zadáním identifikátoru a nalezením prvku

Text vzoru

Obsluha zadá <C>. Na základě <C> se nalezne v seznamu <B> jeden prvek <B> Vybraný <B> se dosadí do daného <A> jako <role B> .

Poznámky k použití

Tento vzor se používá při naplnění běžné asociace vyhledáním prvku v seznamu pomocí zadaného identifikátoru namísto výběrem ze zobrazeného seznamu. Vše ostatní zůstává stejné, jako u předešlého vzoru.

7.3.1.3.4 Vzor pro scénář přidání resp. odebrání prvku agregace ku N v USE CASE SCENARIO

Text vzoru

Obsluha může přidat (odebrat) <B>. [Obsluha může hned editovat nové <B>, viz <název USE CASE>]

Poznámky k použití

Používá se v případě vazby agregace 1:N a označuje scénář práce obsluhy s prvky seznamu v agregaci ku N. V případě přidání prvku a možnosti okamžité editace nového prvku se zvolí volitelná část scénáře s odkazem na jiný USE CASE. V tomto jiném prvku USE CASE je popis editace prvku <B>. V diagramu se pro tento odkaz na jiný USE CASE volí interakce INCLUDE (interakce viz odpovídající kapitola).

Příklad z editace faktury:

Obsluha může přidat nový řádek faktury. Obsluha může hned editovat nový řádek faktury, viz „Editace řádku faktury“.

7.3.1.3.5 Vzor pro scénář editace v USE CASE SCENARIO

Text vzoru

Obsluha zadá <A1>, <A2>, <A3> ...

Poznámky k použití

Prvky *<A1>, <A2>, <A3>* odpovídají zadaným hodnotám. Tyto texty budou přes formulář přeneseny k atributům odpovídajících výskytů informace s odpovídajícími názvy resp. se s nimi bude pracovat v dalším scénáři vůči těmto atributům (například vyhledávání prvků, verifikace prvků apod.). Nemusí se tedy vždy jednat o pouhé přenesení z editačních polí do atributů prvků, ale také o složitější scénáře zpracování.

Příklad na použití vzoru v USE CASE SCENARIO „přihlášení obsluhy“:

Uživatel zadá uživatelské jméno a heslo. Po odsouhlasení se na základě zadaného uživatelského jména a zakryptovaného hesla nalezne daný uživatel v seznamu uživatelů, pokud nenalezen, tak ERR1 atd.

7.3.1.3.6 Vzor pro scénář zadání prvku v kompozici ku jedné v USE CASE SCENARIO

Text vzoru

Obsluha zadá <A>, viz případ užití <název USE CASE>

Poznámky k použití

Prvek v kompozici ku jedné se edituje v jiném případu užití, na který se provádí odkaz. V diagramu se použije interakce INCLUDE (interakce viz odpovídající kapitola).

Příklad:

Obsluha zadá adresu trvalého bydliště, viz „Editace adresy“

7.3.1.3.7 Vzor pro scénář hromadného zpracování v USE CASE SCENARIO

Text vzoru

Pro každý <A> [,který <kritéria>,] se provede <pojmenování operace>, viz případ užití <název USE CASE>.

Poznámky k použití

Vzor slouží k vyjádření scénářů, kdy je třeba provést zpracování N výskytů. Pojmenují se výskyty ve složeném pojmu <A>. Pokud existují kritéria výběru výskytů, uvedou se. Pojmenuje se operace, která se provádí. Zpracování jednoho výskytu se uvádí v jiném případu užití, na který se provede odkaz. V diagramu se uvede jako interakce INCLUDE (viz dále).

7.3.1.3.8 Vzor pro scénář vyhledání instance v seznamu

Text vzoru

Na základě daných <A1, A2, ...> se v seznamu <B> naleznou (nalezne) <B>.

Poznámky k použití

Vzor popisuje část scénáře vyhledání instance v seznamu. Může být použit například při vyhledání instance po zadání hodnot obsluhou anebo při importu údajů od externího systému.

7.3.1.3.9 Specifické scénáře v USE CASE SCENARIO

Kromě ustálených scénářů vzniklých z opakujících se situací existují scénáře specifické, které vyjadřující neopakující se specifické situace a nejsou tvořeny převzetím ze vzorů scénářů.

7.3.1.4 Použití BASIC PATH a ALTERNATE PATH pro EXCEPTION FLOW

Scénář daného případu užití by měl popisovat chování systému a to i v těch případech, kdy dojde k výjimkám a mezním stavům ve scénáři. Jedná se o takové situace, kdy se například obsluha pokusí zadat nepřípustnou hodnotu (příkladem je zadání nuly v částce převodního příkazu), nebo kdy dojde obecněji k vyhodnocení určitých hodnot a zjistí se nepřípustnost resp. je třeba upozornit na danou hodnotu (např. upozornění na nesplnění modulo 11 u rodného čísla nebo čísla účtu apod.). Ošetření takovýchto mezních stavů v dané větvi scénáře se nazývá EXCEPTION FLOW.

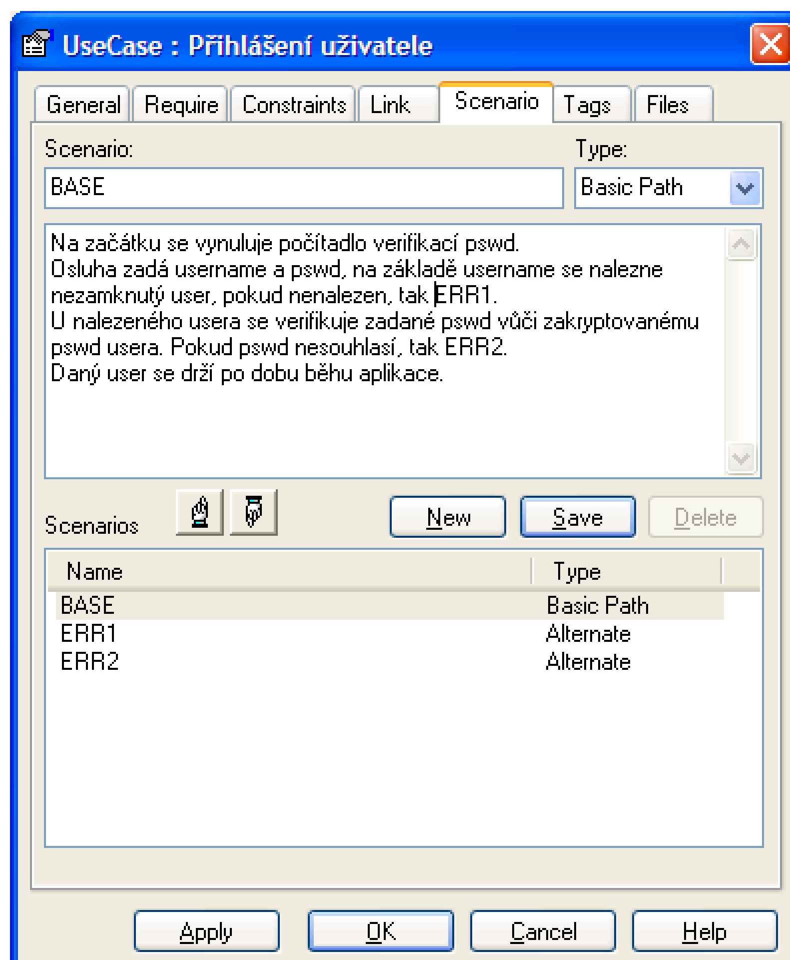
Z hlediska efektivního a přehledného zápisu je výhodnější tyto větve chodu scénáře EXCEPTION FLOW neumísťovat do stejné pozice jako hlavní chod scénáře. Pokud se totiž obě větve dají do téže pozice, čtení scénáře se stává velmi nepřehledným.

Používá se tato konstrukce:

Hlavní větev chodu se umístí do posice tzv. hlavního scénáře V tomto scénáři se provedou odkazy na větvení do tzv. vedlejších scénářů, které se označí návěštím, například ERR<číslo> apod. Celý scénář se tak rozčlení na jednotlivé části:

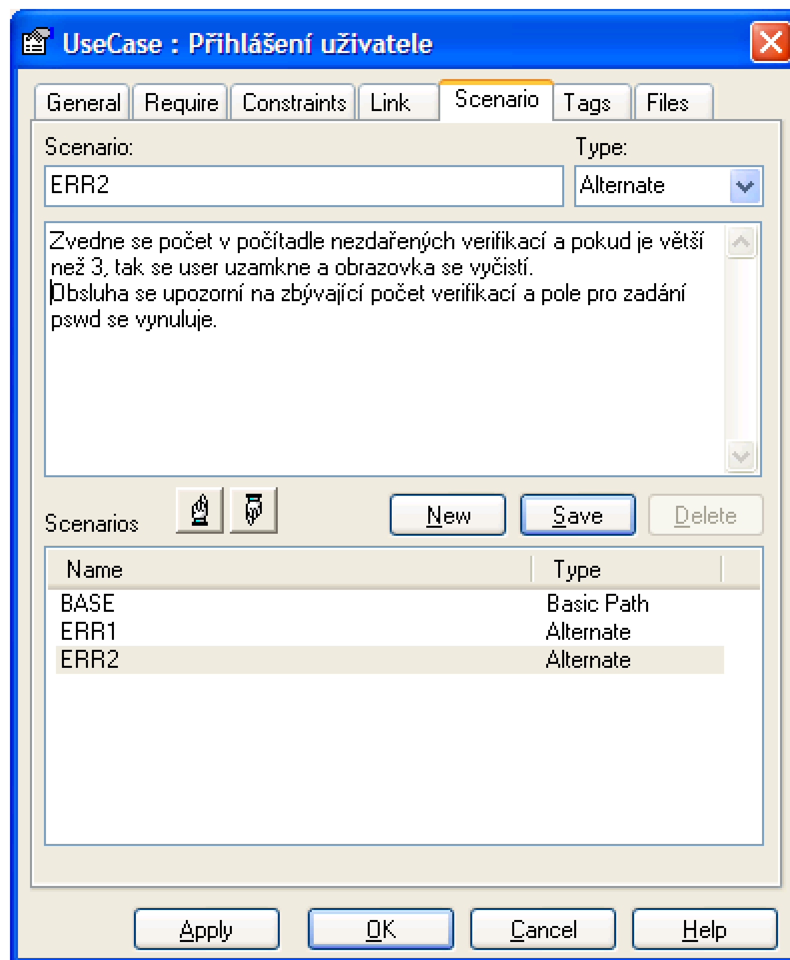
- BASE scénář
- Několik ERR scénářů, které se volají z hlavního BASE scénáře pomocí návěští

Příklad:



obrázek 38 Hlavní větev pro případ užití „Přihlášení uživatele“

Druhý obrázek ukazuje scénář v EXCEPTION FLOW:



obrázek 39 Exception flow

7.3.2. Interakce mezi prvky USE CASE v USE CASE MODELU

Interakce mezi případy užití je směrovou vazbu od jednoho případu užití k druhému případu užití a vyjadřuje doslova použití jednoho případu užití druhým případem užití. V textu scénáře se interakce projeví odkazem typu „viz *název případu užití*", tj. odkazem na druhý případ užití. Čtenář je povinen v daném místě provést odskok ve čtení scénáře, pokračovat ve čtení textu scénáře druhého použitého případu užití od jeho začátku a poté při ukončení druhého scénáře se vrací zpátky do prvého scénáře. Druhý scénář se použije v odkazu jako celek, nelze se odkázat skokem doprostřed scénáře. Pokud vznikne taková potřeba, model není v pořádku a prvek USE CASE se musí rozdělit na menší případy užití.

Všechny typy interakcí mezi případy užití podléhají tomuto pravidlu odskoku ve scénáři. Typy interakce specifikované syntaxí UML (viz dále) určují pouze důvody a způsoby vzájemného použití mezi případy užití. Z tohoto hlediska je informace o typu interakce sice důležitou, ale pouze druhotnou informací, která udává blíže typ použití mezi případy užití.

7.3.2.1 Vnější a vnitřní pohled na prvek USE CASE v interakci

Pokud nějaký prvek USE CASE A používá druhý prvek USE CASE B, tak celkový scénář celého prvého prvku A je nutno chápat jako scénář napsaný v textu scénáře A plus scénář za odkazem v B. Interakce mezi prvky USE CASE tedy ukazuje vnitřní strukturu, která je z vnějšího pohledu na prvek A nezajímavá. Prvek A se chápe jako celek i s prvky, na které se odkazuje.

Častou chybou je nepochopení tohoto principu. Pojem „interakce mezi případy užití" je zavádějící v tom smyslu, že pokud se takto hovoří o interakci mezi prvky USE CASE, má se tím na mysli vlastnost „jak je jeden prvek USE CASE vnitřně strukturován". Interakce mezi případy užití vždy ukazuje informaci ohledně jednoho prvku USE CASE a udává jak vypadá ve své vnitřní struktuře, tj. ukazuje, jak tento USE CASE používá jiné prvky USE CASE.

Tento princip plně odpovídá principu vnějšího a vnitřního pohledu na prvek (viz předešlé kapitoly).

Vysvětlení této problematiky interakce mezi případy užití lze doplnit o přirovnání k volání mezi funkcemi, kde je situace velmi podobná. Volání funkcí je směrové a vždy se hovoří o jedné funkci, která vnitřně volá druhou funkci. Z hlediska vnějšího

pohledu na funkci je její vnitřní struktura nezajímavá a funkce se považuje za jeden celek i se všemi odbočkami ve volání jiných funkcí (ve své struktuře) .

7.3.2.2 Princip maximální opětovné použitelnosti v USE CASE MODELU

Použití jednoho případu užití druhým případem užití vede k tomu, že lze přesně a syntakticky jednoznačně vyjádřit opětovnou použitelnost mezi případy užití, tj. lze dodržet zásadu maximální opětovné použitelnosti i v USE CASE MODELU. To vede k důslednému postupu neopakování se v případech užití a k zavedení interakcí odkazem na opětovně použitelné případy užití. Pokud se tedy opakuje nějaká část scénáře ve dvou případech užití, je povinností zavést interakci a tuto část „vytknout“ a opětovně použít. Při tomto „vytknutí“ se volí odpovídající typ interakce mezi prvky USE CASE.

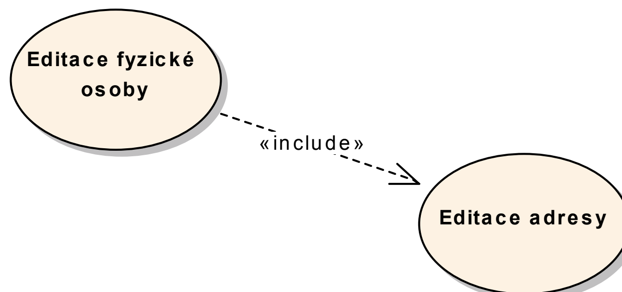
Dodržení této zásady se projeví nejenom ve zhutnění dokumentu UC MODELING, ale navíc se promítne v opětovné použitelnosti až do návrhu informačního systému s opětovně použitelnými scénáři. Tím je například již ve fázi AM z interakcí případů užití zřejmé, že bude opětovně použita již jednou zavedená část formuláře, nebo je zřejmé volání stejného již jednou zavedeného scénáře zpracování apod.

7.3.2.3 Vztah interakce případů užití mezi projekty

Vztah interakce mezi případy užití není omezen pouze na daný projekt, ale lze jej použít i mezi projekty. Tímto se zavádí opětovná použitelnost mezi prvky USE CASE napříč mezi projekty. Například lze vyjádřit opětovnou použitelnost již naprogramovaného případu užití z jiné části jiného systému. Ke vztahu interakce případu užití do jiného projektu je třeba zavést mechanismus, který byl uveden v kapitole CONTROL PACKAGE: „Uživatel“ si musí přilinkovat odpovídající CONTROL PACKAGE, ve kterém je umístěn užívaný prvek USE CASE. Podrobnosti viz Postupky.

7.3.2.4 Interakce INCLUDE v USE CASE MODELU

V interakci INCLUDE jeden prvek USE CASE používá druhý prvek USE CASE, přičemž z hlediska případů užití se jedná o prosté vložení jednoho případu užití do druhého (skládání případů užití, skládání scénářů). Jeden scénář se odvolává na jiný scénář, přičemž povaha tohoto odvolání je prosté vložení jednoho textu scénáře do druhého scénáře. Interakce INCLUDE se značí takto:



obrázek 40 INCLUDE se znázorňuje šipkou se stereotypem <<include>>

V textu se objeví odkaz na případu žití pomocí slovního spojení: *viz <název prvku USE CASE*>. Směr DEPENDENCY v INCLUDE (kdo koho potřebuje - používá) odpovídá směru šipky vztahu INCLUDE.

Nejčastější případy použití této interakce jsou uvedeny ve vzorech scénářů případů užití, viz kapitola SCENARIO PATTERNS.

7.3.2.4.1 Instance prvku USE CASE a interakce INCLUDE

Pro správné modelování interakce INCLUDE je třeba znát tu skutečnost, že prvek USE CASE jako případ užití spadá mezi ty prvky, které se podřizují principu dichotomie „druh versus výskyt“. Pokud zavádíme prvek USE CASE v modelu, potom máme na mysli „druh“, tj. zavádíme možnost užití systému. Pokud okolí konkrétně použije jedno užití, máme na mysli výskyt této možnosti. Jednotlivé výskyty případů užití probíhají stejně, ale mohou mít jiný kontext užití. Tuto skutečnost je třeba mít na paměti při modelování interakcí, protože tam se jedná o vztah na úrovni druhu. Na předešlém obrázku se jedná o vztah druhů, nikoliv výskytů. V odkazu scénáře se může objevit užití v konkrétní roli, tj. kontext použití daného případu užití.

Technologie EFEM (za podpory EA) role v INCLUDE zavádí také roli a multiplicitu v interakci INCLUDE jako povolenou extenzi UML. Role udává konkrétní kontext použití vloženého scénáře.

Například v textu scénáře případu užití předešlého obrázku s INCLUDE se v prvku USE CASE „Editace fyzické osoby“ nachází přesně tato věta:

Obsluha edituje adresu trvalého bydliště, viz „Editace adresy“.

Scénář případu užití „Editace adresy“ je v popisu jednoduchý, v něm se zadává jednoduše ulice, město a PSČ jako tři texty, ale v předešlém obrázku není patrno, že se jedná o editaci trvalého bydliště (jak udává přesný scénář v textu). To je nejednoznačná situace v modelu a může vést ke kolizi. Stačí uvést například další

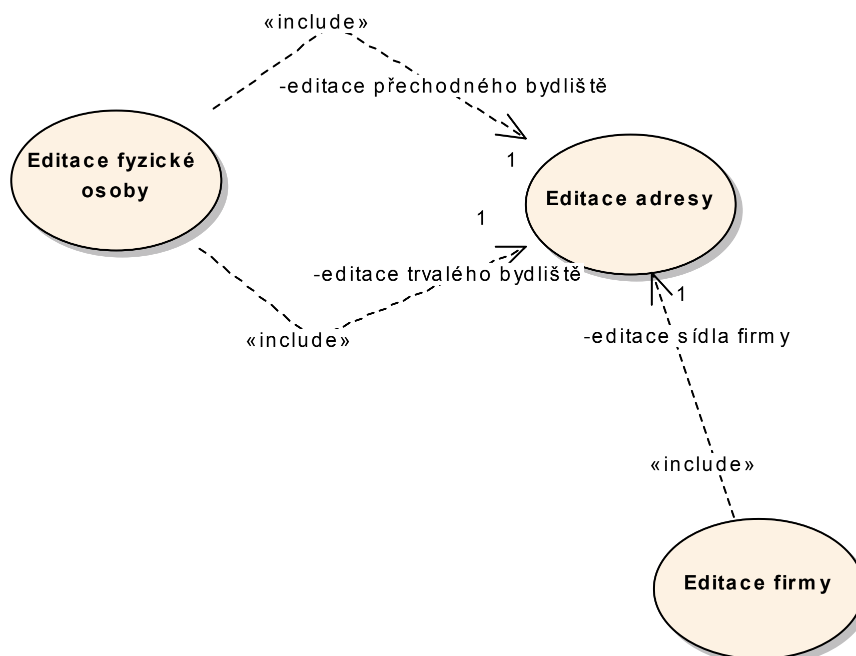
editaci přechodného bydliště a kontext použití se překrývá. V případě textu scénáře se případ užití „Editace adresy“ používá „v roli“ neboli kontextu „Editace adresy trvalého bydliště“. Je třeba mít na paměti, že chybou by bylo napsat následující větu:

Obsluha může zadat adresu trvalého bydliště, viz „zadání adresy trvalého bydliště“.

Opětovně použitelný prvek USE CASE je „Editace adresy“ v libovolném kontextu kterékoli instance scénáře „Editace adresy“ (například editace přechodného bydliště, editace sídla firmy apod.).

Technologie EFEM podporuje a doporučuje stejně jako umožňuje EA tuto syntaxi pomocí role.

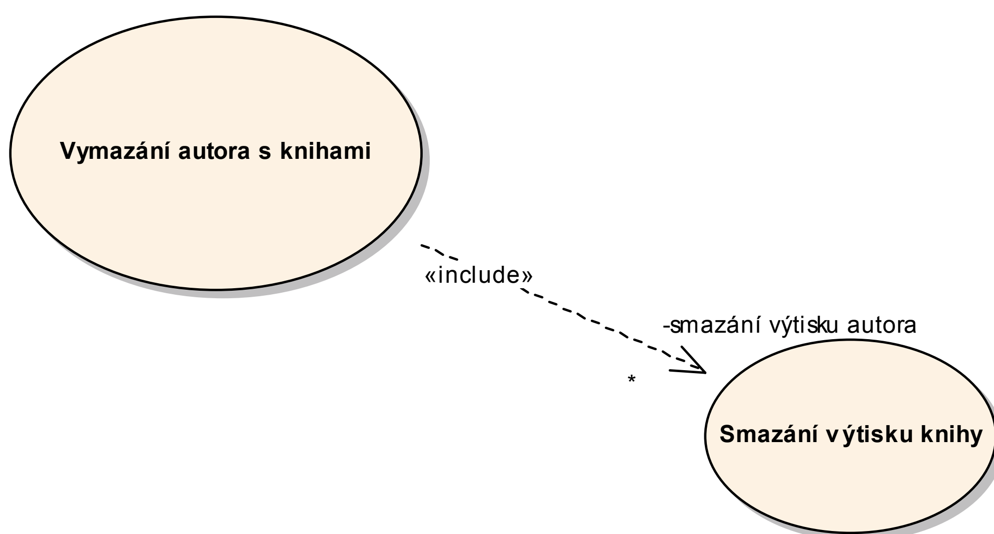
Například diagram s editací adresy vypadá i s rolemi případů užití takto:



obrázek 41 Příklad na role v interakci INCLUDE

Uvedený diagram se chápe takto: Editace fyzické osoby obsahuje editaci adresy ve dvou rolích, v editaci přechodného bydliště a editaci trvalého bydliště. Editace firmy obsahuje editaci adresy v jedné roli editace sídla firmy.

Je možné zavést také multiplicitu různou od jedné (tj. N), pokud se volá scénář v cyklu, resp. obecně několikrát, například:



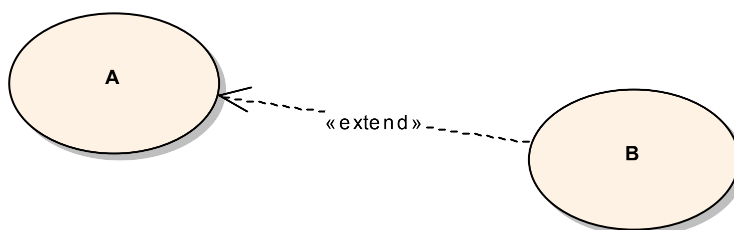
obrázek 42 Příklad na multiplicitu N u role ve vztahu INCLUDE

Vzhledem k jednosměrnosti vztahu nemá smysl zavádět roli a multiplicitu na straně u prvku USE CASE, který obsahuje daný prvek USE CASE (v diagramu na straně, kam nesměřuje šipka).

7.3.2.5 Interakce EXTEND v USE CASE MODELU

Interakce EXTEND je svou povahou stejná jako interakce INCLUDE, pouze nese další doplňující informaci. U interakce EXTEND jeden prvek USE CASE A používá druhý prvek USE CASE B ve smyslu obsažení stejně jako INCLUDE, avšak v prvku A existuje podmínka, při splnění které (tj. při TRUE této podmínky) se prvek USE CASE B použije a při jejím nesplnění se prvek USE CASE B nepoužije. Tím je vyjádřena navíc určitá volitelnost použití druhého prvku USE CASE podle vyjádřené podmínky. Bod, ve kterém se vyhodnocuje daná podmínka, se nazývá EXTENSION POINT a lze jej v modelu v prvku A definovat. Tento EXETENSION POINT také vyplývá přímo z textu scénáře, takže není třeba jej definovat v diagramu povinně.

Je to jedna z mála interakcí v UML, u níž je směr šipky obráceně proti směru DEPENDENCY, což je třeba mít na paměti. Interakce EXTEND se maluje pomocí šipky od použitého k používanému prvku:



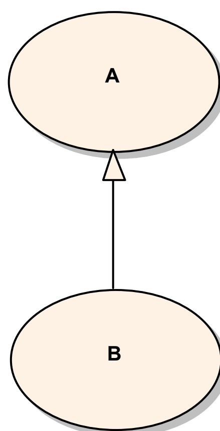
obrázek 43 Interakce EXTEND, prvek USE CASE A používá prvek USE CASE B

Na předešlém obrázku A používá B (proti směru šipky), ale čte se, že B extenduje A. (Pozn.: Z hlediska dodržení směru šipky ve směru použití by bylo výhodnější v UML zavést interakci „je extendován“).

Velmi často se EXTEND používá u stereotypní situace, kdy je ve scénáři třeba vybrat ze seznamu prvků, prvek se však v systému nenachází. V tom případě se může „extendovat“ scénář o zadání nového prvku. Podmínkou je, že prvek se v seznamu nenachází.

7.3.2.6 Interakce GENERALIZATION - SPECIALIZATION v USE CASE MODELU

Podobně jako mezi třídami také mezi prvky USE CASE lze zavést vztah GEN-SPEC. Vztah GEN SPEC se obecně v UML znázorňuje spojnicí s trojúhelníkem, stejně tak i mezi prvky USE CASE:



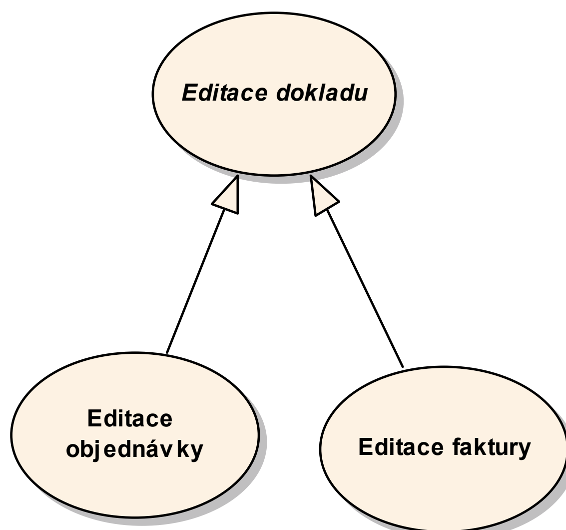
obrázek 44 Vztah GEN-SPEC mezi prvky USE CASE

Ve vztahu GEN SPEC se opět jedná o použití jednoho prvku USE CASE druhým prvkem USE CASE a to ve směru šipky trojúhelníku, tj. na předešlém obrázku prvek B používá prvek A. Avšak v případě GEN-SPEC se již nejedná o vztah použití

zavolání nějaké instance scénáře tak, jak tomu bylo u INCLUDE resp. EXTEND. Prvek B je chápán jako speciálnější prvek k prvku A, takže B se chápe jako specializace případu užití A. Scénář, který je na horní úrovni v A, je takto použit i v prvku B podobně jako při dědění mezi třídami.

Vztah je (jako v každém GEN-SPEC) třeba číst odspodu nahoru, tj. celý případ užití je prvek B plus obecnější prvek A. Při tomto čtení zespodu nahoru se při konkrétním použití prvku USE CASE systému vždy jedná pouze o jednu instanci, tj. vlastnosti se skládají již na úrovni druhu.

Příklad:



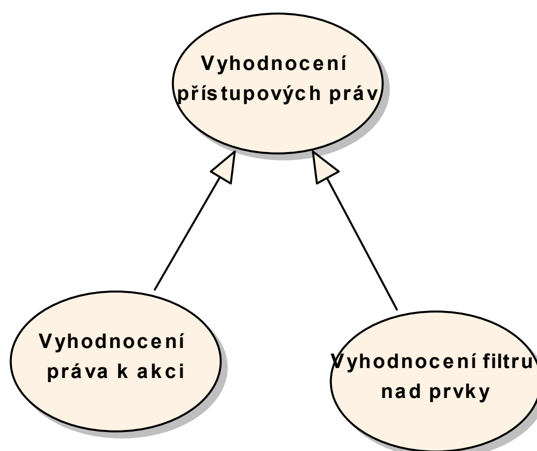
obrázek 45 Příklad na GEN SPEC mezi prvky USE CASE

Oba případy užití „Editace faktury" a „Editace objednávky" používají stejnou část scénář „Editace dokladu", ve kterém je (například) editace datumu vystavení dokladu, což je společné pro oba scénáře. V uvedeném příkladu se instance prvku Editace faktury (podobně Editace objednávky) chápe jako jedna instance dohromady i s předkem, od kterého podědila vlastnosti případu užití. Nejedná se tedy o použití zavoláním jiné instance prvku USE CASE, ale o použití jednoho prvku druhým již na úrovni druhu a poté instanciování jedné instance poskládané v druhu.

Syntaxe UML obecně zavádí u prvku, který vstupuje do vztahu GEN-SPEC označení *„isAbstract"*, v překladu „je abstraktní". Pokud je prvek označen jako abstraktní, nelze jej instanciovat a slouží pouze jako předloha obecnějšího prvku ve vztahu GEN-SPEC. Také prvky USE CASE mohou být abstraktní, tj. což lze v modelu vyznačit a projeví se to kurzívou. Jako příklad viz předešlý obrázek, kdy obsluha nikdy nebude používat případ užití Editace dokladu „napřímo", ale pouze dědice tohoto prvku.

V systému se konkrétně použijí pouze potomci a obsluha instanciuje pouze případ užití Editace faktury resp. Editace objednávky.

Podobně jako u tříd také mezi případy užití platí zástupnost rolí, tj. tam, kde se nějak vyskytuje případ užití A v nějaké roli, lze zavolat také USE CASE B. Lze také ve speciálních případech zavést obdobu přepsání scénáře případu užití ve smyslu definice obecného scénáře na horní úrovni a jeho následné konkretizace (implementace) ve spodním prvku USE CASE, například takto:



obrázek 46 Použití konkretizace obecného scénáře

7.3.2.7 PRECONDITION a POSTCONDITION

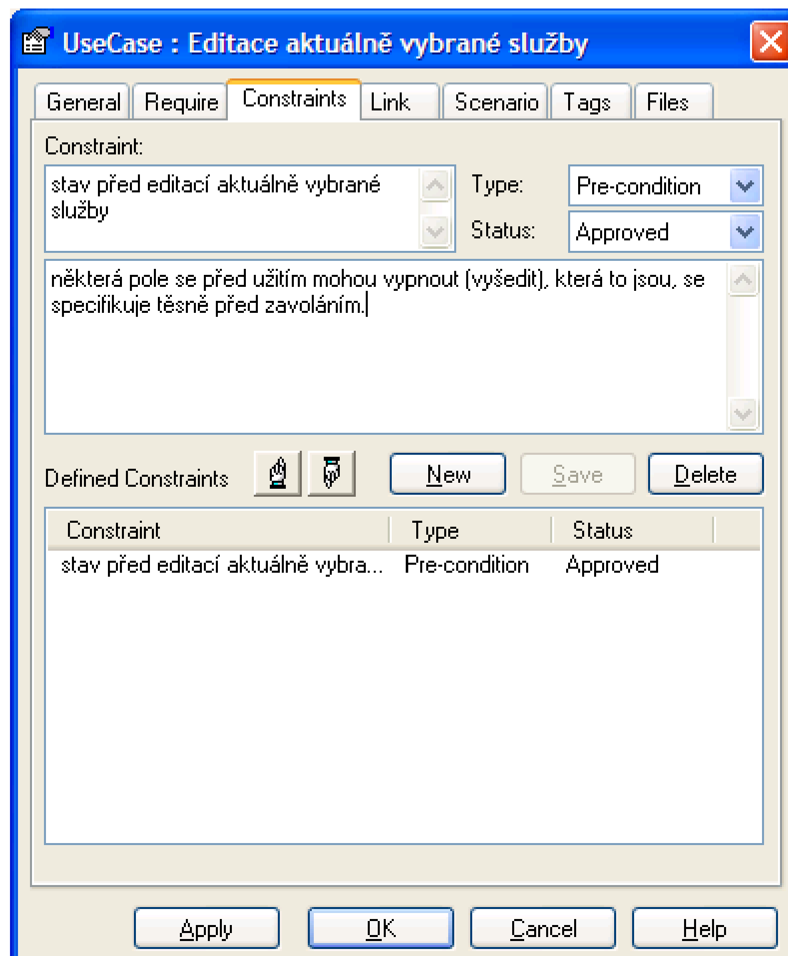
Ke každému případu užití lze přiřadit podmínky nazývané PRECONDITION a POSTCONDITION. Tyto podmínky jsou v UML prvky typu CONSTRAINT (tj. sama textová podmínka je typu BOOLEAN) se stereotypy *<<PRECONDITION>>* a *<<POSTCONDITION>>*. Tyto prvky jsou v UML používané i v jiných modelech, než v USE CASE MODELU. Pomocí těchto dvou prvků lze obecně stanovit stav „před“ a stav „po“ proběhu daného prvku USE CASE.

Prvek POST-CONDITION lze s výhodou použít pro vyjádření požadovaného stavu na konci případu užití, což může dodatečně dobře vysvětlit složitý algoritmus scénáře. Samotný scénář nesmí obsahovat vysvětlující prvky, protože se jedná o přesný popis kroků algoritmu. Konečný stav obsažený v POST-CONDITION může dobře scénář vysvětlit, avšak nikoliv nahradit.

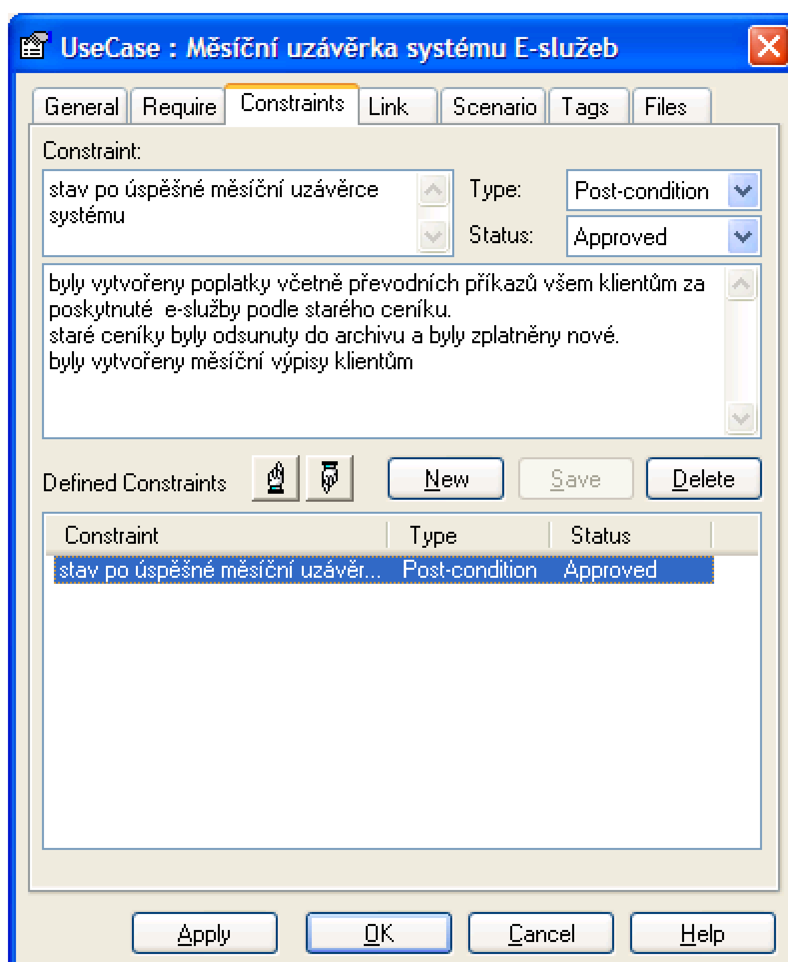
Popisované stavy „před“ mohou mít i charakter proměnných stavů, které se vyplňují těsně před zavoláním případu užití, což se používá při opětovném použití daného

prvku USE CASE (většinou u vztahu INCLUDE). Na tuto skutečnost se upozorní právě v PRE-CONDITION.

Příklad na PRE-CONDITION:



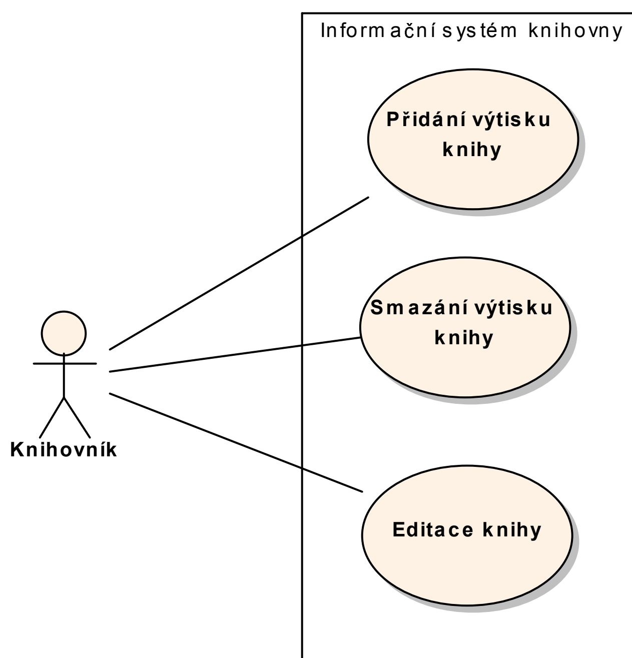
Příklad na POST-CONDITION:



Poznámka: Stav POST-CONDITION v předešlém příkladu nenahrazuje scénář, pouze jej doplňuje.

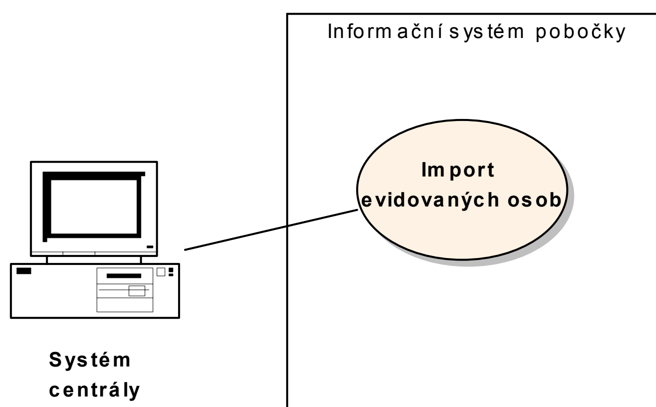
7.3.3. Prvek ACTOR v USE CASE MODELU

Prvek ACTOR znázorňuje vnější objekt mimo systém, tj. objekt z podniku, který komunikuje se systémem. Komunikace prvku ACTOR se znázorní interakcí ACTOR versus odpovídající prvek USE CASE, interakce je typu USE. Tato interakce prochází přes rozhraní systému, což lze znázornit graficky oddělením systému prvků ACTOR přes obdélník uzavírající systém, tzv. BOUNDARY. Implicitním grafickým znázorněním prvku ACTOR je „panáček“ značící živou obsluhu:



obrázek 47 ACTOR jako „panáček"

Prvkem ACTOR může být i „neživý systém“, například externí systém informačního systému. Pro externí systém se volí jako grafický prvek symbol PC:



obrázek 48 Neživý ACTOR

Prvek ACTOR nemodeluje samotný informační systém, ale jeho okolí a tak je třeba vždy tento prvek chápat. Prvek ACTOR zobrazuje kontext použití systému v modelu podniku. Po nalezení tohoto kontextu je pro samotnou tvorbu informačního systému dále nezajímavý.

Je třeba upozornit na skutečnost vypozorovanou z praxe, že přecenění významu prvku ACTOR může vést k chybným krokům při modelování informačního systému. Prvek ACTOR interakcí s prvkem USE CASE pouze provazuje problematiku modelu podniku s informačním systémem. Pro samotnou tvorbu informačního systému je jeho další použití zavádějící.

Pro informační systém platí pravidlo anonymního klienta. Znamená to, že pro tvorbu systému je okolí a tedy prvek ACTOR prvkem nezajímavým. Systém pouze poskytuje svoje rozhraní k použití anonymnímu klientovi. To je důvod, proč je pro designéra a programátora informace o prvku ACTOR, který je mimo systém, absolutně nezajímavá. Designéra zajímá pouze spojnice vůči okolí a jak má být realizována (obrazovka, parser dat apod.), což se doví ze scénáře. Pro správné naprogramování systému je v práci s prvky USE CASE MODELU důležité nalezení všech prvků USE CASE se všemi vzájemnými použitími (interakcemi) a dobře napsanými scénáři USE CASE SCENARIO.

Význam práce s prvky ACTOR v modelování jsou tyto:

- Význam pro inventarizaci procesů podniku. Prvek ACTOR slouží pro verifikaci resp. inventarizaci všech kontextů (souvislostí) použití systému, tj. procesů podniku, které mají být podpořeny informačním systémem. Pomocí výčtu prvků ACTOR se v technice rozkladu BPM ověřuje, zda neexistuje ještě „někdo“ (tj. objekt z okolí), kdo systém bude potřebovat anebo „něco“, co s ním bude komunikovat. Pomocí prvků ACTOR se hledají všechny objekty z podniku, které komunikují s informačním systémem a používají jej. Ověřuje se, zda je opravdu nalezen seznam všech případů užití a nikdo jiný z podniku již nebude systém potřebovat a žádný jiný objekt z podniku již se systémem nebude komunikovat. Je třeba zdůraznit tu skutečnost, že pokud je dobře určen výčet všech procesů podniku (viz technika vyhledávání případů užití rozkladem), tak pokud se autor modelu nedopustil logické chyby, jsou tím také logicky odvoditelné všechny prvky typu ACTOR, protože tyto prvky jsou objekty jakoby „uvnitř“ těchto procesů (hrají role v těchto procesech). Někdy se prvek ACTOR používá pro vyjádření určité potřeby v kontextu použití, tj. používá se při vyhledávání určité množiny prvků BUSINESS PROCESS. Například ACTOR „Knihovník“ implikuje množinu procesů „co všechno dělá Knihovník a proto potřebuje IS“. Při určení všech těchto procesů podniku a následně případů užití je pojem „Knihovník“ jako prvek modelu již mnohem méně zajímavý a z hlediska modelování nemá již takovou roli pro vyhledání prvků USE CASE.
- Význam pro nalezení všech kanálů (interfaců) k okolí. Každá interakce prvku ACTOR se systémem znamená identifikaci nového interfacu vůči okolí v daném kontextu použití. Tato informace je pro designéra velmi zajímavá. Designér navrhuje realizaci pouze spojnice (interakce) mezi prvky ACTOR a případy užití, které protínají rozhraní systému, tam se budou navrhovat interfacy systému.

- Při znalosti všech funkcionalit a interfaců se z prvku ACTOR stává pěkným a výrazným ozdobným prvkem diagramů.

Je nutno zdůraznit, že pro dodavatele IS význam prvků ACTOR stoupá z obchodního důvodu. Zákazník odebírající informační systém většinou v diagramech rád vidí prvky okolí systému, i když z hlediska samotného programu, tj. technologicky vzato, jsou tyto prvky nezajímavé.

7.3.4. Význam prvku ACTOR pro EFEM a časté chyby při jeho zavedení

Základem techniky vyhledávání prvků USE CASE v EFEM je práce s prvky BUSINESS PROCESS a nikoliv s prvky ACTOR. I když i prvky ACTOR jsou významnými prvky modelu (viz předešlý výčet) a dobře zpřehledňují a zkrášlují diagramy (což je pro obchodní dokumenty velmi přínosné), jejich význam speciálně pro vývojářské dokumenty je až sekundární. Nedocenění tohoto přístupu vede k častým chybám v práci s prvky ACTOR.

Existuje technika vyhledávání případů užití používající pouze prvky ACTOR bez modelování rozkladu BPM. Technologie EFEM tuto techniku nedoporučuje pro velmi neefektivní postup, který vede k častým chybám. Tato jiná technologie používá tento přístup: Určí se prvky ACTOR a následně se naleznou všechny případy užití tohoto prvku ACTOR. V tomto postupu je uschována nevyslovená myšlenka o procesech podniku, protože prvek ACTOR se v podniku „nějak chová“ a tím dává žít některým procesům podniku. Technologie EFEM se na rozdíl od tohoto postupu soustřeďuje na tyto procesy a až sekundárně na objekty z těchto procesů, tj. na prvky ACTOR. Znamená to, že z hlediska technologie EFEM je prvek ACTOR určen takto: Byl nalezen BUSINESS PROCESS, který je podporován určitými případy užití (viz technika vyhledávání případů užití). V modelu podniku se vyskytují určité objekty, které v rámci těchto procesů podniku komunikují se systémem. Tyto objekty se jsou prvky modelu typu ACTOR a přistupují k interfacům systému.

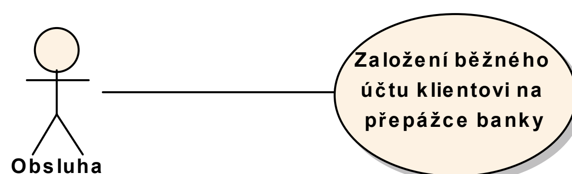
7.3.4.1 Chyba „Neplodné diskuse nad kontextem prvku ACTOR“

Někdy bývá obtížné (a mnohdy i matoucí), jak vlastně pojmenovat jednoduše objekt z okolí, který komunikuje se systémem, tj. jak nazvat prvek ACTOR a tím vyslovit kontext použití od okolí.

Příklad: V bankovním systému byl nalezen případ užití, který se jmenuje „Založení běžného účtu klientovi na přepážce“. Je úmyslně zvolen tak dlouhý název proto, aby bylo ihned zřejmé, jaký bude užitek a jak bude probíhat (přibližně) scénář tohoto

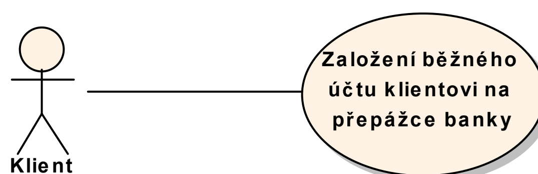
případu užití. Po proběhnutí scénáře (pokud scénář „nespadne“ do EXCEPTION FLOW) bude mít klient založen v informačním systému nový běžný účet.

Otázkou je, kdo vystupuje jako prvek ACTOR vůči tomuto případu užití: Obsluha nebo Klient? Je snad správně model na následujícím obrázku?



obrázek 49 Kontext prvku ACTOR zvolen jako Obsluha

anebo model na tomto obrázku?



obrázek 50 Druhý kontext prvku ACTOR

Oba obrázky se liší pouze názvem prvku ACTOR a ničím jiným.

Existuje argumentace pro oba názvy prvku: Pro volbu prvku s názvem „Klient“ je možné argumentovat tím, že v kontextu klienta se účet zakládá, od něho jdou všechny informace a pro něj tento případ užití existuje. Obsluha je pouze zprostředkovatel stejně jako vyplněný písemný formulář, monitor a klávesnice.

Naopak pro volbu názvu Obsluha hovoří to, že je to ona, kdo sedí za počítačem a tuto osobu tam opravdu vidíme. Klient je až za přepážkou banky a nikoli u počítače.

Co je správně? Podle doporučení EFEM by bylo největší chybou strávit příliš dlouhý čas nad těmito diskusemi.

Pokud je dobře napsán scénář případu užití, tak tyto otázky jsou z hlediska technologie EFEM podružné. Pro EFEM je pro správný a rychlý vývoj základní hledisko efektivity a přesnosti tvorby IS. Znamená to, že při posuzování těchto dvou ukázek je nejdůležitější to, zda je správně napsán scénář daného případu užití, protože podle něj se bude program psát. Pokud je případ užití již nalezen a dokonce již dobře popsán ve scénáři, tak identifikace prvků mimo systém na druhé straně interfacu systému je již podružnou záležitostí. Není však podružný návrh tohoto

interfacu, který bude v obou případech stejný, protože je obsažen ve stejném scénáři.

Existuje však jedno čistě pragmatické hledisko, jak při již nalezeném a popsaném případu užití vhodně vybrat název prvku ACTOR pro dokumentaci z několika možných názvů. Nejlépe je třeba zvolit tu z možností názvu prvku ACTOR, která je logicky nejbližší čtenářům dokumentu. Volí se tedy takový název, který nebude pro čtenáře kolizní a nepovede ke zbytečným diskusím. V předešlém příkladu pokud budou dokumenty v nějaké exportované podobě číst pracovníci banky, je vhodnější zvolit název „Obsluha“ proto, aby se zabránilo neplodným diskusím typu „kdo vlastně sedí u počítače“. (Poznámka: I když z hlediska teorií informace je asi vhodnější název Klient, protože on drží kontext případu užití)

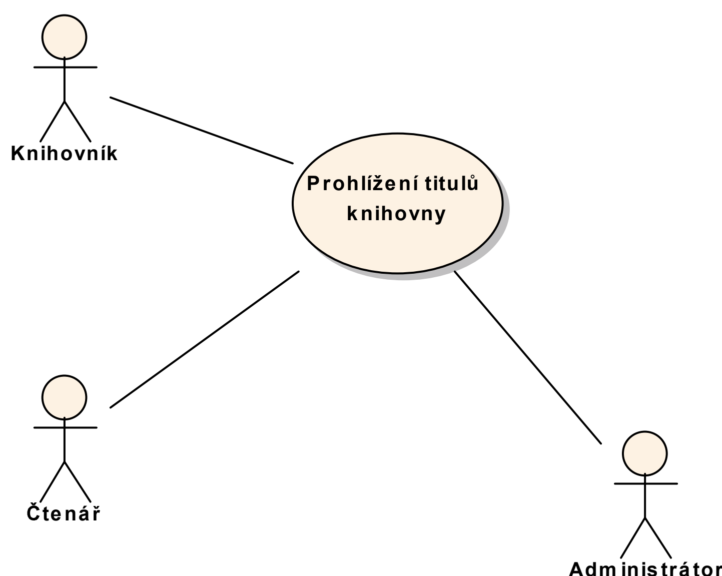
7.3.4.2 Chyba „Příliš mnoho štěňat u misky“

Další častou chybou při určování prvků ACTOR je snaha vyjádřit pomocí tohoto prvku nikoliv užití od okolí, ale skutečnost „kdo všechno může systém použít“, což může být opravdu mnoho objektů z okolí (v podstatě nekonečně mnoho). Posláním prvku ACTOR je identifikovat objekt z okolí, pro nějž je systém určen a určit kanály (interfacy) vůči systému. Přitom prvek ACTOR udržuje kontext použití systému (hraje svou roli).

Nedodržení tohoto principu vede k důsledku k explozi počtu prvků ACTOR. Diagramy USE CASE MODELU potom vypadají tak, že kolem jednoho prvku USE CASE existuje příliš mnoho prvků ACTOR, což skutečně připomíná chumáč štěňat okolo jedné společné misky.

Příklad:

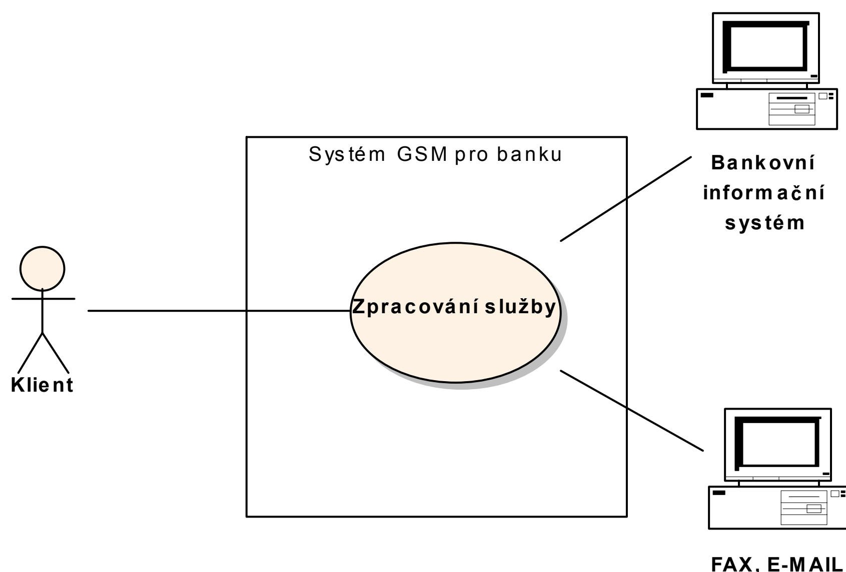
V modelu pro informační systém knihovny byly identifikovány tyto prvky typu ACTOR: Knihovník (mění evidenci knih), Čtenář (půjčování, prohlížení atd.), Administrátor (přístupová práva, uzávěrka apod.), Externí databáze Archivu Národního muzea aj. Autor modelu chtěl vyjádřit tu skutečnost, že případ užití „Prohlížení titulů v knihovně“ mohou použít jak Knihovník, tak Čtenář, tak Administrátor (pokud chce). Proto namaloval chybně tento model:



obrázek 51 Chybný model s efektem "příliš mnoho štěňat u misky"

Pokud autor namaluje předešlý obrázek, vyjádřil tím úplně jinou skutečnost, než jakou původně zamýšlel: V probíhajícím scénáři případu užití „Prohlížení titulů knihovny" existují tři použité kanály (interfacy), jeden v kontextu Knihovníka, druhý v kontextu Čtenáře, třetí v kontextu Administrátora. V daném scénáři „Prohlížení" by každý z nich musel do scénáře něčím přispět, což není pravda. Designér při pohledu na tento obrázek usuzuje, že v daném scénáři budou existovat tři komunikace (kontexty přístupu k systému). V tomto případě pro tento případ užití by bylo vhodné zvolit buď „neutrální" název Obsluha anebo zvolit pouze prvek Čtenář. V druhém případě nastává drobný problém: Neznalým problematiky je třeba vysvětlit, že ten, kdo si prohlíží tituly, není ani Administrátor, ani Knihovník, ale je to ten, kdo vstupuje do prvku ACTOR Čtenář. Název Obsluha je v tomto případě méně kolizní a proto lepší.

Na druhou stranu se může stát, že daný prvek USE CASE vskutku interaguje hned s několika prvky ACTOR a nejedná se o chybu exploze prvků ACTOR. Například v procesu podniku bankovní služby typu GSM nalezneme následující popisu : „Klient GSM pošle zprávu do banky, ta se vyhodnotí v bankovním informačním systému a odpověd se pošle buď zpátky klientovi anebo na fax nebo e-mail". Odpovídající část modelu pro případ užití zpracování služby pak vypadat takto:



obrázek 52 Prvky ACTOR správně identifikují interfacy

Na předešlém obrázku je zřetelně vidět, jaké vstupy a výstupy se budou řešit, jsou to průsečíky interakcí v bodech BOUNDARY s interakcí USE. Nalezly se tři problémy interfaců k řešení, které se skutečně musely řešit až do kódu (práce s modemem GSM, vstupní kanál banky, emailový a faxový klient)

7.4 Kontrola úplnosti dokumentu UC MODELING

Dokument UC MODELING by měl být úplný, což by mělo být zkontrolováno. Pokud se kontrola úplnosti dokumentu neprovede, hrozí bobtnání projektu v pozdějších fázích (například při programování), což je velmi nežádoucí.

Úplnost a konzistence dokumentu **UC MODELING** znamená, že dokument:

- obsahuje všechny funkcionality produktu, tj. všechny případy užití
- v daných funkcionalitách obsahuje scénáře, které úplně popisují algoritmus případu užití

Druhý bod předešlého výčtu lze zkontrolovat přímou četbou jednotlivých scénářů, což není problém. První bod předešlého výčtu se kontroluje obtížněji, avšak i zde lze zavést systematický postup využívající rozklad procesů.

Úplnost dokumentu **UC MODELING** se provádí ve dvou směrech:

- kontrola úplnosti ve vertikálním směru
- kontrola úplnosti v horizontálním směru

Kontrola úplnosti ve vertikálním směru spočívá ve verifikaci každého uzlu rozkladu. Dotaz verifikace zní jednoduše: Pokud se v dokumentu **UC MODELING** proces A rozkládá na procesy B a C, neexistuje ještě proces D, který by měl také rozkládat proces A? Pomocí tohoto postupu se postupuje odshora dolů a sleduje se, zda se na některé případy užití resp. skupiny případů užití resp. celé větve procesů nezapomnělo.

Kontrola úplnosti v horizontálním směru je složitější proces. Používá úvahu spočívající v logice, že pokud se v systému pracuje s nějakou instancí informace, se kterou systém pracuje v daném scénáři, tak se tato instance musela v nějakém kontextu jiného scénáře (tj. jiného případu užití) zrodit, někde se s ní muselo pracovat. Postupuje se tak, že při čtení scénářů se neustále sleduje, že pokud scénář poskytuje nějakou instanci informace něčeho (nebo seznam instancí), tak někde jinde, v jiné části systému, existuje jeden nebo vícero případu užití, kde se tato instance zrodila a byla zadána apod. V některých případech je toto sledování jednoduché. Například ve scénáři se objeví věta: *„…Obsluze se zobrazí seznam firem…“*, z čehož vyplývá, že někde jinde existují případy užití „Založení firmy“ nebo „Import firem“ apod. V některých případech je situace složitější: *„…Obsluze se zobrazí seznam druhů plátna, obsluha vybere plátno, obsluze se zobrazí povolené barvy pro vybraný druh plátna, obsluha vybere barvu…“.* Z toho vyplývá, že někde existuje scénáře umožňující zadávat, editovat apod. povolené barvy pro druh plátna jako kombinace (viz asociativní třída).

Doporučení: Kontrolu úplnosti provádí hlavní analytik projektu resp. jím pověřený pracovník, je vhodné provést navíc brainstorming v týmu tak, že se dokument UC MODELING publikuje na intranetu, tým se s ním seznámí a připraví se několika hodinová diskuse se snahou nalézt nekonzistence dokumentu.

7.5 Použití dokumentu UC MODELING v projektu

Dokument UC MODELING má v projektu hned několik možných použití. Proto by měl vedoucí projektu dbát na to, se tento dokument zodpovědně tvořil. Pokud se vytváří nějaký výstup do dokumentu typu WORD apod. musí být tento dokument vždy uložen v VSS/CVS jako součást dokumentu UC MODELING

7.5.1. Použití dokumentu pro vývoj

Jak bylo řečeno, pro designéra je při tvorbě návrhu jako zadání z dokumentů AM plně dostačující kombinace CLASS MODEL AM plus USE CASE MODEL. Dá se říci, že toto je jeden z nejhlavnějších důvodů, proč je třeba dokument UC MODELING vytvářet. Jedná se totiž o jednu z nejefektivnějších kombinací modelů pro tvorbu IS.

Použitý výstup z dokumentu pro vývojářské účely

Pro vývojáře, kteří potřebují mít zpřístupněn dokument UC MODELING (designér, programátor), je teoreticky dostačující mít k dispozici nástroj EA s možností stáhnout si odpovídající prvky CONTROL PACKAGE.

Lze však doporučit efektivnější postup: Vytvořit HTML výstup z dokumentu UC MODELING a zveřejnit jej na intranetu firmy.

7.5.2. Použití dokumentu pro vedoucího projektu

Pro vedoucího projektu je tento dokument nezastupitelný v tom, že jeho použití zabraňuje bobtnání projektu, vedoucí má přehled o projektu, vidí složitost systému, sleduje postupné řešení atd.

Použitý výstup z dokumentu pro vedoucího projektu

Vedoucí má tyto možnosti:

- Je možné provést výstup do RTF formátu (viz dokument Postupky). Výstup z dokumentu by měl být z hlediska funkcionalit úplný. Tento přístup preferuje vedoucí, který potřebuje tištěnou podobu dokumentu resp. pokud jí osobně dává přednost.
- Vedoucí používá výstup z HTML na intranetu. Nevýhodou je nedostupnost mimo prostor firmy a nelze provádět vlastní poznámky do dokumentu.

- Stránky se publikují na intranetu jednak jako WEB stránky, ale současně se stránky zabalí do zkomprinovaného balíčku (např. ZIP) ke stažení. Dokument se poté dá rozbalit na lokálním stroji jako HTML stránky.

7.5.3. Použití dokumentu pro tvorbu dokumentů strategického modelování

V některých případech nastává situace, kdy je třeba provádět určitá velmi rychlá strategická rozhodnutí ohledně projektu ve velmi raných stadiích projektu, kdy nejsou k dispozici žádné modely. Kromě strategických rozhodnutí designu (např. platforma, typ databáze apod.) je třeba také alespoň částečně odhalit složitost z hlediska analytického náhledu (z hlediska funkcionalit). Problém je v tom, že není čas na provádění AM, protože tyto práce jsou již částí „zaplaceného“ projektu, ale je třeba získat alespoň nějaké relevantní podklady pro rozhodování.

Pro tuto ranou fázi rozhodnutí strategické povahy se vytvářejí rychlé analytické modely v dokumentu s názvem STRATEGIC MODELING. Tento dokument lze chápat jako „nedodělaný, rozpracovaný“ dokument UC MODELING. Konkrétní postup jeho tvorby je specifický a proto viz Postupky EFEM.

Použitý výstup z dokumentu pro strategické modelování

Z daného dokumentu UC MODELING se vytvoří RTF výstup a zpracuje se. Je možné, že v dané chvíli je výhodnější zahájit práce přímo na WORD dokumentu, viz Postupky. Dokument v podobě HTML nemá význam.

7.5.4. Použití dokumentu pro smlouvy s odběratelem

Výstup z dokumentu UC MODELING lze použít také jako dodatek ke smlouvě při dodávce IS odběrateli. Protože tento dokument obsahuje veškerou funkcionalitu strukturovanou do „kapitol“, lze z tohoto dokumentu snadno vytvořit velmi podrobnou přílohu (například nazvanou „Funkční specifikace produktu“) popisující dohodnutou funkcionalitu dodávky SW. Tímto lze předejít nepříjemným kolizím při servisu resp. při vznášení dodatečných požadavků od odběratele.

Použitý výstup z dokumentu pro smlouvy s odběratelem

Z daného dokumentu UC MODELING se vytvoří RTF výstup a zpracuje se. Dokument v podobě HTML nemá význam.

7.5.5. Použití dokumentu pro testování

Testování probíhá na dvou abstraktních úrovních:

- analytické, tzv. funkční testy, ověřují zda systém provádí to, co má
- design testování (technologie) ověřuje kritická místa technologie

Pro testování analytické je třeba, aby tester znal funkcionalitu IS. K tomu mu slouží dokument UC MODELING.

Nástroj EA umožňuje modelovat i prvky pro testování zavedením prvku TEST CASE. Pro účely rychlé dokumentace testů je možné však postupovat i tak, že jednotlivé prvky USE CASE se prohlásí za TEST CASE. V tom případě je postup tvorby dokumentu velmi jednoduchý.

Použitý výstup z dokumentu pro testování

- Tester má zpřístupněn dokument UC MODELING na intranetu v HTML formátu. Z něj přebírá informace o požadovaném chování aplikace.
- Výsledky testů zapisuje zvlášť do dokumentu WORD s názvem ANALYTIC TESTING. Postup tvorby tohoto dokumentu viz Postupky.

7.5.6. Použití dokumentu pro obchodní oddělení

Dokument UC MODELING lze použít také při tvorbě obchodních dokumentů v tom smyslu, že obchodní oddělení může dostat ucelené informace o funkcionalitách systému. V žádném případě však tato dokumentace neobsahuje tolik podrobností, jako původní dokument UC MODELING. Publikování dokumentu UC MODELING v úplné podobě by znamenalo prozradit příliš mnoho know-how firmy.

Použitý výstup z dokumentu pro obchod

Z daného dokumentu UC MODELING se vytvoří RTF výstup a zpracuje se. Dokument v podobě HTML nemá význam.

7.5.7. Použití dokumentu pro uživatelskou dokumentaci

Dokument UC MODELING lze s výhodou použít pro založení dokumentu uživatelské dokumentace a zahájení prací na ní. Využije se té skutečnosti, že rozklad procesů

odpovídá budoucím kapitolám uživatelské příručky a scénáře popisují při určitých úpravách požadované chování obsluhy a systému.

Použitý výstup z dokumentu pro uživatelskou dokumentaci

Zvolí se výstup buď do RTF nebo do HTML podle toho, který z výsledných dokumentů je nejblíže k požadovanému typu dokumentu uživatelské příručky.

KONEC DOKUMENTU SKRIPTA