

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

PROVOZNĚ EKONOMICKÁ FAKULTA

KATEDRA INFORMAČNÍHO INŽENÝRSTVÍ

**OBJEKTIVĚ ORIENTO VANÝ PŘÍSTUP
V PROJEKTOVÁNÍ INFORMAČNÍCH SYSTÉMŮ**

skriptum

2005

Ing. Vojtěch Merunka, Ph.D.

Ing. Robert Pergl

Ing. Marek Pícka

Toto skriptum je určeno především pro posluchače magisterského oboru Informatika na PEF ČZU v předmětech Informační inženýrství a Projektování informačních systémů. Některé pasáže z tohoto textu se také týkají látky probírané v předmětech Objektové metody, Znalostní a databázové systémy a Měření kvality softwaru.

Osnova

1 Sémantická mezera mezi informačním systémem a architekturou stroje	15
1.1 Historické souvislosti, mainframe a PC.....	15
1.2 Architektura informačního systému versus architektura stroje	16
1.3 OOP jako softwarové řešení sémantické mezery	17
2 Vývoj nástrojů a technik OOP k překonávání sémantické mezery.....	18
2.1 Programovací jazyky a prostředí	18
2.2 Databázové systémy	20
2.3 Metody tvorby informačních systémů	22
3 Dnešní stav nástrojů a technik OOP	25
3.1 Přínosy OOP	25
3.2 Problémy OOP	25
3.3 Pokrok v oblasti programovacích jazyků a prostředí.....	28
3.4 Databázové systémy	29
3.5 Metody tvorby informačních systémů	34
3.6 Iterativní a evoluční versus sekvenční model životního cyklu	34
3.7 Rigorózní versus agilní metodiky	35
3.8 Tvorba informačních systémů v kontextu podnikového managementu	37
4 Doporučovaný postup při projektování s využitím OOP	42
4.1 Celopodnikový pohled.....	42
4.2 Model životního cyklu	43
4.3 Postupná transformace pojmů při projektování.....	50
4.4 Zajištění jakosti při business modelování.....	69
4.5 Formální techniky návrhu datové struktury.....	75
4.6 Způsob provádění analýzy	78
4.7 Řízení týmu, softwarové profese	83
4.8 Algoritmizace rozpočtu.....	85
4.9 Softwarové metriky a metoda funkčních bodů	87
5 Vhodná alternativa – agilní metodiky	90
5.1 Úvod – Softwarová krize	90
5.2 Projevy softwarové krize	91
5.3 Riziko - hrob projektu.....	92
5.4 Agilní metodiky a extrémní programování.....	92
5.5 Čtyři parametry	93
5.6 Náklady na změnu	94
5.7 Filosofie XP	95
5.8 Praxe XP	98
5.9 Souvislosti postupů XP	101
5.10 Proces vývoje softwaru metodou XP.....	105
5.11 Strategie řízení a organizace projektů XP.....	106
5.12 Metriky.....	107
5.13 Lidé okolo XP.....	108
5.14 Nasazení XP.....	109
5.15 Porovnání XP a rigorózních metodik.....	111
6 UML.....	115

6.1	Vznik UML.....	115
6.2	Vlastnosti UML	117
6.3	Definice UML.....	118
6.4	Struktura jazyka UML	118
6.5	Společné mechanismy jazyka UML	120
6.6	Architektura	124
6.7	Diagramy UML.....	125
6.8	Metamodel UML	134
7	Příklady objektově orientovaných metodik.....	138
7.1	Shlaer-Mellor	138
7.2	Coad – Yourdon.....	139
7.3	OMT.....	139
7.4	OOAD – Booch	141
7.5	Objectory	142
8	Unified Process	143
8.1	Unified Process - úvod.....	143
8.2	Požadavky	149
8.3	Analýza	155
8.4	Návrh	161
8.5	Implementace.....	165
9	Metamodelování	168
9.1	Architektura pro metamodelování	168
9.2	Metamodelovací prostředky	170
9.3	Použití metamodelování v praxi	172
9.4	Metamodelování – shrnutí	174
10	Použitá literatura	176
10.1	Publikace autorů	176
10.2	Ostatní.....	177
11	Přílohy.....	181
11.1	Slovníček pojmů	181
11.2	Modelovací nástroj Craft.CASE.....	194

Obsah

1 Sémantická mezera mezi informačním systémem a architekturou stroje	15
1.1 Historické souvislosti, mainframe a PC	15
1.2 Architektura informačního systému versus architektura stroje	16
1.3 OOP jako softwarové řešení sémantické mezery	17
2 Vývoj nástrojů a technik OOP k překonávání sémantické mezery	18
2.1 Programovací jazyky a prostředí	18
2.1.1 Objektově orientované programovací jazyky	19
2.1.2 Smíšené (hybridní) programovací jazyky	19
2.2 Databázové systémy	20
2.2.1 Objektově orientovaný datový model	20
2.3 Metody tvorby informačních systémů	22
2.3.1 Otázka transformace zadání od uživatele do podoby objektového modelu	23
3 Dnešní stav nástrojů a technik OOP	25
3.1 Přínosy OOP	25
3.2 Problémy OOP	25
3.2.1 Odklon od původního OOP	25
3.2.2 UML	26
3.2.3 Nedostatečnost metod analýzy	28
3.3 Pokrok v oblasti programovacích jazyků a prostředí	28
3.4 Databázové systémy	29
3.4.1 Formální techniky návrhu objektových databází	29
3.4.1.1 Možnosti uplatnění „klasické“ normalizace v objektových databázích	30
3.4.1.2 Techniky objektově orientované normalizace	31
3.4.1.3 Ambler-Beckovy tři objektové normální formy	32
3.5 Metody tvorby informačních systémů	34
3.5.1 Nejpoužívanější metody řízení a podpory softwarových projektů	34
3.6 Iterativní a evoluční versus sekvenční model životního cyklu	34
3.6.1 Sekvenční model životního cyklu	35
3.6.2 Iterativní model životního cyklu	35
3.6.3 Evoluční model životního cyklu	35
3.7 Rigorózní versus agilní metodiky	35
3.7.1 Rigorózní metodiky	35
3.7.2 Agilní metodiky	36
3.7.3 Příčina sporu	36
3.8 Tvorba informačních systémů v kontextu podnikového managementu	37
3.8.1 Procesy a procesní modely – requirement engineering	38
3.8.2 Myšlenka konvergenčního inženýrství	38
3.8.3 Vztah mezi informačním a řídicím systémem uvnitř organizace	39
3.8.4 Vztah k OOP	41
4 Doporučený postup při projektování s využitím OOP	42
4.1 Celopodnikový pohled	42
4.2 Model životního cyklu	43
4.2.1 Iniclace	46
4.2.2 Konstrukce	46
4.2.3 Dodání	46
4.2.4 Provoz	47
4.2.5 Jednotlivé týmy v procesech	47
4.2.6 Provozní, testovací a vývojová platforma	49
4.3 Postupná transformace pojmů při projektování	50
4.3.1 Metoda BORM	50
4.3.1.1 Použití BORMu v praxi	51
4.3.2 Vývoj pojmu objekt během projektování	52
4.3.3 Fáze expanze a konsolidace	54
4.3.4 Objekty reálného světa (business objekty)	55
4.3.4.1 Metoda OBA	55

4.3.4.2	Diagram ORD.....	57
4.3.4.3	Podrobná analýza procesů.....	59
4.3.4.4	Rozšíření modelu business procesů směrem nahoru.....	59
4.3.4.5	Rozšíření modelu obchodních a správních procesů směrem dolů.....	59
4.3.4.6	Simulace procesů.....	60
4.3.4.7	Změna procesů - Business Process Reengineering.....	62
4.3.5	Logické - konceptuální objekty.....	62
4.3.5.1	Přechod od business objektů ke konceptuálním objektům.....	62
4.3.5.2	Diagramy konceptuálních objektů.....	63
4.3.6	Softwarové - implementační objekty.....	64
4.3.7	Přínos rozdělení modelu na business, konceptuální a softwarové objekty.....	64
4.3.8	Evoluce hierarchií objektů.....	65
4.3.9	Tři dimenze objektového modelu – zjednodušení složitosti.....	67
4.3.10	Chyby kterých je třeba se vyvarovat při modelování.....	68
4.3.11	Zkušenosti.....	68
4.4	Zajištění jakosti při business modelování.....	69
4.4.1	Vztah jakosti a metodiky.....	69
4.4.2	Pozice business modelování v zajištění jakosti.....	70
4.4.3	Charakteristiky jakosti.....	71
4.4.3.1	Funkčnost.....	72
4.4.3.2	Bezporuchovost.....	73
4.4.3.3	Použitelnost.....	73
4.4.3.4	Účinnost.....	74
4.4.3.5	Udržovatelnost.....	74
4.4.3.6	Přenositelnost.....	74
4.5	Formální techniky návrhu datové struktury.....	75
4.5.1	Pojem datový objekt.....	75
4.5.2	Tři objektové normální formy.....	75
4.5.2.1	1ONF.....	76
4.5.2.2	2ONF.....	76
4.5.2.3	3ONF.....	77
4.6	Způsob provádění analýzy.....	78
4.6.1	Metodiky specifické.....	78
4.6.2	Metodiky holistické.....	78
4.6.3	Hledání zadání a systémové modelování.....	78
4.6.4	Matematický model a jeho základní složky.....	79
4.6.4.1	Proměnné v modelu.....	79
4.6.4.2	Struktury modelu.....	79
4.6.5	Tvorba modelu.....	80
4.6.5.1	Systémový řez.....	80
4.6.5.2	Vazby v modelu a systémový řez.....	82
4.6.6	Způsob tvorby modelu.....	83
4.7	Řízení týmu, softwarové profese.....	83
4.7.1	Softwarové profese.....	84
4.7.2	Organizace pracovních týmů.....	84
4.7.2.1	Nestrukturované (dělbá práce dle objemu).....	85
4.7.2.2	Strukturované týmy (dělbá podle profese).....	85
4.8	Algoritmizace rozpočtu.....	85
4.9	Softwarové metriky a metoda funkčních bodů.....	87
5	Vhodná alternativa – agilní metodiky.....	90
5.1	Úvod – Softwarová krize.....	90
5.1.1	Propast mezi vývojáři a uživateli.....	90
5.1.2	Komplexnost systémů.....	90
5.1.3	Opomíjení lidských hodnot.....	90
5.2	Projevy softwarové krize.....	91
5.3	Riziko - hrob projektu.....	92
5.4	Agilní metodiky a extrémní programování.....	92
5.5	Čtyři parametry.....	93
5.6	Náklady na změnu.....	94

5.7	Filosofie XP.....	95
5.7.1	Čtyři hodnoty.....	96
5.7.2	Základní pravidla.....	96
5.7.2.1	Rychlá a kvalitní zpětná vazba	97
5.7.2.2	Přírůstková změna (pravidlo drobných korekcí)	97
5.7.2.3	Hraní na výhru	97
5.7.2.4	Práce v souladu s lidskými instinkty a nikoli proti nim.....	97
5.7.2.5	Cestování nalehko.....	97
5.8	Praxe XP.....	98
5.8.1	Plánovací hra	98
5.8.2	Malé verze	99
5.8.3	Metafora	99
5.8.4	Jednoduchý návrh.....	99
5.8.5	Testování	100
5.8.6	RefaktORIZACE.....	100
5.8.7	Párové programování	100
5.8.8	Společné vlastnictví.....	100
5.8.9	Nepřetržitá integrace	101
5.8.10	40-ti hodinový týden	101
5.8.11	Zákazník na pracovišti.....	101
5.8.12	Standardy pro psaní zdrojového textu	101
5.9	Souvislosti postupů XP	101
5.9.1	Plánovací hra	102
5.9.2	Malé verze	102
5.9.3	Metafora	102
5.9.4	Jednoduchý návrh.....	102
5.9.5	Testování	102
5.9.5.1	Testy jednotek.....	102
5.9.5.2	Funkční testy.....	103
5.9.6	RefaktORIZACE.....	103
5.9.7	Párové programování	103
5.9.8	Společné vlastnictví.....	104
5.9.9	Nepřetržitá integrace	104
5.9.10	Zákazník na pracovišti.....	104
5.10	Proces vývoje softwaru metodou XP	105
5.10.1	Fáze plánování.....	105
5.10.2	Fáze vývoje.....	105
5.10.2.1	Komunikace.....	105
5.10.2.2	Psaní zdrojového textu.....	106
5.10.2.3	Testování.....	106
5.10.3	Fáze nasazení a údržby.....	106
5.10.4	Možné obměny	106
5.11	Strategie řízení a organizace projektů XP	106
5.11.1	Přístup managementu	106
5.11.2	Přijetí odpovědnosti.....	107
5.11.3	Výuka poznatků.....	107
5.12	Metriky	107
5.13	Lidé okolo XP	108
5.13.1	Programátor	108
5.13.2	Zákazník	108
5.13.3	Tester	108
5.13.4	Stopař.....	108
5.13.5	Kouč	108
5.13.6	Konzultant	109
5.14	Nasazení XP	109
5.14.1	Omezení XP	109
5.14.1.1	Velikost týmu a rozsah projektů	109
5.14.1.2	Náklady na změnu	109
5.14.1.3	Charakter projektů	109
5.14.1.4	Charakter zákazníků	110

5.14.1.5	Tým.....	110
5.14.1.6	Prostředí.....	110
5.14.2	Zavádění XP.....	110
5.14.2.1	Přizpůsobení metodiky.....	110
5.14.2.2	Přechod na XP.....	110
5.15	Porovnání XP a rigorózních metodik.....	111
5.15.1	Rigorózní metodiky.....	111
5.15.1.1	Proces.....	111
5.15.1.2	Omezení.....	111
5.15.1.3	Rozsah použití.....	112
5.15.1.4	Dokumentovatelnost procesu.....	112
5.15.1.5	Požadavky na vývojové nástroje.....	112
5.15.2	Metodika extrémní programování.....	112
5.15.2.1	Proces.....	112
5.15.2.2	Omezení.....	113
5.15.2.3	Rozsah použití.....	113
5.15.2.4	Dokumentovatelnost procesu.....	113
5.15.2.5	Požadavky na vývojové nástroje.....	114
5.15.3	Diskuse.....	114
6	UML.....	115
6.1	Vznik UML.....	115
6.2	Vlastnosti UML.....	117
6.3	Definice UML.....	118
6.4	Struktura jazyka UML.....	118
6.4.1	Stavební bloky jazyka UML.....	119
6.4.2	Předměty (things).....	119
6.4.3	Vztahy (relationships).....	119
6.4.4	Diagramy.....	120
6.5	Společné mechanismy jazyka UML.....	120
6.5.1	Specifikace.....	120
6.5.2	Ozdoby (Adornments).....	121
6.5.3	Podskupiny.....	122
6.5.3.1	Klasifikátor a instance.....	122
6.5.3.2	Rozhraní a implementace.....	123
6.5.4	Mechanismy rozšiřitelnosti.....	123
6.5.4.1	Omezení.....	123
6.5.4.2	Stereotypy.....	123
6.5.4.3	Označené hodnoty.....	124
6.6	Architektura.....	124
6.7	Diagramy UML.....	125
6.7.1	Diagram tříd.....	125
6.7.1.1	Použití.....	126
6.7.1.2	Speciální případ – diagram balíčků (Package diagram).....	127
6.7.2	Objektový diagram.....	127
6.7.2.1	Použití.....	127
6.7.2.2	Použité prvky.....	127
6.7.3	Diagram případů užití.....	128
6.7.3.1	Použití.....	128
6.7.3.2	Použité prvky.....	129
6.7.4	Sekvenční diagram.....	129
6.7.4.1	Použití.....	130
6.7.5	Diagram spolupráce.....	130
6.7.5.1	Použití.....	130
6.7.6	Stavový diagram.....	131
6.7.6.1	Použití.....	131
6.7.6.2	Použité prvky.....	132
6.7.7	Diagram aktivit.....	132
6.7.7.1	Použití.....	132
6.7.8	Diagram komponent.....	133

6.7.8.1	Použití	133
6.7.8.2	Použité prvky	133
6.7.9	Diagram nasazení	134
6.7.9.1	Použití	134
6.7.9.2	Použité prvky	134
6.8	Metamodel UML	134
6.8.1	Čtyřvrstvá architektura	135
6.8.2	Struktura metamodelu UML	135
6.8.3	Jazyk OCL	137
7	Příklady objektivě orientovaných metodik	138
7.1	Shlaer-Mellor.....	138
7.2	Coad – Yourdon	139
7.3	OMT	139
7.4	OOAD – Booch	141
7.5	Objectory	142
8	Unified Process.....	143
8.1	Unified Process - úvod	143
8.1.1	Historie UP	143
8.1.2	UP a odvozené metody	144
8.1.3	Zásady použití UP	144
8.1.4	Axiomy UP	144
8.1.5	Iterace v metodice UP	144
8.1.6	Struktura UP	145
8.1.6.1	Fáze začátek.....	145
8.1.6.2	Fáze Rozpracování.....	146
8.1.6.3	Fáze Konstrukce	148
8.1.6.4	Fáze Zavedení	148
8.2	Požadavky	149
8.2.1	Pracovní potup Požadavky	150
8.2.2	Definice požadavků.....	151
8.2.2.1	Specifikace požadavků	151
8.2.3	Modelování případů užití	151
8.2.4	Aktivita metodiky UP: Najít aktory a případy užití	152
8.2.4.1	Hranice systému.....	152
8.2.4.2	Aktoři.....	152
8.2.4.3	Případy užití.....	153
8.2.4.4	Slovník pojmů.....	153
8.2.5	Aktivita metodiky UP: Detail případu užití	153
8.2.6	Kdy modelovat případy užití.....	154
8.3	Analýza.....	155
8.3.1	Detail pracovního postupu Analýza	155
8.3.2	Analytický model	155
8.3.3	Aktivita metodiky UP: Analýza případu užití.....	156
8.3.4	Analytické třídy	156
8.3.5	Hledání analytických tříd	157
8.3.5.1	Analýza podstatných jmen a sloves	157
8.3.5.2	CRC karty	157
8.3.5.3	Další zdroje tříd	158
8.3.6	První verze analytického modelu	158
8.3.7	Analytické balíčky.....	158
8.3.7.1	Analýza architektury.....	159
8.3.8	Aktivita metodiky UP: Analýza případu užití.....	160
8.3.9	Realizace případu užití	160
8.3.9.1	Diagramy interakce.....	160
8.3.10	Diagramy aktivit.....	160
8.4	Návrh.....	161
8.4.1	Detail návrhu	161
8.4.2	Návrhové třídy.....	162

8.4.2.1	Anatomie návrhové třídy	163
8.4.2.2	Správně formulované návrhové třídy	163
8.4.2.3	Agregace nebo dědění.....	163
8.4.3	Upřesnění analytických relací	163
8.4.4	Rozhraní a podsystémy	164
8.4.4.1	Rozhraní.....	164
8.4.4.2	Podsystémy	164
8.4.5	Realizace případů užití – návrh	165
8.4.6	Stavové diagramy	165
8.5	Implementace	165
8.5.1.1	Detail pracovního postupu Implementace	166
8.5.1.2	Komponenty.....	166
8.5.1.3	Nasazení.....	166
9	Metamodelování.....	168
9.1	Architektura pro metamodelování.....	168
9.2	Metamodelovací prostředky	170
9.2.1	COMMA	170
9.2.2	GOPRR.....	171
9.2.3	Rodina standardů OMG.....	171
9.3	Použití metamodelování v praxi.....	172
9.3.1	Využití metamodelu při popisu metodologií.....	172
9.3.2	CASE a Meta-CASE nástroje.....	172
9.3.3	Metamodel při zpracování dat a metadat.....	173
9.3.4	Generický model	174
9.4	Metamodelování – shrnutí.....	174
10	Použitá literatura.....	176
10.1	Publikace autorů	176
10.2	Ostatní	177
11	Přílohy	181
11.1	Slovníček pojmů.....	181
11.2	Modelovací nástroj Craft.CASE.....	194
11.2.1	Popis programu.....	194
11.2.2	Instalace a spuštění programu	194
11.2.3	Hlavní okno – spouštěč (launcher).....	194
11.2.4	Ovládání oken, menu a zobrazení hodnot v Craft.CASE.....	195
11.2.5	Business a konceptuální analýza v Craft.CASE, metoda BORM	195
11.2.6	Možnosti nastavení Craft.CASE	197
11.2.6.1	General properties.....	197
11.2.6.2	Elements properties	198
11.2.7	Metamodel Craft.CASE	198
11.2.7.1	Uzly a spojení (Nodes and connections)	198
11.2.7.2	Kontrola úplnosti a správnosti modelu	199
11.2.7.3	Identita objektů, presentory, kopírování objektů.....	199
11.2.8	Práce s grafickým editorem.....	201
11.2.8.1	Menu.....	201
11.2.8.2	Pomocné prvky při kreslení objektů - uzly.....	203
11.2.8.3	Pomocné prvky při kreslení objektů - spojení	203
11.2.8.4	Hierarchie v diagramech – dekompozice a generalizace.....	204
11.2.9	Příprava k modelování.....	205
11.2.9.1	Nastavení parametrů	205
11.2.10	Business modelování.....	206
11.2.10.1	Požadované funkce.....	207
11.2.10.2	Participanti.....	207
11.2.10.3	Scénáře	208
11.2.10.4	Datové toky	208
11.2.10.5	Diagramy	209
11.2.11	Konceptuální modelování.....	210
11.2.11.1	Třídy	210

11.2.11.2	Globální objekty	211
11.2.11.3	Sady objektů	211
11.2.11.4	Komponenty	211
11.2.11.5	Parametry	211
11.2.11.6	Diagramy	211
11.2.12	Architektury	212
11.2.12.1	Business architektura	213
11.2.12.2	Konceptuální architektura	213
11.2.13	Pomocné hierarchie	213
11.2.13.1	Vytvoření podpůrných hierarchií	213
11.2.13.2	Vazby mezi podpůrnými hierarchiemi	214
11.2.13.3	Propojení mezi podpůrnými hierarchiemi a modelem v Craft.CASE	215
11.2.14	Simulátor	216
11.2.14.1	Příprava simulace	216
11.2.14.2	Provedení simulace	216
11.2.14.3	Záznam simulace	218
11.2.15	Možné uživatelské problémy a jejich řešení	219
11.2.16	XML výstup	220
11.2.16.1	Syntaxe XML souboru	220
11.2.16.2	Datová struktura XML souboru	221
11.2.16.3	Hierarchie tříd XML objektů	222

Seznam vyobrazení

obr. 1. vývoj UML.....	26
obr. 2. 0ONF Amblera a Becka [Ambler 2004]	32
obr. 3. 1ONF Amblera a Becka [Ambler 2004]	32
obr. 4. 2ONF Amblera a Becka [Ambler 2004]	33
obr. 5. 3ONF Amblera a Becka [Ambler 2004]	33
obr. 6. čtyři vrstvy modelu řídicích systémů podniku nebo organizace	40
obr. 7. ICT management [Hall et al. 2004].....	42
obr. 8. čtyři hlavní fáze životního cyklu	43
obr. 9. čtyři hlavní fáze životního cyklu - detail.....	44
obr. 10. obsah procesu INITIATE-JUSTIFY.....	44
obr. 11. obsah dílčího procesu CONSTRUCT-GENERALIZE	45
obr. 12. nastavení procesu konstrukce (z projektu Deloitte&Touche pro T-Mobile ČR)	45
obr. 13. kontrolní seznam k ukončení procesu konstrukce	47
obr. 14. 6 fází životního cyklu vývoje systému v BORMu.....	51
obr. 15. projekty v BORMu za období 1997-2000	52
obr. 16. postupná transformace objektového modelu	53
obr. 17. příklad seznamu funkcí.....	55
obr. 18. příklad scénářů (generováno z Craft.CASE)	56
obr. 19. příklad modelové karty (generováno z Craft.CASE)	57
obr. 20. pojmy diagramu ORD.....	57
obr. 21. příklad diagramu popisujícího proces (generováno z Craft.CASE)	58
obr. 22. příklad simulace procesu – stav před obdržení potvrzení od vedoucího	61
obr. 23. příklad simulace procesu – stav po obdržení potvrzení od vedoucího	61
obr. 24. simulační záznam – seznam událostí objektu Referent a Auto	61
obr. 25. rozhodovací tabulka pro přechod od business ke konceptuálnímu modelu	63
obr. 26. příklad proměny hierarchií objektů – fáze business modelování	66
obr. 27. příklad proměny hierarchií objektů – fáze konceptuálního modelování	66
obr. 28. příklad proměny hierarchií objektů – fáze softwarového modelování	67
obr. 29. příklad úlohy v nenormalizovaném tvaru	76
obr. 30. příklad úlohy v 1ONF	76
obr. 31. příklad úlohy v 2ONF	77
obr. 32. příklad úlohy v 3ONF	77
obr. 33. Mapování reality na model	81
obr. 34. Možné vazby systémového řezu.....	82
obr. 35. Funkční body programovacích jazyků.....	89
obr. 36. Kvalita dodaného IS	91
obr. 37. Exponenciální náklady na změnu	94
obr. 38. Náklady na změnu potřebné pro XP.....	94
obr. 39. vývoj UML	117
obr. 40. Vztahy.....	119
obr. 41. Specifikace případu užití	121
obr. 42. Ozdobený element	122
obr. 43. Označená hodnota.....	124
obr. 44. Architektura	125
obr. 45. Třídní diagram	127
obr. 46. Objektový diagram	128
obr. 47. Diagram případů užití.....	129
obr. 48. Sekvenční diagram.....	130
obr. 49. Diagram spolupráce	131
obr. 50. Stavový diagram	132
obr. 51. Diagram aktivit	133
obr. 52. Diagram nasazení.....	134
obr. 53. Standardní čtyřvrstvá architektura pro metamodelování	135
obr. 54. Metamodel – stavový stroj.....	136
obr. 55. Třídní diagram metodiky Booch.....	142
obr. 56. Struktura UP	145
obr. 57. Hodnotící criteria fáze zahájení	146
obr. 58. Hodnotící kriteria fáze rozpracování	147

obr. 59.	Hodnotící kritéria fáze konstrukce.....	148
obr. 60.	Hodnotící kritéria fáze nasazení	149
obr. 61.	Staffing	149
obr. 62.	Pracovní postup - požadavky.....	150
obr. 63.	Detail pracovního postupu.....	151
obr. 64.	Specifikace požadavku	151
obr. 65.	Aktivita metodiky UP: Najít aktory a případy užití.....	152
obr. 66.	Aktivita metodiky UP: Detail případu užití.....	154
obr. 67.	Specifikace případu užití	154
obr. 68.	Detail pracovního postupu Analýza.....	155
obr. 69.	Aktivita metodiky UP: Analýza případu užití	156
obr. 70.	Detail návrhu	162
obr. 71.	čtyřvrstvá architektura pro metamodelování	169
obr. 72.	COMMA.....	170
obr. 73.	GOPRR příklad - obrázek.....	171
obr. 74.	Schéma Meta-CASE nástroje	173
obr. 75.	Generický model.....	174
obr. 76.	ikona programu.....	194
obr. 77.	hlavní okno (launcher).....	194
obr. 78.	volby hlavního menu	195
obr. 79.	hlavní okno a menu pro konceptuální analýzu	196
obr. 80.	nastavení programu.....	197
obr. 81.	hlavní okno přepnuté do MacOS	198
obr. 82.	nastavení volitelných atributů.....	198
obr. 83.	kontrola úplnosti a správnosti modelu.....	199
obr. 84.	editor diagamů	200
obr. 85.	manipulace s presentory	200
obr. 86.	grafický editor business procesů.....	201
obr. 87.	grafický editor konceptuálních diagramů	201
obr. 88.	nastavení editoru diagramů.....	201
obr. 89.	vkládání objektů do diagramu	201
obr. 90.	editace objektů v diagramu.....	202
obr. 91.	nastavení možností kreslení.....	202
obr. 92.	nastavení pomocné mřížky	202
obr. 93.	aktivace a deaktivace panely s vlastnostmi	202
obr. 94.	reporty a testy	203
obr. 95.	pomocné body uzlů.....	203
obr. 96.	příklad dialogu.....	203
obr. 97.	pomocné body spojení a menu	204
obr. 98.	další objekty na spojeních.....	204
obr. 99.	příslušnost objektů k sobě.....	204
obr. 100.	nastavení dekompozice.....	204
obr. 101.	zobrazení dekompozice	204
obr. 102.	generalizace	205
obr. 103.	nastavení vlastností objektů před modelováním.....	206
obr. 104.	funkce systému	207
obr. 105.	participanti	207
obr. 106.	scénáře	208
obr. 107.	předefinované role participantů v procesech	208
obr. 108.	datové toky.....	209
obr. 109.	diagramy	209
obr. 110.	syntaxe diagramu procesů.....	210
obr. 111.	třída a objekt třídy.....	211
obr. 112.	metody objektů a zprávy mezi objekty.....	212
obr. 113.	globální objekty	212
obr. 114.	sady objektů.....	212
obr. 115.	komponenty	212
obr. 116.	business architektura.....	213
obr. 117.	konceptuální architektura.....	213
obr. 118.	vytváření hierarchií.....	214

obr. 119. příklady hierarchií – organizační struktura a služby podniku	214
obr. 120. vazby mezi hierarchiemi.....	215
obr. 121. vyhledávání spojení mezi hierarchiemi	215
obr. 122. editační a simulační režim	216
obr. 123. příklad simulace.....	218
obr. 124. záznam simulace.....	218
obr. 125. možné chyby a jejich řešení.....	219
obr. 126. XML schéma	221

1 Sémantická mezera mezi informačním systémem a architekturou stroje

Počítače za půl století své existence výrazně změnilly svoje parametry (cena, oblast použití, dostupnost, ...) a zejména mnohonásobně vzrostl jejich výkon. Pokroky pozorujeme hlavně v hardware (větší paměť, rychlejší procesor, celkově menší rozměry, atp.). Mnohem menší změny nastaly v uplynulých letech ve způsobu činnosti samotného počítače (strojový kód, paměť, ...). Je vhodné dodat, že zde se změny měří mnohem hůře než prosté technické parametry počítačů. Věnujme se proto problematice softwaru podrobněji a pokusme se zpochybnit populární tezi o bezproblémovém dosavadním vývoji nástrojů tvorby softwaru.

Na rozdíl od klasických inženýrských disciplin, jakými je třeba stavitelství nebo automobilový průmysl, je softwarové inženýrství stále velmi mladou (což by vadit nemělo) a nevyspělou (což už vadí) disciplínou. Dnes téměř nikoho nenapadne postavit most nebo automobil "na koleně" bez pomoci standardů, metodologických technik, znalostí atp. Software je však stále ve velké míře vytvářen tímto řemeslným způsobem. Statistiky z USA nám ukazují velmi varující důsledky takového počínání. [Ewusi 2003] V devadesátých letech pouze 10% programu byly použity tak, jak byly vytvořeny. Další 10% se mohlo používat po mírném přepracování. Velká část, 20%, je přepracována zásadním způsobem. Tzn., že 20% dodaných programu programátorskými firmami je vráceno a přepracováno většinou pomocí nových kontraktů. Téměř 40% programů bylo dodáno, ale nikdy je uživatelé nepoužívali tak jak původně požadovali a zbytek byl dodán v takovém stavu, že byl nepoužitelný. Tato čísla naznačují, že s programovými produkty to skutečně není tak jednoduché nebo tak krásné, jak se často ukazuje nebo předvádí. Je třeba si uvědomit, že prakticky se používá jen zlomek ze všech vytvořených programů - a to těch, které se osvědčily. Právě proto je třeba hledat příčiny selhávání programů, které je mnohem častější než selhání hardware.

Při tvorbě software se většinou jedná o metodu pokusu a omylu. Výsledný produkt je oproti spolehlivosti a flexibilitě jiných inženýrských výtvorů velmi neuspokojující. Úkolem softwarového inženýrství je od 60. let právě hledání teoretických a praktických prostředků vedoucích k významnému zvýšení spolehlivosti a efektivity programátorské práce v kombinaci s novými architekturami výpočetních systémů. Jak již bylo uvedeno, nové technologie stále počítače zmenšují, zrychlují a zvětšují jejich paměť. Proto bývá zvykem hledat příčiny v oblasti tvorby programu a jsou tak oblíbená tvrzení o nespolehlivosti systémového software a problémech s tvorbou aplikací, což mnohdy vede i k zajímavé argumentaci o ceně tvorby programu.

Je jisté, že samotný proces tvorby software není bezproblémový, ale příčiny tohoto stavu je třeba hledat jinde. Důležité je odhalit podstatu vývojových změn technického a programového vybavení počítačů, tj. podstatu změn hardware a software. Je třeba se zaměřit na celý výpočetní systém jako na jeden celek (tj. spojení software a hardware), a hovořit o jeho architektuře na úrovni stroje a na úrovni výpočetního systému, jak jej chápe uživatel.

1.1 Historické souvislosti, mainframe a PC

Před érou současné výpočetní techniky bylo technické vybavení kanceláří relativně malé, modulární a hlavně levné. K dispozici bylo množství jednoduchých třídících, počítačích a tisknoucích přístrojů, které se relativně jednoduše zapojovaly a konfigurovaly.

S nástupem počítačů se vše změnilo. Počítače byly zpočátku vyvíjeny ve výzkumných laboratořích vládních agentur a univerzit především pro vojenské použití. Jejich výkon umožňoval provádět operace tisíckrát rychleji než jakékoliv dřívější zařízení. Tyto počítače umožňovaly spouštění větších a komplexnějších programů a kombinovat operace, které byly dříve rozděleny mezi několik

jednodušších strojů, do jednoho komplexního běžícího programu. V tomto bodě počítačové historie bylo samozřejmostí, že větší je lepší. Tento přechod měl jeden nepříjemný dopad, který pocítíme dodnes, a to velké náklady do zavádění informačních technologií. Výrobci hardwaru i softwaru investovali velké sumy do vývoje a podpory nových verzí systémů, což ovlivňovalo jejich cenu na trhu. Nové systémy však postupně musely být stále více a více kompatibilní s jejich předchozími verzemi. Jejich zákazníci se snažili zlevnit provoz pomocí současného spouštění co největšího počtu běžících úloh a budováním centrálních výpočetních středisek.

S nástupem osobních počítačů a pracovních stanic, které jsou dnes minimálně stejně výkonné jako dřívější mainframy, ale mají nižší pořizovací a udržovací náklady, došlo k odklonu od mainframů jako jediného možného návrhu výpočetního systému. Mnoho nových softwarových aplikací se dnes může provozovat na PC. Kromě toho, že pro PC existuje velké množství softwarových balíků, které se dají víceméně jednoduše přizpůsobovat pro koncového uživatele, mají PC i další „výhodu“: aplikace mohou obsahovat grafický interface a to nejen okna a tlačítka, ale také nejrůznější multimediální efekty (obrázky, animace, zvuk, obchodní grafika).

Používáním PC se však hlavní problém nevyřešil, pouze se zmodernizoval a převzal styl monolitických aplikací z mainframů na PC. Na mainframech jsme měli jen jeden centrální počítač, na kterém odděleně běželo současně více monolitických aplikací. Nyní jsou PC používány velmi podobným způsobem. Jakmile více aplikací musí sdílet společná data, je nutné použít interfejsy, které zprostředkovávají předávání dat z aplikace do aplikace, což je však další software, jehož provoz vyžaduje další prostředky.

Analogický vývoj se odehrál i poli softwaru. Nastala exploze vývoje zákaznického softwaru, která postupně vytvářela pro trh stále „lepší“ nástroje (vyšší programovací jazyky, generátory aplikací, generátory sestav, CASE prostředky). Zároveň vznikaly nové operační systémy. Vše nakonec vedlo k zakonzervování původní dnes již nevyhovující von Neumannovy (vN) architektury počítače, ke krizi programování a k pokusům ji řešit právě objektově orientovanými technologiemi.

1.2 Architektura informačního systému versus architektura stroje

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

Software, který ovládá vN počítač, se skládá z množství jednoduchých operací, které mění stav počítače, především stav jeho paměti. Příkazy určují kromě požadovaných operací nad daty i to, jak tyto datové operace provádět. Je třeba řídit přesouvání dat v paměti, řídit tok algoritmu. Úkolem programátora je potom bezchybně a beze zbytku všechno popsat včetně pořadí provádění. Takový styl uvažování ale není člověku vlastní. Člověk uvažuje na mnohem vyšší úrovni abstrakce a není mnohdy ani schopen ani ochoten psát úlohy na úrovni strojových instrukcí vN stroje (i když i v této oblasti lze získat zručnost). Výsledkem předchozích úvah je vznik vyšších programovacích jazyků a balíků aplikačních programů a CASE systémů. V těchto systémech se jejich uživatel – analytik či tvůrce softwaru - může vyjadřovat svobodněji a s větším zaměřením na vlastní řešený problém.

Kdyby šlo vše překonat softwarem, tak by se tím problémy s realizací požadavků od uživatelů vyřešily. Naneštěstí jsou rozpory v sémantické mezeře natolik závažné, že programování klasickým způsobem je značně obtížné. Důsledky vN architektury dokonce nutí i ve vyšších programovacích jazycích používat primitivní operace z úrovně stroje. Jedná se především o přiřazovací příkazy umožňující přesuny dat z místa na místo a skoky pro změny toku řízení (příkazy cyklu). Fakt, že se

sémantickou mezerou jsou opravdu veliké potíže, se promítá i do různých vnějších podob operačních systémů a různých aplikačních programů, kde se většinou nedaří dokonale odstínit interní detaily vnitřní architektury.

1.3 OOP jako softwarové řešení sémantické mezery

Všechny výše zmíněné problémy s počítači mají prvotní příčinu v rostoucích požadavcích současných uživatelů na abstraktní komunikaci s počítačem. Bezchybná implementace tak komplexních výpočetních systémů pomocí imperativních programovacích prostředků na klasické vN architektuře je dnes vzhledem ke své programové složitosti téměř neřešitelný problém. OOP je právě zatím jedinou prakticky úspěšnou variantou řešení tohoto problému. [Yourdon 1995, Goldberg 1995] Jeho úspěšnost spočívá v tom, že objektový model výpočtu je postaven na jiné architektuře stroje než je klasická vN a tvůrce systému tak není omezen vyjadřovacími prostředky imperativního programování. Dnes, od konce 90. let, se výhody OOP již neprojevují jen na speciálně konstruovaných počítačích (několik jich bylo skutečně za posledních 25 let sestrojeno), ale začínají se výrazně prosazovat i na klasické architektuře stroje, kde jsou chybějící „objektové“ vlastnosti doplňovány během kompilace a nebo softwarovým virtuálním strojem běžícím jako aplikace v daném operačním systému.

2 Vývoj nástrojů a technik OOP k překonávání sémantické mezery

V předchozí části textu byl diskutován vznik OOP jako reakce na existující sémantickou mezeru. OOP řeší problémy se sémantickou mezerou metodou snižování složitostí spojených s tvorbou moderního softwaru na stávající příliš primitivní architektuře stroje. Základem OOP je myšlenka vyšší abstrakce při tvorbě softwaru, důraz na modularitu, znovupoužitelnost a standardizaci.

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

2.1 Programovací jazyky a prostředí

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

Pro naivní uživatele výpočetní techniky je příznačné, že pojem „objektově orientovaný“ ztotožňují s výhodami, které přináší grafická uživatelská rozhraní současných softwarových systémů. Literatura [Abadi 1996, Lacko 1996, Biermann 1990] se vesměs shoduje na několika kritériích, která musí systém splňovat, aby ho bylo možné považovat za objektově orientovaný, což lze také považovat za vymezení objektu jako abstraktního datového typu:

1. Údaje a jejich funkčnost, čímž rozumíme množinu operací, které lze s danou skupinou údajů provádět, jsou spojeny do jediné logické entity nazývané objekt. Operace jsou nazývány metodami. Hodnoty údajů i kódy metod jsou v objektu uzavřené (tzv. zapouzdření dat) a jsou přístupné jen tvůrci objektu. Uživateli objektu stačí znát jen specifikaci metod jako tzv. rozhraní objektu, které je jediným a postačujícím prostředkem pro práci s příslušným objektem.
2. Objekty mají schopnost dědit své vlastnosti od jiných objektů a jsou sdružovány podle svých vlastností do tříd objektů. Objekty mající různé údaje stejného typu a stejnou množinu operací nad nimi jsou instance téže třídy. Třídy objektů jsou v systému uspořádány do orientovaného grafu podle dědičnosti. Je-li možné dědit vlastnosti jen z jediné třídy, hovoříme o jednoduchém dědění. Dědění nám umožňuje navrhovat nové objekty pouze tím, jak se liší od těch již dříve vytvořených, jejichž vlastnosti dědí.
3. Různé objekty jsou schopné na stejnou zprávu (volání operace nad údaji objektu) různě reagovat v závislosti na svém konkrétním obsahu a konstelaci svého okolí, což je označováno jako polymorfismus.
4. Objekty mají kromě dědění také vzájemné vazby skládání, závislosti a delegování.
5. Metody jsou programy obsahující operace nad údaji objektu. Metody jsou součástí objektů.
6. Identita objektů je nezávislá na jejich datovém obsahu.

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

Je úplnou samozřejmostí, že objektově-orientované principy neslouží jen k lepšímu návrhu grafického uživatelského rozhraní. Dobrý program se rozezná právě podle toho, že objekty používá i „uvnitř výpočtu“ a ne jenom pro ovládání prvků grafického uživatelského rozhraní nebo pro tvorbu vstupních formulářů či výstupních sestav.

2.1.1 Objektově orientované programovací jazyky

Vznik objektově orientovaného přístupu je spojen s takzvanými "ryze objektově orientovanými" programovacími jazyky. Tyto programovací jazyky jsou nazývány jako jazyky založené na čistých objektově orientovaných prostředích (EPOL - environment-based pure object languages). Nejznámějšími jazykem této kategorie je Smalltalk.

Mezi společné vlastnosti EPOL patří skutečnost, že se nejedná pouze o kompilátory příslušných jazyků, které by pracovaly pod nějakým klasickým operačním systémem. EPOL také vlastní kompletní vývojová prostředí pro tvorbu programů a vlastní, vesměs grafické, nadstavby operačních systémů pro podporu běhu programů. Tyto grafické nadstavby mohou spolupracovat s klasickými grafickými operačními systémy či nadstavbami, jako jsou například MS Windows, X-Window nebo Mac OS. EPOL je proto možné považovat za objektově orientované operační systémy (či nadstavby operačních systémů) s možností nejen vytvářet, ale i spouštět programy.

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

Mezi čisté dynamické obj. orientované systémy patří jazyky vhodné především pro rozsáhlé úlohy z oblasti umělé inteligence, simulace, počítačové grafiky, databází i z jiných oborů. Jsou to jazyky: Simula, Smalltalk, CLOS, Flavors, Dragoon, Eiffel, Beta, Mainsail, ESP a Object-Prolog. Protože byly navrhovány od samého počátku jako obj. orientované, tak je jejich výhodou mj. i čistá syntaxe a úplné obj. orientované prostředí se všemi vlastnostmi.

2.1.2 Smíšené (hybridní) programovací jazyky

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

I když tyto jazyky umožňují využívat většinu OO vlastností, tak je pro ně charakteristická ta vlastnost, že v nich lze OO zásady "šidit" používáním klasických prostředků. (Stejně snadno jako je možné ve Fortranu nebo Basicu obcházet zásady strukturovaného programování). Jejich kompilátory

jsou proto často i schopny přeložit "neobjektově orientované" programy jakoby psané pro jejich „předka“, ze kterého byly vytvořeny.

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

2.2 Databázové systémy

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

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

- a) Objektově relační datový model představuje evoluční trend vývoje. Jde o doplnění relačního datového modelu o možnost práce s některými datovými strukturami, které známe z oblasti objektově orientovaných programovacích jazyků. Většina výrobců velkých relačních databázových systémů (např. Oracle) zvolila tuto variantu. Objektově relační datový model ale ve svých principech zůstává původním relačním datovým modelem.
- b) Objektově orientovaný datový model, který představuje revoluční trend vývoje. Jde o nový datový model, který není postaven jako rozšíření relačního datového modelu. Do jisté míry zde jde o renesanci původního síťového datového modelu, který je doplněn o možnost práce s objekty tak, jak je známe z objektového programování. [Loomis 1994]

2.2.1 Objektově orientovaný datový model

Objektový datový model se výrazně liší od relačního datového modelu. [Bertino et al. 1995, Silbershatz 2004] Tabulky jsou v ODM pouze jedna z možných forem výstupní prezentace uložených dat. ODM se nicméně může podobat strukturám síťových databází, jak byl implementován v systémech IDMS. Na ODM je možno nahlížet jako na renesanci síťového datového modelu. Při určité míře zjednodušení lze připustit vztah:

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

Základní charakteristiky objektového datového modelu jsou následující:

1. Objektová databáze podporuje více typů množin objektů. (na rozdíl od relačního datového modelu, kde je relační tabulka jediným „druhem množiny“). Společně se označují termínem

collection (české označení zatím chybí, většina autorů používá termín „sada“ či „kolekce“). V konkrétních databázových systémech to může být až několik desítek různých typů různých vlastností tak, jak je známe z knihoven objektových programovacích jazyků. (Je to například Array, List, OrderedCollection, SortedCollection, Set, Bag, Dictionary, ...)

2. Objektová databáze rozlišuje mezi pojmem třída objektů a množina (kolekce) objektů. Třída je jen realizace datového typu objektů a množina je jen úložiště pro objekty. Na rozdíl od tabulek v RDM, kde role třídy a množiny splývají dohromady, v ODM nemusíme pracovat pouze s množinami, které obsahují jen všechny objekty jedné třídy. Můžeme mít například více množin objektů stejného typu i množinu obsahující objekty z různých tříd. (Pokud takové objekty mají díky polymorfismu nějaké společné atributy, tak nám nic nebrání je držet pohromadě a nad takovou množinou provádět například selekci.)
3. Objekty se skládají z vnitřních datových složek (což mohou být opět jiné objekty) a z metod, které představují funkční stránku každého objektu. Známe nejen tzv. přístupové metody, které jen přímo manipulují s datovými složkami objektu (zapisují nové hodnoty datových složek a nebo čtou hodnoty datových složek), ale i metody složitější, které vypočítávají z objektů taková data, jenž v objektu nebyla jednoduše uložena jako jedna z jeho datových složek. Toto známe z objektového programování. Pro ODM je ale důležité si uvědomit, že mezi atributy objektů patří nejen jejich datové složky (jako v RDM), ale i metody poskytující další data.
4. Polymorfismus objektů nevzniká pouze děděním tříd. Pokud mají objekty společné atributy, tak jsou polymorfní i když jejich třídy mezi sebou nedědí.
5. Každý objekt má svoji vlastní identitu, což v objektové databázi znamená, že v rámci jednoho paměťového prostoru má každý objekt systémem přidělen jednoznačný identifikátor obvykle označovaný jako OID (Object IDentifier). OID plní úlohu ukazatele do virtuální paměti. OID každého objektu zůstává stejný, i když se v objektu změní všechny jeho datové složky nebo metody. OID se také samozřejmě nemění při změnách objektu na fyzické úrovni, jako např. změna jeho umístění v operační paměti nebo na disku. Vzhledem k existenci konceptu OID můžeme rozlišovat mezi pojmem rovnost dat objektu a totožnost objektu (Dva objekty se shodnými daty ještě nemusí být totožné). Praktický důsledek konceptu OID je ten, že v ODM není potřeba objektům vytvářet primární klíče. Toto RDM nezná, neboť identita relačních záznamů je dána jen hodnotami atributů. V některých objektových databázích (např. Gemstone) mohou OID zůstat skryté pod aplikačním rozhraním SRBD. V takové databázi potom její uživatel vidí objekty, které se přímo propojují a skládají mezi sebou.
6. V ODM lze v bázi dat pracovat i s takovou soustavou objektů, která je sama o sobě aplikací. Objektová databáze nemusí sloužit jen jako úložiště dat, se kterým manipuluje externí program. Algoritmy programu lze „rozpustit“ v metodách objektů přímo uložených v objektové databázi. Tvorba databázové aplikace na straně klienta je potom velmi zjednodušená, protože v extrémním případě se může jednat jen o prezentační rozhraní výpočetního systému, který celý pracuje „uvnitř“ objektového databázového serveru.
7. Na rozdíl od běžných objektových programovacích jazyků mohou objekty v ODM migrovat mezi různými třídami, v systému může existovat současně více verzí jedné třídy. Různí uživatelé podle svých přístupových práv mohou mít dostupné různé atributy na stejných objektech.

Objektové databáze se také liší ve filosofii práce s daty vzhledem k uživatelům databáze. Relační i síťové databáze svým uživatelům poskytují prostředky, jak lze pomocí programů běžících na klientském počítači pracovat s daty na discích na serveru, takže na logické úrovni se všechna data databáze prezentují svým uživatelům jako uložená na disku vzdáleného počítače a odtamtud přístupná. Objektové databáze ale vytvářejí svým uživatelům na svých rozhraních zdání toho, že

všechna data jsou přítomna na klientu, jen jsou navíc sdílená s ostatními. S daty se tedy pracuje podobným způsobem jako s běžnými proměnnými z vyšších programovacích jazyků nebo aplikačních programů.

2.3 Metody tvorby informačních systémů

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

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

1. OMT - Object Modelling Technique, jejímž autory jsou J. Rumbaugh, M. Bláha, W. Premerlani, F. Eddy a W. Lorensen. Metoda byla poprvé publikována nakladatelstvím Prentice Hall v knize "Object-Oriented Modelling and Design" v roce 1991. Technika je zajímavá tím, že je v podstatě hybridní technikou, zahrnující jak vybrané objektové, tak i klasické nástroje. Nejčastěji se používala pro návrh databázově orientovaných aplikací s objektovým klientem a relačním serverem. [Blaha 1995, Rumbaugh et al. 1991]
2. Coad-Yourdonova metoda, jejímiž autory jsou P. Coad a E. Yourdon (autor velmi používané metody klasické strukturované analýzy a návrhu). Metoda byla poprvé publikována v časopise American Programmer v roce 1989 a posléze v knihách obou autorů nakladatelství Yourdon Press. Z uvedených technik je nejsnazší co do počtu používaných pojmů. [Yourdon 1994]
3. OOSD - Object-Oriented Software Development, jejímž autorem je G. Booch. Metoda byla poprvé publikována v roce 1991 v knize nakladatelství Benjamin/Cummings "Object-Oriented Design with Applications". Tato poměrně velmi komplexní metodologie byla určena pro týmový vývoj rozsáhlých aplikací v jazyce C++ a Smalltalk, a ze všech zde vyjmenovaných metod pokrývá nejvíce objektových vlastností. [Booch 1994]
4. OOSE - Object-Oriented Software Engineering autora Ivara Jacobsona. Metoda je velmi kvalitně popsána ve stejnojmenné knize. (Alternativní název metody je „Objectory“) Vzhledem k tomu, že metoda má původ ve skandinávské škole, je velmi zajímavá pro aplikace z oblasti simulace a řízení. Jacobsonova metoda se jako první začala zabývat problematikou získávání a modelování informací v prvních fázích životního cyklu ještě před budováním konceptuálního diagramu. Úvodní technika této metody - „use case“ byla adoptována i do ostatních objektových metodologií a je dodnes používána. [Jacobson 1992]
5. Object-Oriented Structured Design notation autorů A.I. Wassermanna, P.A. Pirchera a R.J. Mullera, publikovaná poprvé v časopise IEEE Computer č. 23(3) 1990. Metoda je zajímavá především notací používaných diagramů, která ovlivnila následné metodologie.

6. OOER notation autorů K. Gormana a J. Choobineha, poprvé publikovaná na 23. mezinárodní konferenci o informatice (computer science) na Havaji v létě 1991. Metoda používá notaci ER diagramů v klasické Chenově syntaxi, rozšířené o objektové vlastnosti. Je výhodná pro použití v objektově orientovaných databázích.
7. Unifikovaný modelovací jazyk UML podporují od roku 1996 autoři G. Booch, J. Rumbaugh a I. Jacobson pod záštitou firmy Rational Inc. Metoda začala být publikována průběžně na Internetu a v sérii knih, vydávaných firmou Rational Inc, která dodává také vlastní CASE nástroj Rational Rose. UML je dnes doporučovaným průmyslovým standardem pro notaci (způsob kreslení) diagramů a je stále ve vývoji. Představuje sjednocení myšlenek původních metod svých autorů na platformě OMT. Ve srovnání s původní OMT je patrný posun směrem k větší míře podpory objektových vlastností a určitý odklon od původních „hybridních“ vlastností OMT - například zrušení datového modelování pomocí DFD. Je třeba mít na paměti, že UML je ve skutečnosti jen jazyk, tedy návrh standardu k zakreslování nejrůznějších objektových diagramů, který se navíc stále vyvíjí, mění a doplňuje. [UML 2004]
8. Metoda J. Martina (jako E. Yourdon známý i z dřívějšího období) a J. Odella. Metoda byla publikována v sérii knih nakladatelství Martin Books, a představuje jako UML pokus o sjednocení objektových zkušeností předchozích metod. Používá velké množství nejrůznějších diagramů a pojmů. [Martin et al. 1995]

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

2.3.1 Otázka transformace zadání od uživatele do podoby objektového modelu

I když je tvorba objektového modelu usnadněna výše uvedenými metodami a nástroji, které říkají co a jak lze při modelování použít, tak se jedná o velmi obtížný a přitom klíčový problém. Zatím není k dispozici jednoznačný a všeobecně uznávaný návod, který by posloužil k efektivní transformaci zadání problému do formální podoby modelu. Velkou roli zde hraje zkušenost návrháře a jeho schopnost komunikovat se zadavateli. Značným problémem je rozpoznání objektů a jejich vlastností při modelování zadání pro vytvářený systém. K řešení těchto problémů byly vyvíjeny různé různé složité techniky. Tou nejjednodušší – až naivní, a také samozřejmě nejméně účinnou – je návod vycházející z jazykové analýzy textu zadání (podstatná jména = objekty, slovesa = metody, přídavná jména = atributy). [Rumbaugh et al. 1991] Mezi sofistikovanější metody, které byly pro tento účel vyvinuty, patří:

1. Object Behavioral Analysis, která slouží k získávání prvotní představy o objektovém modelu. jedná se o iterativní techniku používající především tzv. modelující karty. Touto technikou začíná projekt tvorby informačního systému a jejím výstupem je objektová informace dostatečná k sestavení prvotního modelu. [Rubin et al. 1992]
2. Behavioral Constraints [Goldberg 1995], která slouží k transformaci prvotního objektového modelu na konceptuální metodou „filtrace“ skrze formálně vyjádřená pravidla, která omezují použití jednotlivých možných hierarchií mezi objekty.
3. Applying Patterns - využívání vzorů (poprvé publikováno P. Coadem na Internetu, dále například [Beck 1997]). Technika slouží ke stanovení vhodné struktury objektů v konceptuálním modelu pomocí znovuvyužívání objektových vzorů, jenž jsou vlastně formalizované části znalostí z předchozích projektů, které jsou v podobě konceptuálních

subgrafů a jsou schopny se vzájemně kombinovat a v nových podmínkách hrát nové role. Používání této techniky spočívá ve vytváření konceptuálního modelu jako orientovaného grafu metodou skládání a substitucí subgrafů představujících jednotlivé vzory.

4. Structural Transformations [Nierstrasz 1990, Shiver et al. 1987, Goldberg 1995], jenž je zvláštním případem předchozí techniky pro potřeby postupné transformace objektového modelu od modelu popisujícího zadání k modelu popisujícímu řešení.

3 Dnešní stav nástrojů a technik OOP

Od roku 1993 probíhá standardizační proces v objektové technologii a to jak na aplikační úrovni (jazyky), tak i na fyzické úrovni (komunikační protokoly, formáty v paměti). Jsou již výsledky ve formě zpráv sdružení OMG¹. V rámci norem ANSI existují schválené standardy nebo zatím jen návrhy objektových programovacích jazyků. Každým rokem ve světě probíhá několik konferencí, kde je diskutována jak praxe, tak i příspěvky k matematické teorii OOP.

3.1 Přínosy OOP

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

Dnešní roli OOP lze charakterizovat následovně:

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

3.2 Problémy OOP

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

3.2.1 Odklon od původního OOP

Došlo k odklonu od „čistého“ OOP. Hybridní řešení, která byla zamýšlena původně jako dočasný kompromis z důvodu technologické nouze (jak již bylo diskutováno), se začala používat jako hlavní

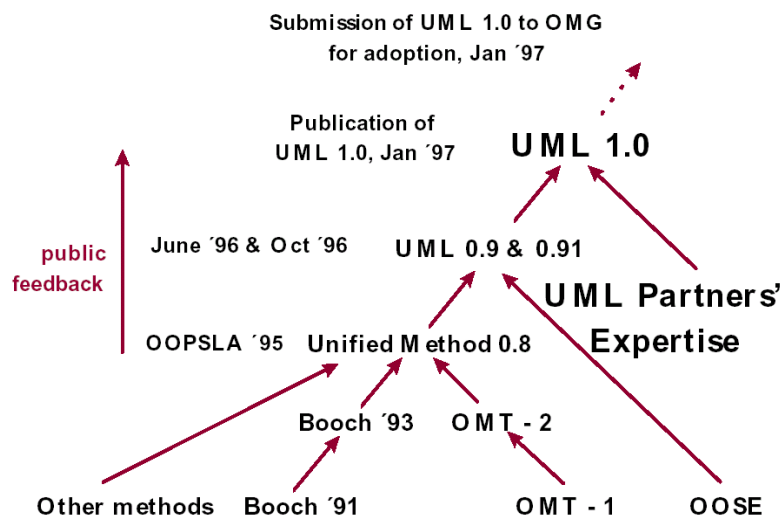
¹ OMG = The Object Management Group. Mezinárodní sdružení firem, vysokých škol a dalších institucí, které se zabývá standardizací OOP, podporou aplikovaného výzkumu a jeho uplatněním v praxi. (<http://www.omg.org>)

prostředek pro tvorbu programů. Tato skutečnost ale skrývá kromě svých praktických výhod jeden velký problém. Jde totiž o to, že po teoretické stránce je hybridní OOP jen ad-hoc vybranou podmnožinou celého paradigmatu tak, jak se ji povedlo implementovat v jazycích C++ a podobných.

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

3.2.2 UML

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



obr. 1. vývoj UML

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

UML je „jen“ grafický jazyk. To by samo o sobě nemělo vadit – je dobře, že od roku 1996 máme konečně pro objektové modelování k dispozici standard. Problém je ale v tom, že k „univerzálnímu“ jazyku nemáme odpovídající metodiku a tak se za metodiku mnohdy považuje samotná znalost UML. Drtivá většina odborné literatury o UML svým čtenářům předstírá, že metodika problém není, a že se

naučí modelovat v UML tím, že se naučí kreslit jednotlivé diagramy. Sami máme podobné negativní zkušenosti i s náplněmi školicích kurzů „moderního objektového modelování“ .

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

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

nedostatečný a vývojáři se musejí při kódování vracet k modelované problematice a upřesňovat detaily systému.

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

1. UML není metoda, je to „jen“ zobrazovací prostředek.
2. UML je komplikovaný – kdo neovládá programování, tak se ho obtížně učí a je velmi pravděpodobné, že ho nesprávně chápe
3. UML nezdůrazňuje, které pojmy se mají používat ve fázi analýzy a které až ve fázi návrhu a implementace.
4. UML málo podporuje čistě objektové programovací jazyky a objektové databáze.
5. UML málo pomáhá tam, kde součástí projektu je hledání a upřesňování zadání.
6. Modelování se v UML často interpretuje jen jako zobrazování budoucího zdrojového kódu aplikace (nebo dokonce jen jako zobrazování dat uvnitř této aplikace).

3.2.3 Nedostatečnost metod analýzy

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

3.3 Pokrok v oblasti programovacích jazyků a prostředí

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

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

3.4 Databázové systémy

První objektově orientované databáze se objevily již ve druhé polovině 80. let a vznikly na základě potřeby uchovávat a databázově zpracovávat v pokud možno nezměněné podobě data z programů napsaných v tehdy se rozvíjejících objektově orientovaných programovacích jazycích. Ve srovnání s relačními databázemi, které v té době byly na vrcholu vývoje, to byly systémy velmi neefektivní a málo výkonné, protože se jednalo o experimentální programy psané jako aplikace v nějakém objektovém programovacím jazyce.

Po více než 15 letech vývoje je však situace jiná. Z praxe již známe případy, kdy nasazení objektové databáze vyřešilo problémy, které relační systém nedokázal zvládnout. Objektové databázové aplikace se objevují již i v ČR. Dnešní objektové databáze mají srovnatelný výkon s velkými relačními systémy – zvládají stovky transakcí za sekundu a tisíce současně připojených uživatelů. Význam objektových databází v blízké budoucnosti ještě poroste, protože již dnes existuje celá řada aplikací, kde objektové databáze prakticky prokazují svoje přednosti. Společnou vlastností těchto aplikací je velké množství komplexních datových struktur a jejich proměnlivost za chodu systému, které způsobují problémy relačním databázím. Takové systémy mohou pracovat až se stovkami a tisíci různých vzájemně poskládaných datových typů reprezentovaných třídami objektů. Dotazy nad takovými objekty ještě navíc vyžadují vysokou míru vzájemného polymorfismu. (V takových systémech kupříkladu potřebujeme klást dotazy nad množinami obsahující prvky různého typu. A zároveň očekáváme, že při přidání nového datového typu se nebudou muset přepisovat již hotové dotazy.) Typickým příkladem takových systémů jsou datové sklady a znalostní systémy, které jsou charakteristické dlouhodobým shromažďováním velkého množství nově vznikajících různorodých dat. Takové systémy jsou charakteristické nejen pro řízení velkých podniků, ale také v různých evidenčních systémech státní správy, zdravotnických systémech, informačních systémech obsahujících ekologické informace, zemědělských informačních systémech, historiografických informačních systémech atp.

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

Ještě je v tomto kontextu vhodné poznamenat, že relační databáze fungují velmi dobře v oblastech, kde během života systému nedochází k požadavku na změnu struktury databáze a na přidávání dalších datových typů. Relační systém může být výkonný i pokud se databáze skládá z velkého množství záznamů, ale uložených v malém počtu jednoduše strukturovaných relačních tabulek. Trend vývoje tedy spíše naznačuje, že se budou používat všechny dnes známé datové modely.

3.4.1 Formální techniky návrhu objektových databází

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

Je sice možné převzít relační techniky, ale potom dostaneme jen „relační databázi v objektovém prostředí“ a nevyužijeme všechny vlastnosti, které ODM má.

Jinou možností je převzít schéma objektů a tříd tak, jak jsou navrženy v aplikaci, která má s databází pracovat. To už je lepší, protože právě proto byl objektový datový model vyvinut. Jenomže struktura objektů výhodná pro algoritmy v softwarové aplikaci může podstatně komplikovat jejich efektivní databázové zpracování. Struktury mnoha návrhových vzorů totiž využívají dynamické vazby mezi objekty, řetězení metod atp.

V různých pramenech se lze setkat s tvrzeními o objektových databázích, které tuto problematiku zjednodušují a prohlašují například, že objektovou databázi není třeba normalizovat a že objektová databáze je mnohonásobně rychlejší než relační. Tvrzení o zbytečnosti datové normalizace můžeme nalézt i v renomované literatuře.

Tvrzení o rychlosti platí, ale jen pro případ, kdy se podaří navrhnout objektové schéma tak, aby obsahovalo přímo propojené objekty mezi sebou na rozdíl od relační databáze, která musí používat spojení od cizího klíče z jedné tabulky na primární klíč druhé tabulky. Jinými slovy – pokud se podaří objektové schéma navrhnout v duchu zásad síťového datového modelu.

S normalizací to je ještě složitější. Uvažuje se o využití refaktoringu a návrhových vzorů. V relačním datovém modelu jsou jednotkou funkční závislosti samotné atributy – tedy datové složky v záznamech a ne celé záznamy. Proto se bez normalizace v RDM neobejdeme. Objektový datový model ale pracuje s objekty, které navenek vystupují jako nedělitelné jednotky. Proto není problém normalizace tak náročný jako v RDM, jenže to neznamená, že neexistuje.

3.4.1.1 Možnosti uplatnění „klasické“ normalizace v objektových databázích

Nepochybně i v objektovém datovém modelování jsou užitečné formální techniky správného návrhu datové struktury. Potřebujeme je ze stejných důvodů, jako to činíme u relačních databází – tedy pro řešení možných problémů s redundancí a konzistencí dat. Podívejme se proto nejprve, proč nelze jednoduše použít techniky známé z relačních databází. Pro relační datový model je k dispozici:

1. datová normalizace,
2. metoda syntézy atributů a
3. metoda datové dekompozice atributů.

Objektový datový model ale není pokračováním relačního datového modelu. (Popis vlastností objektového datového modelu je například v [Hruška 1995, Kroha 1995, Bertino et al. 1995, Silberhatz et al. 2004]) Pro použití formálních technik se relační a objektový datový model od sebe podstatně liší a to především v následujícím:

- a) Objekty mají nejen datové složky jako relační záznamy, ale i metody. Je tedy otázkou, jak objekty popisovat. Používá se pojem "chování" (angl. behavior) objektu, které obsahuje datové složky i metody. Pro potřeby databázového modelování někteří autoři pracují také s pojmem "atribut", což jsou zvnějšku viditelné datové vlastnosti objektu. U takových atributů se potom nerozlišuje, zda mají implementační původ v datové složce nebo v metodě.
- b) Postupy relační datové normalizace jsou odvozeny od konceptu funkční závislosti mezi atributy. Atributy v relační databázi jsou "základní stavební jednotkou" a jsou to skalární a atomické hodnoty, z nichž se skládají jednotlivé relace. V objektovém datovém modelu ale pracujeme s celými objekty, které z "relačního" pohledu sdružují několik atributů dohromady. Navíc, atributy těchto objektů mohou být také dalšími objekty. To znamená, že "základní stavební jednotka" objektového datového modelu na rozdíl od relačního není ani atomická ani skalární.
- c) Třetím problémem jednoduchého převzetí relačních technik je koncept identity objektů. V relačním datovém modelu je identita tvořena hodnotou vybraných atributů, které mají úlohu

primárních klíčů relací. V objektovém datovém modelu je ale identita objektu zachována i v případech, kdy jsou mu změněny hodnoty všech jeho atributů. Proto lze vytušit, že například čtvrtá a pátá relační normální forma nemají v objektové databázi uplatnění, neboť v relačních databázích jsou používány k řešení problémů závislostí mezi atributy ve sdružených primárních klíčích.

3.4.1.2 Techniky objektově orientované normalizace

Od přelomu 80. a 90. let se objevilo několik různých prací (např. [Burleson 1999, Wai 1992]). První práce se věnovaly různým rozšířením relačních technik, ale v poslední době se již můžeme setkat s pracemi věnovanými přímo objektovým databázím.

Nootenboomova OONF

Podle holandského autora Henka Nootenbooma [Nootenboom 2004] jsou první tři normální formy univerzálně platné pro relační i objektové databáze. Jako náhradu za relační čtvrtou a pátou normální formu (a pravděpodobně i BCNF) zavádí koncept jediné objektové normální formy, která má následující definici:

A collection of objects is in OONF if it is in 3NF and contains meaningful data elements only.

Univerzální platnost 1NF, 2NF a 3NF z relačních technik je zajisté legitimní názor, ale definice OONF raději zůstane bez komentáře.

Chodorkovského ONF, 4ONF, 5ONF a 6ONF

Práce [Khodorkovsky 2002] zavádí pojem objektové normální formy (ONF), která se týká "správného" vztahu mezi daty objektu a metodami objektu. Pravidla takto zavedené objektové normální formy jsou potom doplňována ke klasickým "relačním" definicím 4NF, 5NF (a také k 6NF, která je autorovým zpřesněním 5NF). Autor tato doplnění klasických definic označuje jako 4ONF, 5ONF a 6ONF.

Tuto práci lze považovat za kvalifikovanější vyjádření podobných myšlenek jako v předchozím případě. I zde podle autora platí, že 1NF, 2NF a 3NF jsou pro relační i objektové databáze společné.

"Čínská" ONF

Práce [Yonghui et al. 2001] zavádí jedinou objektovou normální formu jako náhradu za všechny relační normální formy. Za objektový datový model se ale v této práci považuje stromová hierarchie tak, jak s ní pracuje jazyk XML. Takže se problematiky návrhu objektových databází, jak je chápeme my, netýká.

"Australsko-Švýcarská" ONF

Autoři [Zahir et al. 1997] zde zavádějí jednu ONF pomocí více typů funkčních závislostí mezi objekty. Konkrétně "path dependency" týkající se skládání objektů a navigability mezi objekty, "local dependency" týkající se vztahů vnitřních složek objektu a "global dependency" týkajících se požadavků na aplikaci. Struktura objektů je v ONF tehdy, pokud jsou uživatelské požadavky na aplikaci zpětně odvoditelné ze vztahů mezi objekty. Tato práce je především zajímavý příspěvek k problému testování souladu navrženého objektově orientovaného modelu aplikace se zadáním na ni.

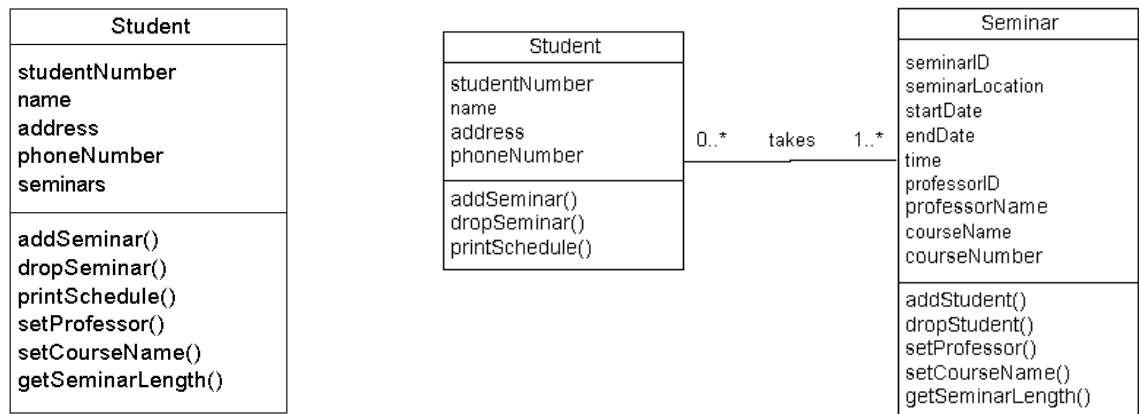
Tento problém s návrhem objektových databází souvisí, ale není to metoda správného návrhu z pohledu předcházení redundancím, nekonzistencím atp.

3.4.1.3 Ambler-Beckovy tři objektové normální formy

Na internetu [Ambler 2004] a také v knihách [Ambler 1997] jsou zavedeny tři objektové normální formy pro objektově orientované aplikace, které jsou do určité míry analogické s první, druhou a třetí relační normální formou. Autoři sami tyto normální formy označují 1ONF, 2ONF a 3ONF a hovoří o nich jako o nástroji pro normalizaci tříd objektů komplementární s technikou návrhových vzorů. Podívejme se na jejich návrh podrobněji:

1ONF

A class is in 1ONF when specific behavior required by an attribute that is actually a collection of similar attributes is encapsulated within its own class. An object schema is in 1ONF when all of its classes are in 1ONF.



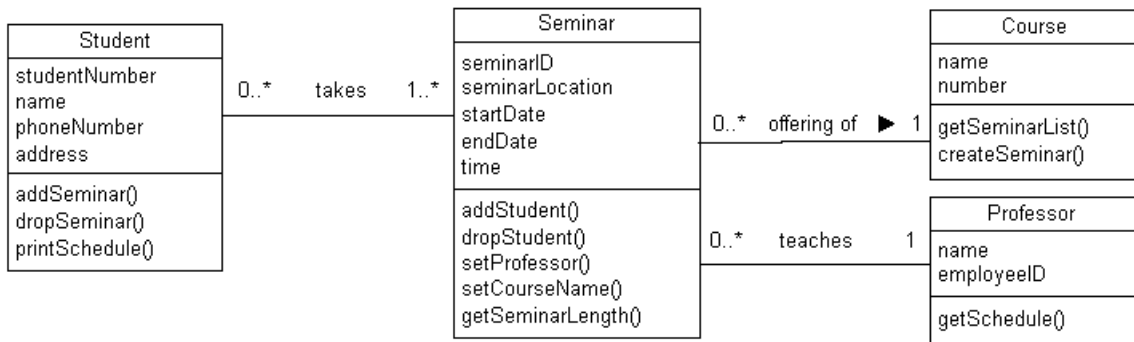
obr. 2. 0ONF Amblera a Becka [Ambler 2004]

obr. 3. 1ONF Amblera a Becka [Ambler 2004]

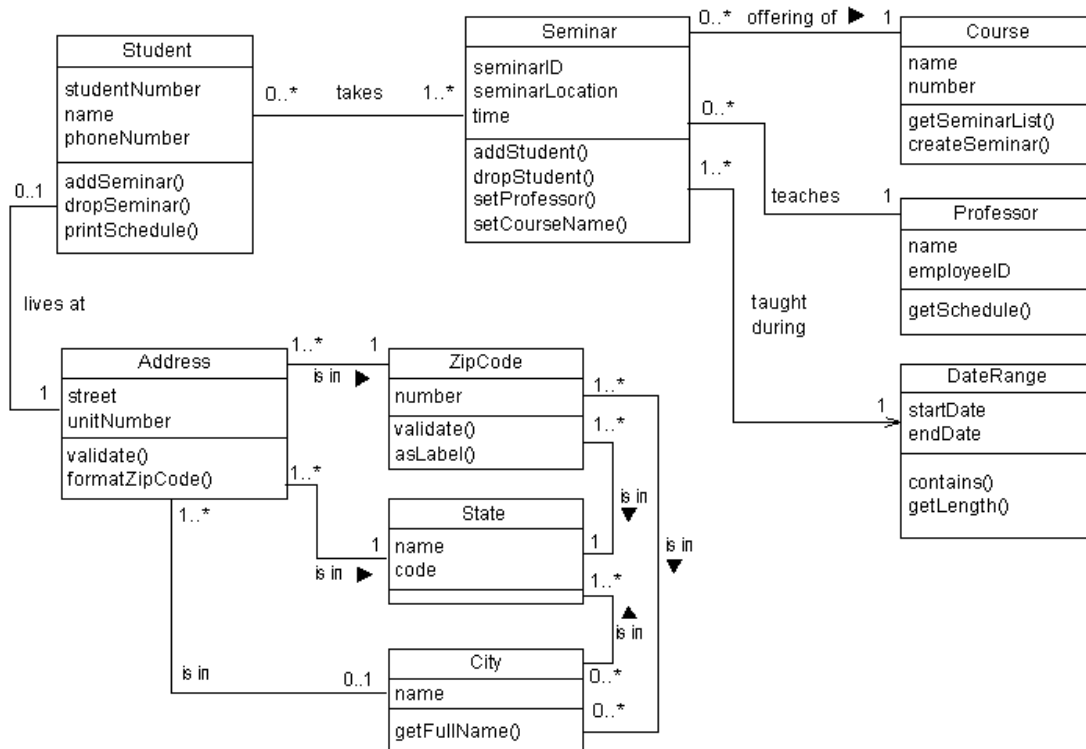
Z definice i příkladu je zřejmé, že autoři chtěli první normální formu postavit analogicky k první relační normální formě, což se jim také povedlo. Poněkud matoucí je ale skutečnost, že objekt může být v nenormalizovaném tvaru i tehdy, když již má přidruženou kolekci objektů ve vlastních třídách. – Na příkladu totiž třída Student skládá kolekci seminars již ve tvaru 0ONF. Z datového pohledu se tak kolekce seminars z 0ONF téměř neliší od vazby takes v 1ONF a rozdíl mezi 0ONF a 1ONF je víceméně jen v metodách třídy Student.

2ONF

A class is in second object normal form (2ONF) when it is in 1ONF and when "shared" behavior that is needed by more than one instance of the class is encapsulated within its own class(es). An object schema is in 2ONF when all of its classes are in 2ONF.



obr. 4. 2ONF Amblera a Becka [Ambler 2004]



obr. 5. 3ONF Amblera a Becka [Ambler 2004]

Jak ukazuje definice i příklad, tak druhá objektová normální forma požaduje vyčlenit vlastnosti, které jsou sdílené více objekty, do samostatných objektů. Tato normální forma je podle mého názoru velmi srozumitelná a dává analogické výsledky, jako druhá relační normální forma.

3ONF

A class is in third object normal form (3ONF) when it is in 2ONF and when it encapsulates only one set of cohesive behaviours. An object schema is in 3ONF when all of its classes are in 3ONF.

I v této třetí a poslední objektové normální formě lze rozpoznat, že dává analogické výsledky, jako třetí relační normální forma. Jde tu o to, že pokud máme v nějakém objektu vlastnosti, které by samy mohly být objektem, tak je vyčleníme do nového samostatného objektu. Pokud tak neučiníme, tak budou objekty držet více navzájem různých skupin vlastností, které autoři v definici označují jako "cohesive behaviors". Smysl této normální formy je nepochybný.

Ambler-Beckův přístup je zatím nejlepším výsledkem v oblasti formálních metod objektového datového modelování. Některé definice jsou sice formulovány příliš "programátorsky", ale tento přístup má svoje praktické uplatnění, což můžeme potvrdit z vlastních zkušeností z výuky i poradenství.

3.5 Metody tvorby informačních systémů

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

3.5.1 Nejpoužívanější metody řízení a podpory softwarových projektů

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

- a) ... je jednodušší a je důsledně procesně orientovaný ve srovnání s RUPem, na který lze nahlížet jen jako na obrovskou sadu jednotlivých objektových i neobjektových nástrojů a technik, ze kterých si projektový manažer musí umět správně vybrat.
- b) ... zabývá se i fází údržby a provozu již vyrobeného systému, čímž dobře do procesu začleňuje správu softwarové konfigurace, podporu uživatelům a vytváří předpoklady pro zpětnou vazbu na zahájení nových projektů. RUP se postimplementačními fázemi nezabývá.
- c) ... lépe podporuje tvorbu v čistých objektových jazycích a databázích (důraz na znovupoužitelnost, refactoring, komponentový přístup, ...) a lépe do celkového procesu začleňuje využívání metrik. RUP je šitý na míru konzervativnější technologii hybridních programovacích jazyků (např. C++ nebo Java) a relačních databází.
- d) ... lze využít i pro jiné metody analýzy a návrhu, než z dílny UML. Amblerův přístup je slučitelný například s metodou BORM a dokonce i s extrémním programováním.

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

3.6 Iterativní a evoluční versus sekvenční model životního cyklu

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

3.6.1 Sekvenční model životního cyklu

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

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

3.6.2 Iterativní model životního cyklu

Iterativní model je založený na myšlence, že je možné se vracet do předchozích fází vývoje projektu za účelem zpřesnění zadání. Nejznámější varianta tohoto přístupu používaná v OOP je prototypový model a spirální model. Právě skutečnost, že lze zahájit tvorbu dříve, než jsou známy všechny požadavky na systém a použít zkušenost z prvotní implementace pro upřesnění zadání, měla za následek velkou oblibu tohoto stylu v komunitě OOP.

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

3.6.3 Evoluční model životního cyklu

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

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

3.7 Rigorózní versus agilní metodiky

3.7.1 Rigorózní metodiky

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

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

3.7.2 Agilní metodiky

Agilní metodiky jsou samotnými autory označovány jako metodiky, které umožňují vytvořit řešení velmi rychle a optimálně a toto řešení dovolují pružně přizpůsobovat. Jedná se o několik metodik, z nich nejpopulárnější je takzvané extrémní programování (XP). Agilní přístup (AP) byl s velkým nadšením přijat v komunitě programátorů pracujících s OOP. Představitelé těchto přístupů z celého světa v roce 2001 vydali společný dokument „Agile Manifesto“ a vytvořili alianci pro „agilní vývoj softwaru“. [AP 2001, Beck 2003, Beck 2002]

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

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

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

Přestože doposud uvedená informace o agilním přístupu vypadá krajně podezřele, tak agilní přístup má v praxi dobré výsledky, což můžeme potvrdit i my². Autoři agilních metodik jsou většinou pro věc nadšení, ale také velmi vzdělaní lidé. Někteří z nich odborně publikují i v oblasti formálních technik a jiných metod řízení projektů, což dokladují například i citace Amblera a Becka v tomto textu.

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

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

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

Podrobný popis agilních metodik s důrazem na extrémní programování je obsahem samostatné kapitoly sekce tohoto textu.

3.7.3 Příčina sporu

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

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

² Například modelovací nástroj Craft.CASE, který je popisován v příloze této práce a jehož zadavatelem/analytikem je autor, je programován podle zásad XP.

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

Rigorózní metodiky pro vývoj s pomocí OOP mají za sebou dlouhý vývoj. Většinou vznikly postupnou změnou z klasických strukturovaných metodik. Dnes používané metodiky mají svůj původ ve výzkumu a zkušenostech s tvorbou softwaru pomocí OOP v 90. letech. Zde jsou příčiny, proč nespĺňují očekávání a selhávají. Zároveň se zde skrývá vysvětlení, proč došlo ke vzniku AP. Příčiny jsou dvě:

- a) První spočívá ve změně stylu programování, jak byla popsány v kapitole 3.2.1. Rigorózní metodiky selhávají proto, že se nedostatečně vyrovnaly se změnou nástrojů a technik pro programování a implementaci. Jednoduše řečeno; rigorózní metodiky stále dostatečně nezaznamenaly výraznou kvalitativní změnu ve způsobu tvorby programů a stále předpokládají, že ve fázi implementace je jen potřeba z celkového modelu úlohy vyprodukovat zdrojový kód, který se dodá kompilátoru k překladu. Vývojáři proto oprávněně hledí s nedůvěrou na rigorózní metodiky, když na jedné straně vyžadují tvorbu perfektního konceptuálního modelu a na druhé straně nevyužívají možnosti, které mají dnešní vývojové prostředí. Naproti tomu AP nejen umožňuje, ale přímo vyžaduje plnou podporu vlastností nových vývojových prostředí a navíc bez potřeby modelování.
- b) Další problém spočívá ve schopnosti rigorózních metodik správně, účinně a rychle zachytit, analyzovat a zobrazit zadání od uživatelů. K modelování se dnes téměř výhradně používá UML. V kapitole 4.2.2 byla diskutována jeho nedostatečná podpora úvodních fází modelování. AP je totiž také odpovědí na to, že UML nepodporuje dobře úvodní analýzu. Proto AP radí raději žádnou analýzu zadání nedělat a místo toho řešit projekt po částech a pravidelně spolupracovat s uživatelem. Bohužel vinou nedostatků rigorózních metodik je dnes AP prakticky úspěšné i v oblastech, kde by uživatelé byli schopni a dokonce by preferovali na začátku projektu formulovat požadavky – nebo jejich podstatnou část. Ale protože je v UML obtížné je srozumitelně modelovat, tak se raději nemodeluje vůbec a přistupuje se k AP, přičemž uživatelé musí průběžně komunikovat během vývoje. Toto je problém větších projektů, například specifických aplikací pro business, systémů pro řízení technologií (v telekomunikačním průmyslu nebo distribuce energií), atd. Ve všech těchto oblastech je třeba projekty plánovat a zadání je z podstatné části známé předem.

3.8 Tvorba informačních systémů v kontextu podnikového managementu

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

3.8.1 Procesy a procesní modely – requirement engineering

Právě procesy a procesní modely jsou ověřenou a v praxi používanou metodou pro analýzu, návrh a implementaci organizačních změn za aktivní spoluúčasti zadavatelů (interview, workshopy, ...). Těmito problémy se zabývá poměrně nedávno konstituovaný obor aplikované informatiky, který je anglicky označován „requirement engineering“. Více lze nalézt kupříkladu v [Kotonya et al. 1999, Hammer et al. 1994]. Z objektově orientovaného procesního modelu lze dobře s aktivní pomocí zadavatelů najít a) funkce, b) strukturu, c) rozsah požadovaného systému a d) také role budoucích uživatelů vytvářeného systému.

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

3.8.2 Myšlenka konvergenčního inženýrství

Objektová technologie může naštěstí poměrně jednoduše modelovat jak softwarové systémy, tak i systémy podnikové či organizační, které můžeme souhrnně podle některých autorů nazvat jako systémy sociotechnické. Právě proto, že jedna technologie slouží k modelování obojího, tak není nemožná myšlenka modelovat podnikový a informační systém ne jako modely dva, ale jako jeden model a změny a vlastnosti procesů přímo promítat do změn a vlastností softwaru a naopak. Tento přístup je podrobně popsán například v [Taylor 1995].

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

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

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

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

Objektový přístup je sice předpokladem, ale samotná technika, ani drahý CASE nástroj zde nestačí. Pokud při modelování neexistuje vedoucí pracovník s vizí, který je schopen a ochoten změny prosadit, tak není možné projekt úspěšně realizovat a zahájení projektu je ztrátou času a peněz. Této zodpovědnosti se vedoucí pracovníci nemohou jednoduše zbavit tím, že objednájí drahé konzultační

služby a jmenují do funkce vedoucího projektu podřízeného pracovníka, u kterého není v souladu zodpovědnost a povinnosti s potřebnou mírou autority a pravomocí.

3.8.3 Vztah mezi informačním a řídicím systémem uvnitř organizace

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

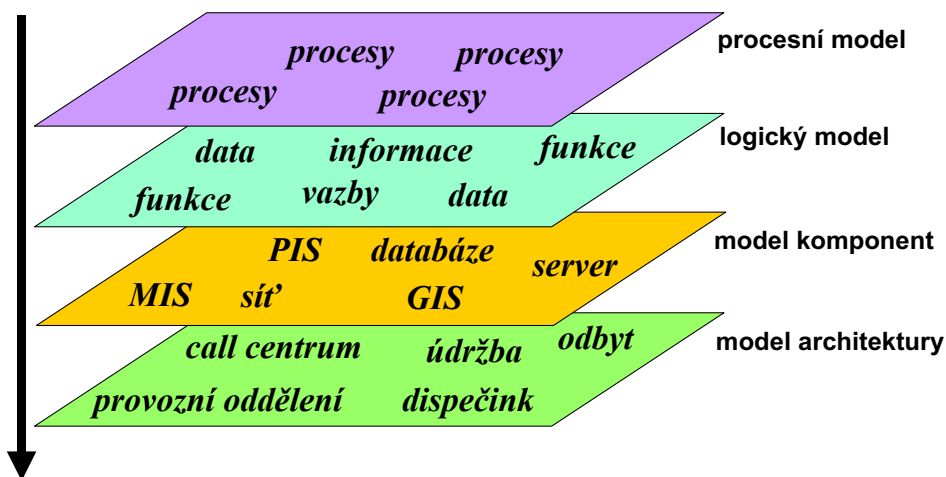
- 1) Očekává se, že problémy řídicí a organizační povahy, které organizace má, musejí být řešeny vybudováním či nákupem „nového informačního systému“ (nebo jeho nové komponenty). = Příčina problémů v řídicím systému se nesprávně chápe jen jako nedostatečné vybavení informačními technologiemi.
- 2) Očekává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

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

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

řídicí systém = organizační struktura + pravidla, řídicí funkce, kompetence, legislativní podpora, ... + informační systém.

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



obr. 6. čtyři vrstvy modelu řídicích systémů podniku nebo organizace

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

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

Dochází-li ke změně na jakékoliv úrovni, tak je vhodné provést rozbor dopadů této změny na okolní úrovně postupně až k procesům. Například rozhodnutí vedení organizace „postavíme si nové výpočetní středisko“ nebo „v budově XY uděláme oddělení Z pro styk se zákazníky“, které se týká úrovně architektury, má smysl pouze tehdy, víme-li jaké subsystémy z nadřazené úrovně tento zásah vyžadují, jaké logické důsledky z další nadřazené úrovně to přinese a kterých procesů z nejvyšší úrovně se bude změna týkat, tedy zda procesy změníme, vylepšíme, zrušíme nebo nastavíme nové. Podobné nebezpečí na úrovni komponent v sobě skrývá například záměr „začneme používat GIS“, zavedeme „webový portál na internetu“ atd. Jestliže se totiž rozhodnutí provádějí pouze na úrovni, kam prvek patří, tak reálně hrozí, že jediné, co změna organizaci přinese, jsou zbytečně vyhozené peníze a pro mnoho lidí přidělení práce navíc.

3.8.4 Vztah k OOP

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

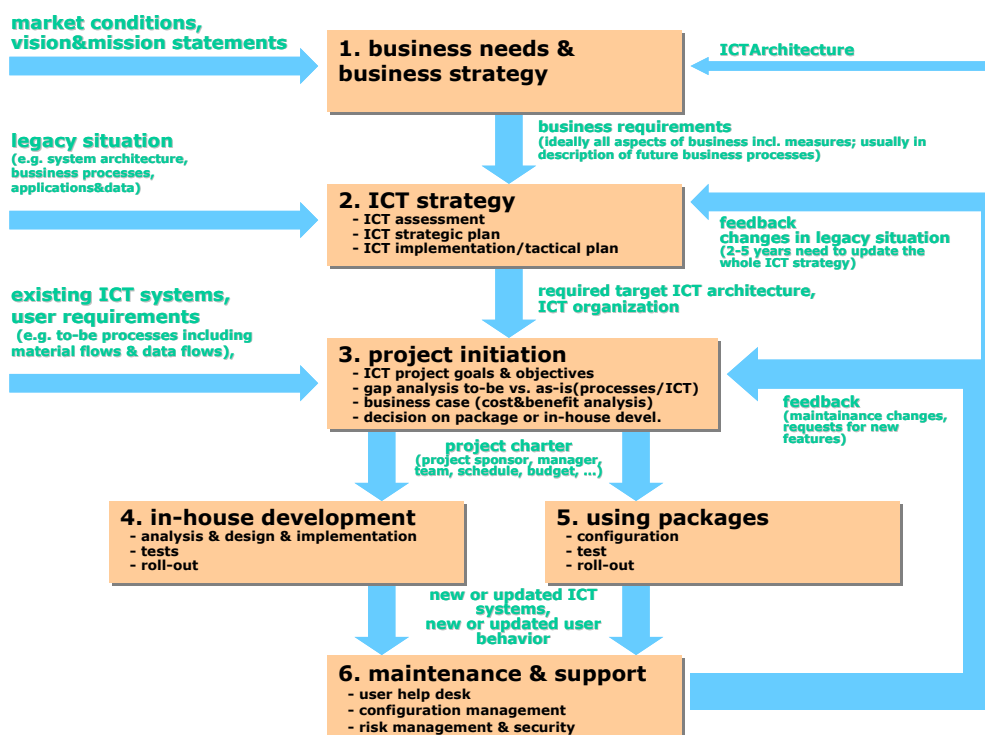
Situace s AP je ještě méně radostná. Filosofie AP jde zcela proti zde popsaným zásadám. Žádná analýza, žádné rozbory a reorganizace business procesů, žádné organizační změny. Jen co nejrychlejší implementace podle okamžitých potřeb uživatele.

4 Doporučovaný postup při projektování s využitím OOP

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

4.1 Celopodnikový pohled

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



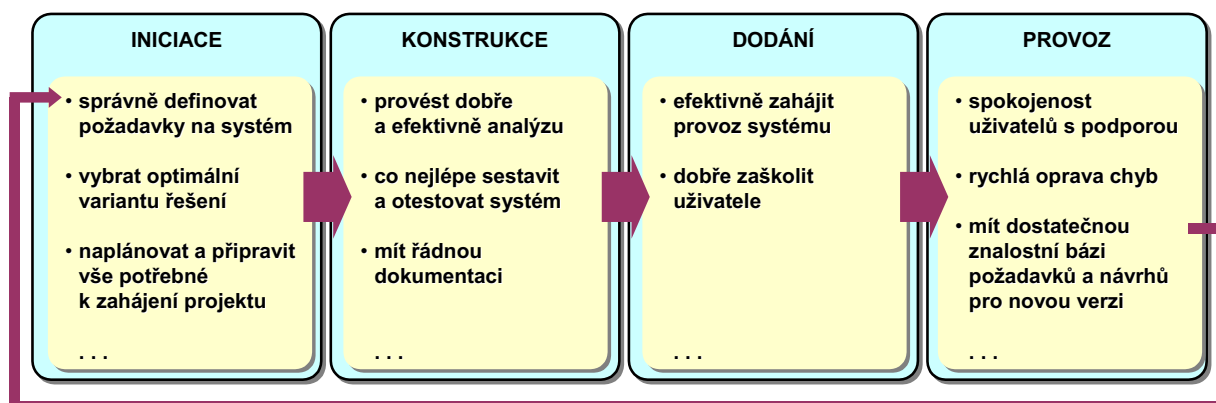
obr. 7. ICT management [Hall et al. 2004]

Na tomto schématu jsou následující bloky činností:

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

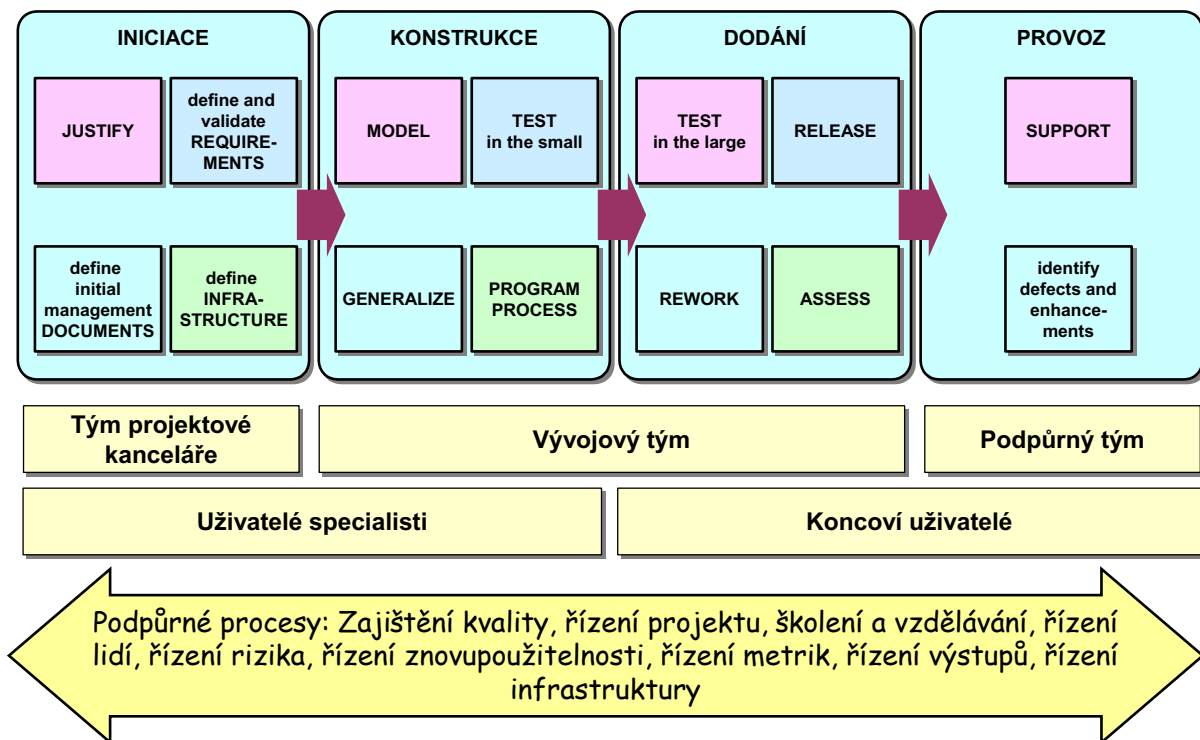
4.2 Model životního cyklu

Přípravu i realizaci software je nutno vidět jako proces. Následující návrh řešení je aplikací původního amblerova přístupu popsaného v kapitole 3.5.1. Tento přístup, jak již bylo řečeno, nahlíží na tvorbu softwaru očima manažera z perspektivy projektového řízení. To znamená, že zde popisujeme něco trochu jiného, než klasické metody analýzy a návrhu, které jsou v tomto chápání metodami „pro dělníky“. (Při určité míře zjednodušení to znamená, že například celá metoda „extrémního programování“ nebo OMT může být považována za konkrétní náplň jediné fáze konstrukce zde popisovaného přístupu).



obr. 8. čtyři hlavní fáze životního cyklu

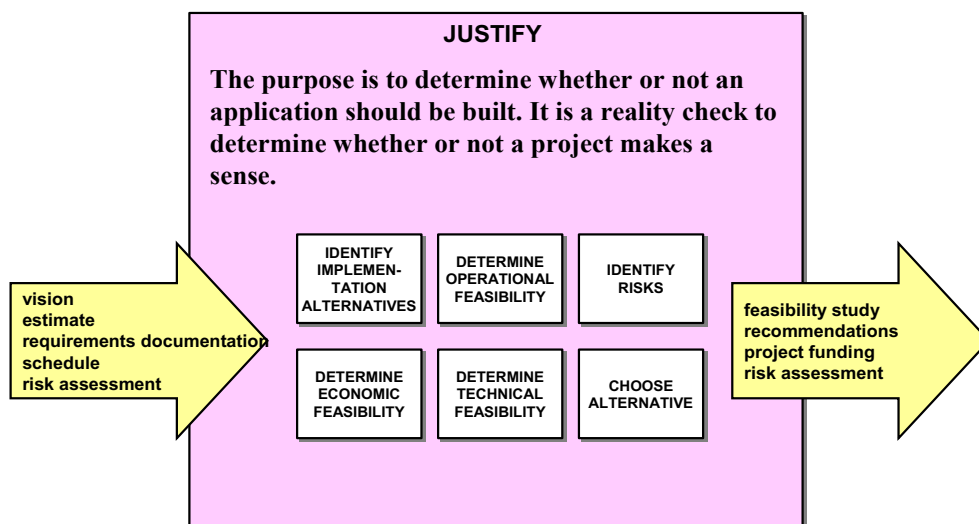
Jak je uvedeno na obrázku, celý proces je členěn na čtyři fáze pojmenované „iniciace“, „konstrukce“, „dodání“ a „provoz“. Každá z fází se skládá ze dvou až čtyř dílčích procesů.



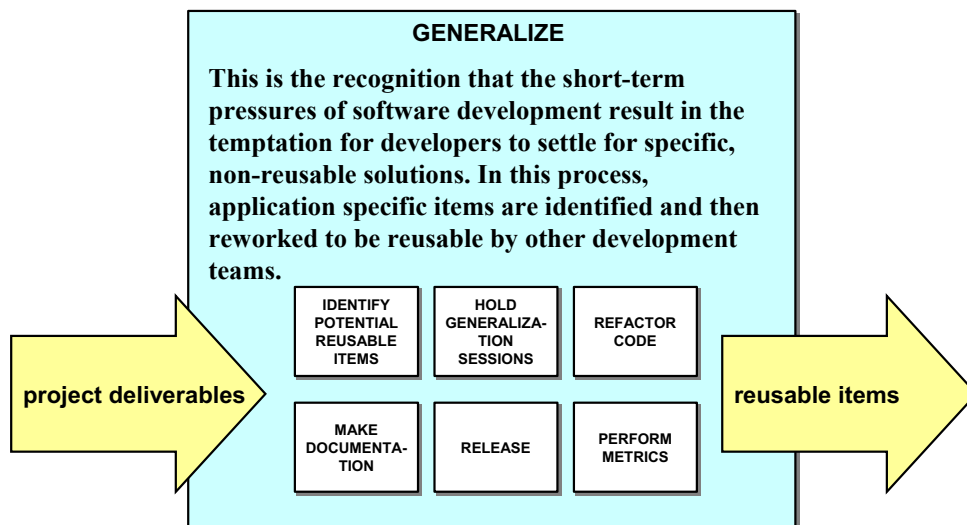
obr. 9. čtyři hlavní fáze životního cyklu - detail

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

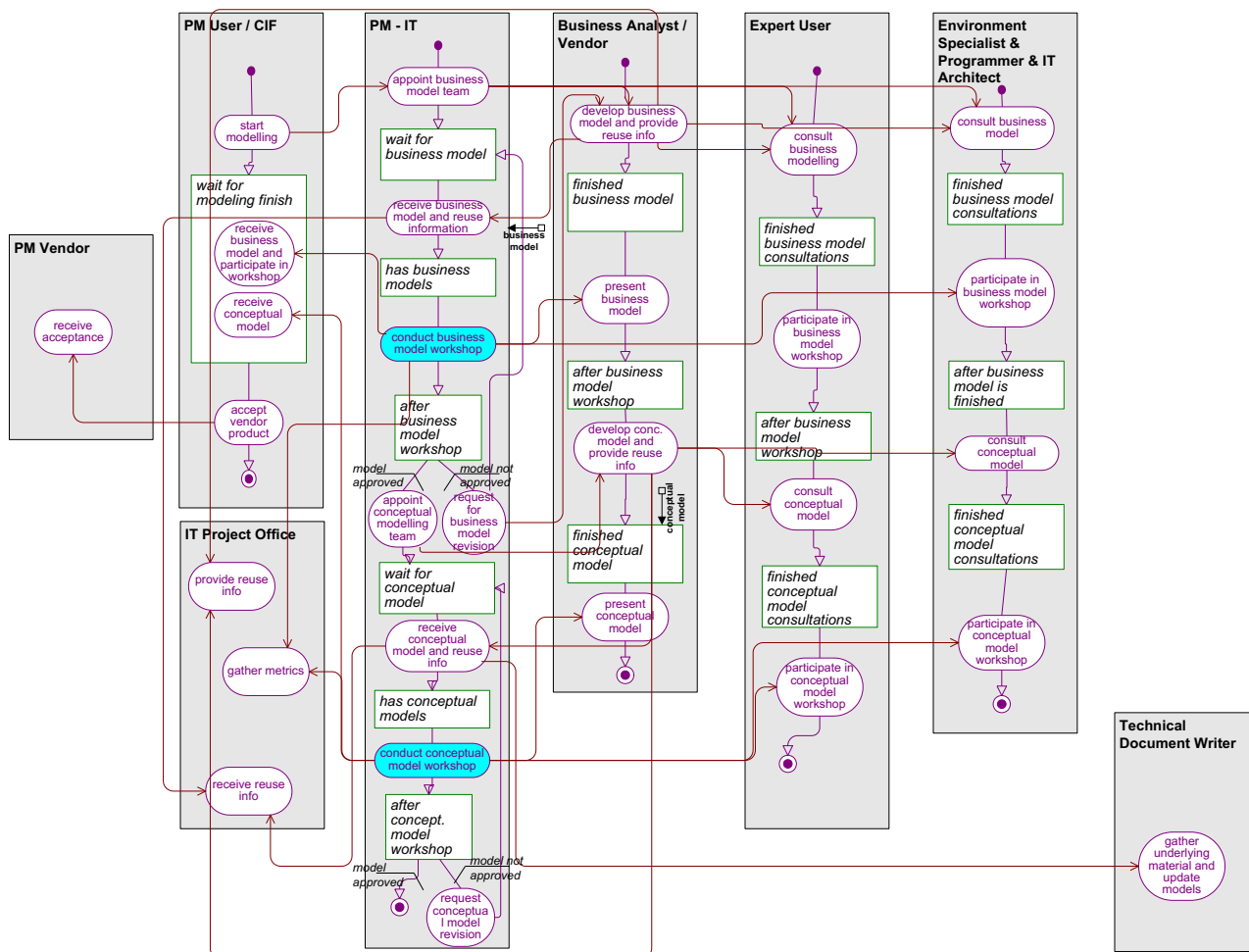
Každý z dílčích 14 procesů je možné v konkrétních podmínkách modelovat ve formě procesní mapy, kde lze vyznačit podrobný výčet potřebných aktivit a vymezení rolí účastníků procesu. Na obrázcích je příklad podrobnějšího obsahu dílčích procesů a příklad procesní mapy z konkrétního projektu firmy Deloitte&Touche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



obr. 10. obsah procesu INITIATE-JUSTIFY



obr. 11. obsah dílčího procesu CONSTRUCT-GENERALIZE



obr. 12. nastavení procesu konstrukce (z projektu Deloitte&Touche pro T-Mobile ČR)

4.2.1 Iniclace

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

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

4.2.2 Konstrukce

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

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

4.2.3 Dodání

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

- a) Vydání (release) je proces, který zajišťuje přechod na provozní platformu, řešení problémů s instalací na této platformě a přípravu provozní infrastruktury. Jeho součástí jsou i podpůrné a

Tým projektové kanceláře

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

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

Tým „help desku“

Je to stálý tým s těmito úkoly:

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

Zahajovací tým

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

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

Pracovní tým

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

1. Zodpovídá za úspěšné vytvoření systému za dodržení vstupních omezujících podmínek

2. Produkuje všechny dohodnuté výstupy (software, dokumentace, příručky, rozčlenění na opakovatelně použitelné prvky, sběr hodnot příslušných metrik aj.).
3. Zodpovídá za projekt a jeho výsledek v osobě manažera projektu. Manažerem projektu by neměl být technik-programátor a při implementaci většího cizího systému ani šéf IT zadavatele.
4. Zajišťuje spolupráci manažera projektu s projektanty pro oblast řešení technických složek projektu.

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

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

4.2.6 Provozní, testovací a vývojová platforma

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

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

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

4.3 Postupná transformace pojmů při projektování

V této kapitole bude popsán soubor objektově orientovaných technik a metod, které je vhodné použít během konstrukce softwarové aplikace. Postupy zde uvedené tvoří metodu BORM, která je původní českou metodikou pro analýzu a návrh informačních systémů. BORM byl publikován především v [Merunka et al. 2000, Carda et al. 2003] a v zahraničí také v [Knott et al. 2003, Liu et al. 2005, Knott et al. 2000].

4.3.1 Metoda BORM

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

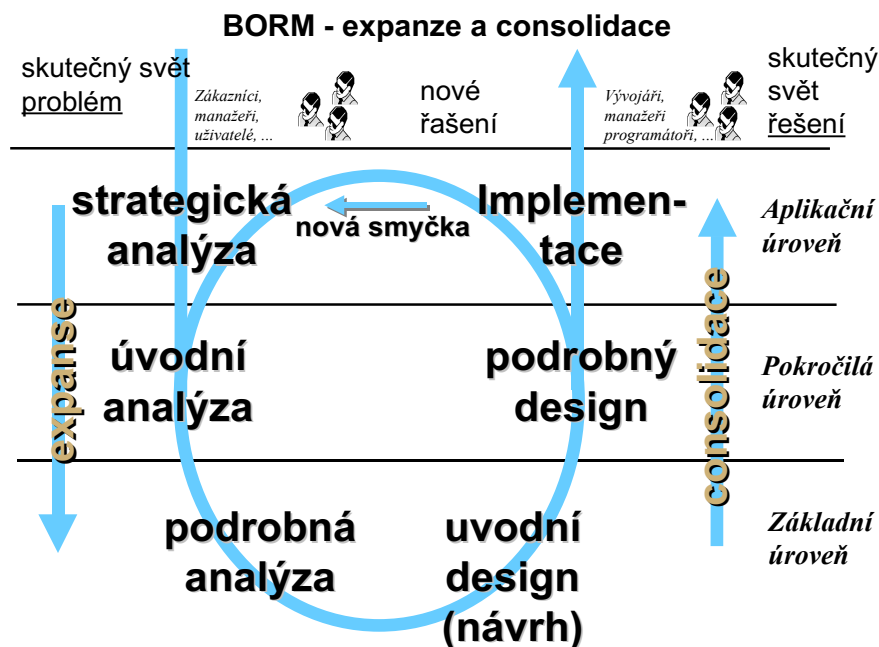
Práce na BORMu byla od svého počátku součástí grantu VAPPIENS (research project Various Programming Paradigms in Integrated Environments), který je součástí programu "Know How Fund of Czech Academic Link Programme" Britské rady (British Council). Od roku 1996 je další vývoj podporován firmou Deloitte&Touche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

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

BORM rozlišuje 6 fází životního cyklu vývoje systému:

- 1) Strategická analýza. Zde dochází k vymezení samotného problému, je stanoveno jeho rozhraní, jsou rozpoznány základní procesy, které se v systému a také v jeho okolí mají odehrávat.

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



obr. 14. 6 fází životního cyklu vývoje systému v BORMu

4.3.1.1 Použití BORMu v praxi

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

Následující tabulka ukazuje rozsahy vybraných konkrétních projektů prováděných mezinárodní poradenskou firmou Deloitte&Touche metodou BORM v průběhu let 1997-2000.

projekt	počet systémových funkcí	počet scénářů	počet procesních diagramů	počet objektů ³ (participantů)	průměrný počet stavů na objekt	průměrný počet aktivit ⁴ na objekt
Agrární komora ČR (analýza veřejného inf. systému pro obchod ovocem a zeleninou)	4	7	7	26	4	4
Zdravotnický komplex (klinika, fakultní nemocnice, ...) – BPR organizační struktury	6	12	12	18	10	12
Společnost zabývající se celostátním bezdrátovým přenosem TV, radio, telef. a datových signálů (BPR projekt a přechod společnosti na tržní prostředí)	4	9	9	14	8	8
Krajská elektroenergetická společnost (analýza zákaznického inf. systému)	12	19	19	23	12	12
Krajská elektroenergetická společnost (analýza a prototyp inf. systému pro evidenci a likvidaci poruch)	19	31	34	27	13	14
Krajská plynárenská společnost (BPR celého podniku)	28	81	97	210	11	12
Krajská plynárenská společnost (BPR celého podniku)	23	60	63	120	12	12
Podnikový autoprovoz (analýza, návrh a implementace distrib. IS firmy s několika pobočkami v ČR)	7	5	8	12	9	11
IS pro podporu a evidenci financování projektů vědeckého pracoviště (analýza, návrh a implem. webového IS)	10	12	5	11	6	7

obr. 15. projekty v BORMu za období 1997-2000

4.3.2 Vývoj pojmu objekt během projektování

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

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

- a) Aplikační úroveň představuje okruh znalostí a dovedností, se kterými přichází jedna nebo druhá skupina do běžného pracovního styku. Pojmy z aplikační oblasti jsou proto pro příslušníka skupiny známé a srozumitelné. Bohužel jsou však nejméně vzdálené pojmům aplikační úrovně skupiny druhé a informační systémy proto nelze stavět pouze na této úrovni.
- b) Základní úroveň představuje komunikační optimum mezi velmi od sebe vzdálenými aplikačními úrovněmi. Největší roli zde hraje konceptuální modelování, které dovoluje dostatečně srozumitelné a formální diagramové nástroje, které jsou na jedné straně ještě sledovatelné

³ Objekty, které tvoří obsah datových toků na komunikacích mezi aktivitami participantů nejsou započítány.

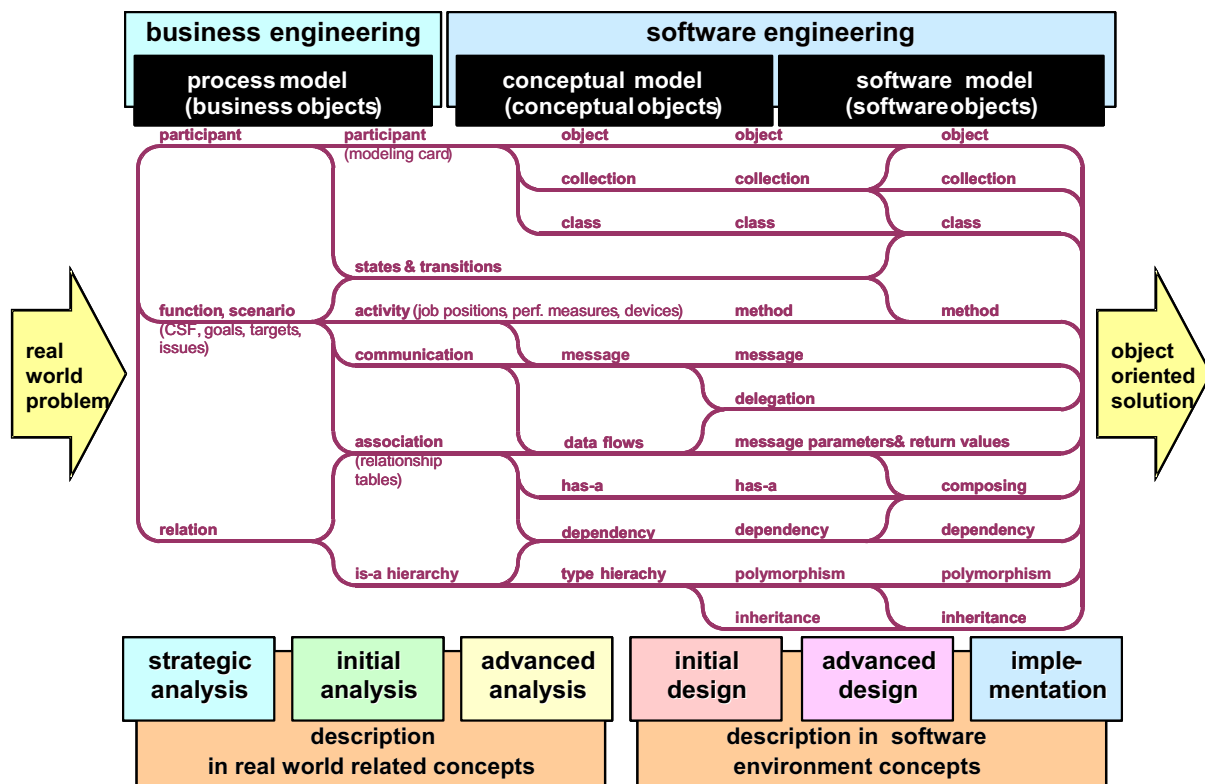
⁴ v projektech BPR každá aktivita navíc obsahuje sadu přibližně 6-10 dalších údajů jako například vazbu na cíl (business goal), popisy pracovních činností, popisy pracovních míst, čas potřebný pro vykonání dané aktivity atd.

„niprogramátory“ z první skupiny a zároveň na druhé straně poskytují dostatek informací pro příslušníky druhé skupiny.

- c) Pokročilá úroveň je tvořena množinou nástrojů, technik a znalostí, které v první skupině dovolují najít správný konceptuální model a druhé skupině tento model umožňují transformovat do softwarové podoby. Rozdíl mezi dobrým a špatným analytikem je právě v největší míře dát mírou znalostí z této úrovně.

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

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



obr. 16. postupná transformace objektového modelu

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

1. Softwarové objekty, se kterými se pracuje v závěrečných fázích vývoje IS za účelem softwarové implementace. Tyto objekty obsahují pojmy přímo odpovídající konstrukcím z objektových programovacích jazyků a nebo standardu UML.
2. Konceptuální objekty, se kterými se pracuje v prostředních fázích vývoje IS. Tyto objekty obsahují základní pojmy objektově orientovaného paradigmatu, jako například polymorfismus objektů, zapouzdření, skládání, delegování, klasifikace objektů podle různých dimenzí, závislost objektů, třídy a množiny objektů atd. Je pravda, že mnohé z konceptuálních pojmů jsou shodné se softwarovými pojmy, ale značnou část z nich je třeba při přechodu na softwarové objekty transformovat, protože současné používané programovací jazyky podporují pojmy OOP pouze omezeným způsobem. Zhruba řečeno je rozdíl mezi konceptuálními a softwarovými objekty závislý na použitém programovacím prostředí a je proto např. v případě C++ větší, než při použití Smalltalku. Smyslem tvorby modelu s konceptuálními objekty je snaha mít implementačně nezávislou ale dostatečně podrobnou dokumentaci softwarového návrhu, která by byla použitelná i pro inovace systému po změně technologie. To by nebylo možné, kdyby se modelovalo jen podle možností aktuálního programovacího prostředí.
3. Objekty reálného světa, (anglicky jako „business objects“). Tyto objekty vyplňují mezeru mezi zadáním - tj. chápáním na aplikační úrovni zadavatele a mezi konceptuálním objektovým modelem. Objekty reálného světa podporují pouze vybrané pojmy OOP a většinu pojmů ponechávají na pozdějších transformacích. Model objektů reálného světa je použitelný nejen pro zahájení analýzy softwarové aplikace, ale i pro podporu práce např. manažerů pro tvorbu modelů při rozhodování, aniž by projekt vždy nutně končil softwarovou implementací.

4.3.3 Fáze expanze a konzolidace

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

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

Zbývající fáze od konceptuálního modelu až k finálnímu systému složenému ze softwarových objektů se označují jako stadium konzolidace. Je tomu tak proto, že v těchto etapách se model, který je produktem předchozí expanze, postupně stává fungujícím programem. To znamená, že na nějakou myšlenkovou "expanzi" zadání zde již není prostor ani čas. V tomto stadiu se také počítá s tím, že od některých idejí z expanzního stadia bude třeba upustit vzhledem k časovým, kapacitním, implementačním nebo i intelektuálním schopnostem - odtud tedy název tohoto stadia. (Takto odstraněná informace však může být v budoucnu základem analýzy nové verze systému).

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

4.3.4 Objekty reálného světa (business objekty)

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

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

4.3.4.1 Metoda OBA

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

Metoda vznikla počátkem 90. let na základě zkušeností s aplikacemi různých technik JAD (Joint Application Design) a CRC (Class-Responsibility-Collaborator) pro potřeby objektové analýzy a návrhu a implementace v objektově orientovaných programovacích jazycích. [Rubin et al. 1992, Bellin 1997]

- | |
|---|
| <ol style="list-style-type: none">1. Zaměstnanci (referenti) používají auta pro svoje služební účely2. Někteří zaměstnanci (referenti) dostávají dlouhodobě přidělená auta pro služební účely3. Vedoucí zaměstnanců rozhodují o přidělení služebních aut zaměstnancům na dlouhodobou výpůjčku4. Autoprovoz zajišťuje nákup či pronájem aut a jejich technický servis5. Autoprovoz přiděluje auta zaměstnancům na dlouhodobou výpůjčku6. Zaměstnanci požadují přidělení referentského vozidla na konkrétní služební cestu7. Autoprovoz přiděluje referentská auta zaměstnancům8. Vedoucí zaměstnanců potvrzuje žádost autoprovozu o výpůjčku auta v půjčovně pro služební cestu zaměstnance v případě, že není k dispozici referentské vozidlo pro krátkodobou výpůjčku, nebo rozhodne o dočasném přerušení dlouhodobé výpůjčky |
|---|

obr. 17. příklad seznamu funkcí

Jedná se o iterativní techniku začínající řízeným interview se zadavateli a pracující s různými typy formulářů, tabulek a modelových karet, ke kterým přísluší sada postupů a pravidel. Podrobnější popis OBA analýzy lze například nalézt v [Rubin et al. 1992]. Jednotlivé kroky OBA analýzy, jak jsou použity v BORMu, jsou následující:

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

2. krok - definování objektů pomocí modelových karet. V tomto kroku se pro každý rozpoznáný objekt z předchozího kroku vytvoří jeho modelová karta, která obsahuje jméno objektu, seznam aktivit objektu a s ním související seznam s modelovaným objektem spolupracujících objektů. Předpokládá se, že pro každý rozpoznáný spolupracující objekt je také vytvářena jeho modelová karta.
3. krok - klasifikace objektů. V tomto kroku dochází k přidání další informace k modelovým kartám jednotlivých objektů. Modelové karty jsou tříděny podle různých kritérií a na základě nalezených podobností se seznam modelových karet upřesňuje a doplňuje.
4. krok - sestavení tabulky vztahů mezi objekty. Tabulka vztahů v nejjednodušší podobě vyjadřuje jaký objekt má vztah s jiným objektem.
5. krok - modelování životních cyklů objektů. V tomto kroku se pro každý rozpoznáný objekt s pomocí informací v tabulce scénářů, modelových kartách a tabulkách vztahů sestaví životní cyklus objektu jako sled jeho stavů a přechodů mezi těmito stavy v podobě procesního diagramu. Tento poslední krok lze v případě první iterace provést ihned po kroku 1. a 2. a teprve poté provést kroky 3. a 4.

Scenario 1	Derived from functions: 2 , 3 , 4 , 5 , 8	Phase: as is
Initiation: Rozhodnutí vedoucího	Action: Přidělení služebního auta zaměstnanci na dlouhodobou výpůjčku	Result: Zaměstnanec má přiděleno služební auto na dlouhodobou výpůjčku
Auto (default)		
Referent (performs)		
Vedoucí (approves)		
Vedoucí autoprovozu (cooperates)		
Scenario 2	Derived from functions: 2 , 3 , 4 , 6	Phase: as is
Initiation: Rozhodnutí vedoucího	Action: Zrušení dlouhodobé výpůjčky služebního auta	Result: Zaměstnanec vrací dlouhodobě vypůjčené auto do autoparku, autopark po převzetí auta zajistí technickou prohlídku
Auto (default)		
Referent (is informed)		
Vedoucí (is responsible)		
Vedoucí autoprovozu (cooperates)		

obr. 18. příklad scénářů (generováno z Craft.CASE)

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

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

1. poskytuje prostředky pro dokumentování projektu od samého počátku,

2. modelové karty a další výstupy OBA jsou znovupoužitelné v dalších podobných projektech (například jako návrhové vzory) a
3. úsilí vynaložené při sestavování scénářů a životních cyklů objektů lze využít při návrhu optimální funkčnosti uživatelského rozhraní.

Collaborators of: Referent in diagram 'výpůjčka auta':	Auto	Vedoucí	Vedoucí autoprovozu
start: požaduje auto		✓	
čeká na rozhodnutí vedoucího: dostává potvrzení žádosti		✓	✓
čeká na rozhodnutí vedoucího: dostává zamítnutí žádosti		✓	
čeká na přidělení auta: je informován o přidělení auta			✓
čeká na přidělení auta: je informován o ukončení rezervace		✓	
dostal přidělené auto: vyzvedává auto z autoparku	✓		
má auto ve službě: vrací auto do autoparku	✓		

obr. 19. příklad modelové karty (generováno z Craft.CASE)

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

4.3.4.2 Diagram ORD

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

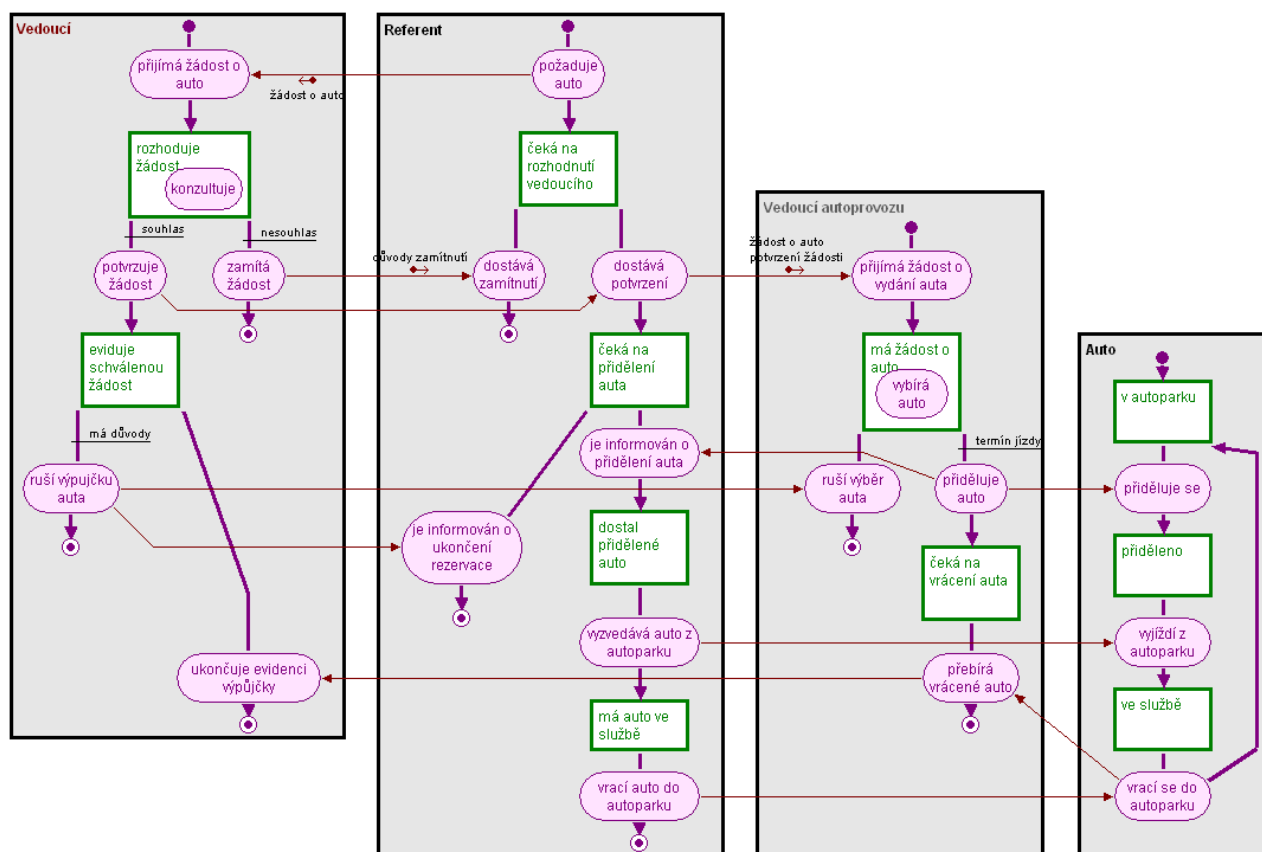
Pojem	symbol	Popis
Objekt = Participant	Obdélník se jménem zobrazeným uvnitř v levém horním rohu.	Objekt (je označován jako "Participant") představuje účastníka modelovaného procesu.
Stav	Menší obdélník odlišený barvou a typem písma pro pojmenování stavu kreslený dovnitř symbolu pro objekt.	Stavy vyjadřují postupné změny participantů v čase.
Asociace	Silná černá šipka s plným zakončením mezi participanty a nebo stavy. U šipky se píše popis, který blíže specifikuje charakter vazby.	Asociace vyjadřují datově orientované vztahy mezi participanty a nebo jejich stavy, protože se mohou v čase měnit. Vyjadřují jednotným způsobem vztahy, které mohou být později upřesněny jako skládání, dědění nebo závislost objektů.
Aktivita	Ovál propojený čarou s participantem nebo jeho stavem. Ovály mohou být kresleny také dovnitř k nim příslušných objektů.	Aktivita reprezentují jednotlivé aspekty chování objektů tak, jak byly rozpoznány pomocí scénářů v modelových kartách.
Komunikace	Šipka, která propojuje aktivity mezi sebou. Malé pojmenované šipky kreslené rovnoběžně k hlavní šipce komunikace vyjadřují datové toky.	Komunikace vyjadřují sled provádění a vzájemnou závislost aktivit různých objektů mezi sebou. datové toky mohou být vedeny oběma směry.
Přechod	Šipka s nevyplněným trojúhelníkovým zakončením, která propojuje aktivity a stavy jednoho objektu.	Součástí přechodu je také aktivita, ze které přechod vychází. Přechod tedy představuje činnost, kterou je třeba vykonat, aby objekt změnil svůj stav.
Podmínka	Přeškrtnutí s textovým popisem u komunikace nebo u propojení aktivity a objektu.	Podmínkou se vyjadřuje omezená platnost komunikace nebo aktivity.

obr. 20. pojmy diagramu ORD

ORD dovoluje modelovat jednotlivé procesy současně dvojím způsobem:

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

Vzhledem k tomu, že modelovaný proces je konstruován jako propojení rolí (stavů a přechodů) účastnících se objektů, tak ORD dovoluje jednoduchým a nenásilným způsobem zachytit přesný průběh modelovaného procesu a poskytuje tím i prostředky pro ověřování jeho správnosti. (Do ORD totiž není například možné přidat aktivitu, která by nenavazovala na nějaký již přítomný stav nebo nebyla vázána nějakou komunikací s jinou aktivitou.) Tyto vlastnosti, které přímo vyplývají z použité teorie⁵, jsou velmi dobře využitelné v interview, ve kterých se diagram sestavuje nebo verifikuje.



obr. 21. příklad diagramu popisujícího proces (generováno z Craft.CASE)

⁵ ORD je diagram, kde každý objekt (participant) je modelován jako Mealyho automat. Při simulaci se využívá synchronizace pomocí Petriho sítě.

4.3.4.3 Podrobná analýza procesů

Pro podrobnou analýzu podnikatelských a správních procesů nelze vystačit se sadou základních pojmů. Součástí analýzy podnikatelských a správních procesů je v neposlední řadě analýza pracovních činností, systemizace pracovních míst, simulace procesů a návrh nové organizační struktury odvozené ze struktury procesů.

Pro konstrukci podrobného objektově orientovaného modelu podniku je stejně jako pro modely informačních systémů klíčový pojem procesu, participantu a aktivity, které v tomto kontextu interpretujeme následovně:

- a) Proces. Pro popis procesů nám slouží scénáře OBA a jejich podrobné rozpracování v podobě procesních diagramů ORD úplně stejně jako při modelování informačních systémů. V konkrétních modelech velkých organizací je takových procesů typicky 50 až 100.
- b) Participant. V perspektivě modelování organizačních a správních procesů jsou objekty – participanty jednotlivé funkční jednotky podniku. Může to být například útvar vedoucího provozu, oddělení obchodu, zákaznické centrum, nejrůznější provozní útvary, oddělení reklamace, úsek generálního ředitele apod. V konkrétních modelech velkých organizací je takových participantů nejčastěji 100 až 200. Mezi participanty je možné vytvářet asociace a vazby skládání. Participantem mohou být jak velké úseky – např. obchodní oddělení, tak i jednotky malé například na úrovni jednoho pracoviště. Kritériem pro rozpoznání participantu není velikost nebo oficiální zařazení v podnikové hierarchii ani prostorové vymezení, ale jen a pouze existence jednoznačné a pro participant charakteristické množiny aktivit.
- c) Aktivita je jedna konkrétní činnost, kterou provádí konkrétní participant v konkrétním procesu. Je to například „vyřízení objednávky“ nebo „posouzení reklamace“ nebo „vyjednávání stavebního povolení“. Aktivity mohou měnit stavy svých participantů. Aktivity by také měly mezi sebou komunikovat. V konkrétních modelech velkých organizací je takových aktivit (všech participantů ve všech procesech) asi 1000 až 5000. Pokud uvnitř aktivity dokážeme rozpoznat sled dalších aktivit, které různě komunikují, tak je účelné aktivitu rozdělit a mezi nimi ještě najít potřebné stavy. Naopak pokud je v modelu několik aktivit pohromadě, které komunikují stejným způsobem nebo nekomunikují vůbec, tak nemá smysl je rozlišovat a je vhodné je sloučit do jedné.

4.3.4.4 Rozšíření modelu business procesů směrem nahoru

Pro rozhodování nad podnikovými a správními procesy je třeba znát jejich souvislosti s dalšími atributy organizace. Takových atributů je několik druhů a zpravidla mezi ně patří cíle, úkoly, problémy a kritické faktory úspěchu. V konkrétních případech můžeme model rozšířit i o další.

Během podrobné analýzy při interview se uvedené atributy rozpoznávají a sleduje se jejich vliv a souvislosti s procesy. Nejčastějším způsobem prezentace těchto důležitých informací jsou tabulky – například tabulka procesů a cílů, kde řádky jsou jednotlivé procesy, sloupce jsou jednotlivé cíle a na průsečících procesů a cílů se vyznačuje míra ovlivnění příslušného cíle příslušným procesem.

4.3.4.5 Rozšíření modelu obchodních a správních procesů směrem dolů

Participanty a aktivity jsou při velmi podrobné analýze příliš hrubým nástrojem. Jejich úlohou je napomáhat při modelování procesů – především při jejich členění na menší části, které jsou spolu logicky propojeny. Pro podrobnou analýzu je však třeba získat informaci i o následujících údajích,

kteře jsou užitečné pro úvahy o optimální podobě procesů a na jejich základě formulovaného zadání pro informační systém:

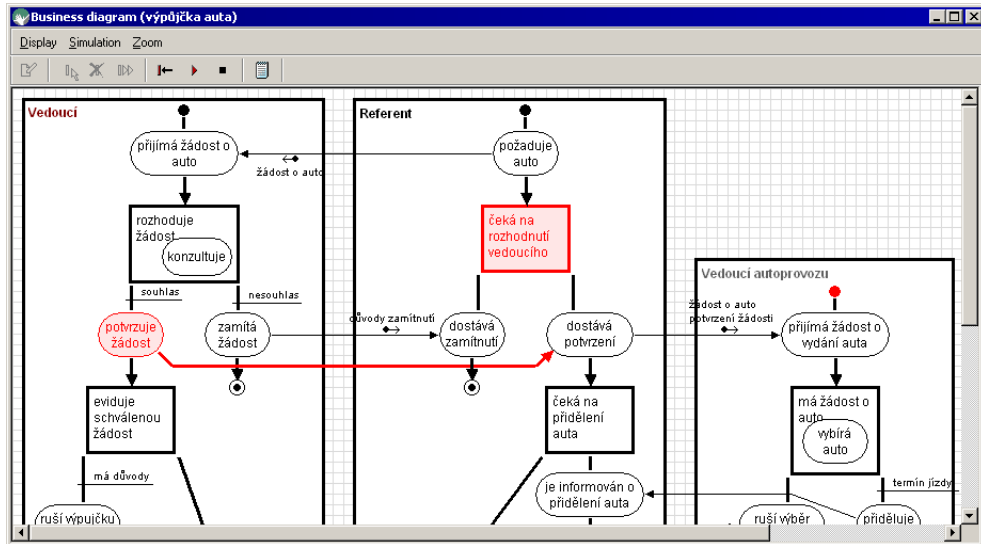
1. Pracovní místa. Je to konkrétní popis pracovního místa daného pracovníka s případným počtem konkrétních pracovníků na této pozici. Je to například „Elektroinstalatér“, „Řidič stavebního stroje“ nebo „Sekretářka ředitele obchodního úseku“. Tuto úlohu nemůže převzít účastník. Členění účastníků do takových podrobností, které odpovídají pracovním místům, je sice možné, ale výsledný model by byl velmi komplexní, obsahoval by příliš mnoho opakujících se nebo podobných prvků, takže by byl velmi nepřehledný. V konkrétních modelech velkých organizací je takových pracovních míst nejčastěji 500 až 1000.
2. Popis pracovní činnosti. Je to konkrétní popis týkající se plnění jednoho pracovního úkolu. Je to například „zajištění pracoviště“ nebo „oprava vozidla“. Podobně jako u pracovních míst není vhodné spojit pojem pracovní činnosti s aktivitou. Průměrná aktivita v podrobném modelu obsahuje 2 až 5 takových pracovních činností. Celkem to znamená asi 1000 až 5000 činností. Mohou se ale i vyskytnout aktivity s jedinou pracovní činností.
3. Zařízení je konkrétní pracovní pomůcka nebo stroj, který je potřeba k vykonání dané aktivity (například „služební auto“ nebo „osobní počítač“ nebo „ochranný oděv“).
4. Software. Zde jsou na mysli komponenty nebo moduly zamýšlených nebo již existujících informačních systémů, které jsou potřeba pro vykonání dané aktivity.

Uvedené možnosti rozšíření modelů podnikových a správních procesů slouží například jako podpora návrhu optimální struktury procesů a od toho se odvíjející organizační struktury a vylepšených popisů pracovních činností jednotlivých pracovních míst. Možností, jak využít a dále rozpracovat modelovou informaci je opravdu mnoho. Jednou z možností je například její využití pro optimalizaci počtu pracovníků na jednotlivých pozicích metodou zjišťování časových ekvivalentů jednotlivých úkonů vzhledem k celkovému času daného pracovní dobou. Další možností je například využití informace jako podklady pro konstrukci uživatelských rozhraní budovaných komponent informačních systémů.

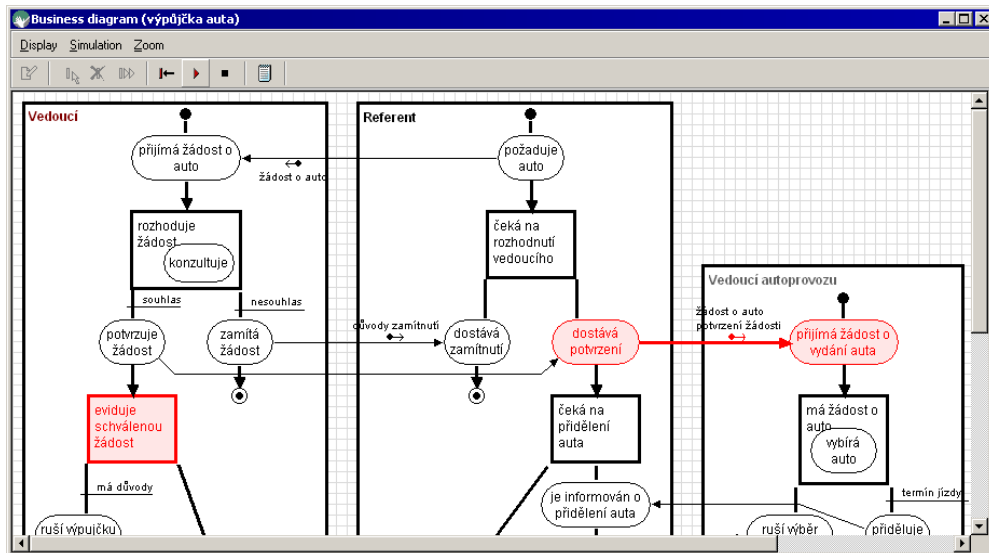
4.3.4.6 Simulace procesů

Modely obchodních a správních procesů, pokud jsme použili⁶ CASE nástroj, je možné použít i jako podklad pro sestavení simulačních modelů, se kterými lze provádět simulační experimenty. Do stavů a přechodů účastníků může být doplněna informace o časovém průběhu. Máme-li k dispozici simulační software, tak můžeme modelovou informaci výhodně využít k podrobnému ověření funkčnosti a smysluplnosti navrhovaných procesů přímo se zadavatelem. Dokonce i v případě, kdy speciální simulační software není k dispozici, může být výhodné naprogramovat funkční prototypové řešení zamýšlené aplikace, které poslouží podobným způsobem, jakým by posloužil simulační model.

⁶ Obrázky byly pořízeny pomocí simulačního modulu nástroje CraftCASE, jehož popis je v příloze.



obr. 22. příklad simulace procesu – stav před obdržení potvrzení od vedoucího



obr. 23. příklad simulace procesu – stav po obdržení potvrzení od vedoucího

name	role	system
Auto	default	Participant
Referent	performs	Participant
Vedoucí	approves	Participant
Vedoucí autoprovo	cooperates	Participant

Log
0: Auto / start
0: Referent / start
2: Auto / v autoparku
2: Referent : požaduje auto
5: Referent : požaduje auto -> žádost o auto -> Vedoucí : přijímá žádost o auto
6: Referent / čeká na rozhodnutí vedoucího
12: Referent / čeká na rozhodnutí vedoucího : dostává potvrzení žádosti
15: Referent / čeká na rozhodnutí vedoucího : dostává potvrzení žádosti -> žádost o auto, potvrzení žádosti
16: Referent / čeká na přidělení auta
22: Auto / v autoparku : přiděluje se
22: Referent / čeká na přidělení auta : je informován o přidělení auta
24: Referent / dostal přidělené auto
24: Auto / přiděleno
26: Referent / dostal přidělené auto : vyzvedává auto z autoparku
28: Auto / přiděleno : vyjíždí z autoparku

obr. 24. simulační záznam – seznam událostí objektu Referent a Auto

4.3.4.7 Změna procesů - Business Process Reengineering

Modelování a změna podnikatelských a správních procesů (business procesů) je klíčovou technikou při provádění BPR (Business Process Reengineeringu). BPR lze definovat jako zamyšlení a zásadní přeměnu fungování dané organizace. [Hammer et al. 1994] Hlavní vlastnost charakteristická pro BPR je jeho primární zaměření na procesy – tedy na to, co podnik dělá, či musí dělat, a teprve sekundárně – na základě poznanych procesů - na organizační strukturu, která by měla činnosti realizovat. BPR se provádí se ve dvou etapách:

1. V první etapě (etapa AS-IS = tak, jak to je) se podrobně modeluje stávající stav podnikových procesů. To je důležité pro exaktní vymezení předností a především nedostatků podniku. Bez této nepopulární fáze totiž nelze zodpovědně modelovat budoucí stav. Pro analytika je kriticky důležité v této fázi udržet projekt tak, aby důsledně popisoval pouze stávající stav se všemi nedostatky i bizarnostmi, neboť interview se zadavateli mají zpravidla tendenci sklouznout k povídání o chtěných nebo zamýšlených strukturách a k zatajování popisu problémů se stávající strukturou.
2. Po precizním vyhodnocení výsledků první etapy (měření, porovnávání, konzultace) se přistupuje k druhé etapě (etapa TO-BE = jak by to mělo být). Zde se navrhuje a podrobně popisují nové procesy a z informací v nich uložených se nakonec provádí nový návrh.

V BORMu se přístup BPR doporučuje u složitějších projektů k upřesnění a verifikaci prostředí, do kterého se informační systém projektuje, a na základě kterého se formulují funkční požadavky na tento informační systém.

4.3.5 Logické - konceptuální objekty

S nimi se v BORMu setkáváme ve středních fázích vývoje systému. Model tvořený konceptuálními neboli logickými objekty stojí jakoby napůl cesty mezi zadáním a řešením. Proto se také zpočátku na tyto objekty a jejich vazby nahlíží z perspektivy analýzy, kdy je ještě dovoleno použít model z konceptuálních objektů použit k upřesnění modelu zadání, ale zároveň model tvořený konceptuálními objekty slouží pro zahájení návrhu, kdy již považujeme problém za rozpoznáný a začínáme jej konsolidovat do počítačově realizovatelné podoby.

4.3.5.1 Přejchod od business objektů ke konceptuálním objektům

Přejchod od objektů reálného světa ke konceptuálním objektům je možné stručně popsat následovně:

1. Nejprve se provede podrobný popis objektů reálného světa a naleznou se vazby „je jako“ (is-a) a asociace.
2. Na základě znalosti objektů reálného světa se naleznou třídy a množiny objektů.
3. Vazby „je jako“ (is-a) poslouží pro sestavení hierarchie typů.
4. Asociace mezi objekty a provázání objektů pomocí komunikací poslouží pro nalezení vazeb skládání (agregace) a pro posílání zpráv mezi metodami objektů.

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

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

obr. 25. rozhodovací tabulka pro přechod od business ke konceptuálnímu modelu

4.3.5.2 Diagramy konceptuálních objektů

Diagramy konceptuálních objektů jsou na rozdíl od diagramů objektů reálného světa poměrně známé a používané již od začátku 90. let. Někdy se však nesprávně ztotožňují s diagramy objektů softwarových, protože se v nich objevují velmi podobné nebo zcela shodné pojmy. Rozdíl je však ve způsobu nazírání na tyto pojmy a vazby. V BORMu je pro tyto diagramy použit upravený UML. Jsou to tyto úpravy a doplnění:

- UML nerozlišuje mezi polymorfismem, děděním, hierarchií typů a hierarchií „je jako“. V našem přístupu tyto vazby graficky rozlišujeme (Vysvětlení rozdílu je v kapitole 4.3.8).
- UML nerozlišuje mezi pojmem třída objektů a množina objektů. V našem přístupu tyto vazby rozlišujeme a zavádíme nový grafický symbol pro množinu objektů a pro třídu jako objekt sám o sobě. Pojmy třída a množina v BORMu se z důvodu jednoduchosti a srozumitelnosti modelují jednotně pouze pro objekty reálného světa.
- V našem přístupu klademe důraz na dynamickou stránku problému. Metody objektů včetně zobrazených podrobností jejich vazeb (zpráv) na jiné metody jiných objektů jsou nedílnou součástí popisu systému a používají se ve všech typech diagramů. Proto jsme zavedli samostatný grafický symbol pro metodu a pro zprávu poslanou z metody k jiné metodě.
- UML dovoluje v jednom konkrétním objektovém diagramu použít současně vazby a pojmy na různé úrovni abstrakce – tedy míchat pojmy objektů reálného světa s konceptuálními a softwarovými. V jednom diagramu je možné mít například současně vyznačené asociace spolu s děděním atd. V našem přístupu doporučujeme používat pojmy postupně, tak jak se během modelování od sebe odvozují.

4.3.6 Softwarové - implementační objekty

Se softwarovými objekty se v BORMu setkáváme až v závěrečných fázích životního cyklu vývoje systému, kdy je třeba model postupně transformovat do takové podoby, která je vyžadována pro fyzickou realizaci systému v podobě programu v daném programovacím jazyce. Právě dokončení modelu tvořeného softwarovými objekty na takové úrovni podrobností, které již odpovídají výrazovým prostředkům použitého jazyka, se považuje za okamžik ukončení objektově orientovaného návrhu a zahájení implementace. Přeměnu modelu tvořeného strukturou konceptuálních objektů a model softwarových objektů lze stručně popsat následujícím způsobem:

- a) Přeměna hierarchie typů na hierarchii dědění mezi třídami a případná transformace vícenásobné dědičnosti na jednoduchou.
- b) Doplnění tříd o atributy a metody, které umožní realizovat stavy a přechody (současné objektové jazyky se stavy a přechody přímo nepracují).
- c) Využití návrhových vzorů pro optimalizaci a pro kontrolu proveditelnosti modelu v daném programovacím jazyce.
- d) Pokud to cílové prostředí vyžaduje, tak nahradit vazby závislosti.
- e) Pokud to cílové prostředí vyžaduje, tak nahradit vazby delegování.
- f) Napojení modelu na struktury v existujících systémech.

4.3.7 Přínos rozdělení modelu na business, konceptuální a softwarové objekty

Podívejme se ještě jednou na důvody používání tří objektových modelů během tvorby systému:

„business“ modelování

Nejprve je doporučeno sestavit model zadání v kategoriích business objektů. Zde je výhodné nepoužívat žádné specificky softwarové pojmy, protože je na ně příliš brzy. Důležitější je zde dosažení porozumění mezi zadavatelem a analytikem a jeho řádné dokumentování. Programátorské pojmy zde nepoužíváme také proto, že některá vazby známé v reálném světě mají jiný význam, než ve světě softwarového kódu (např. dědičnost) a nebo je dnes používané programovací jazyky nepodporují vůbec (například delegování nebo závislost).

V této fázi modelování se také zabýváme i vztahy mezi participanty, které nakonec nebudou součástí funkčnosti budované aplikace. Jejich modelování ale může být důležité pro upřesnění zadání a pro provedení nezbytné reorganizační změny v okolí projektovaného informačního systému.

Konceptuální modelování

Konceptuální model vzniká transformací z předchozího modelu, ze kterého se vyberou ty objekty a vazby, které reprezentují budovaný informační systém. Tyto objekty a vazby jsou potom základem pro sestavení modelů popisujících konceptuální analýzu budovaného systému. (Což je místo, až od kterého klasické metodiky – jako např. OMT – začínají projektování). I když je tento konceptuální model již popisem budovaného systému, tak není vhodné „skočit“ přímo do světa implementace a při výběru pojmů a vazeb se omezit jen na vlastnosti cílového implementačního prostředí. Tato dokumentace totiž musí sloužit i při možných pozdějších změnách a rozšířeních systému, přechodu na jinou technologii atd.

Softwarové modelování

Tato poslední fáze je procesem, kdy se sestavený konceptuální model konsoliduje do takové podoby, že je ho možné implementovat v příslušném programovacím prostředí. Rozdíly mezi konceptuálním a softwarovým modelem tedy nespočívají jen v různé míře zobrazení detailu, ale hlavně v zohlednění konkrétních implementačních omezení jak ze strany programovacího prostředí, tak i ze strany dříve vyrobených softwarových komponent, se kterými musí nový systém spolupracovat, a které téměř nikdy nemají ideální znovupoužitelnou strukturu.

4.3.8 Evoluce hierarchií objektů

Nové typy se v objektových systémech realizují většinou pomocí tříd, přičemž ale programátoři vědí, že novou třídu do systému lze vyrobit nejen pomocí dědění, ale i skládáním. Z toho proto vyplývá, že hierarchie dědění v implementačním modelu a hierarchie typů v analytickém modelu jednoho systému nemusí vždy znamenat totéž. Navíc při používání dědičnosti máme v programovacích jazycích nástroje pro skrývání metod a dat děděných objektů. Na hierarchii tříd objektů se proto v BORMu nahlíží trojím způsobem podle následujících kritérií:

1. Z pohledu návrháře – tvůrce nových objektů. Tato hierarchie je hierarchií dědičnosti, protože dědičnost je programátorským nástrojem pro tvorbu nových tříd. Její místo je ale až ve fázi softwarového modelování.

2. Z pohledu uživatele – analytika nebo aplikačního programátora, který potřebuje již hotové objekty použít ve svém systému. Tento pohled, který předchází implementaci, lze ještě podrobně dělit na

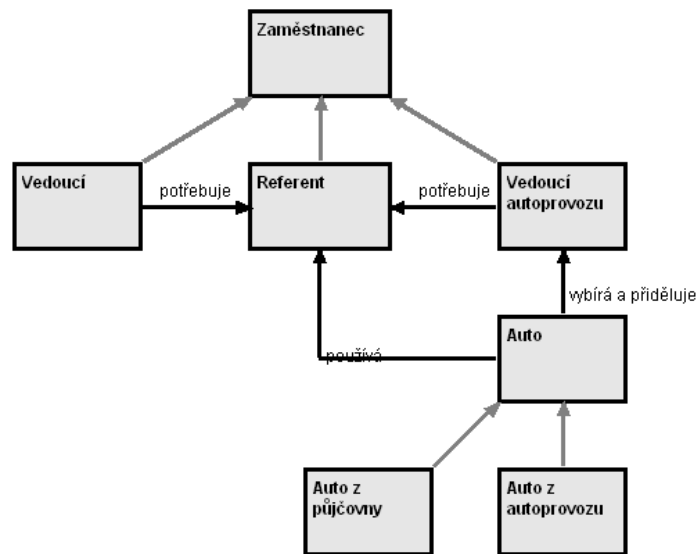
2.1. Z pohledu polymorfismu – objekty na nižších úrovních hierarchie potom musejí být schopny dostávat stejné zprávy, jako objekty vyšších úrovní. Právě tato hierarchie je hierarchie typů. Její místo je ve fázi konceptuálního modelování.

2.2. Z pohledu aplikační domény – instance tříd na nižších úrovních potom musejí být prvky stejné domény, kam patří instance tříd nadřazené třídy. To znamená, že doména nižší úrovně je podmnožinou domény vyšší úrovně. Tato hierarchie je anglicky označována jako IS-A, česky ji můžeme přeložit „je jako“ (nebo „patří k“). Od hierarchie typů se může v konkrétních případech lišit proto, že se nezabývá jen chováním objektů na rozhraní, ale i datovým obsahem objektu a jeho konkrétní rolí v modelovaném systému. Její místo je ve fázi „business“ modelování.

U jednoduchých úloh je samozřejmě pravda, že uvedené tři hierarchie jsou totožné. Proto se tímto problémem programátorské kuchařky příliš nezabývají a všechny tři typy hierarchií považují za dědičnost. U komplexnějších úloh však toto tvrzení neplatí a to především při návrhu systémových knihoven, které se opakovaně znovupoužívají při návrhu konkrétních systémů. V některých objektových programovacích jazycích lze dokonce nalézt prostředky pro oddělení typů a tříd.

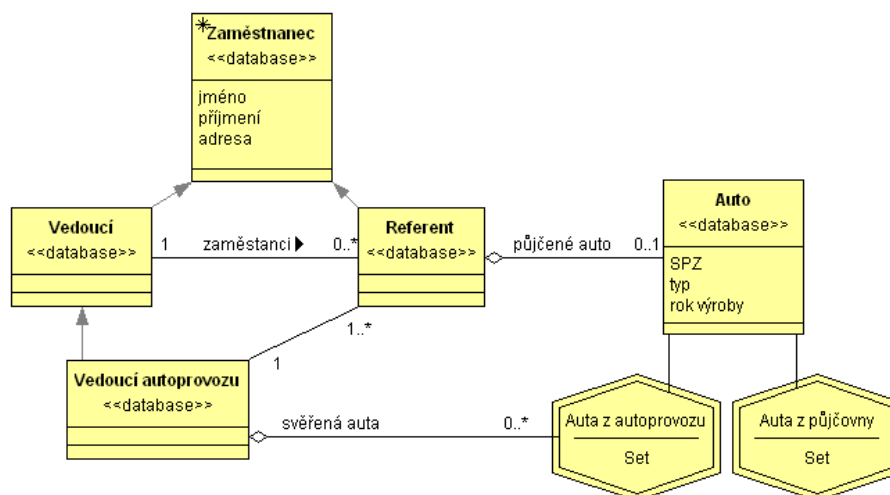
V BORMu se nejprve pracuje pouze s hierarchií „je jako“, z ní se později odvodí hierarchie typů a nakonec se na základě typů navrhne hierarchie dědičnosti. Tento přístup, kdy se s hierarchiemi objektů pracuje v různých fázích vývoje různě, zabraňuje nadměrnému a nesprávnému použití dědičnosti v průběhu implementace. Postupnou evoluci hierarchií budeme demonstrovat na následujícím příkladu, do kterého patřil i příklad procesu simulace a OBA a je popsán v [Carda et al. 2003]. Ve fázi business modelování byla sestavena hierarchie objektů, kde v horní části diagramu je

participant vedoucí, referent a vedoucí autoprovozu jako tři disjunktní podmnožiny pod participantem zaměstnanec:



obr. 26. příklad proměny hierarchií objektů – fáze business modelování

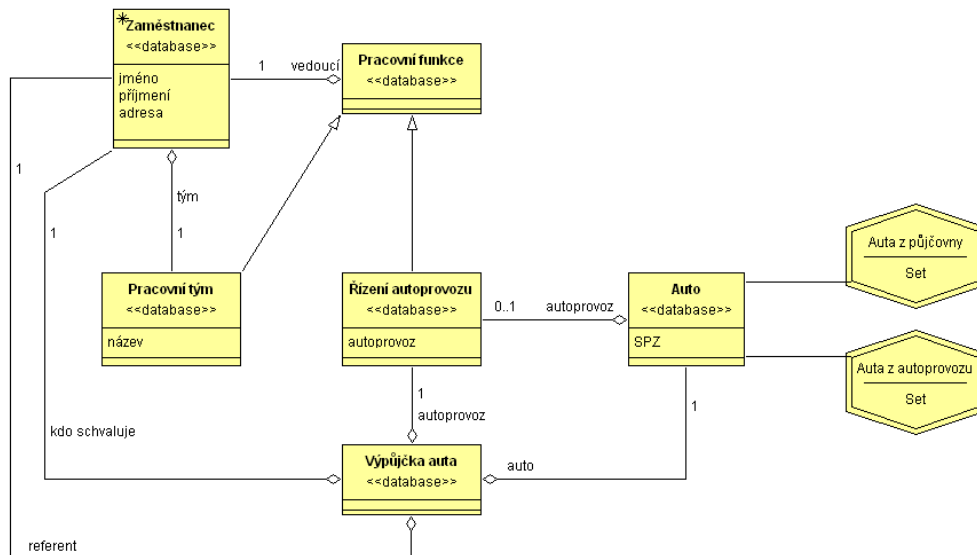
V následné fázi konceptuálního modelování došlo nejen ke zpřesnění modelu, ale také ke změně hierarchie. Kritériem už totiž není příslušnost do domény, ale chování objektů. Proto je teď vedoucí autoprovozu „podtypem“ vedoucího. (vedoucí autoprovozů se chovají stejně jako vedoucí a k tomu mají ještě další vlastnosti). Tuto hierarchii nebylo vhodné použít dříve, protože bychom touto abstrakcí zmátli zadavatele systému z praxe, kam se systém projektuje. V reálném světě totiž žádný vedoucí autoprovozu není zároveň vedoucím zaměstnanců, ale jde o jinou funkci. Ve fázi business modelování totiž hierarchie znázorňuje vztah domén (kdo je kdo), tady ale znázorňuje vztahy v chování a vlastnostech (kdo je jako).



obr. 27. příklad proměny hierarchií objektů – fáze konceptuálního modelování

A nakonec ve fázi softwarového modelování muselo dojít k dalšímu přepracování, protože systém nevznikal na zelené louce, ale jeho implementace se musela napojit na databázi všech zaměstnanců (v diagramu třída označená hvězdičkou). Nebylo proto možné z předchozí fáze rozpoznat podtypy

implementovat jako různé třídy. Proto se musel použít návrhový vzor „Player-Role“, který různé typy a podtypy objektů dovoluje implementovat bez dědění jen pomocí skládání (v příkladu to je skládání k objektu pracovní funkce).



obr. 28. příklad proměny hierarchií objektů – fáze softwarového modelování

Jak ukazuje tento příklad z praxe, tak není v praxi často možné realizovat „krásný“ objektový návrh právě z důvodu nutnosti zachování a napojení se na předchozí struktury. Na druhou stranu by bylo chybou, kdyby analytici přeskakovali prostřední fázi konceptuálního modelování a z analýzy zadání by se rovnou začali věnovat implementaci. Vždy je totiž prospěšné vědět a mít dokumentované, jaký je „ideální“ konceptuální model systému. Tato znalost je totiž velmi užitečná při údržbě, rozšiřování, opravách apod.

4.3.9 Tři dimenze objektového modelu – zjednodušení složitosti

V ideálním případě si lze v BORMu představit jediný souhrnný diagram⁷ pro celý systém. Rozsáhlost modelu u velkých systémů však prakticky znemožňuje s takovými diagramy pracovat. Proto je třeba pracovat s menšími diagramy, které vždy popisují pouze část systému. Široce používanou možností zjednodušení složitosti modelů jsou dekompozice a hierarchie. Je však také možné použít ještě jeden způsob, jak snížit složitost:

Prvky a vazby v objektově orientovaném systému lze třídit a filtrovat podle toho, jestli nesou informaci o datech nebo o funkcích nebo o změnách v čase. Tato tři kritéria si potom můžeme představit jako tři rozměry abstraktního znalostního prostoru, ve kterém je objektově orientovaný model vytvářen:

1. Vynecháním časového rozměru získáme pohled, který zobrazuje vztah dat a funkcí mezi sebou. Zobrazíme-li potom funkce ve větší podrobnosti před daty (objekty), tak dostaneme pohled, který se v standardním UML nazývá „object interaction diagram“. Zobrazíme-li naopak data (objekty ve větší podrobnosti) před funkcemi (metodami), tak

⁷ BORM používá standard UML, ale rozšiřuje ho o nové symboly, které v kombinaci se stávajícími dovolují vytvářet další druhy diagramů.

do této skupiny také může patřit i diagram, který zobrazuje objekty, třídy, atributy a metody včetně objektových hierarchií (dědění, skládání) a je v standardní UML nazýván jako „object-class diagram“.

2. Vynecháním datového rozměru získáme pohled, který zobrazuje vztahy funkcí (metod) v závislosti na čase. Typickým představitelem je sekvenční diagram z UML.
3. Vynecháním funkčního rozměru získáme pohled, který zobrazuje závislost dat v čase. Takový typ diagramu v standardním UML není⁸. V tomto pohledu jde totiž o zobrazení, jak se v jednotlivých stavech v čase mění datová struktura systému. V BORMu k tomuto slouží již dříve popsany procesní diagram. Pro každý stav, který budeme chtít zobrazit, lze jako jeho dekompozici sestavit samostatný zjednodušený „object-class“ diagram definující data a vazby daného stavu. Právě odlišnosti mezi těmito diagramy jednotlivých stavů (jiné vazby, jiné hodnoty proměnných, změny v kardinalitě, ...) názorně ukazují průběh datových změn systému v čase a napomáhají jeho softwarové implementaci.

4.3.10 Chyby kterých je třeba se vyvarovat při modelování

Při vytváření struktury objektového modelu nejčastěji dochází vlivem podcenění výše uvedených postupů a ovlivnění neobjektovými technikami návrhu k následujícím chybám, které jsou diskutovány například v [Abadi 1996, Larrison et al. 2002, Molhanec 2004]:

- **Podcenění možností skládání objektů** a i jiných hierarchií na úkor **přeceňování dědičnosti**. Je to způsobeno tím, že dědění je pojmem novým, a proto se na něj v literatuře zabývající se výkladem OOP klade větší důraz, než na skládání, které již dříve bylo určitým způsobem využíváno.
- **Podcenění možností metod**, které vede k jejich omezení na pouhou manipulaci se složkami objektů (metody pouze typu "ukaz" a "nastav"). Vzhledem k provázanosti datové a funkční stránky v OOP je velmi vhodné s některými metodami počítat přímo v návrhu datové struktury a tím ušetřit na objektových vazbách a datových attributech objektů.
- **Zjednodušení modelu výpočtu směrem k vN** projevující se v omezení parametrů zpráv a atributů objektů na pouze skalární data typu „znak“ nebo „číslo“ atp.
- **Chybné stanovení hranice**, která určuje, kdy se různé objekty mají ještě modelovat hierarchií různých tříd a kdy se již jedná o objekty s různým datovým obsahem ale stejné třídy.
- **Nerozlišení pojmů** třída objektů a množina objektů v konceptuálním modelu a z toho vyplývající nevhodná implementace typů pomocí tříd.

4.3.11 Zkušenosti

Metoda BORM a především její možnosti analýzy v počátečních fázích vývoje projektu byla v letech 1996 až 2000 prakticky použita firmou Deloitte&Touche například v projektech pro pražské zdravotnictví, Ústav pro státní informační systém ČR, elektroenergetiku, zemědělství,

⁸ UML zná jen diagram stavů a přechodů, který se týká jen jedné třídy objektů. Zde ale hovoříme o celém modelu se všemi objekty, jak na sebe vzájemně působí.

telekomunikace a plynárenství. Ve všech projektech se ukázalo, že BORM lze dobře využívat jako nástroj pro provádění business process reengineeringu. Zkušenosti s mapováním business a workflow procesů pomocí BORMu prokázaly, že BORM dovoluje ve srovnání s jinými metodami velmi rychle získávat přesné a verifikované výsledky. Výsledky takové analýzy také velmi dobře slouží pro podrobnou a úplnou specifikaci zadání softwarového projektu, i když větší část projektů bylo zaměřených na organizační poradenství a nemělo za cíl tvorbu informačního systému.

Přínosy metody BORM ve tvorbě softwaru byly ověřeny při použití programovacích nástrojů VisualWorks/Smalltalk-80, Visual Basic, .NET a Control Web, přičemž vytvářené aplikace jsou vesměs typu klient-server a využívají relační databázové systémy Oracle, rozhraní ODBC a nebo objektovou databázi Gemstone. BORM je využíván ve firmě e-Fractal s.r.o., která je jedním z největších obchodníků na českém internetu a zabývá se také tvorbou softwaru na zakázku, především webových informačních systémů využívajících objektovou databázi.

Součástí vývoje metodologie BORM byl i výběr vhodného CASE nástroje. V současné době je metoda BORM již implementovaná v nástroji Metaedit Plus finské firmy MetaCase⁹. Od listopadu 2004 také probíhá pod patronací Deloitte&Touche implementace vlastního nástroje Craft.CASE, jehož popis je v příloze tohoto textu.

4.4 Zajištění jakosti při business modelování

4.4.1 Vztah jakosti a metodiky

Jakost produktu lze definovat jako „souhrn podstatných vlastností produktu, které určují míru uspokojení daných (obecně očekávaných) a stanovených potřeb uživatele produktu, v případě užití produktu stanoveným způsobem“ [Vaníček 2004]. Oproti tomu jakost procesu je v normě definována jako „stupeň splnění požadavků souborem inherentních znaků“.

Současně s trendem zvyšování jakosti softwarových produktů však dochází k nežádoucímu jevu: k odtrženému chápání pojmu jakosti. Pokud má jakost dostát své definici uvedené výše, nelze ji chápat zúženě pouze jako soubor obtěžujících norem a nařízení, které je třeba dodržovat, aby firma získala certifikát, kterým se může honosit. Tento přístup je bohužel dosti rozšířený a vede k určité skepsi o tom, že snahy vedoucí k zajištění jakosti budou znamenat „lepší produkt“. Pokud se totiž nad definicí jakosti hlouběji zamyslíme, můžeme prohlásit, že v podstatě všechny metodiky tvorby softwaru mají za cíl zvýšit jakost výsledného produktu, neboť jedním z hlavních cílů všech metodik je, aby zákazník obdržel produkt (program, systém či službu), který naplňuje jeho očekávání. **Péče o jakost výsledného produktu musí být nedílnou součástí každé metodiky, pomocí které je produkt vytvářen.**

Pro každou metodiku tvorby softwaru bychom si měli položit několik otázek:

1. Jakým způsobem zajišťuje metodika jakost výsledného produktu nebo k ní přispívá?
2. K dosažení jakosti by měly být v metodice pravidla a postupy. Jak jsou tato pravidla a postupy kontrolovatelné a měřitelné?

⁹ MetaEdit je výsledkem výzkumného projektu na univerzitě v Juväskylä. Autor této práce se podílel v letech 1998-2000 na implementaci BORMu do tohoto nástroje, který je od té doby na PEF ČZU používán ve výuce projektování a softwarového inženýrství.

3. Jakost produktu je z principu pružný aspekt – může být jiná v závislosti na potřebách uživatele, obstaravatele event. zainteresovaných třetích stran. Je metoda dostatečně pružná, aby byla schopná umožnit maximalizaci jakosti ve všech (běžných) situacích?

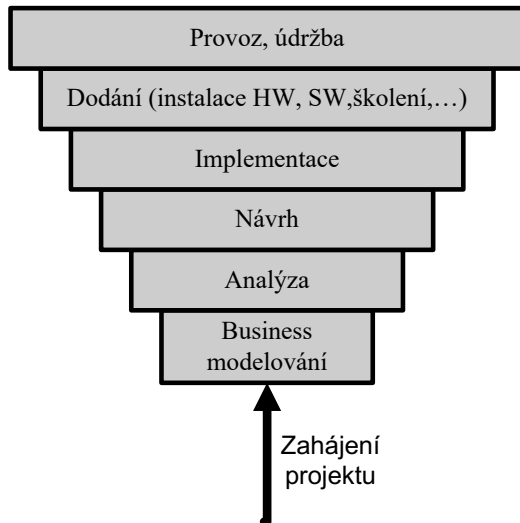
Normy týkající se jakosti softwarových produktů se bohužel věnují především pouze poslední otázce, zabývají se tedy již výsledným produktem. Součástí těchto norem (např. současná ISO/IEC 14598) jsou sice části týkající se plánování, řízení a vývoje, jedná se však o obecné postupy, které nejsou vytvořeny pro žádnou konkrétní metodiku. Jakost procesu a tedy i metodik se snaží řešit normy ISO 9000. Tyto normy jsou však zcela obecné a týkají se jak managementu jakosti procesu výroby serek, tak i lokomotiv. Z toho nelze očekávat, že jejich výstupem budou nějaké míry. Neexistuje norma na management jakosti procesu vývoje IS nebo SW, existuje jen obecný předpis jak by měly vypadat. Proto nelze zobrazit míry jakosti procesu do měr jakosti produktu. Míry jakosti IS a SW produktu v řadě 9126 nejsou zdaleka dokonalé, ale nějaké byly navrženy v 9126-2, 9126-3 a 9126-4 jsou. Míry jakosti procesu ale žádné navrženy nejsou. Zcela obecně pro všechny typy produktů ani existovat nemohou.

Důsledkem toho může při používání konkrétní metodiky dojít ke dvěma negativním jevům:

1. Omezení možností metodiky v důsledku nutnosti dodržení normativního postupu.
2. Nevyužití speciálních možností, které má sama metodika pro zajištění jakosti.

Z tohoto důvodu je třeba zabývat se zajištěním jakosti nejen na obecné úrovni, ale též studovat možnosti zajištění jakosti konkrétně pro určitou metodiku.

4.4.2 Pozice business modelování v zajištění jakosti



Proces vzniku softwarového produktu je z hlediska zajištění jakosti obrácenou pyramidou. Pyramida se směrem nahoru rozšiřuje, což symbolizuje, že každá fáze staví na předchozí a přidává další výstupy. Jakost procesu v dané fázi životního cyklu může stavět pouze na jakosti procesu fáze předchozí a jakost dále rozšiřovat. Pokud je např. v programu funkce špatně naimplementována, jakostní dodání produktu na tomto nic nezmění. Máme sice mechanismy, které zajišťují zpětné zvýšení jakosti, např. testování odhalí chyby implementace, ovšem jakmile dojde k závadě na jakosti, vždy to znamená náklady navíc, čas navíc, zvýšení stresu a další negativní jevy.

Fáze, která hraje klíčovou roli v zajištění jakosti výsledného produktu je fáze analýzy a především business

obr. 1. Vývoj IS z hlediska zajištění jakosti modelování, a to ze dvou důvodů:

1. Jde o první krok.
2. Při business modelování získáme od uživatele poznatky o jeho potřebách a požadavcích ohledně budoucího systému. Při opětovném pohledu na definici jakosti je jasné, že toto je naprosto klíčová záležitost, jelikož jakost je založena na subjektivní potřebě uživatele. Skvělý software, který nepřináší uživateli očekávaný užitek může být sice z různých hledisek výjimečně dobrý, není však jakostní!

4.4.3 Charakteristiky jakosti

Jakým způsobem tedy zajistit jakost ve fázi business modelování? Různí klienti budou mít různé požadavky na software a informační systémy v závislosti na svých prioritách, strategii a cílech, ke kterému má informační systém sloužit. V případě softwaru obsluhujícího jadernou elektrárnu bude prioritou bezpečnost a bezporuchovost, zatímco uživatelský komfort a minimální doba k zaškolení nebudou tak klíčové. V případě veřejného informačního terminálu tomu bude přesně naopak. Aby bylo možné stanovit úroveň jakosti odděleně podle skutečných potřeb, bylo dohodnuto rozdělit jakost softwarových produktů na šest kategorií nazvaných *charakteristiky jakosti*, které mají další *podcharakteristiky*. Shrňme si je zde pro pohodlí čtenáře, v dalším textu nebudou komplexně vysvětleny, detaily lze nalézt v literatuře [Vaníček 2004].

1. Funkčnost (Functionality)
 - i. Funkční přiměřenost (Suitability)
 - ii. Přesnost (Accuracy)
 - iii. Schopnost spolupráce (Interoperability)
 - iv. Bezpečnost (Security)
 - v. Shoda ve funkčnosti (Functionality Compliance)
2. Bezporuchovost (Reliability)
 - i. Zralost (Maturity)
 - ii. Odolnost vůči vadám (Fault Tolerance)
 - iii. Schopnost zotavení (Recoverability)
 - iv. Shoda v bezporuchovosti (Reliability Compliance)
3. Použitelnost (Usability)
 - i. Srozumitelnost (Understandability)
 - ii. Naučitelnost (Learnability)
 - iii. Provozovatelnost (Operability)
 - iv. Atraktivnost (Attractiveness)
 - v. Shoda v použitelnosti (Usability Compliance)
4. Účinnost (Efficiency)
 - i. Časové chování (Time Behaviour)
 - ii. Využití zdrojů (Resource Utilisation)
 - iii. Shoda v účinnosti (Efficiency Compliance)
5. Udržovatelnost (Maintainability)
 - i. Analyzovatelnost (Analysability)
 - ii. Měnitelnost (Changeability)
 - iii. Stabilita (Stability)
 - iv. Testovatelnost (Testability)
 - v. Shoda v udržovatelnosti (Maintainability Compliance)

6. Přenositelnost (Portability)
 - i. Přizpůsobitelnost (Adaptability)
 - ii. Instalovatelnost (Installability)
 - iii. Slučitelnost (Co-existence)
 - iv. Nahraditelnost (Replaceability)
 - v. Shoda v přenositelnosti (Portability Compliance)

Rozeberme si nyní, které aspekty můžeme pozitivně ovlivnit již při business modelování a jakým způsobem.

4.4.3.1 Funkčnost

Funkčnost je vymezena jako schopnost informačního systému či softwarového produktu obsahovat funkce, které zabezpečují předpokládané nebo stanovené potřeby uživatele při používání systému za stanovených podmínek. Na rozdíl od některých ostatních charakteristik, tato charakteristika zajímá vždy všechny zúčastněné strany.

Tato charakteristika tedy zjišťuje, **zda** jsou funkce vůbec zabezpečeny, **nikoliv jak**.

4.4.3.1.1 Funkční přiměřenost (Suitability)

Tato charakteristika požaduje, aby pro danou úlohu byla vždy k dispozici funkce, která naplňuje potřeby zákazníka¹⁰. Při business modelování v metodice BORM shromažďujeme požadavky na funkčnost především ve dvou krocích: při tvorbě *seznamu funkcí* a *tabulky scénářů*. Seznam funkcí obsahuje přehled funkcí z hlediska podniku z vnějšího pohledu, tabulka scénářů navazuje na seznam funkcí a zaměřuje se na analýzu funkcí vnitřních, tedy do určité míry skrytých pohledu zákazníka firmy. Z hlediska zajištění jakosti je podstatné se soustředit na oba tyto kroky BM.

4.4.3.1.2 Přesnost (Accuracy)

Přesnost požaduje přesné výsledky. Ačkoliv do značné míry jde o implementační záležitost týkající se např. zaokrouhlovacích chyb a algoritmů, ve fázi BM je třeba si se zákazníkem ujasnit, jaká míra přesnosti je pro konkrétní výstupy požadována. Tyto informace je třeba zanést do všech tabulek a diagramů BORMu.

4.4.3.1.3 Schopnost spolupráce (Interoperability)

Jedná se o schopnost spolupráce IS s ostatními IT systémy ve firmě. Součástí BM tedy musí být analýza IT prostředí firmy, na jejímž základě je poté možné se rozhodnout pro konkrétní technologie a přístupy a pro vyhovující, ne však zbytečně nákladnou a rozsáhlou úroveň schopností spolupráce. BORM se soustřeďuje na komplexní procesní analýzu podnikového prostředí, tato subcharakteristika je tedy v BORMu automatická.

¹⁰ *Zákazníkem* neboli *klientem* zde budeme rozumět podnik, instituci, pro kterou vyvíjíme IS. Zákazníka tohoto podniku či instituce budeme potom označovat výrazem *zákazník firmy*.

4.4.3.1.4 Bezpečnost (Security)

Na bezpečnost dat je v současnosti kladen vysoký důraz. S klientem je třeba si vyjasnit, jakou míru bezpečnosti potřebuje pro jednotlivé části systému. Systém musí být v maximální možné míře bezpečný, na druhou stranu snaha o implementaci přehnané bezpečnosti může zvýšit náklady, snížit uživatelský komfort, a tím snížit jakost systému. Speciální bezpečnostní požadavky je vhodné zahrnout do seznamu funkcí a následně rozpracovat až do detailního OBA diagramu.

4.4.3.1.5 Shoda ve funkčnosti (Functionality Compliance)

Shoda hodnotí míru přizpůsobení IS lokálním požadavkům, zvyklostem a normám. Zajištění shody je *jedním z klíčových úkolů BM*, jelikož se jedná o jedinou fázi při tvorbě IS, kde je možné toto zajistit (pokud nepočítáme dodatečné korekční mechanismy).

Tato podcharakteristika je doplňována ke všem charakteristikám jakosti a má všude stejný význam, nebudeme ji tedy u dalších charakteristik již rozvádět.

4.4.3.2 Bezporuchovost

Tyto charakteristiky se zaměřují na schopnost systému vyvarovat se poruchám, v případě výskytu poruch na míru zachování výkonu a na obnovení systému po odstranění poruchy. Podobně jako v případě přesnosti 4.4.3.1.2 je ve fázi BM třeba si ujasnit potřeby zákazníka. Jde především o tzv. *Business critical* části systému, tedy části, které jsou podstatné pro běh firmy. Postupy a odpovědnosti v případě selhání systému též musí být popsány v *Disaster Recovery Plan*, který je součástí firemního *Business Continuity Plan*. Tyto postupy je též velmi vhodné provádět metodikou BORM, jelikož je vyladěna na co nejpřesnější a nejobsažnější zachycení procesu.

4.4.3.3 Použitelnost

Uvedené podcharakteristiky se do značné míry prolínají. Pro nás jsou podstatné dvě skupiny:

4.4.3.3.1 Srozumitelnost (Understandability)

charakterizuje snadnost porozumění možnostem systému a práce s ním. Tato podcharakteristika bude především zajímat zákazníka, který kupuje hotový software. Při vývoji informačních systémů na zakázku, na který se zde zaměřujeme, není tato podcharakteristika až tolik relevantní.

Nicméně v této souvislosti je třeba podotknout, že zákazník (ačkoliv si to nemusí připouštět) mnohdy nemá přesnou představu, co by měl od systému požadovat a jaký užitek (tzv. *Business Value*¹¹) mu přinese. Součástí práce analytika tedy často bývá spolu se zákazníkem vyjasnit i tyto otázky a podle toho přizpůsobit diagramy.

4.4.3.3.2 Naučitelnost, Provozovatelnost a Atraktivnost

jsou podcharakteristiky, které se do značné míry prolínají a zabývají se snadností a příjemností obsluhy systému. Zahrnují též úsilí, které je třeba vynaložit na školení personálu.

¹¹ V terminologii normalizace jakosti se hovoří o Quality in use.

Zde můžeme konstatovat, že hlavní odpovědnost za tyto aspekty spadá do fáze návrhu IS, implementace a dokumentace. Analytik se však musí zaměřit na shodu v použitelnosti (viz 4.4.3.1.5) a do analytické dokumentace zanést všechny potřebné údaje. Tyto údaje nejsou procesního charakteru, metodikou BORM je tedy nepostihneme.

4.4.3.4 Účinnost

4.4.3.4.1 Časové chování (Time Behaviour)

Zajímá nás především *asymptotické* časové chování, tedy jak se budou měnit účinnostní charakteristiky systému (odezva, propustnost, ...) při nárůstu objemu zpracovávaných dat. Časové chování je ovlivňováno při implementaci použitými technologiemi, výběrem vhodných algoritmů a vnitřních optimalizací systému. Zlepšení jedné charakteristiky bývá za cenu zhoršení charakteristiky jiné. Implementátoři proto potřebují znát co nejpřesnější odhady, kolik dat a jakým způsobem se bude zpracovávat a jaké budou nároky na jednotlivé účinnostní charakteristiky. Je tedy úkolem analytika se zabývat i těmito otázkami. Přijdou na řadu sice až ve druhé fázi, kdy již máme hotové business procesy a datová schémata, jsou však stále do značné míry součástí business modelování. Časové chování ovšem nebývá typicky klíčovým požadavkem aplikace navrhované metodou BORM.

4.4.3.4.2 Využití zdrojů (Resource Utilisation)

Jedná se o analýzu efektivity využití všech ostatních zdrojů, a to jak hardwarových, softwarových, tak i nároků na obsluhu. Platí to samé, co pro předchozí podcharakteristiku.

4.4.3.5 Udržovatelnost

Vyjmenované podcharakteristiky specifikují schopnost hotového IS být modifikován. Modifikace zahrnují opravy nedostatků, vylepšování, adaptaci vzhledem ke změnám prostředí, změnám požadavků a změnám funkční specifikace. Hodnotí fázi identifikace (analyzovatelnost), fázi provedení změn (měnitelnost) a možnost otestování systému po změně, tj. udržení jakosti. Stabilitnost adresuje riziko negativního vlivu na funkčnost systému z důvodů zásahů do něj.

Změny hotových IS a obecně změny zadání při vývoji i po dokončení jsou velkým problémem a způsobují často značný nárůst nákladů a problematickou funkčnost systému. Jsou vyvíjeny metodiky, které se snaží tento stav zlepšit, jmenujme zde především rodinu tzv. *agilních metodik* viz [Odkaz do textu](#). Vhodné použití těchto metodik umožňuje značně zvýšit flexibilitu systému.

Společným rysem těchto přístupů je však značné omezení fáze modelování. Je tedy třeba, aby projekt byl svým charakterem vhodný pro tyto metodiky, v případě nutnosti použití klasických metodik je však udržovatelnost omezena. V takovém případě je třeba při business analýzách v co největší míře identifikovat směr budoucích změn a systém na ně připravit.

4.4.3.6 Přenositelnost

do značné míry souvisí s udržovatelností 4.4.3.5. Jde zde o schopnost IS být přenesen do jiného prostředí, a to z hlediska softwarového i hardwarového. Platí zde – podobně jako v předchozích charakteristikách a podcharakteristikách – nutnost ujasnit si se zákazníkem současné a budoucí potřeby z hlediska předpokládané dynamiky jeho podnikání a zanesení do diagramů BORMu.

4.5 Formální techniky návrhu datové struktury

Je třeba ještě věnovat pozornost formálním technikám návrhu datových struktur. Všechny doposud uvedené práce jsou významným přispěním do této problematiky, ale v této oblasti je OOP stále na samotném počátku. Komunita analytiků a návrhářů od techniky objektové normalizace očekává následující vlastnosti:

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

4.5.1 Pojem datový objekt

Nejprve musíme definovat, co rozumíme datovým objektem. Takový objekt je uchovávan v databázi a slouží "jen" k ukládání a manipulaci dat. Není to objekt, který by obsluhoval pomocí svých algoritmů nějakou část aplikace, tak navrhuji nepracovat zvlášť s datovými složkami a s metodami a zavést jediný pojem "atribut". Atributem budeme rozumět datovou vlastnost objektu, která je součástí rozhraní daného objektu. Nebudeme rozlišovat, zda je příslušný atribut implementovaný do objektu vloženým údajem a nebo zda je výsledkem zpracování dat objektu pomocí metody.

Nabízí se tu otázka, zda takové zjednodušení není příliš velké. Například Ambler-Beckův přístup (popsaný v kapitole 3.4.1.3) přímo pracuje se vztahem dat a metod a využívá ho ve svých definicích. Pro datové objekty si ale takové zjednodušení můžeme dovolit. Samozřejmě, že požadavek na vyvážený vztah mezi daty a metodami objektů je legitimní. Každý správný objekt by měl mít jen takové metody, které pracují s jeho daty. Ale zde hovoříme o hromadném zpracování dat, kdy se snažíme navrhnout schéma tak, aby co nejvíce vyhovovalo nějakému konkrétnímu zadání. To znamená, že dobrý návrhář musí brát v úvahu i business procesy, návyky a další požadavky a omezení reálného prostředí, pro které modeluje. Takže například adresa bydliště je nepochybně samostatný strukturovaný objekt třídy `Adresa`, ale pro uživatele databáze osob je mnohem přirozenější manipulovat se složkami adresy přímo z objektů třídy `Osoba` a ne být nucen se muset na složky adresy každé osoby dostávat dvěma operacemi, kde první nejprve vyvolá objekt adresy vcelku, aby potom druhá operace nad takto získanou adresou zpřístupnila číslo domu nebo změnila PSČ.

4.5.2 Tři objektové normální formy

Všechny výše uvedené požadavky nejlépe splňuje upravená podoba Ambler-Beckova přístupu. Tato verze již byla několikrát vyzkoušena (např. v [Bartoška et al. 2004]) a je součástí výuky. Jde o

postup, který by měl při modelování předcházet všem možným následným úvahám o použití dědění, skládání a dalších vazeb mezi objekty je vhodný pro fázi konceptuálního modelování.

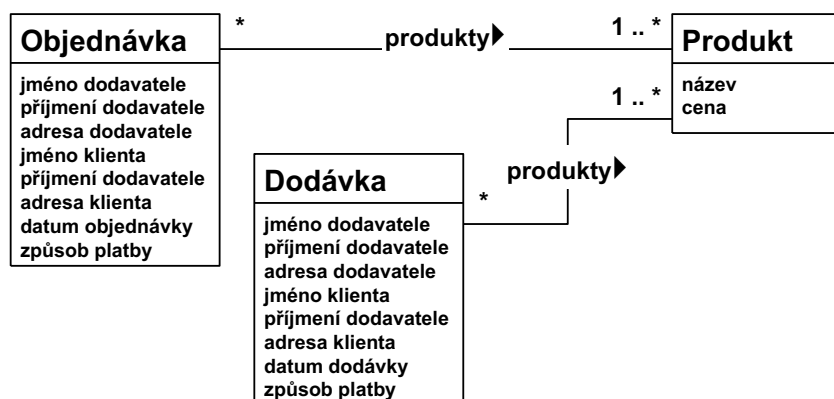
4.5.2.1 1ONF

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



obr. 29. příklad úlohy v nenormalizovaném tvaru

Na obrázku je příklad úlohy v nenormalizovaném tvaru a stejná úloha v 1ONF. Původní přístup je doplněn o pojistku pro případ, že návrháři sami nerozpoznají opakované skupiny atributů a nevyčlení je do samostatné třídy. Podle našich praktických zkušeností je pravidlo v tomto znění skutečně potřeba. Problém není vždy tak triviální jako ve zde uvedeném případě. Opakované atributy se mohou ukrývat i pod různými na první pohled nenápadnými názvy.



obr. 30. příklad úlohy v 1ONF

4.5.2.2 2ONF

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

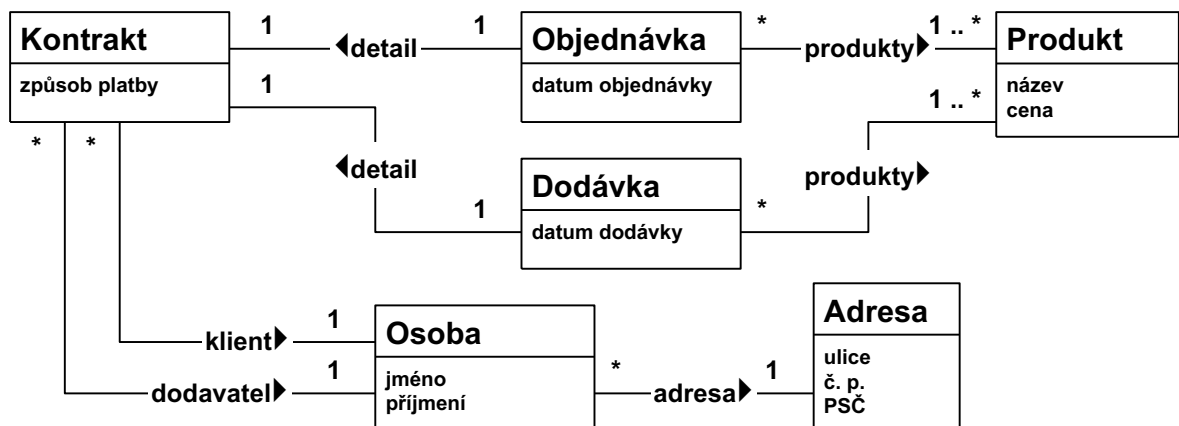


obr. 31. příklad úlohy v 2ONF

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

4.5.2.3 3ONF

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



obr. 32. příklad úlohy v 3ONF

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

Zde předložený přístup s sebou přináší řadu otázek a námětů k přemýšlení, jak jej rozšířit. Jedním z nich je otázka, jak do tohoto přístupu zahrnout vztah IS-A i další používané vazby mezi objekty, jejichž použití by mělo následovat po transformaci do 3ONF. Nabízí se například úvaha, že správným využitím IS-A vztahů by se mohla zabývat následná čtvrtá normální forma. Jinou samostatnou otázkou je také potenciální podobnost zde zavedeného pojmu "atribut" objektu s pojmem "aspekt" aspektově orientovaného paradigmatu programování.

4.6 Způsob provádění analýzy

Klíčovou fází tvorby informačního systému (IS) je analýza. Dobře provedená analýza je rozhodujícím předpokladem kvality výsledného produktu. Během posledních desítek let jsme svědky vzniku řady metodik zabývajících se analýzou IS. Tyto metodiky můžeme z hlediska přístupu k systémovému modelování rozdělit do dvou skupin:

1. metodiky specifické,
2. metodiky holistické.

4.6.1 Metodiky specifické

Mezi specifické metodiky patří prakticky všechny přístupy vznikající v průběhu šedesátých let. Jedná se především o rodinu strukturovaných metodik SA/SD i většinu metodik objektových (OMT). Tyto metodiky jsou typické svojí orientací na *informační systém*, nikoliv na realitu. Obsahují řadu více či méně komplexních diagramů a postupů, pomocí nichž lze zachytit část reality, která je důležitá pro implementaci budoucího IS.

Tyto metodiky vychází z předpokladu, že víme CO bude budoucí informační systém a řešíme pouze JAK bude vypadat.

4.6.2 Metodiky holistické

Holistické metodiky, které jsou reprezentovány především metodikou BORM, jsou relativně mladou záležitostí. Jejich hlavním rysem je komplexní přístup k problematice analýzy IS. Hlavním cílem je uchopení a zmapování celé reality prostředí a vytvoření abstraktního modelu, který je nezávislý na budoucí implementační platformě a technologiích. Výsledek business analýzy je pak určitým způsobem transformován na model vhodný pro počítačovou implementaci (viz např. 4.3.2).

Tento přístup se tedy nezaměřuje pouze na IS, ale komplexním způsobem analyzuje prostředí v celé své složitosti, vazbách a souvislostech. Teprve po důkladném zmapování se hledá pozice budoucího IS, jeho úloha v organizaci a struktura. Hlavní otázkou tedy je CO? a otázka JAK? je až na druhém místě. Holistické metodiky však nejsou v rozporu se specifickými. Zjednodušeně se dá říci, že tvoří jakýsi „obal“, který umožňuje v druhé fázi (JAK) efektivní použití metod specifických.

Práce pod metodou BORM, jakožto zástupce holistické metodiky, klade větší nároky na abstraktní myšlení analytika či konzultanta a účinná adopce tohoto přístupu vyžaduje osvojení nových dovedností a uvědomění si souvislostí problematiky.

4.6.3 Hledání zadání a systémové modelování

Zabývejme se dále problematikou, co vlastně má být obecně obsahem analýzy, tj. jakým způsobem uchopit modelovanou realitu a vytvořit analytický výstup. Postup vytváření modelu je analogický problematice **systémového modelování**. V dalším textu si problematiku rozebereme na jakési obecné organizace (podniku).

4.6.4 Matematický model a jeho základní složky

Jako teoretický aparát pro systémové modelování použijeme matematické modelování (MM). Nejprve si vymežíme pojem informační systém z hlediska matematického modelu.

V matematickém modelu můžeme rozlišit dvě základní složky, ze kterých se model skládá: proměnné a struktury.

4.6.4.1 Proměnné v modelu

V matematickém modelu se obecně můžeme setkat s těmito typy proměnných:

Vstupní proměnné endogenní. Tyto proměnné nás zajímají jako vstupy. V business analýze se jedná o vnější podněty, materiál, požadavky, které vstupují do procesu v organizaci. V metodice BORM jsou částečně obsaženy v *tabulce funkcí* a jsou upřesněny v průběhu dalších kroků.

Vstupní proměnné exogenní. Těmto proměnným se v MM říká také „vysvětlující“. Jedná se o vstupy, které přímo nesouvisí s činností organizace. Může se jednat o pomocné technické, legislativní, logistické a jiné vstupy ve formě materiální, služeb i know-how.

Výstupní proměnné. Obsahují výstup. Jedná se tedy o produkty organizace materiální (výrobní) i nehmotné (služby, know-how).

Vnitřní proměnné. Vnitřní (pomocné) proměnné, které charakterizují vnitřní vazby v modelu. V business analýze jim odpovídají vnitřní jednotky organizace a infrastruktura.

Pomocné proměnné. Pomocné proměnné používané v modelu. Tyto proměnné typicky nemají svůj obraz v realitě, ale jsou nutné pro kompletní zobrazení logiky organizace. Může se např. jednat o prvek „podpora zákazníkům“, kde toto oddělení přímo v podniku není, tato funkcionality však přítomna je a z nějakého důvodu ji potřebujeme mít v modelu ohraničenu.

4.6.4.2 Struktury modelu

Struktury modelu nám v podstatě určují vědecké odvětví, do kterého model spadá. V oblasti matematického modelování se obecně můžeme setkat s těmito typy struktur:

Analytické struktury z oblasti matematické analýzy, lineární algebry, diferenciálního počtu, maticového počtu a dalších odvětví.

Geometrické struktury. Model je popsán grafickými prostředky: body, přímkami, křivkami, grafickými závislostmi. Používá se aparát analytické geometrie a matematické analýzy.

Topologické struktury. Modely spadají do oblasti teorie grafů, teorie automatů, Petriho sítí, apod.

Vyšší struktury. Artefakty z oblasti schématických diagramů, například elektrotechnických, aj.

Mezi uvedenými strukturami lze identifikovat hierarchie podle úrovně abstrakce. Například struktury skládající se z elektronických obvodů mohou být ve své podstatě grafy. Grafy lze převést na matice, atd.

Pro nás je podstatné, že výchozím bodem v metodice BORM jsou slovní struktury, které jsou poté rozpracovány do topologických struktur (diagramů)¹². Používají se i maticové výstupy (tzv. *modelové karty*).

¹² Diagram OBA má charakter konečného automatu a Petriho sítě.

Z hlediska systémového modelování je ve strukturách třeba zachytit tyto aspekty:

Datovou strukturu. Z hlediska MM se jedná o zachycení vnitřních a případně pomocných proměnných. Prvky datové struktury v metodice BORM jsou ve fázi business modelování především *participanti*. V pozdějších fázích (konceptuálního) modelování pak přichází na řadu diagram objektů a tříd.

Procesy. Procesy zachycují různým způsobem vztahy mezi vstupními proměnnými, vnitřními proměnnými a výstupními proměnnými MM. V metodice BORM je klíčový procesní diagram (OBA).

4.6.5 Tvorba modelu

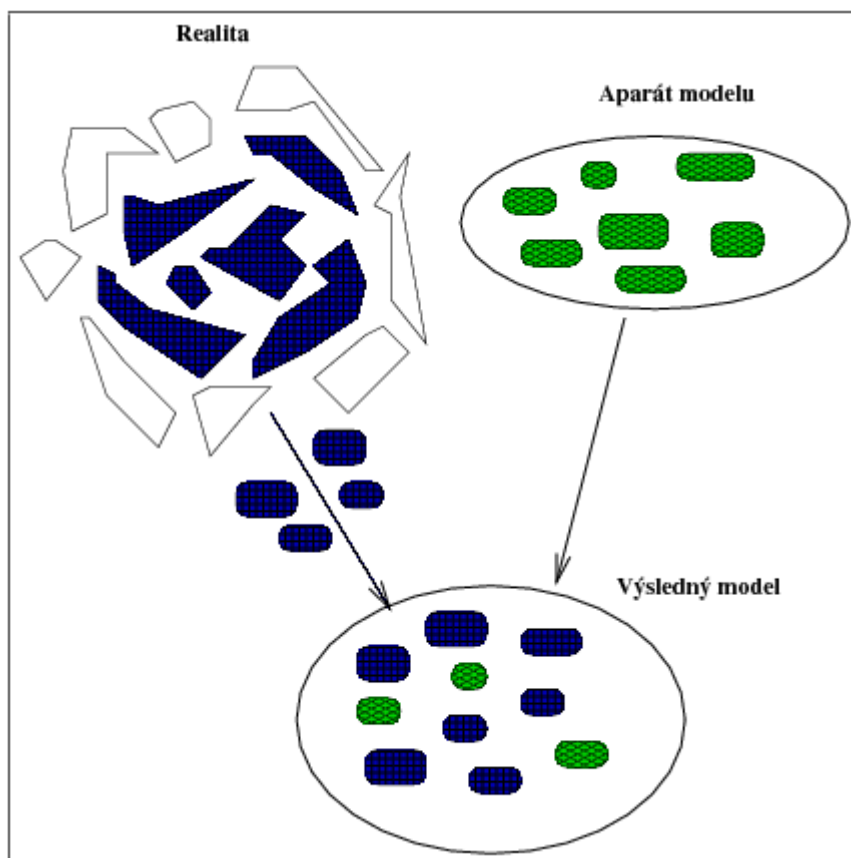
Tvorba modelů v metodice BORM má svá pravidla a specifika. Pokud se ovšem na tvorbu modelu podíváme z nejobecnější možné perspektivy, z perspektivy matematického modelu, dostáváme se k těmto čtyřem základním úkonům:

1. volba vstupů,
2. volba výstupů,
3. volba prvků a jejich hranic,
4. identifikace vazeb mezi prvky.

Kroky nemusí jít nutně v uvedeném pořadí a některé z nich mohou být podle charakteru situace a úlohy triviální.

4.6.5.1 Systémový řez

Obecně je třeba vytvořit tzv. *systémový řez*, tj. definovat podle nějakého úhlu pohledu podmnožinu reality a přiřadit různou důležitost jejím proměnným. Na obr. 33 systémovému řezu odpovídají vybarvené polygony.



obr. 33. Mapování reality na model

Ve výsledném modelu potom můžeme rozlišit dva typy prvků: prvky mající svůj původ v realitě a prvky aparátu modelu. Množství prvků aparátu modelu závisí na použitém modelu a velký poměr velikosti druhé skupiny k první znamená, zjednodušeně řečeno, že výsledný model obsahuje mnoho balastu netýkajícího se modelované problematiky. Naopak čím je větší poměr velikosti první skupiny ke druhé, tím je model bližší realitě, což má řadu výhod. Díky tomu, že metodika BORM je navržena právě s ohledem na složitost reality a možnosti jejího věrného zobrazení, je z tohoto hlediska na vysoké úrovni. Konkrétní měřeními podložené údaje však nemáme k dispozici.

Celou situaci ohledně obecného mapování reality na model zachycuje obr. 33. Obrázek též znázorňuje zjednodušení reality zachycené v modelu a skutečnost, že model i aparát jsou vždy ohraničené (konečné), ale realita nikoliv.

Při tvorbě systémového řezu je z hlediska MM klíčovým pojmem *izolovatelnost*. Izolovatelnost nám určuje, nakolik je prvek nezávislý na okolí, které zůstane mimo systémový řez (a tedy i organizaci). Tato závislost by měla být minimální. Obecně volíme mezi rozšířením systémového řezu a akceptováním omezení a komplikací vyplývajících ze závislosti. Příkladem může být situace, kdy firma má vozový park řešený outsourcingem. Odpadá tak přímá agenda vozového parku a může být diskutabilní, zda-li tuto agendu zahrnout do analýzy. Její zahrnutí zesložití a zneprůhlední vypovídací schopnost modelu a přidělá práci analytikům, její nezahrnutí však může mít za následek např. problémy v zobrazení procesu přidělování aut referentům a jeho souvislosti.

Na problematiku systémového řezu narazíme při práci v metodice BORM prakticky ve všech fázích a je také jednou z hlavních příčin, proč je přirozené, že jsou složitější modely zpravidla vytvářeny iterativně.

4.6.5.2 Vazby v modelu a systémový řez

Z hlediska vazeb mezi prvky modelu v souvislosti se systémovým řezem obecně rozlišujeme tyto případy:

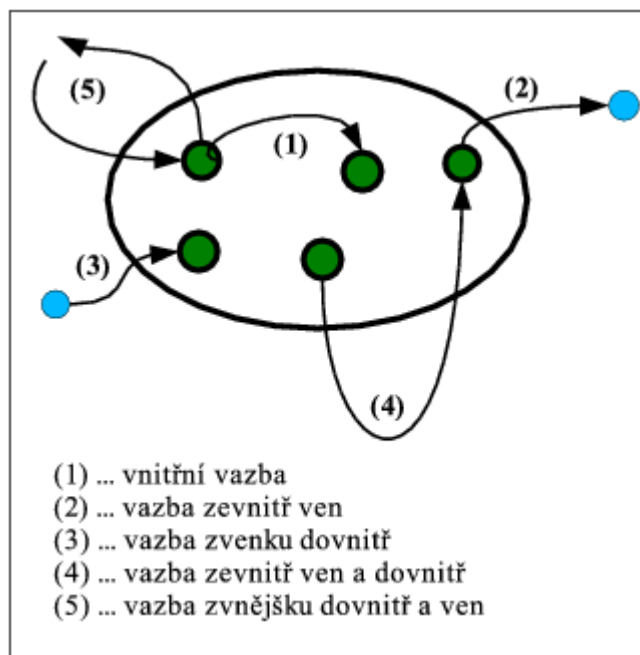
Vnitřní vazba. Jedná se o vazbu uvnitř modelu mezi prvky. Z hlediska systémového řezu bezproblémové.

Vazba zevnitř ven. Model tedy ovlivňuje zbytek systému mimo systémový řez. Pokud jsme si jisti, že ovlivněná část reality není součástí systémového řezu, není třeba se touto situací zabývat.

Vazba zvnějšku dovnitř. Z hlediska aparátu matematického modelování všechny vstupy do systému musí být ve formě vstupní proměnné. Musíme tedy buď vazbu nahradit pomocným vstupem (exogenní vstupní proměnnou) nebo rozšířit systémový řez. Prakticky to znamená to, že do procesu organizace nemůže nic vstupovat „bokem“, ale jen přes přesně definované a vymezené organizační jednotky¹³.

Vazba zevnitř ven a dovnitř. Pokud se nám dvě vazby zvnějšku dovnitř a zevnitř ven podaří identifikovat jako jednu takovouto vazbu, může být výhodné rozšířit systémový řez. To ale opět závisí na vazbách vnějšího prvku. Příkladem může být opět autopark zmíněný výše.

Vazba zvnějšku dovnitř a ven. Pokud prvek systémového řezu účastníci se této vazby nemá v systému jinou funkci, je možné ho vyřadit (Toto však není případ na obrázku, kde prvek má ještě vnitřní vazbu). Takovým humorným ilustrativním příkladem by mohl být případ, kdy analytik omylem zanesl do systémového řezu oddělení, které náhodou bylo zrovna ve stejné budově jako analyzovaná organizace, ale patřilo k úplně jiné firmě. Takové oddělení by pak nemělo žádné vazby na naši organizaci a v systému by jen tak „viselo“.



obr. 34. Možné vazby systémového řezu

Jednotlivé situace znázorňuje obr. 34. Vazby jsme ale diskutovali jen ve vztahu k jednomu prvku v systému. Při skutečné analýze je třeba brát v potaz řetězové vazby. Jelikož v praxi bývají vazby

¹³ Výjimkou se může zdát být elektronický styk s klienty např. formou webového rozhraní přímo do systému. V tomto případě je však vstupním bodem (proměnnou) zadávací formulář, který je svým způsobem.

často značně složité a provázané, nejedná se o jednoduchou problematiku. Velkou roli zde hraje cit, intuice a zkušenosti analytika.

4.6.6 Způsob tvorby modelu

Problematika způsobu tvorby modelu je značně rozsáhlá a je v literatuře z různých pohledů detailně rozpracována. Z hlediska systémového modelování obecně znamená proces tvorby modelu definovat prvky a jejich vazby. Tento problém lze řešit čtyřmi přístupy (metodami):

- metoda shora dolů (dekompozice),
- metoda zdola nahoru (syntéza),
- metoda zevnitř ven,
- ostrůvková metoda.

Metoda shora dolů. Postupuje se metodou dekompozice, tj. začne se od hrubého návrhu, který postupně zjemňujeme. Tato metoda je vhodná pro modelování problémů, ve kterých analytik má schopnost se takto abstraktně orientovat. Typicky se jedná o problémy z oblasti matematické a vědecké, nicméně lze nalézt i případy z business praxe, pro které lze o této metodě uvažovat.

Metoda zdola nahoru. Začíná se od nejjemnější granularity a budují se postupně struktury s větší úrovní abstrakce. Tato metoda je vhodná pro problémy, kde jsou známy detailně všechny prvky a vazby, ze kterých se postupně vytváří celky na větší úrovni abstrakce.

Metoda zevnitř ven. Model se buduje od některé části systému. Nejlepší metoda pokud určitou část systému analytik zná lépe než ostatní. Vzniká zde ale riziko či zpomalení práce v případě, že se narazí na slepou uličku, nebo není zřejmé, jakým směrem by se měla analýza provádět. Vhodné je proto mít k dispozici konzultanta¹⁴, který je schopen analytikovi v případě potřeby asistovat.

Ostrůvková metoda. Jedná se o metodu zevnitř ven, která je aplikována postupně na různé části systému, dokud se nám takto vzniklé nepropojí. Ačkoliv tato metoda není příliš citována v publikacích, je to ideální metoda pro business modelování, má však i svá úskalí. Především je zde klíčová role manažera projektu, který musí ohlídat, aby se analýza nezvrhla do zmateného přeskakování „z jednoho na druhé“.

4.7 Řízení týmu, softwarové profese

Práce na větším softwarovém celku musí být určitým způsobem řízena, má-li být efektivně dosaženo zamýšlených cílů. Řízením projektu se zabývá manažer, jehož hlavní směry činnosti jsou následující:

- Komunikace se zadavateli. Manažer klade návrhy na softwarové projekty různým klientům. Každý takový návrh musí obsahovat odůvodnění a časové a finanční nároky.
- Plánování projektu. Úkolem manažera je vypracovat harmonogram projektu a stanovit pravidla, kterými se bude projekt řídit včetně jednotlivých etap a jejich návaznosti. Je třeba zajistit způsob validace produktu, různé konfigurace, nároky na zaškolení a údržbu.

¹⁴ Zde ve významu jakéhokoliv člověka, který se v analyzovaném prostředí pohybuje a orientuje.

- Řízení rozpočtu je jednou z klíčových rolí manažera. Patří sem mimo náklady na vlastní tvorbu i náklady např. na techniku, školení a cestování.
- Úkolem manažera je rovněž výběr osobností a týmů pro řešení projektu a jeho složek. Při výběru je třeba respektovat složitost požadavků i rozpočet projektu.
- Kontrola stavu projektu a schopnost reagovat na problémy, které vznikají při samotné tvorbě díla.
- Prezentace výsledků během projektu a na konci projektu jak zadavateli, tak i uživatelům.

4.7.1 Softwarové profese

Kromě manažera vyžadují jednotlivé etapy vytváření programového celku další různé typy profesí. Jedná se o následující profese (z historických důvodů se uvedené profese vesměs zjednodušeně nazývají „programátor“):

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

Na straně zadavatele je třeba ještě mít gestora/kontaktní osobu, která ručí za uzavření etap projektu – je to pojistka, že projekt jde správným směrem a že se nestane, že zadavatelé na konci vývoje řeknou, že to chtěli jinak. Při tvorbě informačních systémů je skutečně veliký rozdíl mezi tím, kdy zadavatel řekne „to nechci“ (během projektování) a nebo „to jsem nechtěl“ (nad dokončeným systémem).

4.7.2 Organizace pracovních týmů

Při tvorbě softwaru je nutná spolupráce řešitelů. Vzájemná komunikace řešitelů u složitých projektů vyžaduje až přes 50% celého času. Přiměřené organizaci týmu je proto třeba věnovat dostatečnou pozornost. Pracovní týmy se dělí na základě jejich organizace na:

4.7.2.1 Nestrukturované (dělba práce dle objemu)

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

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

4.7.2.2 Strukturované týmy (dělba podle profese)

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

4.8 Algoritmizace rozpočtu

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

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

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

$$M = Z \diamond E,$$

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

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

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

- zcela nová řešení ($k = 3,6$, $n = 1,2$),
- varianty známých řešení s novými prvky ($k=3$, $n=1,12$) a
- varianty známých řešení ($k = 2,4$, $n = 1,05$).

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

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

kde m je konstanta pro různé typy úloh ($m = 0,32$ pro nová řešení, $m = 0,35$ pro střední typ úlohy a $m = 0,38$ pro variantní řešení). Potřebný počet N řešitelů je potom určen jako

$$N = E / T.$$

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

$$E = K \times R^n \times x_1 \times x_2 \times \dots \times x_m,$$

kde příslušná úprava může představovat až násobky či zlomky (až $5\times$ □ není výjimkou) původní hodnoty. Doporučované hodnoty jednotlivých koeficientů x_i jsou následující:

- Požadovaná spolehlivost produktu, která je přímo úměrná obtížnosti řešení v rozsahu 0,75 až 1,4.
- Požadovaná adaptabilita na změny prostředí, která je přímo úměrná obtížnosti řešení v rozsahu 0,87 až 1,3.
- Rozsah použitých databází, která je přímo úměrná obtížnosti řešení v rozsahu 0,94 až 1,16.
- Složitost produktu ve smyslu požadavků na speciální vstupy a výstupy či algoritmy zpracování, která je přímo úměrná obtížnosti řešení v rozsahu 0,7 až 1,65.
- Časová náročnost řešení (např. míra požadavků na časové odezvy na hranici možností použité techniky která vyžaduje detailnější algoritmy), která je přímo úměrná obtížnosti řešení v rozsahu 1,0 až 1,66.
- Paměťová náročnost řešení (např. velikost požadované paměti na hranici možností použité techniky která vyžaduje detailnější algoritmy), která je přímo úměrná obtížnosti řešení v rozsahu 1,0 až 1,56.
- Dostupnost techniky pro řešení (zda je technika k dispozici nebo zda se ještě vyvíjí), která je přímo úměrná obtížnosti řešení v rozsahu 0,87 až 1,15.
- Schopnost řešitelského týmu, která je nepřímo úměrná obtížnosti řešení v rozsahu 1,46 až 0,71.

- Míra znalosti řešené aplikace, která je nepřímo úměrná obtížnosti řešení v rozsahu 1,29 až 0,82.
- Míra znalosti použitých programovacích prostředků, která je nepřímo úměrná obtížnosti řešení v rozsahu 1,21 až 0,9.
- Výkonnost použitých programovacích prostředků, která je nepřímo úměrná obtížnosti řešení v rozsahu 1,24 až 0,83.
- Míra využívání moderních programovacích technik, která je nepřímo úměrná obtížnosti řešení v rozsahu 1,24 až 0,82.
- Požadavky na zkrácení časového rozvrhu oproti odhadu, která je nepřímo úměrná obtížnosti řešení v rozsahu 1,23 až 1,0.

4.9 Softwarové metriky a metoda funkčních bodů

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

Softwarové metriky se dělí podle časového kritéria na:

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

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

- ✓ rojektové metriky (project metrics) sloužící především k měření veličin potřebných k určování a porovnávání charakteristik jednotlivých projektů jako celku z pohledu projektového řízení během všech fází životního cyklu. Pro objektový přístup to jsou například následující ukazatele:
 - míra kvality řídicího procesu, (v současné době rozpracováváno v sadě norem ISO),
 - míra kvality vývojového procesu, (v současné době rozpracováváno v sadě norem ISO),
 - velikost vyvíjené aplikace (počet scénářů, tříd, subsystémů, modelů, ...),
 - velikost pracovního týmu (průměrný počet člověko-dní na scénář, třídu či objekt, průměrný počet vyvinutých entit na pracovníka, ...) a
 - plánovací ukazatele (počet iterací ve vývoji, počet uzavřených subdodávek, počet etap projektu, ...).
- ✓ Návrhové metriky (design metrics) zabývající se podrobnostmi vyvíjené softwarové aplikace a slouží jako efektivní podpora pro testování produktů, vzájemné porovnávání produktů a nástroj pro určování silných a slabých míst návrhu pro potřeby měření, opravování a vylepšování jednotlivých komponent vyvíjené aplikace. Pro objektový přístup to jsou například následující ukazatele:
 - velikost metod (průměrný počet posílaných zpráv na metodu, prům. počet příkazů na metodu, rozsah komentářů, stupeň polymorfismu, ...)
 - velikost objektů (průměrný počet instančních a třídních metod na třídu, rozsah komentářů, průměrný. počet instančních a třídních proměnných na třídu, ...) a

- míra vazeb mezi objekty (průměrná hloubka dědičnosti, průměrný počet složek objektů, závislost objektů, znovupoužitelnost objektů a metod, průměrný počet odstíněných metod na třídu, rozsah komentářů,...).

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

kde N je celkový počet funkčních bodů požadované aplikace,

- inputs označuje počet požadovaných uživatelských vstupů do aplikace,
- outputs označuje počet požadovaných uživatelských výstupů z aplikace,
- queries označuje počet požadovaných uživatelských dotazů do aplikace,
- interfaces označuje počet požadovaných datových rozhraní aplikace do jiných aplikací a
- algorithms označuje počet algoritmů, které bude aplikace používat ve svých výpočtech.

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

programovací jazyk	počet příkazů na jeden funkční bod (SPR Inc., 2002)	počet příkazů na jeden funkční bod (Albrecht & Behrens, 1983)
Smalltalk	21	21
Ada 95	49	-
Java	53	-
C++	53	-
Basic	-	64
Lisp	-	64
Prolog	-	64
Modula-2	-	71
PL/1	-	80
RPG	-	80
Pascal	-	91
Fortran	-	106
COBOL	107	106
Algol	-	106
C	128	150
Assembler	-	320

obr. 35. Funkční body programovacích jazyků

5 Vhodná alternativa – agilní metodiky

5.1 Úvod – Softwarová krize

V kapitole 1 jsme se zabývali problematikou sémantické mezery a jejím dopadem na implementaci informačních systémů. Sémantická mezera je však jen dílčím problémem. Při vývoji komplexního softwarového vybavení narážíme na celou řadu dalších problémů. Tyto problémy zejména za poslední dvě desetiletí dosáhly alarmujících rozměrů a stále více se hovoří o tzv. softwarové krizi. Jedná se o závažný jev, kdy i přes značný výkon výpočetní techniky není možné dostatečně uspokojovat poptávku společnosti po stále složitějších kvalitních programových systémech.

Takzvaná softwarové krize je výslednicí mnoha příčin. Jednotlivé vlivy nejsou stejné v čase ani v prostoru¹⁵. Můžeme si však jmenovat nejzávažnější společné jmenovatele:

5.1.1 Propast mezi vývojáři a uživateli

Tento problém má kořeny v minulosti. Výpočetní technika v ranných fázích vývoje sloužila především akademickým a vědeckým kruhům. Také řešené problémy byly spíše výpočetního a tudíž exaktního rázu. Programátoři tedy byli většinou i uživateli nebo k nim měli velmi blízko.

Dnes je však třeba, aby vývojáři často vyvíjeli aplikace řešící problémy oborů, které jsou jim velmi vzdálené a naopak například burzovní makléř si těžko bude sám vyvíjet informační systém. Jsme tedy svědky rozrůstající se *propasti mezi vývojářem a uživatelem* a vývoj na základě introspekce vývojářů již nemůže fungovat. A to je právě nejzávažnější příčinou softwarové krize. Již nestačí dřívější postupy analýzy, vývoje, testování. Je třeba využívat postupy, které umožní vývojářům co nejlépe pochopit svět uživatelů a jejich potřeby a na druhou stranu pomohou uživatelům lépe komunikovat s vývojáři a sdělovat jim svoje potřeby a očekávání. Systém je kvalitní jen tehdy, když jsou s ním spokojeni uživatelé a přináší jim hodnotu výrazně vyšší než obtíže spojené s jeho nasazením a používáním¹⁶.

5.1.2 Komplexnost systémů

Druhým aspektem, kde nutně narážejí původní postupy, je rozrůstající se *komplexnost systémů*, a to jak na úrovni domény řešeného problému - zvětšuje se počet poskytovaných funkcí, je třeba pracovat s velkým množstvím složitě provázaných dat, apod., tak na úrovni domény technologické - vzrůstají požadavky na distribuovanost, bezpečnost, schopnosti uživatelského rozhraní, atd.

5.1.3 Opomíjení lidských hodnot

Ukazuje se, že tento bod není ani v oblasti softwarového inženýrství zanedbatelný. Jako u každého lidského oboru, i softwarové inženýrství lze provádět dvěma různými přístupy. Můžeme například, jak nám ekonomické poučky ranného kapitalismu velí, hledět jen na svůj prospěch, tj. především zisk na úkor obchodního partnera¹⁷ a uvnitř vývojářské firmy pak egoistická snaha získat prospěch na úkor

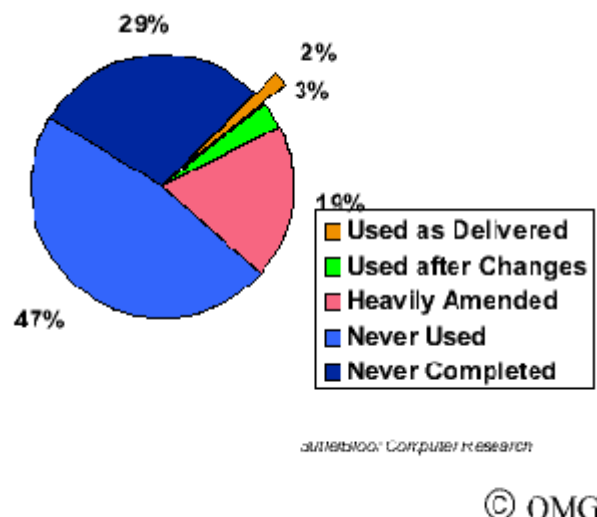
¹⁵ Časem je myšlen historický vývoj a prostorem jsou myšleny různé projekty v přibližně stejném období.

¹⁶ Problematika jakosti při vývoji IS je diskutována v kapitole **moje kapitola o jakosti**.

¹⁷ zde se jedná především o vztah zadavatel(uživatel)-dodavatel(vývojářská firma).

kolegů. Nebo se lze chovat kooperativně, tj. firma jako celek se snaží, aby zákazník byl spokojen a uvnitř firmy zaměstnanci pracují v atmosféře vzájemného respektu a podpory. A zákazník se snaží svým otevřeným a vstřícným přístupem usnadnit vývoj systému. Tento přístup se začíná ve vyspělých ekonomikách stále více prosazovat, u nás se však ještě bohužel často setkáváme s přístupem vytlačovat morální a etické principy z businessu a považovat je za brzdu, slabost, neschopnost prosadit se. Ukazuje se ale, že hrají velmi důležitou roli a jejich potlačením dosáhneme často opačného efektu.

5.2 Projevy softwarové krize



obr. 36. Kvalita dodaného IS

Samotný pojem „softwarová krize“ je značně abstraktní a diskuse kolem něj částečně filosofická. Negativní projevy softwarové krize jsou však velmi hmatatelné a kvantitativně měřitelné.

Jmenujme si především

Vysoké náklady na vývoj - Software je velmi drahý, protože jeho vývoj je náročný a plný rizik.

Zpoždění harmonogramu - tj. neschopnost dodat funkční systém do dohodnutého termínu.

Nedodání systému - Různá rizika projekt zcela ochromí a zabrání jeho dokončení („Never Completed“ v grafu na obr. 36)

Nevyhovující systém - Projekt je sice dokončen, ale systém nelze používat. Graf na obr. 36 ukazuje, že toto je nejčastější situace. V lepším (bohužel však méně častějším) případě lze systém používat po dodatečných (a často nákladných a zdlouhavých) úpravách („Heavily Amended“ v grafu). Často je však systém prakticky zcela nepoužitelný. Důvody tohoto stavu mohou být:

Nepochopení zadání - systém neodpovídá požadavkům uživatelů.

Poruchovost - systém sice splňuje zadání, ale množství chyb ho zcela ochromuje.

Neudržitelný systém - Ani po úspěšném nasazení nemusí být vyhráno. Je třeba provádět podporu a údržbu systému po celou dobu jeho životnosti. Také musí být možné (do určité míry) systém přizpůsobovat měnícím se požadavkům.

Procento případů, kdy je dodaný systém alespoň trochu přínosný, je dle grafu na obr. 36 pouhých 25%, což je alarmující.

5.3 Riziko - hrob projektu

Je jasné, že příčiny projevů zmíněných v sekci 5.2 se nalézají ve způsobu organizování procesu tvorby softwarového projektu¹⁸. Celou situaci ovšem komplikují *rizika*, se kterými musí dobrá metodika počítat.

Rizika, se kterými je třeba počítat, jsou především:

Nepochopení (části) zadání - I přes dokonalý systém získávání specifikace se může stát, že dojde k nedorozumění.

Změny zadání - jsou noční můrou všech projektových manažerů i programátorů. Častým přístupem je omezit co nejvíce změny dokonalou specifikací zadání před začátkem práce. To je vždy výhodné pro vývojářskou firmu, málokdy ale pro zákazníka. Pokud je cílem softwarové firmy spokojenost zákazníka, je třeba přestat se před tímto fenoménem schovávat.

Technologická rizika - Jedná se o rizika různě závažných selhání hardwarového a softwarového vybavení při vývoji. Do tohoto bodu však můžeme pro naše potřeby zahrnout i technologické problémy týmu způsobené nedostatkem znalostí a zkušeností s vývojovými nástroji a postupy.

Fluktuace pracovníků - je velmi běžným problémem, který nesmí být podceněn. Především je třeba zabránit tomu, aby projekt byl životně závislý na jednom nebo dvou pracovnících.

Neschopnost dodržet plán/požadavky - I pokud se žádné z předchozích rizik nenaplní, projekt se stejně může dostat do potíží vinou vnitřních nesrovnalostí. Příčinami může být např. špatná komunikace s vedením nebo mezi jednotlivými pracovníky, nevhodné pracovní prostředí, problémy v kolektivu, atd. Příčiny mohou být někdy velmi špatně odhalitelné a mohou se i kombinovat.

5.4 Agilní metodiky a extrémní programování

Extrémní programování je jednou z tzv. *agilních metodik* (AM), což je skupina metodik, které vycházejí z toho, že jedinou cestou, jak prověřit správnost navrženého systému, je jej co nejrychleji vyvinout, předložit zákazníkovi a na základě zpětné vazby upravit. Pod agilní metodiky jsou dnes zařazovány následující metody a přístupy:

- Adaptivní vývoj software (Adaptive Software Development, ASD)
- Feature-Driven Development (FDD)
- extrémní programování (Extreme Programming, XP)
- Lean Development
- SCRUM
- Crystal metodiky.

Tyto přístupy se navzájem liší v mnoha aspektech, ale jsou spojeny svým zaměřením na lidi, výsledky, minimalizaci metod a maximalizaci spolupráce (podrobně viz [AP 2001]). My se v této kapitole budeme podrobně věnovat extrémnímu programování.

¹⁸ Je myšlena organizace v obecném slova smyslu, nejen způsob řízení ze strany managementu.

Extrémní programování je jedna z agilních metodik. Jejím autorem je Kent Beck a za základní publikaci je považována jeho kniha [Beck 2002]. Kent Beck má za sebou dlouhou a bohatou kariéru vývojáře ve Smalltalku a systémového konzultanta. Extrémní programování je velmi mladá metodika (1997), principy v ní užívané jsou však všechny dlouho dobře známé. Její název vychází z filosofie vzít pár osvědčených věcí a ty dotáhnout do extrémů. Je dobré testovat? Tak budeme testovat extrémně často. Je dobré integrovat? Tak budeme integrovat každou chvíli. Víc hlav víc ví? Tak budeme u počítače sedět vždy dva! Může se zdát, že tak nepřináší tolik potřebnou revizi zastaralých způsobů, ale opak je pravdou. Výsledkem jsou velmi revoluční, mnohdy až kontroverzní doporučení.

5.5 Čtyři parametry

Existuje celá řada možností, jak kvantifikovat model vývoje programového vybavení. Kent Beck v [Beck 2002] používá systém čtyř řídicích parametrů (nebo proměnných), který je pro potřeby výkladu XP velmi názorný a zcela dostačující. Tyto čtyři parametry jsou:

- náklady
- čas
- kvalita
- funkcionalita

Důležité je, že tyto parametry nejsou nezávislé a jejich vzájemné závislosti jsou poměrně komplikované. Podrobnější studium těchto závislostí přesahuje rozsah tohoto textu, spokojíme se pouze se dvěma konstatováními:

1. Každý z parametrů vyžaduje citlivé nastavení.
2. Vedení může týmu předepsat hodnoty maximálně *tří* ze čtyř proměnných.

Vysvětleme si trochu první tvrzení:

Náklady - Příliš málo financí bude znamenat problémy s technickým vybavením, podporou a motivací týmu. Na druhou stranu více finančních prostředků, než je třeba, nepřinese projektu podstatné zlepšení ostatních parametrů a může dokonce zvýšit režii¹⁹. Pravdou ale také je, že většinou všichni vystačí s nižšími prostředky, než které by chtěli.

Čas - Je jasné, že nedostatek času se podepíše na všech třech zbylých parametrech, je však dobré systém co nejdříve nasadit do provozu, protože zpětná vazba od uživatelů používajících systém v provozu má daleko vyšší kvalitu než jakékoli druhy formálních verifikací.

Kvalita - Nastavení tohoto parametru také ovlivňuje do určité míry všechny ostatní a je třeba ho uvážit podle konkrétního nasazení produktu.

Funkcionalita - Je třeba především vysvětlit zákazníkům, že menší šíře zadání může zlepšit ostatní parametry. Systém však musí dobře plnit svoje určení a nesmí se ušetřením funkce snížit jeho hodnota.

Druhé tvrzení se může narážet na odpor řídicích pracovníků, ale jak ukazuje praxe, pokud jsou týmu nuceny našponované hodnoty všech čtyř proměnných, nedosáhne se požadovaných výsledků často ani u jedné z nich. Zvláště při zvýšených požadavcích na jednu proměnnou musíme očekávat

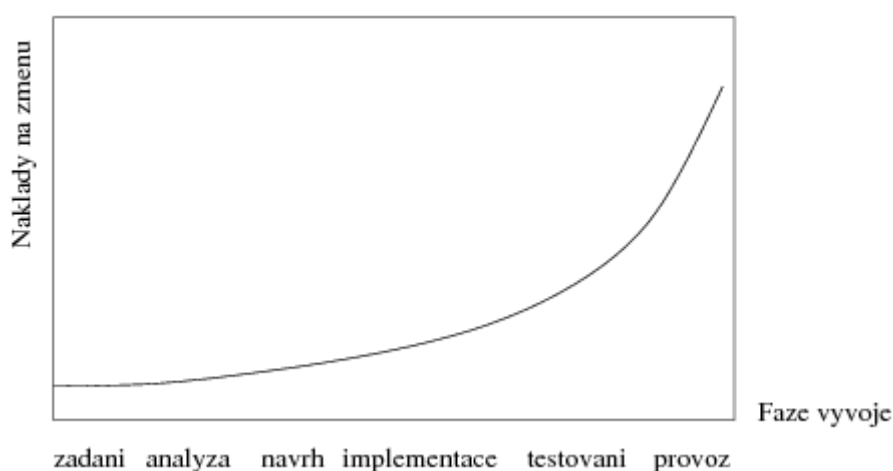
¹⁹ Např. nákup a instalace nových nástrojů, které však ve skutečnosti nejsou potřeba.

prudkou změnu jiné proměnné či proměnných. Nereálné požadavky zvyšují stres, což se negativně projeví na celém procesu vývoje.

V běžných přístupech se obvykle snažíme ukotvit funkcionalitu (šíři zadání) a jí přizpůsobit náklady a čas. Pro XP (a obecně pro agilní metodiky) ovšem bývá typické, že náklady a čas jsou fixní, zatímco funkcionalita je proměnná. Toto vše při imanentní maximalizaci kvality kódu.

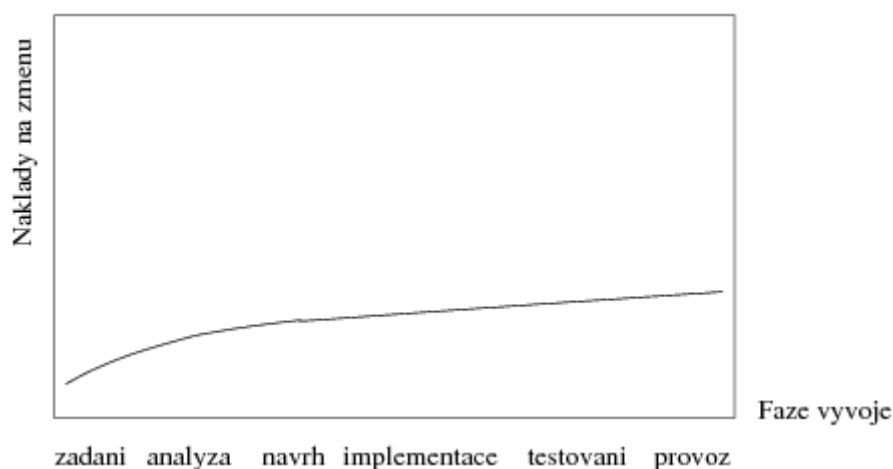
5.6 Náklady na změnu

Nejdůležitějším předpokladem úspěšného nasazení XP je *změna křivky závislosti nákladů na změnu na čase této změny*. Graf na obr. 37 ukazuje dříve typický exponenciální průběh, který nutí k rozsáhlým analýzám, specifikacím a z ekonomických důvodů vylučuje pružnost na změny.



obr. 37. Exponenciální náklady na změnu

Jak si uvědomíme při studiu principů XP, v případě, že tuto křivku nejsme schopni (či často ochotni) změnit, nemá smysl se dále zabývat XP. Tento neblahý průběh však není naštěstí ve většině případů neměnný²⁰ a dnes máme k dispozici mocné nástroje, které tuto křivku dokáží ohnout na průběh podobný obr. 38.



obr. 38. Náklady na změnu potřebné pro XP

²⁰ výjimky viz. sekce v kapitole *Nasazení*.

Jsou to především

Lepší programovací jazyky - Jazyky na vyšší úrovni abstrakce umožňují se oprostit od technických detailů a rozsáhlé standardní knihovny ušetří spoustu práce. Změny je potom možné provádět snadněji a s menším vlivem na ostatní kód. Zde jsou největší devizou *objektové jazyky*, zvláště pak jazyk Smalltalk. Ačkoli je XP v principu nezávislé na programovacím jazyku, nejlepších výsledků je dosahováno právě u týmů používajících Smalltalk a příbuzné technologie (např. objektové databáze).

Mocnější programovací nástroje - Dnes máme k dispozici moderní grafická uživatelská rozhraní, vizuální programování, vizualizační nástroje a nástroje podporující refaktorizaci (tj. změnu kódu bez změny jeho funkčnosti, viz 5.8.6).

Lepší programovací postupy - Jmenujme především návrhové vzory (více v [Beck 1997], [Gamma et al.]).

Standardy - Stále více se začínají prosazovat standardy, což především velmi zjednodušuje implementaci, sdílení kódu a reinženýring. Standardy dnes existují snad pro všechny oblasti výpočetní techniky - od síťových protokolů, výměny dat distribuovaných systémů (SOAP) i lokálních (XML) až po programovací jazyky a prostředky (SQL). Nesmíme též opomenout blahodárný vliv ustalujících se konvencí pro psaní kódu (např. názvy proměnných, velká/malá písmena, atd.).

Jakmile se nám podaří ohnout křivku nákladů na změnu, otevírají se nám obrovské možnosti. A právě extrémní programování jich dokáže skvěle využít.

5.7 Filosofie XP

Než se pustíme do samotného výkladu principů, nebude od věci dát na chvíli slovo Kentu Beckovi, aby nám pomohl nahlédnout do filosofie stojící za XP:

„Všechny metodiky se zakládají na strachu. Pokoušíme se získat návyky, které mají zabránit obavám, aby se staly skutečností. XP není v tomto ohledu odlišné od žádné jiné metodiky. Rozdíl je v tom, jaké obavy jsou do XP zapuštěny. V XP se odráží můj strach do té míry, do jaké je XP mým dítětem“. (Kent Beck, převzato z [Beck 2002])

Autor především zmiňuje tyto obavy:

- Konání zbytečné práce.
- Konání práce, na kterou člověk není hrdý.
- Zrušení projektů z důvodu
 - nesprávných obchodních rozhodnutí,
 - nesprávných technických rozhodnutí činěných vedením společnosti místo programátora.
- Strávení života pouze prací

Autorovy převratné myšlenky jsou hlavně dílem věcí, ze kterých strach nemá:

- Psaní zdrojového textu.

- Názorové změny.
- Pokračování v práci, aniž by bylo známé vše o budoucnosti.
- Spoléhání se na ostatní lidi.
- Upravování analýz a návrhu běžícího systému.
- Psaní testů

Ačkoli se jedná právě o věci, ze kterých mají všichni největší strach, Kent Beck a rozrůstající se skupina jeho následovníků praktikujících XP, dokazují, že právě takové radikální překročení niterných obav může být tou pravou cestou.

5.7.1 Čtyři hodnoty

Nasazení metodik extrémního programování vyžaduje především silnou motivaci a disciplínu všech zúčastněných - vedoucích pracovníků, programátorů, testerů, zákazníků i dalších²¹. Základem XP postupujícím všechna pravidla jsou *čtyři hodnoty*. Je důležité je mít stále na zřeteli a nesklouznout k zlovykům. Tyto hodnoty jsou:

- komunikace
- jednoduchost
- zpětná vazba
- odvaha

Všechny postupy XP v sobě obsahují minimálně jednu tuto hodnotu.

Rozveďme si tedy (stále na dost abstraktní úrovni) trochu tyto čtyři hodnoty:

Komunikace - Aby (jakýkoli!) projekt zdárně fungoval, je nutné intenzivně, ale *upřímně* a *smyslupně* komunikovat. A to na úrovni člen týmu-člen týmu (programátoři, testeri, ...), tým-vedení a zadavatel-dodavatel. Zde to vypadá jako všem známé, ale málo používané předsevzetí. Uvidíme však, že síla XP je v tom, že nás opravdu ke komunikaci *donutí*.

Jednoduchost - Cokoli programujeme by mělo být nejjednodušší možné, co ještě splňuje zadání. To je nutný předpoklad k tomu, abychom udrželi kód.

Zpětná vazba - Vždy může dojít k nedorozumění. Sebegeniálnější plánování nemůže počítat se vším. I ten nejužasnější programátor dělá chyby. Jedině zpětná vazba na všech možných úrovních nám poskytne možnost se zotavit z chyb s co nejmenšími problémy.

Odvaha - Samotné zavedení XP vyžaduje notnou dávku odvahy. Odvahu vyžadují i závažná rozhodnutí, která jsou někdy velmi nepříjemná (např. zahodit velkou část kódu). Bez odvahy se nelze dostat kupředu.

5.7.2 Základní pravidla

Praxe XP je založena na několika obecných pravidlech, která je nutné přijmout.

²¹ Dále uvidíme, že XP zavádí i netradiční nové role.

5.7.2.1 Rychlá a kvalitní zpětná vazba

Abychom mohli provádět správné korekce včas, musíme mít rychlou a kvalitní odezvu na korekce předcházející. Rychlejší zpětná vazba také podstatně zlepšuje proces učení a získávání zkušeností. Tento princip souvisí s principem přírůstkové změny (5.7.2.2).

5.7.2.2 Přírůstková změna (pravidlo drobných korekcí)

Tento princip je v XP využíván mnoha různými způsoby a říká, že bychom nikdy neměli dělat velké změny najednou, ale lepší je všechny problémy řešit pomocí řady těch nejmenších změn, které ještě mají vliv.

Pravidlo drobných korekcí je jedním z momentů, proč XP je tak odolné proti *riziku změny zadání*²². Toto pravidlo v přeneseném slova smyslu říká: Když chcete dobře řídit auto, je třeba provádět neustálé drobné korekce směru v závislosti na okamžité situaci. Nelze řídit tak, že vůz pouze nasměrujete na vzdálený bod na horizontu a strnule držíte volant.

Síla pravidla drobných korekcí je však ještě větší: Účinně se eliminuje také riziko vnitřních potíží, jelikož korekce volantem neprovádíme jen podle vnějších vlivů (změny zadání), ale také podle vnitřních podmínek.

5.7.2.3 Hraní na výhru

Měli bychom hrát na výhru a ne jen tak, aby se neprohrálo. To je také jeden z momentů, proč některé projekty zkrachují - popíše se spousta papíru, uskuteční se spousta porad, každý dělá jen takové věci, aby mohl na konci prohlásit, že on vše udělal podle standardních procedur a krach tedy nemůže být jeho vina. Aby tým (v obecném slova smyslu) byl úspěšný v tvrdém boji, musí každý dělat vše, co týmu pomůže vyhrát a nedělat nic, co k výhře nepomůže. Samozřejmě zde zůstává odpovědnost za svou práci a rozhodnutí.

5.7.2.4 Práce v souladu s lidskými instinkty a nikoli proti nim

Pokud má být metodika nejen „laboratorně“ kvalitní, ale také prakticky přínosná, musí počítat s přirozeným chováním skupiny a jejích jedinců, pro které je určena. Vznesené hodnoty jsou jistě skvělým vodítkem, XP si však uvědomuje, že lidskou přirozeností je důraz na osobní zájmy. Je tedy třeba v metodice podpořit, aby osobní zájmy také sloužily dlouhodobým zájmům týmu.

I přes zdánlivou „divokou neorganizovanost“ bude ve většině firem, pro které je metodika XP určena, znamenat zvýšení disciplíny. Formálnější postupy totiž obvykle lidi natolik znechutí, že je neprovádějí z přesvědčení, že jsou přínosné, ale jen proto, „že to vedení chce“. Takové metodiky se pak často stávají jen pláštíkem formality zahalujícím divoké praktiky.

5.7.2.5 Cestování nalehko

Nemůžeme očekávat rychlý postup, když táhneme spousta zavazadel. Námi udržované nástroje by měly mít tyto vlastnosti:

²² Rozhodně se dá tvrdit, že změna je v XP běžnou událostí.

- malý počet,
- jednoduché,
- hodnotné.

Tým praktikující XP by měl být vždy připraven změnit směr, ať už dojde ke změně návrhu, možnosti používat lepší technologie nebo změně ve složení týmu.

5.8 Praxe XP

Nyní si vyjmenujeme a vysvětlíme oněch pověstných *dvanáct postupů*, ve kterých je obsažena celá praktická část XP. Kdykoli se hovoří o XP, diskutuje se právě (a bohužel většinou pouze) o těchto postupech.

Uvedme si nejdříve přehledně všech dvanáct postupů:

1. Plánovací hra (*The Planning Game*)
2. Malé verze (*Small Releases*)
3. Metafora (*System Metaphor*)
4. Jednoduchý návrh (*Simple Design*)
5. Testování (*Continuous Testing*)
6. Refaktorizace (*Refactoring*)
7. Párové programování (*Pair Programming*)
8. Společné vlastnictví (*Collective Code Ownership*)
9. Nepřetržitá integrace (*Continuous Integration*)
10. 40-ti hodinový týden (*40-Hour Work Week*, nověji *Sustainable Pace*)
11. Zákazník na pracovišti (*On-site Customer*)
12. Standardy pro psaní zdrojového textu (*Coding Standards*)

5.8.1 Plánovací hra

Vývoj softwaru v XP je vždy rozvíjející se dialog mezi možným a žádoucím. Zákazník a dodavatel spolupracují na zadání, aby zákazník mohl dostat co největší hodnotu v co nejkratším čase. Plánovací hra se skládá z těchto kroků:

1. Zákazník sepíše seznam požadovaných funkcí systému (obvykle funkčních celků) formou karet zadání (*User Story*²³). Každá karta nese určité jméno a je na ní v několika větách popsána požadovaná funkčnost.

²³ V praxi XP bývá častěji používána anglická terminologie, v textu se však budeme používat české výrazy.

2. Dodavatel zanalyzuje jednotlivé karty, odhadne náročnost (typicky člověkohodiny) a určí vazby mezi funkcemi (tj. kterou funkci je třeba mít hotovu pro implementaci jiné). Také odhadne, kolik je tým schopen vyprodukovat práce za jednu iteraci.
3. Zákazník poté určí pořadí, ve kterém se jednotlivé karty budou implementovat (tj. důležitost funkci).

Pro interní potřeby týmu je běžné složitější karty dekomponovat do tzv. úkolů (*Engineering Tasks*) a jednotlivé karty implementovat jako soubor úkolů. Vazby mezi kartami jsou potom implikovány vazbami mezi úkoly.

5.8.2 Malé verze

Rychle uvedeme minimální možný funkční systém do provozu a pak uvolňujeme malé verze ve velmi krátkých cyklech (iteracích). Všechny nové verze by měly obsahovat jen ta nejcennější zadání. Verze by však měla dávat smysl jako celek, tj. je třeba vždy implementovat celou kartu.

Délka iterace není nijak pevně stanovena a je třeba ji přizpůsobit objemnosti systému. Musí být nejkratší možná, ale dostatečně dlouhá na to, aby se s rezervou dala implementovat alespoň jedna karta. V praxi se délka iterace pohybuje typicky od jednoho týdne do jednoho, dvou měsíců.

5.8.3 Metafora

O systému hovoříme pomocí jednoduše sdíleného příběhu o tom, jak má celý systém fungovat. Počítač můžeme třeba připodobňovat k ploše stolu, výpočet důchodu k tabulce, atd. Metafora pomáhá všem v projektu pochopit základní prvky a vztahy mezi nimi. Příběh musí být snadno sdílen obchodně i technicky orientovanými lidmi. Metafora slouží k lepšímu dorozumění a snazšímu vytvoření mentálního modelu.

5.8.4 Jednoduchý návrh

Systém by měl být navržen co nejjednodušeji, jak je to v daný okamžik možné. Nepotřebnou komplikovanost odstraníme hned, jakmile je zjištěna. Požadavky se zítra mohou změnit, tak implementujeme jen to, co je nutné pro dnešní funkčnost.

Pro přívlastek „nejjednodušší“ v XP definujeme čtyři omezení v prioritním pořadí:

1. Systém (zdrojový text společně s testy) musí komunikovat se vším, s čím chceme, aby komunikoval (tj. kterou funkci je třeba mít hotovu pro implementaci jiné).
2. Systém nesmí obsahovat duplicitní zdrojový text (z bodů 1. a 2. se skládá pravidlo „jeden a pouze jeden“).
3. Systém by měl mít minimální počet tříd.
4. Systém by měl mít minimální počet metod.

Kdykoli přijdeme na to, že některá část systému porušuje tato pravidla, návrh opravíme. Permanentně mažeme vše, co nemá žádný účel buď komunikační nebo výpočetní. Viz též bod 5.8.6.

5.8.5 Testování

Testování je jedna z věcí, která je v XP dovedena do „extrému“. Existují dva typy testů:

Testy jednotek (*Unit Tests*) - tyto testy píše programátoři *před* implementací funkce. Jedná se o nezávislé a zautomatizované testy pro *všechny* části systému, které by se mohly porouchat. Není samozřejmě možné testovat úplně každou metodu v každé situaci, ale je třeba naučit se odhadovat, které testy by se mohly vyplatit. Klíčovou rolí zde hraje zkušenost. Všechny tyto testy musí probíhat na 100%.

Funkční testy (*Functional Tests n. Acceptance Tests*) - tyto testy píše zákazníci za pomoci testerů, a to pro každou kartu. Nemusí být vždy zautomatizované. Jedná se o testy, kterými si zákazník ověřuje, že systém pracuje tak, jak si představoval. Zpravidla není nutné, aby tyto testy z počátku běžely na 100%, s postupem času se procento úspěchu s opravami chyb a implementací dalších funkcí zlepšuje.

5.8.6 RefaktORIZACE

RefaktORIZACE je změna systému, jež nemění jeho chování, ale vylepšuje některé kvality přímo nesouvisející s funkčností systému, např. jednoduchost, pružnost, pochopitelnost, výkonnost. V XP se nejedná o doplňkovou činnost, ale je jedním z pilířů, na kterých XP stojí. K refaktORIZACI se uchylujeme, kdykoli zjistíme její potřebu. Na možnost refaktORIZACE bychom se měli zaměřit před i po přidání každé funkce do systému. Velké refaktORIZACE provádíme v souladu s principem přírůstkové změny (sekce 5.7.2.2) po malých krocích.

5.8.7 PÁROVÉ PROGRAMOVÁNÍ

Veškerý produkční kód (tj. kód, který je součástí dodávaného systému) píše společně dva programátoři sedící u jednoho počítače. Ten, který v daný okamžik třímá klávesnici a myš, píše kód a řeší aktuální problém. Druhý programátor kontroluje správnost kódu a přemýšlí o celkové koncepci. Důležité je, aby šlo o vyrovnaný dialog a ne jen o koukání přes rameno.

Pro tvorbu párů nejsou stanovena žádná pevná pravidla. Seskupení do párů je dynamické a může se i v průběhu dne měnit. Je vhodné si jako partnera vzít někoho, kdo má již nějaké zkušenosti s funkcí, kterou se chystáme implementovat, ale obvykle je tvorba párů náhodná. Samozřejmě, že s někým se nám pracuje lépe, s někým hůře, ale situace, kdy je obtížné páry tvořit svědčí o vážných problémech ve složení týmu.

5.8.8 Společné vlastnictví

Všichni mohou měnit jakoukoli část systému v jakoukoli dobu. Všichni jsou tedy obeznámeni se všemi částmi systému, ačkoli na různé úrovni. Jestliže pár pracuje a vidí příležitost ke zlepšení části zdrojového textu, začne jej zdokonalovat, pokud mu to usnadní život. Všichni tedy přijímají zodpovědnost za celý systém. Společné vlastnictví je zdánlivě šílenou ideou. V kapitole o souvislostech postupů XP si v sekci 5.9 si však vysvětlíme, že v XP tomu tak není.

Kód je třeba psát tak, aby z něj bylo zřejmé, čeho jsme chtěli docílit (tzv. *Intention-revealing*). Je proto vhodné používat jazyky co nejvyšší úrovně (z univerzálních jazyků nejlépe Smalltalk, Eiffel, atd.). Pokud jsme nuceni používat jiné jazyky, je třeba o to více náležitě komentovat a strukturovat.

5.8.9 Nepřetržitá integrace

Kdykoli je nějaký úkol dokončen, systém sestavujeme a integrujeme. Integraci typicky děláme i několikrát denně. Po integraci je každý zodpovědný za to, aby všechny testy jednotek běžely opět na 100%. Nepřetržitá integrace vyžaduje jistou režii navíc spojenou s odstraňováním kolizí, práce navíc však nebude nikdy mnoho, protože neustálá refaktorizace způsobí rozložení systému na mnoho malých objektů a metod, za kterými nestojí dlouhá práce. Nespornou výhodou je, že nesrovnalosti v chápání části systému odhalíme hned v zárodku.

5.8.10 40-ti hodinový týden

Není důležité, jestli týden bude mít 35 či 45 hodin, ale programátoři nesmí být trvale přepracovaní. Jinak se snižuje schopnost soustředění, což se negativně projeví na jejich práci. Když je třeba, lze krátkodobě dělat přesčasy, ale ne více jak jeden týden v kuse. Ukazuje se, že přílišná potřeba přesčasů je příznakem vážného problému v projektu, který se takto nevyřeší.

5.8.11 Zákazník na pracovišti

V týmu by v ideálním případě měl sedět skutečný zákazník, který je k dispozici na plný úvazek a odpovídá na otázky týmu, řeší spory a určuje priority v menším měřítku. Tímto zákazníkem by měl být jeden z budoucích uživatelů systému nebo alespoň někdo, kdo přesně zná potřeby skutečných uživatelů (*Customer Proxy*).

Zákazník na pracovišti vytváří hodnotu v projektu tím, že píše testy funkcionality. Značně také snižuje riziko projektu tím, že se nemusí plánovat tolik dopředu a nepíše se zdrojový text, aniž by se přesně vědělo, jakým testům má vyhovovat a které testy lze vynechat.

Pokud není fyzická přítomnost zákazníka možná, je třeba si alespoň zajistit nepřetržitou telefonickou dostupnost a časově flexibilní meetingy.

5.8.12 Standardy pro psaní zdrojového textu

Je třeba vytvořit standardy pro psaní zdrojového textu, které zlepši čitelnost kódu a komunikaci přes zdrojový kód. Tyto standardy musí všichni vývojáři bezpodmínečně dodržovat. V ideálním případě by nemělo být rozpoznatelné, kdo psal kterou část systému.

5.9 Souvislosti postupů XP

To, co činí XP mocným, jsou vzájemné úzké vazby mezi jednotlivými principy a pravidly. Všechna pravidla spolu navzájem souvisí a podporují se, což činí celou metodiku velmi stabilní. Pro maximální efektivitu všech postupů je vhodné praktikovat všechny postupy. Pokud z nějakého důvodu nechceme nebo nemůžeme některé pravidlo řádně přijmout, nedojde však ke zhroucení procesu. Pravdou také je, že i uváženým adoptováním pouze jednotlivého principu, či hodnoty můžeme zkvalitnit dosavadní proces vývoje softwaru.

Všechny principy jsou natolik provázané, že jednotlivá tvrzení není možné lineárně vysvětlit či dokonce dokázat. Teprve prostudování všech principů XP umožňuje pochopit, proč XP opravdu může fungovat a všechny postupy mají hluboké opodstatnění. V této kapitole si ukážeme proč ta či ona koncepce může fungovat.

5.9.1 Plánovací hra

Jak je možné začít s obecným neformálním zadáním?

- Zákazník je součástí týmu, takže zadání nemusí být na počátku úplné a v případě potřeby ho lze ihned upřesnit.
- Verze uvolňujeme v krátkých intervalech, takže jakékoli chyby plánu mají dopad nejvýše několik málo týdnů či měsíců.

5.9.2 Malé verze

Jak se může dařit vylepšovat software po malých kouskách? Díky neustálé integraci a testům máme vlastně stále k dispozici funkční systém!

5.9.3 Metafora

Jak může nepřesná metafora stačit pro komunikaci?

- Rychlá a konkrétní zpětná vazba ze skutečného zdrojového textu a testů naši metaforu upřesní.
- Zdrojový text neustále refaktorizujeme a tím zdokonalujeme naše chápání, co metafora znamená v praxi.

5.9.4 Jednoduchý návrh

Jak může stačit jednoduchý návrh bez přihlédnutí k budoucím potřebám? Většinou stačí a ušetřený čas můžeme využít pro případy, které si vyžádají více snahy. Zní to jako loterie a skutečně tomu tak je. Předpokládáme, že nám stačí udělat funkci triviálně a doufáme, že univerzálnější funkčnost nebudeme potřebovat. Těšíme se na to, že se požadavky stejně změní a funkci třeba zahodíme. Spoléháme, že pokud to „šťěstí“ mít nebudeme, dodatečnou funkčnost se nám podaří bez větších potíží dodělat, až bude potřeba. Jak to může fungovat? Především máme křivku nákladů na změny z obr. 38. To by však nestačilo, klíčovou roli zde hrají právě synergické vazby s dalšími principy.

5.9.5 Testování

5.9.5.1 Testy jednotek

Neustálé testování má nesporné klady. Nebude ale zabírat příliš mnoho času programátorům a snižovat tak efektivitu? Ne,

- Návrh děláme co nejjednodušší, takže i psaní testů není příliš obtížné.
- Programujeme s partnerem, takže i testy děláme společně.
- Testy nám jasně řeknou, kdy jsme hotovi a neztrácíme tedy čas ověřováním splnění požadavků.

- Je efektivnější otestovat funkci a chybu hned opravit, než se k tomu vracet po nějaké době.

5.9.5.2 Funkční testy

Proč zákazník má ztrácet čas a psát funkční testy? Zákazník sice vydá energii na napsání funkčních testů, v průběhu vývoje však potom má možnost bez další námahy přesně sledovat, nakolik systém vyhovuje jeho požadavkům.

5.9.6 Refaktorizace

Ani refaktorizací se v XP nemusíme bát.

- Díky společnému vlastnictví můžeme provádět změny kdykoli to potřebujeme.
- Díky standardům pro psaní zdrojového textu nemusíme před refaktorizací upravovat formát.
- Jednoduchý návrh refaktorizace usnadňuje.
- Máme testy, kterými po každé refaktorizaci ověříme funkčnost systému, takže jsme méně náchylní k porušení něčeho, aniž bychom o tom věděli.
- Díky párovému programování jsme méně náchylní k děláním chyb.
- Díky nepřetržité integraci během několika hodin zjistíme, kdybychom svou refaktorizací způsobili někomu jinému kolizi.

Díky jednoduchému návrhu nebudou ani refaktorizace příliš složité. A moderní nástroje, často integrované přímo do vývojových prostředí umožňují často dělat refaktorizace jedním stisknutím tlačítka.

5.9.7 Párové programování

Jedná se snad o nejdiskutovanější pravidlo XP. Jeho odpůrci se obávají, že párové programování je plýtvání lidskými zdroji. Ukazuje se však, že po určitém zácvičení v párovém programování je pár více než dvakrát produktivnější než jednotlivý programátor a produkuje výrazně kvalitnější kód!

Párové programování má ovšem jednu nepopíratelnou výhodu: Partneři ne navzájem hlídají, aby nesklouzli k programátorským neřestem. Lidé mají zvláště pod stresem tendenci vracet se ke svým zlozvykům a XP „šidit“.

Z hlediska ostatních postupů je důležité, že

- Standardy pro psaní zdrojového textu redukují malicherné handrkování.
- 40-hodinový týden omezuje nedorozumění a hádky z přepracování.
- Páry píšou společně testy, což jim dává možnosti srovnat svoje chápání věcí před tím, než se pustí do jejich implementace.
- Páry mají metaforu, na jejímž základě mohou rozhodovat o názvech a základním návrhu.
- Páry pracují v rámci jednoduchého návrhu, takže oba partneři chápou, o co jde.

5.9.8 Společné vlastnictví

Pojem společné vlastnictví kódu u softwarových inženýrů nutně vyvolává obraz satana, proti kterému bojují: džungle neorganizovaného chaosu, kdy se lidem pod rukama mění kód. V celém procesu XP je však společné vlastnictví nejen nezbytné, ale také bezpečné:

- Nepřetržitá integrace snižuje pravděpodobnost kolizí.
- Na druhou stranu ale každý implementuje svoji funkci mimo systém, takže pro realizaci jedné funkce (do příští integrace) se nemůže nikomu měnit kód pod rukama.
- Díky neustálé refaktorizaci se systém rozpadne na mnoho malých objektů, čímž se snižuje pravděpodobnost, že dva páry programátorů změní stejnou třídu či metodu ve stejné době.
- Pokud ke kolizi přeci dojde, bude úsilí nutné k urovnání změn malé, neboť každá z nich představuje jen malou část vývoje.

5.9.9 Nepřetržitá integrace

Argumenty proti nepřetržité integraci by mohly být

- nebezpečí vzniku kolizí
- nebezpečí poškození dosud vykonané práce

V XP jsou však tyto negativní rysy omezovány těmito pravidly:

- Můžeme rychle spouštět testy, takže víme, že jsme nic neporušili.
- Programováním v párech se zmenšuje série změn pro integraci na polovinu.
- Refaktorizací zdrojového textu vzniká více menších částí, což snižuje pravděpodobnost kolizí.

Je samozřejmé, že počáteční fáze vývoje jsou z hlediska kolizí problematictější. Je proto dobré si začátek naplánovat tradičním způsobem a záhy přejít na XP. XP je kouzelné v tom, že zatímco u běžných metodik narůstající složitost systému zhoršuje koordinaci, u XP je tomu naopak.

5.9.10 Zákazník na pracovišti

To, že tým má stále k dispozici někoho, kdo má představu, jak by měl budoucí systém vypadat, umožňuje vypustit náročné a nákladné postupy sběru dat a složité formální analýzy pro ujasnění zadání.

Pro zákazníka může být nepříjemné postrádat zapůjčeného pracovníka. Tento pracovník však zřejmě nebude vytížen po celou dobu a může se tak i věnovat své práci, což snižuje ekonomickou ztrátu pro zákazníka způsobenou jeho absencí. Zákazníkovi je třeba vysvětlit, že výhody, které pro něj z XP plynou (možnost změn zadání, zmenšení rizik, urychlení nasazení systému, atd.) daleko přesahují tuto ekonomickou ztrátu. Pokud však má zákazník pocit, že mu nový systém nepřinese hodnotu vyšší, než je práce jednoho člověka, měl by zvážit, zdali má vůbec cenu takový systém budovat.

5.10 Proces vývoje softwaru metodou XP

Nyní si shrneme, jak vypadá typický proces tvorby softwaru pod XP podle probraných postupů.

5.10.1 Fáze plánování

Zákazník projeví zájem o vybudování systému²⁴. Sejde se s týmem, kde společně vytvoří karty zadání. Tým zadání zanalyzuje - obvykle si jednotlivé karty ještě rozdělí na menší úkoly (*Engineering Tasks*), ohodnotí jednotlivé karty předpokládanou náročností a vytvoří relace prerekvizit.

Zákazník si prostuduje analýzu a může některé funkce (např. z důvodu přílišné finanční náročnosti) třeba zrušit nebo i odstoupit od celého projektu. Pokud se projekt bude realizovat, zákazník ohodnotí karty důležitostí pro něj, tj. určí pořadí, ve kterém (s ohledem na prerekvizity) budou jednotlivé funkce realizovány. Domluví se podrobnosti (způsob financování, délka iterací, velikost verzí, kdo bude dělat zákazníka na pracovišti, atd.) a může se začít s vývojem.

5.10.2 Fáze vývoje

V počáteční fázi vývoje se (ve spolupráci se zákazníkem na pracovišti) ještě dopřesní nejasnosti a provede se analýza pro první iteraci. Pak začne vlastní vývoj v párech, který brzy dostane typický pravidelný rytmus:

1. analýza úkolu
2. napsání testů
3. implementace úkolu
4. spuštění testů (+ev. oprava)
5. integrace
6. spuštění testů (+ev. oprava)

Kdykoli někdo zjistí, že lze něco v systému zjednodušit, provede refaktorizaci a opět ji ověří testy.

5.10.2.1 Komunikace

Všichni spolu hojně a otevřeně komunikují a při diskusích o systému používají metaforu. Každé ráno se udělá malá porada (tzv. „ve stoje“), kdy se všichni navzájem zběžně informují, na čem pracují a jaké problémy řeší. Poté se vytvoří páry, které také spolu intenzivně komunikují o řešení aktuálního problému. Všichni pak komunikují se zákazníkem na pracovišti, aby měli jasnou představu o všech vlastnostech systému. Otevřeně též komunikuje tým s vedením a navzájem se podporují. Na základě otevřené komunikace má vedení možnost v začátcích podchytit problémy a drobnými korekcemi vést tým správným směrem.

²⁴ Budeme předpokládat, že tento zákazník již ví, jak XP funguje (co se od něj jakožto integrální součásti vývoje očekává), nebo, vědom si všech výhod XP, je ochotný na XP přistoupit.

5.10.2.2 Psaní zdrojového textu

Programátoři samozřejmě píšou zdrojový kód, který je součástí výsledného produktu. Také však píšou kód „nanečisto“, pokud potřebují podložit abstraktní úvahy konkrétními experimenty. Zdrojový text slouží také jako prostředek komunikace a interní dokumentace. Všichni používají domluvené standardy pro psaní zdrojového textu, což eliminuje třenicové společenství a usnadňuje refaktorizaci.

5.10.2.3 Testování

Všichni často testují (viz výše). Jeden speciální tester je též k dispozici zákazníkovi pro psaní funkčních testů. Spouštíme všechny testy jednotek po každé refaktorizaci a změně systému.

5.10.3 Fáze nasazení a údržby

Jakmile je hotova první minimální verze, systém je nasazen do provozu. Po dokončení každé verze provede obvykle vybraný tester ještě tradičnější testy (např. na výkonnost) a systém je předán zákazníkovi k otestování funkčními testy. Případné problémy jsou - dle závažnosti - buď ihned řešeny nebo je jejich řešení odloženo do další verze. Nasazení systému tedy není v XP nějakým jedním zvláštním dnem. Systém je uveden do provozu co nejdříve a paralelně jsou vyvíjeny nové verze a opravovány nedostatky zjištěné provozem.

5.10.4 Možné obměny

Výše je popsán základní proces XP navržený Beckem v [Beck 2002]. Týmy praktikující XP ho obvykle používají lehce obměněný a doplněný. Je například možné ke každé kartě zadání přidělit osobu (obvykle jí bývá kouč - o kouči více viz 5.13.5), která je zodpovědná za ověření funkčnosti před dodáním a po dodání zajišťuje styk se zákazníkem v případě problémů.

5.11 Strategie řízení a organizace projektů XP

5.11.1 Přístup managementu

Ačkoli je XP značně demokraticky založeno, potřeba řízení zde zůstává. Nesmí ovšem jít o tvrdé vymáhání vlastních představ, ale vzájemný dialog. Optimální „střední cesta“ je opět charakterizována již představenými principy:

Komunikace - Manažer musí být schopen otevřeně komunikovat s týmem, ale především si musí umět získat důvěru, aby i tým otevřeně komunikoval s ním.

Přijatá odpovědnost - Úkolem manažera je zdůrazňovat to, co je nutné udělat, a nikoli zadávat práci.

Kvalitní práce - Mottem manažera by mělo být: „Snažím se jim pomáhat, aby mohli odvádět ještě lepší práci“ a ne „Pokouším se je přimět k tomu, aby odváděli slušnou práci“.

Přírůstková změna - Manažer poskytuje neustálé jemné vedení, nikoli jen velký manuál s pravidly na začátku.

Cestování nalehko - Manažer nesmí vyvolávat přílišnou režii, jako např. dlouhé porady, nepřiměřeně dlouhé výkazy, atd.

Proces řízení projektu XP je ovšem do značné míry decentralizovaný. Část řízení přebírají též role stopaře a kouče (viz sekce 5.13.4 a 5.13.5).

5.11.2 Přijetí odpovědnosti

Aby mohla jakákoli firma fungovat, musí existovat (co možná nejvíce) spravedlivý systém oceňování pracovníků. Pro ten jsou nezbytné podklady založené na odpovědnosti. I přes svou neformálnost XP poskytuje jasně definovaný a kontrolovatelný mechanismus odpovědnosti. Pro celý proces, především ve vztahu k motivovanosti a principu „hraní na výhru“ (sekce 5.7.2.3) je však důležité, aby odpovědnosti byly *přijímány* a ne „přidělovány“. Pak je už jen věcí správného oceňování a motivace, aby si lidé nebrali jen jednoduché a líbivé úkoly, ale měli chuť se poprat i s těmi nepříjemnými.

Co se týká podkladů pro personální hodnocení programátorů, i přes společné vlastnictví kódu zde není problém. Každý programátor totiž vždy přijímá zodpovědnost za určité úlohy (jakožto části funkčnosti systému). Porovnáním těchto zodpovědností např. s funkčními testy lze snadno získat podklady pro hodnocení. Není snad ovšem třeba zdůrazňovat, že jen takovéto mechanické hodnocení nestačí. Stejně jako jinde je i zde třeba přihlížet k okolnostem, schopnostem jedince formulovat a vyjádřit svůj názor, schopnosti dobré týmové spolupráce, atp.

5.11.3 Výuka poznatků

Žádná kvalitní metodika není soubor dogmat. Je tedy třeba získávat poznatky o jednotlivých principech a správně je sladit s konkrétním prostředím, lidmi, procesem, atd. Výrazně v tom může pomoci detailní studium metodiky, vždy jsou ale rozhodující poznatky z praktického nasazení. Toto je důležité si uvědomit především pro řídicí pracovníky, ale i ostatní členy týmu.

5.12 Metriky

XP je velmi neformální metodikou, ale i zde mají metriky své pevné místo. Slouží především jako zdroj zpětné vazby pro tým i vedení. Manažer by měl tyto metriky shromažďovat a aktualizovat. Podle potřeby by pár grafů mělo být viditelně vyvěšeno a pomáhat při vedení týmu. Používanými metrikami jsou:

- Časový harmonogram
- Počet testů v systému (pokud je třeba vést tým k většímu psaní testů)
- Výsledky testů funkcionality
- atd.

Metriky prezentujeme účelově, tj. musí být nějakou hodnotnou zpětnou vazbou, obsahovat nějaké sdělení týmu a motivovat.

5.13 Lidé okolo XP

Metodika XP vyžaduje úpravu náplně práce a rozdělení zodpovědností osob okolo projektu. Také zavádí některé speciální role.

5.13.1 Programátor

Je výkonnou silou týmu, který musí především umět dobře psát testy jednotek a programovat. Musí také umět refaktorizovat. Důležité je, aby byl schopen dobře komunikovat. V XP však programátor není manuálním dělníkem, ale přebírá značnou zodpovědnost za celý proces. Hraje vždy týmově a nikdy není „osamělým vlkem“.

5.13.2 Zákazník

Jak již víme, zákazník (tedy jeho zástupce) je také členem týmu. Zákazník píše zadání a s pomocí testera testy funkcionality. Musí tedy mít co nejlepší představu o nárocích na vyvíjený systém a musí být schopen dobře formulovat zadání. Komunikační schopnosti jsou samozřejmostí.

5.13.3 Tester

Jelikož většinu testovací odpovědnosti přebírají programátoři, tester je v týmu zaměřen ve skutečnosti na zákazníka. Je odpovědný za pomoc zákazníkovi při výběru a psaní testů funkcionality a jejich vyhodnocování. Stará se též o dobré fungování testovacích nástrojů.

V praxi obvykle tester provádí též „výstupní kontrolu“ po každé iteraci a může začlenit další tradiční testy (např. odolnost proti chybám uživatele, zátěžový test, atd.)

5.13.4 Stopař

Jedná se o novou roli zavedenou v XP, i když v každém běžném projektu obvykle tuto roli někdo vykonává. Stopaře bychom mohli nejuvýstižněji označit za „interního dokumentátora pro potřeby týmu“. Sbírá poznatky z vývoje, vede různé protokoly (např. o závadách) a má nad celým procesem určitý nadhled. Důležité ale je, že není jen pasivním pozorovatelem, ale pozitivně se snaží působit na tým, je jakýmsi „svědomím týmu“.

Role stopaře, stejně jako kouče, ovšem nevykonávají specializované osoby, ale jedná se obvykle o programátory nebo testery, kteří mají větší zkušenosti a určité organizační schopnosti.

5.13.5 Kouč

Kouč (*Coach*) je odpovědný za proces jako celek. Má hluboký vhled do celého procesu a jasně ho koriguje tím správným směrem. Někdy ale sáhne i k tvrdším zásahům, pokud jsou třeba. Důležitá je dovednost neříkat přímo, co vidí, ale říkat to tak, aby to spatřil také tým. Kouč tedy (stejně jako ve sportu) je určitou „otcovskou formou vedení“. Zvláště v raných fázích přechodu k XP je role kouče důležitá, časem by odpovědnost za proces měli co nejvíce přejímat programátoři sami. V běžné praxi bývá kouči přidělována též role odpovědné osoby například za konkrétní kartu zadání.

5.13.6 Konzultant

Jelikož se členové týmu často přesouvají a párují a spravují kolektivní kód, není možné si pěstovat specializace. Pokud tedy někdy dojde k potřebě nějaké hluboké technické znalosti, tým osloví experta na daný problém. I tento konzultant však je „vtažen“ do XP a jeho konzultace se řídí principy XP. Tým tedy vytvoří testy pro předkládaný problém a bude se chtít postup naučit, protože ho může potřebovat v budoucnu. Také bude zpochybňovat práci konzultanta - hledat jednodušší řešení. A nakonec si to vše přepíše podle svých standardů. Konzultant neznalý principů XP by se mohl cítit dotčený, ale pokud jeho cílem je opravdu týmu pomoci a ne jen za tučný peníz dávat na odiv svoji genialitu a výjimečnost, musí tento postup respektovat.

5.14 Nasazení XP

XP je vynikající metodika, pokud by však byla používána nevhodně, nadělala by více škody než užítku. V této kapitole si tedy probereme podmínky pro úspěšné nasazení XP a omezení této metodiky.

5.14.1 Omezení XP

5.14.1.1 Velikost týmu a rozsah projektů

XP je určeno pro malé týmy do deseti programátorů. Ve větším počtu je problém s narůstající komunikací a společným vlastnictvím. Omezení na velikost týmu také omezuje velikost projektů.

I velké softwarové firmy pracující na mamutích projektech však mohou výhody XP využít. Zatímco celý projekt je řízen tradičně, některé jeho části jsou řešeny v XP. Jedná se především o ty části, které svým charakterem odpovídají sekcím 5.14.1.3 a 5.14.1.4.

5.14.1.2 Náklady na změnu

Aby XP mohlo fungovat, nutným předpokladem jsou náklady na změnu podle grafu na obr. 38 na straně 94. Tuto problematiku jsme probírali v sekci 5.6.

5.14.1.3 Charakter projektů

Výhody XP se nejvíce projeví u projektů, kde se v průběhu vývoje často mění specifikace. Projekty vhodné pro XP však musí vyhovovat v těchto bodech:

- Velikost projektu odpovídá velikosti týmu (sekce 5.14.1.1).
- Přiměřený rozsah a odborná náročnost problému (v XP všichni musí znát všechno).
- Nesmí být v rozporu s některými principy XP. U některých projektů může být třeba problém s nemožností získávání rychlé zpětné vazby, svázání určitými nutnými formálními postupy (např. ISO9001) nebo nemožnost automaticky a rychle testovat software (např. u programů pro řízení technologických zařízení).

- XP není vhodné (přesněji řečeno je zbytečné) pro projekty, kde je nutné před započítím provést důkladnou analýzu a té se striktně držet (např. některé vládní či armádní zakázky).
- Systém musí být granulovatelný, aby ho bylo možné rozdělit na karty zadání a dodávat přírůstkově.
- Systém musí mít objektové rysy. XP není vhodné například pro projekty zaměřené na datové modelování, jako jsou Data Warehouse.

Pomocí XP je možné vyvíjet nejen zakázkové systémy, ale i tzv. „krabicové produkty“. Roli zákazníka potom přebírá marketingové oddělení, které identifikuje zadání, které chce trh, kolik z každého zadání je zapotřebí, v jakém pořadí by měla být zadání implementována. atd.

5.14.1.4 Charakter zákazníků

O zákazníkovi jsme hovořili v sekci 5.13.2 v souvislosti rolí v XP. Aby projekt mohl běžet v XP, musí zákazník být ochoten tuto hru hrát. Pokud toto nelze, nemá smysl o XP uvažovat. Proto také XP není příliš vhodné pro firmy, kde zákazníkem je pokaždé někdo jiný. Naopak velmi dobré je, pokud máme stále zákazníky „zkušené v XP“.

5.14.1.5 Tým

Týmu není možné XP nutit, to by nefungovalo. Všichni (nebo alespoň naprostá většina) musí opravdu chtít vyvíjet metodou XP a aktivně pěstovat potřebné dovednosti a kvality. Přijetí XP neznamená ani tak změnu metodiky, jako změnu přístupu a práci na sobě.

5.14.1.6 Prostředí

Pro úspěšné provozování XP je důležité mít i vhodné pracovní prostředí. Je nutné, aby členové týmu seděli v jedné místnosti a aby u všech počítačů bylo možné pohodlně programovat v párech. Podrobnější tipy na uspořádání pracoviště lze nalézt třeba v [Beck 2002] na straně 69-71.

5.14.2 Zavádění XP

5.14.2.1 Přizpůsobení metodiky

Jak už bylo na několika místech textu zmíněno, XP nelze používat mechanicky v jeho čisté podobě, ale je třeba ho citlivě přizpůsobit konkrétnímu prostředí, lidem, zaměření firmy. To je asi nejtěžší, neboť nesmíme sklouznout k přizpůsobování XP na základě strachu nebo pohodlnosti. Při zavádění XP bychom proto měli svá rozhodnutí konzultovat s těmi, kdo s XP již mají nějaké zkušenosti.

5.14.2.2 Přejít na XP

Jelikož je XP mladá metodika, v naprosté většině případů dnes týmy k XP budou přecházet od svých dřívějších postupů. Přejít k XP musí být postupný a uvážený. Pokud potřebujeme nějaký starý kód, není vhodné ho celý převádět do XP najednou. Paralelně s vývojem nového kódu v XP

začneme postupně převádět starý kód, a to - opět dle principů XP - od těch nejdůležitějších (nejčastěji používaných) částí. Postupné musí být též zavádění návrhu, plánování a řízení. Je evidentní, že přechod k XP dočasně sníží produktivitu, ale to je nutná daň. Důležitý je dobrý pocit, který by měl všechny změny doprovázet.

5.15 Porovnání XP a rigorózních metodik

Po probrání problematiky XP a agilního přístupu se nyní můžeme provést detailnější srovnání s rigorózními metodikami, které bylo nastíněno v kapitole 3.7. Porovnání provedeme z hlediska těchto aspektů: Stručně charakterizujeme proces, omezení, rozsah použití, dokumentovatelnost procesu a požadavky a využití možností vývojových nástrojů.

5.15.1 Rigorózní metodiky

5.15.1.1 Proces

Rigorózní metodiky se vyznačují těmito kroky:

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

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

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

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

5.15.1.2 Omezení

Klasický model z důvodů zmíněných v předcházejícím odstavci neumožňuje snadno provádět změny zadání v průběhu realizace projektu. Vývojové firmy se proto snaží předejít v maximální možné míře této situaci, typicky uzavíráním co nejzávaznějších a nejpresnějších smluv se

zákazníkem. Je vyžadováno, aby zákazník měl na počátku velmi přesnou představu o funkcích a vzhledu výsledného systému. Pro zákazníka je však toto často velmi problematický požadavek. Nejedná se jen o problém komunikace a přesné specifikace aktuální představy, ale může též dojít během doby vývoje softwaru ke změně situace u zákazníka, se kterou na počátku nepočítal. Při klasickém modelu je doba dodání systému, i přes používání moderních nástrojů, relativně dlouhá, toto riziko je tedy nezanedbatelné. Zvláště významné je potom u dynamicky se rozvíjejících firem.

5.15.1.3 Rozsah použití

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

5.15.1.4 Dokumentovatelnost procesu

Proces řízení je možné předem plánovat, je možné vytvářet formální dokumentaci, lze zahrnovat různé normy na provádění analýzy, dokumentaci, testování, atd.

5.15.1.5 Požadavky na vývojové nástroje

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

Na druhou stranu, klasické metodiky nevyužívají dostatečně potenciál, který nabízí moderní vývojová prostředí a přidružené technologie: vývoj systému za běhu, automatizované testování, správa společného kódu, atd. Přesněji řečeno, využívají ho jen v „malém“, tj. v rámci vývoje kusů softwaru, nikoliv však pro zlepšení celého procesu (pro bližší pochopení této teze viz následující sekce).

5.15.2 Metodika extrémní programování

5.15.2.1 Proces

Proces vývoje je orientován na zákazníka. Metodika akceptuje, že zákazník nemá na začátku přesnou představu o budoucím softwarovém produktu (systému) a teprve v průběhu vývoje se představa upřesní. Dodání systému probíhá v těchto krocích:

1. Neformální zadání sestávající ze zadání funkčních celků (*User Stories*).
2. Analýza funkčních celků.
3. Seřazení funkčních celků podle jejich hodnoty pro zákazníka (*business value*).
4. Dodávání systému po jednotlivých funkčních celcích.

V XP je tedy co nejrychleji dodána minimální funkční verze systému obsahující nejdůležitější funkce a v každé další verzi je dodán další funkční celek. Uvolňování verzí probíhá ve srovnání s klasickým přístupem mnohem častěji (týdny, max. několik měsíců). Na základě zkušeností z provozu je potom upravován obsah dalších verzí. Nové verze neobsahují jen nové funkce, ale i požadované úpravy většího rozsahu funkcí hotových. Tento přístup je podobný tradiční technice prototypování, rozdíl je ale v tom, že není připravován prototyp, který se poté zahodí, ale zákazník přímo pracuje s narůstající částí hotového systému.

Je třeba si uvědomit rozdíl mezi dekompozicí v klasické metodice a funkčními celky v XP. V XP jsou funkční celky postupně **dodávány**, zatímco dekompozice v klasickém přístupu je prováděna za účelem oddělené **implementace** (více programátory). (XP ve skutečnosti také obsahuje tento typ dekompozice: *User Stories* jsou dekomponovány na *Engineering Tasks*).

Jelikož nejsou na počátku přesně známy požadavky na budoucí systém, není před zahájením implementace vypracována podrobná analýza celého systému. Místo toho jak ve fázi počáteční analýzy, tak v průběhu celého vývoje zákazník velmi úzce spolupracuje s vývojovým týmem.

5.15.2.2 Omezení

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

5.15.2.3 Rozsah použití

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

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

5.15.2.4 Dokumentovatelnost procesu

XP je značně neformální a v jeho průběhu se nepoužívají žádné manuálně vytvářené dokumenty, protože díky využívání vlastností moderních vývojových prostředí nejsou tak potřebné. Pokud je z formálních důvodů nějaká další dokumentace vyžadována, je třeba ji vypracovávat navíc, mimo proces XP. V současné době není též zpracováno zařazení XP do formálních a standardizovaných řídicích procesů.

5.15.2.5 Požadavky na vývojové nástroje

XP plně využívá (a tedy i vyžaduje) moderní objektově orientované programovací jazyky spolu s pokročilými technologiemi vývojových prostředí. Typickým příkladem je plně objektový jazyk Smalltalk-84 a prostředí Cincom VisualWorks s integrovanou podporou pro automatické testování, refaktorizaci, správu společného kódu, tvorbu dokumentace, atd.

5.15.3 Diskuse

Ze srovnání je patrné, že extrémní programování lépe využívá možností moderních objektových programovacích jazyků a vývojových nástrojů k prolomení některých závažných omezení klasických metodik, především nepružnosti na změny zadání a znovupoužitelnosti výsledných systémů a jejich částí. Na druhou stranu je však omezen rozsah jeho použití, a to především rozsahem projektů a jejich charakterem. V XP je též problematičtější provádění postupů zajišťování jakosti a dokumentace v souladu s formálními předpisy a je náročnější na kvalitu týmu. Hlavní předností XP je jeho tolerance k nepřesnému a neúplnému zadání a schopnost pružně reagovat na změny zadání v průběhu vývoje. Pravdou je, že XP je v praxi v řadě případů úspěšně používáno, není však obecně nahrazením klasického přístupu a nesnaží se ho ani popírat.

6 UML

Modelovací jazyk UML (Unified Modeling Language, unifikovaný modelovací jazyk) byl vytvořen jako univerzální standard pro vizuální modelování systémů. Přestože je nejčastěji spokojován s modelováním objektově orientovaných softwarových systémů, tak má mnohem širší využití, což je možno pomocí zabudovaných rozšiřovacích mechanismů.

Jazyk UML byl navržen proto, aby spojil nejlepší existující postupy modelovacích technik a softwarového inženýrství. jako takový je explicitně navržen takovým způsobem, aby jej mohly implementovat nástroje CASE (Computer Aided Software Engineering). Tato koncepce vychází ze skutečnosti, že rozsáhlé projekty navrhující informační systémy se bez CASE nástrojů neobejdou. Takto vytvořené diagramy jsou srozumitelné jak pro lidi, ale také jdou následně automaticky zpracovávat (např. generovat kód).

Je důležité si uvědomit, že definice jazyka UML neobsahuje žádný druh metodiky modelování. Přirozeně, že určité aspekty metodiky můžeme najít v každém z elementů, z nichž se UML skládá. Samotný jazyk UML však poskytuje pouze vizuální syntaxi, kterou můžeme využít při vytváření svých modelů.

Jazyk UML se nazývá unifikovaný. To vyplývá jednak z historie objektového modelování, ale také z toho, že se UML snaží o unifikaci pojmů a přístupů z různých domén (viz [OMG 2003]). UML se snaží o unifikaci těchto domén:

- **Vývojový cyklus.** Jazyk UML nabízí vizuální syntaxi pro modelování celého vývojového cyklu – od požadavků na analýzu, přes implementaci, až po nasazení jednotlivých komponent.
- **Aplikační domény.** Jazyk UML byl vytvořen pro modelování problémů z různých aplikačních domén. Od systémů reálného času, až po systémy pro rozhodování.
- **Implementační jazyky a platformy.** Jazyk UML je nezávislý na jakémkoliv programovacím jazyce a jakékoliv platformě. Lze ho používat pro modelování systémů implementovaných v čistě objektově orientovaných jazycích (Smalltalk, Java), lze ho i použít pro hybridní objektové jazyky (C++), tak i pro neobjektové.
- **Vývojové procesy.** Jazyk UML je nezávislý na použité metodě návrhu a vývoje informačního systému.
- **Vlastní interní pojmy.** Jazyk UML se snaží o vnitřní jednotu a konzistenci pomocí malé množiny interních pojmů. Je pravda, že v této oblasti není úplně dokonalý.

6.1 Vznik UML

Objektově orientované jazyky se rozvíjejí už od 70. let (Smalltalk). Jejich hlavní rozšíření začalo ale až v druhé polovině 80. let spolu s jazykem C++ a s rozšířením GUI (Graphics User Interface). V této době začalo být jasné, že se při návrhu objektově orientovaných aplikací neobejdeme bez vhodných metod analýzy a návrhu. Protože byl trh neobsazený a existovaly velká poptávka po objektově orientovaných metodikách, tak se na přelomu 80. a 90. let vzniklo mnoho prací zabývajících analýzou a návrhem objektově orientovaných aplikací. Klíčové základní práce o objektové analýze a návrhu je objevily mezi roky 1988 až 1992. Byly to knihy a autoři:

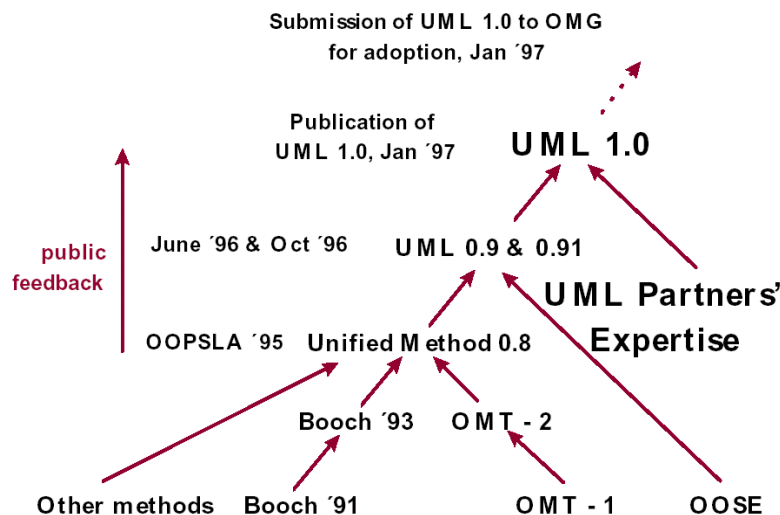
- Sally Shlaer a Steve Mellor napsali dvojici knih ([Shlaer 1989] a [Shlaer 1991]) o analýze a designu. Jejich přístup se vyvinul v jejich metodu Recursive Design approach (1997).

- Peter Coad a Ed Yourdon vyvinuli z Coadovu na prototypování orientovaného přístupu metodu a napsali o ní knihy [Yourdon 2001a], [Yourdon2001b], [Coad 1993] a [Coad et al. 1995].
- Smalltalkovská komunita v Portlandu přišla se svým odpovědností řízeným návrhem (Responsibility-Driven Design – viz. [Wirfs-Brock 1990]) a CRC kartami ([Beck 1989]).
- Grady Booch vycházel ze své práce ve firmě Rational se systémy založenými na jazyce Ada a napsal několik knih o své metodě – viz. [Booch 1994] a [Booch 1996].
- Jim Rumbaugh vedl tým v laboratořích General Electric a přišel se svou metodikou OMT (Object Modeling Technigue) – viz [Rumbaugh et al. 1991] a [Rumbaugh 1996].
- Jim Odell a James Martin napsali knihy na základě svých dlouholetých zkušeností s návrhem business informačních systémů a informačním inženýrstvím – viz [Martin 1994].
- Ivar Jacobson napsal své knihy na základě jeho zkušeností s programováním telefonních přepínačů u firmy Ericsson. Ve svých pracích zavedl koncept užitého případu – více [Jacobson et al. 1992]..

V roce 1994 na konferenci OOPSLA se objevily první návrhy na sjednocení v té době používaných metodik. Zejména sdružení OMG²⁵ mělo zájem na standardizaci používaných objektových metodik. To bylo potřeba, protože téměř každá z výše uvedených metodik měla svou vlastní notaci. Toto snažení však bylo v té době neúspěšné. Na této konferenci (OOPSLA 94) oznámily pánové Booch a Rumbought pokus o sjednocení svých metodik (a vstoupení pana Rumbaughta do firmy Rational). Jejich nová „sloučená“ by měla ty nejlepší podmínky k definování standardu silou, protože jejich metodiky měli dohromady obsazenou více než polovinu trhu. Dalším pokusem o sjednocení objektových metodik byla metodika Fussion ([Coleman et al. 1994]). Ta se ovšem nerozšířila.

K dalšímu pokroku došlo za rok na konferenci OOPSLA 95. Tam Booch a Rumbought představili první návrh své unifikované metody – verzi 0.8. Dál oznámili, že firma Rational koupila Ivara Jacobsona s jeho metodikou Objectory. Následující rok (1996) firma Rational změnila jméno svého návrhu na UML a předala svůj návrh konsorciu OMG, které ho začalo vyvíjet. První verze jazyku UML (tj. 1.0) OMG vydalo v lednu 1997. Zdroje vzniku a počáteční vývoj jazyku UML je znázorněna na následujícím obrázku. V současné době (červen 2005) je aktuální verze 1.5 a verze 2.0 je připravená k přijetí.

²⁵ OMG = The Object Management Group. Mezinárodní sdružení firem, vysokých škol a dalších institucí, které se zabývá standardizací OOP, podporou aplikovaného výzkumu a jeho uplatněním v praxi. OMG vzniklo původně za účelem definice standardu pro distribuované objekty CORBA (Common Object Request Broker Architecture) a následně se začalo věnovat standardizaci na poli objektů. (<http://www.omg.org>)



obr. 39. vývoj UML

6.2 Vlastnosti UML

Základní teze jazyka UML uvedené v [OMG 2003] jsou:

Abstrakce, zaměření se na relevantní detaily při současném ignorování jiných vlastností, je klíč k poznávání a komunikaci. Důvody pro to jsou následující. Každý komplexní systém je nejlépe uchopitelný pomocí malé skupiny relativně nezávislých pohledů. Žádný jednotný pohled není dostatečný. Každý model by měl být vyjádřen na různé úrovni výstižnosti. Nejlepší modely mají přímé spojení s realitou.

Základní cíle UML podle [OMG 2003]:

- Poskytnout uživatelům použitelný vizuální modelovací jazyk pro vytváření a výměnu smysluplných modelů.
- Poskytnout mechanismy rozšiřitelnosti a specializace pro rozšíření základních konceptů.
- Vytvořit specifikaci nezávislou na konkrétních programovacích jazycích a procesech vývoje a analýzy.
- Poskytnout formální základ pro pochopení modelovacího jazyka.
- Podporovat rozvoj objektových nástrojů.
- Podporovat vývojové koncepty jako jsou komponenty (components), spolupráce (collaborations), pracovní rámce (frameworks) a vzory (patterns).
- Zahrnout nejlepší postupy (best practice).

UML, pokud to jde, se snaží používat grafickou již zažitou syntaxi z různých zdrojů (metod) a nesnaží se ji měnit. Důležité je tedy skloubení syntaxe. UML přináší pouze několik nových konceptů. Je však nutné si uvědomit, že mnoho myšlenek, obsažených v těchto konceptech, bylo obsaženo v jiných individuálních metodách a teoriích. UML se tedy snaží tyto myšlenky uspořádat do soudržné struktury.

Podle [OMG 2003] UML zahrnuje tyto nové koncepty:

- Mechanismy rozšíření (stereotypy, omezení, pojmenované hodnoty),
- vlákna a procesy,

- distribuce a souběžnost,
- vzory, spolupráce,
- diagram aktivit pro modelování obchodních procesů,
- rozhraní a komponenty,
- jazyk pro omezení (OCL – Object Constraint Language).

Jednou z nejdůležitějších charakteristik UML je jeho nezávislost na metodologiích. Možná i z toho pramení jeho široké rozšíření jakožto implementačního jazyka. UML jako prostředek na zachycení popisu prostředí je nezávislý na metodologii používané pro správné provedení analýzy a designu.

6.3 Definice UML

Definicí jazyka UML lze rozdělit na 4 části (více [Rumbaugh et al. 1998]):

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

Notace UML vyjadřuje jeho syntaxi. Při tvorbě modelů pomocí jazyka UML postupně pomocí digramů vyjadřujeme jednotlivé stránky systému (statickou, dynamickou, funkční). Pro zachycení těchto stránek modelu je v UML (do verze 1.5) osm respektive devět²⁶ typů diagramů. Jazyk UML ve své podstatě umožňuje vytváření nových typů diagramů kombinací těchto devíti základních druhů, ale toto je proti smyslu unifikace. Jejím smyslem je, aby stejné pohledy na model systému bylo možno vyjádřit tak, aby tyto pohledy byly všeobecně srozumitelné.

Diagramy UML sestavené podle pravidel syntaxe musí mít pevně definovaný význam. To je úkolem sémantiky UML. Tato sémantika je vyjádřena metamodelem UML (více v části odkaz).

Ne všechny vlastnosti lze vyjádřit pomocí diagramů. Ke každému prvku UML můžeme přidat omezení, které přesněji definuje vlastnosti prvků. Pro definici těchto omezení definice jazyka UML obsahuje jazyk OCL (Object Constraint Language). Více o tomto jazyku se lze dočíst v části odkaz nebo přímo v definici UML (viz. [OMG 2003]) a nebo v [Warmer 1998].

Součástí specifikace UML jsou také definice výměnných formátů dat. Pomocí nich můžeme přenést diagramy vytvořené pomocí UML do jiných modelovacích nástrojů – pro to slouží formát XMI (XML Metadata Interchange), nebo pro převod rozhraní tříd do CORBA IDL (Interface Definition Language).

6.4 Struktura jazyka UML

Dále si popíšeme strukturu jazyka UML (více [Rumbaugh et al. 1998]). Ta se skládá z těchto součástí:

- **Stavební bloky** – jsou to základní prvky modelu, relace a diagramy.

²⁶ Podle toho, zda objektový diagram je počítán za samostatný typ.

- **Společné mechanismy** – jsou to obecné způsoby, jimiž v jazyce UML dosáhneme specifických cílů.
- **Architektura** – pohled v jazyce UML na architekturu navrhovaného systému.

6.4.1 Stavební bloky jazyka UML

Podle [Rumbaugh et al. 1998] je jazyk UML sestaven z pouhých třech stavebních bloků:

- **Z předmětů** (things), to jsou samotné elementy modelu,
- **vztahů** (relationships), ty určují vzájemné vztahy mezi předměty (určují, jak spolu dva, nebo více předmětů souvisí),
- **diagramů**, to jsou pohledy na model UML. Diagramy ukazují kolekce předmětů (a vztahů mezi nimi) a vizualizují co bude systém dělat (analytický pohled) a jak to bude dělat (návrhové diagramy).

6.4.2 Předměty (things)

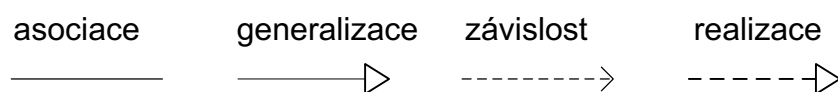
Předměty (rovněž věci nebo abstrakce) dělíme v UML na:

- **Strukturní abstrakce** (structural things), jsou to základní stavební prvky modelu UML, jako jsou třídy, rozhraní, případ užití, komponenta;
- **chování** (behavioural things), jež vyjadřují děje v jazyce UML, například interakce, stav;
- **seskupení** (grouping things), to jsou balíčky (package), které slouží k seskupování významově souvisejících prvků modelu do skupin;
- **poznámky** (annotational things), poznámky, které lze ke každému prvku připojit.

6.4.3 Vztahy (relationships)

Relace (vztahy) v modelu UML umožňují zachytit vzájemné vztahy mezi předměty modelu. Relace zachycují významový (sémantický) vztah mezi předměty. Jazyk UML rozlišuje vztahy:

- **Asociace** (Association) – popisuje spojení mezi předměty.
- **Závislost** (Dependency) – znázorňuje to, že změna v určitém předmětu ovlivňuje význam závislého předmětu.
- **Zobecnění** (Generalization) – jeden element je specializací druhého elementu (a naopak druhý element je generalizací prvního). Pomocí této vazby se vyjadřují ISA hierarchie.
- **Realizace** (Realization) – je to vazba mezi klasifikátory, kdy jeden klasifikátor určuje dohodu, jejíž uskutečnění zaručuje druhý klasifikátor.



obr. 40. Vztahy

6.4.4 Diagramy

Modely v jazyce UML jsou především vyjádřeny pomocí diagramů²⁷. Ale pozor! Diagram není model. Diagram je pouze okno či pohled na část tohoto modelu. Každý nový prvek přidaný do diagramu je automaticky přidáván i do modelu, ale při rušení prvku z diagramu prvek v modelu zůstává – to je proto, že se jeden prvek může vyskytovat v mnoha diagramech. Model je tedy něco jako repositář prvků a znalostí o nich.

Do verze UML 1.5 je v definici [OMG 2003] zahrnuto 9 typů diagramů, rozdělených do dvou skupin podle toho, jestli zachycují dynamickou nebo statickou podstatu systému. Statický model zachycuje předměty a strukturní relace mezi předměty. Dynamický model naproti tomu zachycuje způsob, jakým na sebe předměty navzájem působí, aby bylo dosaženo požadovaného chování systému.

Statický model systému lze zachytit pomocí diagramů:

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

Pro zachycení dynamického chování systému lze použít diagramy:

- diagram případů užití
- sekvenční diagram
- diagram spolupráce
- stavový diagram
- diagram aktivit.

6.5 Společné mechanismy jazyka UML

Jazyk UML obsahuje čtyři společné mechanismy, které jsou používány konzistentně v celém jazyku. Popisují čtyři strategie používané v modelování objektů, jež jsou opakovaně používány v různých kontextech jazyka UML. Jsou to specifikace, ozdoby, podskupiny a mechanismy rozšiřitelnosti.

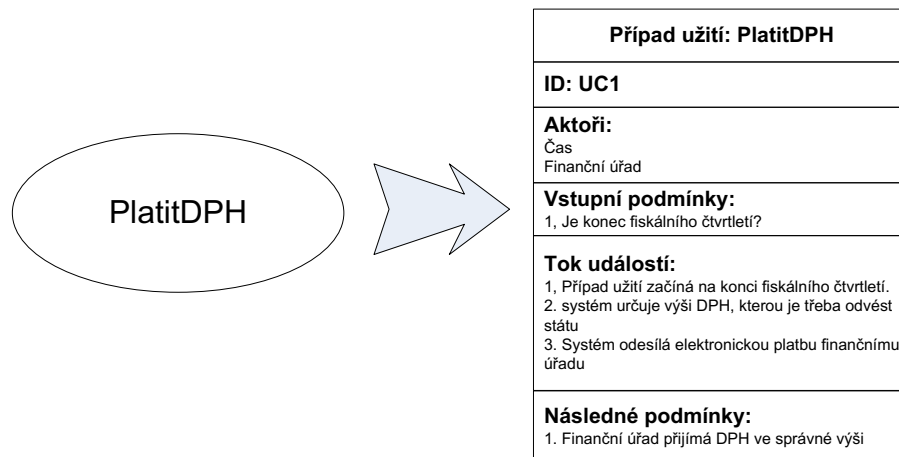
6.5.1 Specifikace

Modely UML mají alespoň dva rozměry – grafický, pomocí kterého lze vizualizovat model pomocí diagramů a symbolů (ikon), a textový, pomocí něhož specifikujeme jednotlivé elementy modelu. Specifikace jsou textovým popisem jednotlivých elementů modelu.

Pro ilustraci zvolme příklad (viz obrázek 41). Jednotlivý případ užití (use case) má grafickou podobu vyjádřenu pomocí oválu se jeho jménem uvnitř. Jeho podrobnou specifikací může být seznam s těmito prvky: jeho jménem, s účastníci se aktory, vstupními podmínkami, tokem událostí

²⁷ Ale ne jenom diagramy, potřebujeme například znát význam jednotlivých prvků v diagramu (sémantiku). K tomu slouží specifikace prvků – sémantický základ.

(záznamem procesu případu užití) a následnými podmínkami, které platí po skončení děje případu užití.



obr. 41. Specifikace případu užití

Množina specifikací je jádrem modelu. Je uložena v repositáři modelu. Tato množina specifikací vytváří sémantický základ modelu, který udržuje celý model pohromadě a dává mu smysl. Různé digramy jsou různými pohledy nebo obrazovými znázorněními do tohoto modelu.

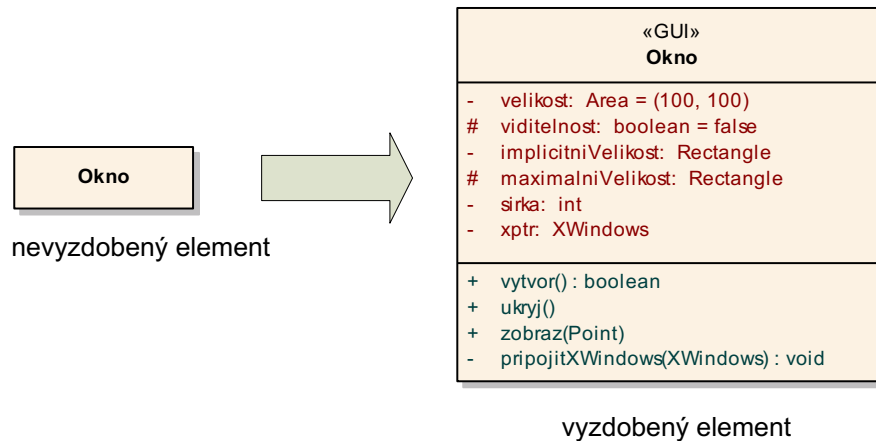
Jazyk UML je velmi pružný a flexibilní při tvorbě modelu. V praxi modely mohou být:

- zjednodušené, což znamená, že jednotlivé prvky jsou v repositáři modelu sice obsaženy, ale v daném diagramu nejsou znázorněny – typicky pro zjednodušení diagramu,
- neúplné – určité elementy mohou v modelu chybět (například proto, že dosud nejsou namodelovány, nebo daná oblast nás nezajímá),
- nekonzistentní – model dokonce může obsahovat protimluvy.

Existence pravidel o neúplnosti a nekonzistenci modelu je důležitá zejména pro postupnou evoluci modelů, kdy je model postupně vyvíjen a prodělává noho změn. Ovšem výsledné modely mají být konzistentní a natolik úplné, aby umožňovali implementaci modelu do podoby softwarového produktu. Typický postup při tvorbě modelu je, že se začíná tvorbou grafického modelu, k němuž se následně přidává sémantika.

6.5.2 Ozdoby (Adornments)

Jednou z vlastností jazyka UML je, že elementy modelu jsou vyjádřeny jednoduchým symbolem, ke kterému lze přidávat mnoho „ozdob“ a tím ho obohacovat o doplňující informace. Příklad pro třídu je na obrázku 42.



obr. 42. Ozdobený element

Obvyklý postup je, že nejdříve prvky modelu mají pouze několik ozdob, a při posupném upřesňování modelu přidáváme další podrobnosti. Tyto další podrobnosti (tj. ozdoby) má cenu k modelu přidávat pouze v tom případě, když zvyšujeme čitelnost, nebo srozumitelnost diagramu, nebo v případě když zdůrazňujeme nějakou důležitou funkci modelu.

6.5.3 Podskupiny

Podskupiny (common divisions) popisují různé způsoby pohledu na prvky systému. V UML známe dva druhy takového pohledu:

- klasifikátory a instance,
- rozhraní a implementace.

6.5.3.1 Klasifikátor a instance

Jazyk UML předpokládá, že můžeme mít abstraktní představu o typu předmětu, ale i představu o konkrétní instanci této abstrakce. Abstraktní představa o předmětu je klasifikátor a konkrétní představy o předmětu jsou jeho instance. Asi nejlepší ilustrací klasifikátoru a instance je Třída a objekt. Třída je klasifikátor a jeho instancí je objekt.

V jazyce UML je instance obvykle znázorněna stejným symbolem jako odpovídající klasifikátor. Názvy instancí jsou na symbolu podtrženy.

Jazyk UML definuje tyto klasifikátory:

- Aktor – představuje roli vnějšího uživatele systému.
- Třída – obsahuje popis množiny objektů sdílejících stejné vlastnosti.
- Role klasifikátoru – je klasifikátor omezený na určitou roli v interakci.
- Komponenta – je fyzická a vyměnitelná součást systému, která odpovídá jednomu nebo více rozhraním, která implementuje.
- Datový typ – je typ, jehož hodnoty nemají vlastní identitu. Například v jazycích C++ a Java používané typy int, float a char.

- Rozhraní – je kolekce operací používané k určení služby poskytované třídou nebo komponentou.
- Uzel – představuje fyzický element obsahující spustitelný kód, který zastupuje výpočetní prostředek.
- Signál – je asynchronní zpráva předávaná mezi objekty.
- Subsystem – je seskupení elementů, z nichž je možno specifikovat chování poskytované obsaženými elementy.
- Příklad užití – je popisem posloupností činností, které systém volá pro uspokojení potřeb uživatele.

6.5.3.2 Rozhraní a implementace

Jazyk UML vychází ze zásady oddělení toho, co předmět vykonává (jeho rozhraní) od toho co vykonává (jeho implementace). Rozhraní definuje dohodu, která zaručuje, čím se budou jednotlivé implementace řídit.

6.5.4 Mechanizmy rozšiřitelnosti

Autoři jazyka UML si uvědomili, že návrh naprosto univerzálního modelovacího jazyka, který by uspokojil všechny současné i budoucí potřeby všech uživatelů, není možný. Proto do něj zabudovali tři jednoduché mechanismy, umožňující jeho rozšiřitelnost. Jsou to omezení (constraints), stereotypy (stereotypes) a označené hodnoty (tagged values).

6.5.4.1 Omezení

Omezení (constraints) jsou omezující podmínky, které rozšiřují sémantiku elementu tím, že umožňují k němu přidávat nová pravidla. Omezující podmínka je textový řetězec uzavřený do složených závorek {}. Tento text specifikuje podmínku nebo pravidlo, které musí být pravdivě vyhodnoceno. Tímto způsobem omezujeme určité chování daného elementu. Pro zápis vlastního mezení typicky používáme jazyk OCL (viz část 3.3.10).

6.5.4.2 Stereotypy

Definice stereotypu podle [Rumbaugh at al 1998] je:

Stereotyp zastupuje určitou variantu v daném modelu existujícího elementu, který má sice stejnou podobu (atributy, relace), ale používá se s jiným záměrem.

Stereotypy umožňují vytvářet nové elementy modelu založené na stávajících elementech. Název stereotypu se vloží do dvojitéch lomených závorek (<<stereotyp>>) a připojí se k danému elementu. Každý element může mít nejvíce jeden stereotyp.

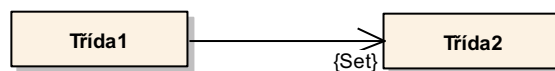
Jakýkoli stereotyp může definovat sadu označených hodnot a podmínek, které platí pro daný, stereotypem označený, element. Ke stereotypu můžeme přidružit rovněž samostatný symbol, barvu nebo texturu. Tímto způsobem můžeme rozšířit grafickou notaci jazyka UML.

Stereotypem tedy zavádíme do UML nové elementy. Těmto novým elementům také musíme definovat jejich sémantiku (význam). To se typicky provádí poznámkou do modelu při malých

počtech nových stereotypů, a nebo externím dokumentem definujícím nové stereotypy. Pokud stereotypy významně mění smysl elementu, tak k jeho definici můžeme také použít model v jazyce UML. Určité oblasti (například business modelování nebo databázové modelování) mají předdefinovaný okruh stereotypů. Jsou to tzv. profily (například UML profile for database modeling – viz [Amber 2005]). Hlavní výhodou, která mluví pro používání těchto profilů je to, že sice používáme specifické rozšíření UML pro své modely, ale s okolím jsme se dohodli na významu těchto nových prvků vnášených do modelu. Tím naše modely zůstanou srozumitelné okolnímu světu (a také CASE nástrojům).

6.5.4.3 Označené hodnoty

V jazyce UML se za vlastnost označuje jakákoliv hodnota sdružená s elementem modelu. Většina elementů má mnoho předdefinovaných vlastností. Pokud potřebujeme k elementu připojit novou vlastnost, která není standardně předdefinovaná standardem UML, potom použijeme tzv. označenou hodnotu (tagged value). Její obecná syntaxe je čárkami oddělený seznam dvojic jméno = hodnota, který je uzavřený do složených závorek. Tedy označené hodnoty mají tuto syntaxi: {tag1 = hodnota1, tag2 = hodnota2, ...}. Někdy mohou být označené hodnoty bez přiřazení hodnoty – tj. pouze {tag}.



obr. 43. Označená hodnota

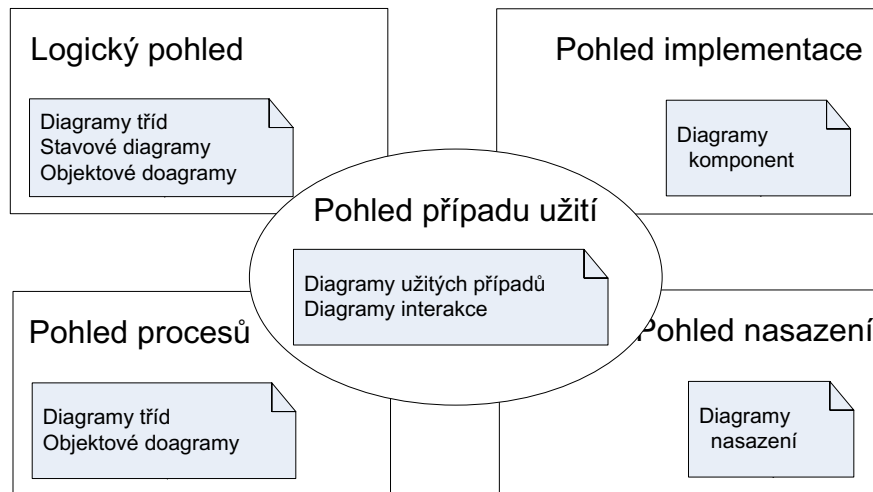
Příklad označené hodnoty je na obrázku. Tam jsou zachyceny dvě třídy, Vztah mezi nimi je 1:N, čili instance jedné třídy jsou uloženy v kolekci. Vlastnosti kolekce určuje označená hodnota {set}, která určuje, že tato kolekce je typu množina (tj. nelze v ní ukládat duplikované prvky).

6.6 Architektura

V knize The Unified Modeling Language Reference Manual (viz [Rumbaugh at al 1998]) je architektura systému definována takto :

Architektura je organizační struktura systému, včetně jeho rozkladů na součásti, jeho propojitelnosti, interakce, mechanismů a směrných zásad, která proniká do návrhu systému.

Architektura je zachycením strategických aspektů vyšší struktury systému. Pro zachycení všech podstatných aspektů architektury systému, definuje jazyk UML čtyři pohledy: logický pohled, procesní pohled, pohled implementace a pohled nasazení. Všechny tyto pohledy jsou integrovány do pátého pohledu, jímž je pohled případu užití. Více na obrázku 44.



obr. 44. Architektura

Jednotlivé pohledy architektury UML jsou:

- Logický pohled. Zachycuje slovník oblasti problému jako množinu tříd a objektů. Důraz je přitom kladen především na zobrazení způsobu, jakým objekty a třídy tvořící základ systému implementují jeho chování.
- Procesní pohled. Modeluje spustitelná vlákna a procesy jako aktivní třídy. J to procesně orientovaná varianta logického pohledu, která obsahuje stejné artefakty.
- Pohled implementace. Modeluje soubory a komponenty, které utvářejí hotov kód systému. Slouží jednak ke znázornění závislosti mezi komponentami, a také jak spravovat konfiguraci množin vytvořených z těchto komponent. Umožňuje definici verze systému.
- Pohled nasazení. Modeluje fyzické nasazení komponent na fyzické uzly systému (to jsou typicky počítače a periferie). Umožňuje modelování distribuce komponent na příslušné uzly distribuovaného systému.
- Pohled případů užití. Ostatní čtyři pohledy jsou odvozeny z pohledu případů užití. Tento pohled zachycuje základní požadavky kladené na příslušný systém.

Tato architektura (říká se jí 4+1) vzniká postupně během modelování systému.

6.7 Diagramy UML

Diagramy v jazyce UML jsou pohledy na model. V zásadě rozlišujeme statický pohled na model (diagram tříd, objektový diagram, diagram komponent, diagram nasazení) a dynamický (diagram případů užití, sekvenční diagram, diagram spolupráce, stavový diagram, diagram aktivit).

6.7.1 Diagram tříd

Třídní diagram znázorňuje statickou strukturu modelu, konkrétně třídy, jejich vnitřní strukturu, a jejich vztahy k ostatním třídám. Diagram tříd se sestává z množiny prvků, jako jsou třídy, balíčky a jejich vzájemné vztahy. Třídy (a třídní diagramy) mohou být organizovány pomocí balíčků.

6.7.1.1 Použití

Diagram tříd se používá k znázornění následujících struktur:

- • Důležité třídy a jejich vzájemné vztahy. Diagram tohoto typu slouží repositář nejdůležitějších tříd a je dobrý pro udělení si celkové představy o modelu. Tento model patří do logického pohledu na architekturu.
- • Funkčně závislé nebo podobné třídy.
- • Třídy patřící do jednoho balíčku.
- • Důležité ISA hierarchie a hierarchie skládání.
- • Důležité struktury znázorňující perzistentní třídy a vztahy mezi nimi. Tento diagram je důležitý pro implementaci databázového ukládání těchto objektů.
- • Balíčky a jejich závislosti, zejména pro znázornění jejich členění do vrstev.
- • Třídy účastnící se realizace specifického případu užití.
- • Třída, její atributy, metody a vztahy s ostatními třídami.

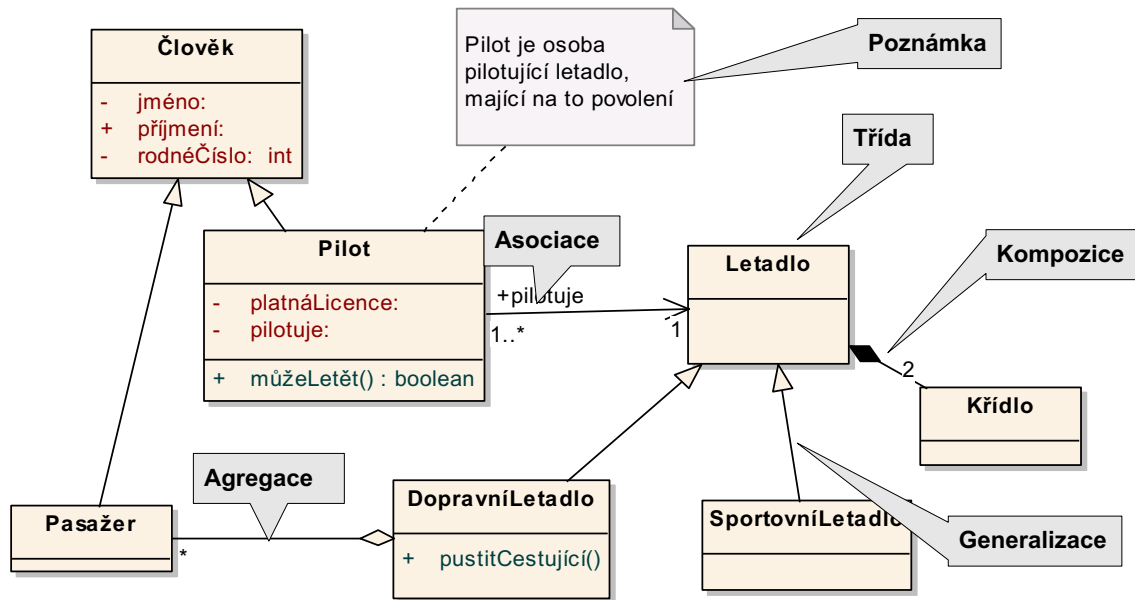
Je vhodné použít pro každou strukturu vlastní třídní diagram. Typicky se používá pouze část z těchto možností použití třídního diagramu. Každá třída z modely musí být znázorněna alespoň v jednom z těchto třídních diagramů.

Použité prvky

V třídním diagramu lze použít tyto prvky:

- • Třída – třída je v [Rumbaugh 1998] definována jako deskriptor množiny objektů, které sdílejí stejné atributy, operace, metody, relace a chování. To znamená, že třída popisuje tyto objekty. Můžeme jí tedy považovat za šablonku objektu,
- • Balíček – balíček je prvek, pomocí kterého strukturujeme model. Balíček v třídním modelu obsahuje třídy, které dělí do vzájemně soudržných jednotek. Mezi balíčky existují závislosti (a nesmí existovat kruhové závislosti).
- • Vazby – vyjadřují v třídním diagramu vztahy mezi třídami.
 - generalizace – pomocí generalizace vyjadřujeme ISA hierarchie (vztah nadtyp-podtyp), v objektově-orientovaných programovacích jazycích je typicky generalizace implementována pomocí dědění.
 - asociace – znázorňuje vztah (relaci) mezi třídami. Asociace může být pojmenovaná, mít pojmenované role, násobnosti a mohou mít označenu navigovatelnost (šipkou). Asociace dokonce může mít své vlastní atributy a metody – potom je třídou (mluvíme o asociační třídě).
 - agregace – agregace je silnější vztah mezi třídami než asociace, vlastnosti agregace lze shrnout takto:
 - celek občas existuje nezávisle na součástech, jindy je na nich závislý,
 - součásti mohou existovat nezávisle na celku,
 - součást může být sdílena více celky,
 - agregace je asymetrická – tj objekt nemůže být součástí sama sebe.

- kompozice – kompozice je silnější vztah než agregace, hlavní rozdíl mezi agregací a kompozicí je, že navíc každá součást patří pouze jednomu celku (zatímco u asociace je možno součásti sdílet).
- závislost – závislost je relace mezi dvěma elementy, v níž změna jednoho elementu promítá do druhého elementu. Závislostí znázorňujeme vazbu klient-server.



obr. 45. Třídní diagram

6.7.1.2 Speciální případ – diagram balíčků (Package diagram)

Diagram balíčků není definován standardem UML, ale je užitečný pro znázornění závislostí mezi balíčky. Je to vlastně třídní diagram v kterém jsou znázorněny pouze balíčky.

6.7.2 Objektový diagram

Objektový diagram slouží k zachycení jednoho, konkrétního stavu objektového systému. Je to vlastně fotografie systému. Je instancí třídního diagramu.

6.7.2.1 Použití

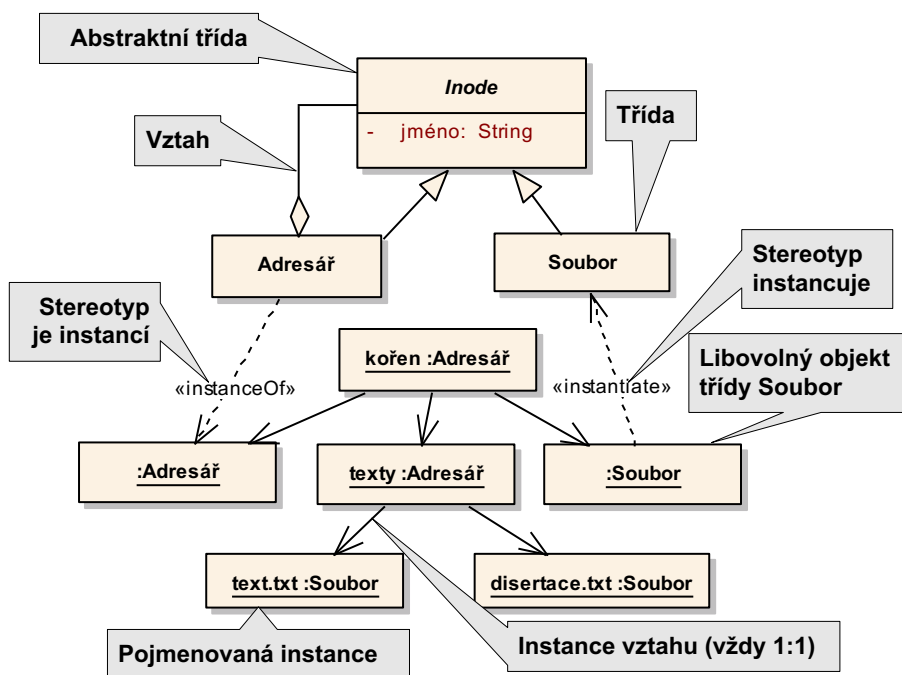
Objektový diagram se používá, když je není jasná struktura, jakou by měl mít třídní diagram. Práce a přemýšlení nad méně abstraktním objektovým diagramem je typicky jednodušší. Na obrázku obr je ukázán vztah mezi třídním a objektovým diagramem.

6.7.2.2 Použité prvky

Objektový diagram používá tyto prvky:

- Objekt – je instancí třídy. V diagramu se objekt kreslí stejně jako jeho třída (pomocí obdélníku), rozdíl je v tom, že u objektu je jeho jméno podtrženo.

- Vztahy mezi objekty – jsou to instance asociací, agregací a kompozic z třídního modelu. Mezi objekty mají vždy vztah 1:1.



obr. 46. Objektový diagram

6.7.3 Diagram případů užití

Tento diagram slouží k modelování požadavků. V diagramu užitých případů je zachycen pohled na systém zvnějšku. Systém vidíme z hlediska externích uživatelů systému (v jazyku UML se jim říká aktori).

6.7.3.1 Použití

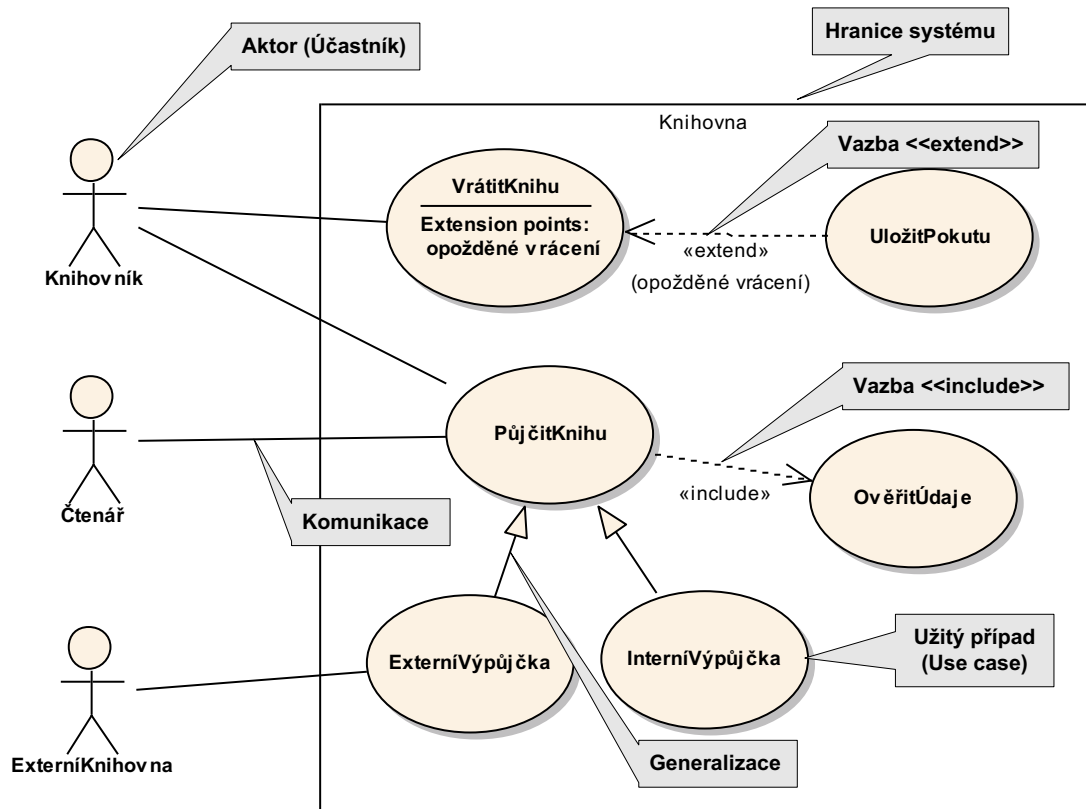
Neexistují striktní pravidla, co by měl diagram užitých případů zachycovat. Má ukazovat to, o čem se analytik domnívá, že je důležité k zachycení vztahů v modelu. Diagramy užitých případů mohou například zachycovat tyto případy:

- Nejdůležitější užití případy a jejich vzájemné vztahy a vztahy s aktory. Diagram tohoto typu může sloužit k zachycení globálního pohledu na model. Je součástí pohledu užitých případů na architekturu.
- Aktoři patřící k jednomu balíčku užitých případů.
- Aktor a všechny užití případy, kterých se účastní.
- Užití případy pracující se stejnými informacemi.
- Užití případy používané skupinou aktorů.
- Užití případy patřící k jednomu balíčku užitých případů.
- Užití případy, které jsou součástí té samé sekvence.
- Užití případy vyvíjené najednou.
- Specifický užití případ a jeho vztahy k ostatním užitým případům a aktorům.

6.7.3.2 Použité prvky

V diagramu užitých případů lze použít tyto prvky:

- Aktoři – jsou to role, přidělené osobám nebo předmětům používajícím daný systém.
- Případy užití – jsou to činnosti, které mohou aktoři se systémem vykonávat.
- Hranice systému – ohraničení kolem případů užití, pomáhá nám k určení velikosti systému. Aktoři jsou vždy vně systému
- Relace
 - Komunikace – je vazbou mezi aktorem a případem užití.
 - Generalizace – může existovat jak mezi případy užití, tak mezi aktory.
 - Vazba <<include>> – je to vztah skládání mezi případy užití. Používáme jí, pokud nějaký případ užití se používá ve více jiných případech užití.
 - Vazba <<extends>> – umožňuje rozšiřování případů užití o jiné případy užití. Rozšiřující případ užití se vykoná v případě, pokud je splněna podmínka definovaná tzv. bodem rozšíření (extension point). Na příkladě na obrázku xxx se vkládaný případ užití UložitPokutu vykoná pouze v případě, pokud je splněna podmínka definovaná bodem rozšíření (tj. opožděné vrácení knihy).



obr. 47. Diagram případů užití

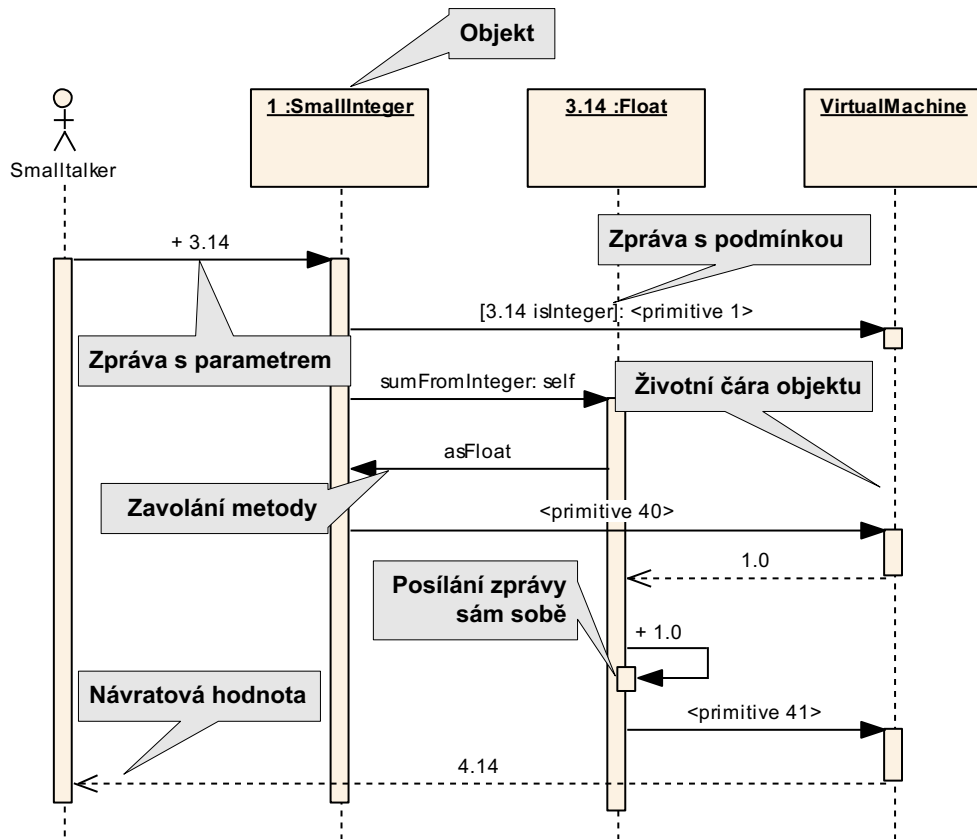
6.7.4 Sekvenční diagram

Sekvenční diagram patří mezi diagramy interakce (ještě spolu s diagramem spolupráce). Tyto diagramy typicky slouží k zachycení průběhu případu užití. V diagramu se odehrává vzájemná

komunikace mezi objekty. Sekvenční diagram se zaměřuje zejména na znázornění časových závislostí.

6.7.4.1 Použití

Sekvenční diagram se používá pro upřesnění a definici případu užití.



obr. 48. Sekvenční diagram

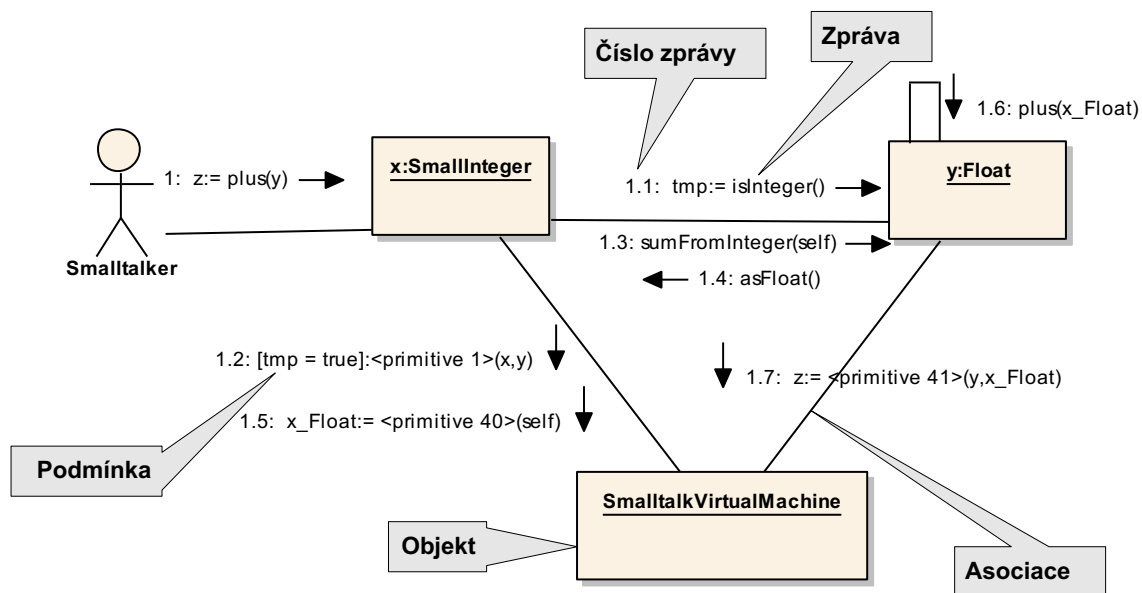
6.7.5 Diagram spolupráce

Diagram spolupráce je isomorfní (tedy lze je mezi sebou transformovat) s sekvenčním diagramem²⁸. Na rozdíl od něho, se však více zaměřuje na vyjádření vzájemných vztahů mezi objekty, než na časovou posloupnost.

6.7.5.1 Použití

Diagram spolupráce se používá pro upřesnění a definici případu užití.

²⁸ Příklady zde uvedené ukazují tu samou věc – návrhový vzor Double dispatching, který se používá například ve Smalltalku pro počítání s čísly.



obr. 49. Diagram spolupráce

6.7.6 Stavový diagram

Stavové diagramy jsou důležitou pomůckou pro modelování dynamického chování reaktivních objektů. V následující části zmiňované diagramy aktivit jsou speciálními případy stavových diagramů. Liší se zejména použitím a bohatostí syntaxe (stavový diagram má bohatší syntaxi). Diagramy aktivit se používají zejména při modelování business procesů, jichž se účastní několik objektů. Stavové diagramy se používají k modelování životního cyklu jednoho reaktivního objektu. Reaktivní objekty:

- reagují na vnější události,
- jejich životní cyklus je modelován jako řada stavů, přechodů a událostí,
- jejich chování je důsledkem předchozího chování.

Stavový diagram obsahuje právě jeden stavový automat pro jeden reaktivní objekt.

6.7.6.1 Použití

V objektově orientovaném modelování můžeme použít stavové automaty k modelování dynamického chování reaktivních objektů. Jsou to:

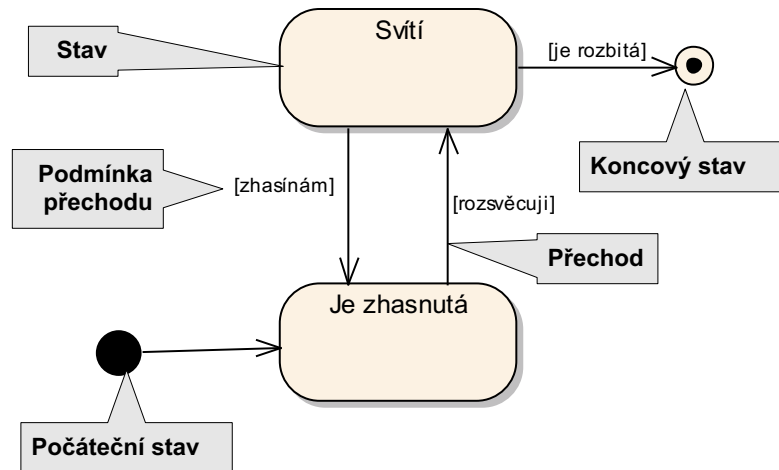
- třídy,
- případy užití,
- podsystémy,
- celé systémy.

Stavové automaty se nejčastěji používají pro modelování dynamického chování tříd.

6.7.6.2 Použité prvky

V základním stavovém diagramu se používají tyto prvky:

- Stav – stav modelovaného systému.
- Přejchod – přechod reprezentuje přechod mezi stavy stavového automatu.
- Podmínka – určuje podmínku přechodu do dalšího stavu.



obr. 50. Stavový diagram

6.7.7 Diagram aktivit

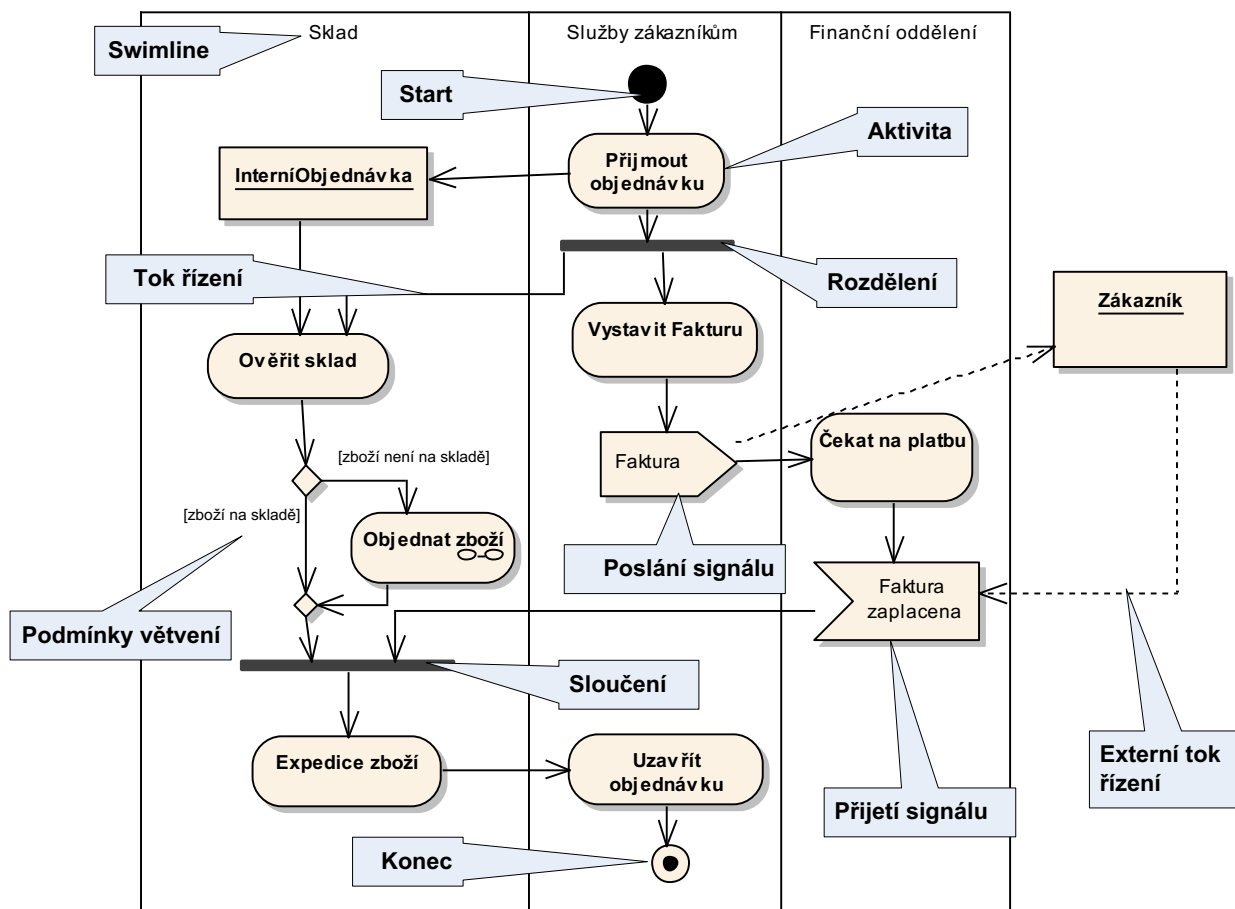
Diagramy aktivit jsou „objektově orientovanými diagramy toků“. Díky nim lze modelovat jakýkoliv proces jako kolekci aktivit a přechodů mezi nimi. Diagram aktivit lze připojit k libovolnému modelovanému elementu a umožní nám modelovat jeho chování.

6.7.7.1 Použití

Diagramy aktivit jsou obvykle připojeny k (a lze tedy jimi modelovat)::

- případům užití,
- třídám,
- rozhraním,
- komponentám,
- uzlům,
- spolupracím,
- operacím a metodám.

Pomocí diagramů aktivit se častokrát modelují business (podnikatelské a správní) procesy.



obr. 51. Diagram aktivit

6.7.8 Diagram komponent

Diagram komponent ukazuje statickou strukturu implementačního modelu. Ukazuje soubor statických elementů, jako jsou komponenty, subsystémy a jejich závislosti.

6.7.8.1 Použití

Diagram komponent se typicky používá pro znázornění těchto struktur:

- Implementačních subsystémů a jejich závislostí.
- Organizaci implementačních subsystémů do vrstev.
- Komponent (souborů zdrojového kódu) a jejich kompilačních závislostí.
- Komponent (aplikací) a jejich běhových závislostí.
- Důležitých struktur komponent, například pro znázornění jejich typického použití.

6.7.8.2 Použité prvky

V komponentovém diagramu se používají tyto prvky:

- Komponenta – jsou fyzické, nahraditelné části systému, které obalují implementaci a poskytují množiny specifikovaných rozhraní.
- Rozhraní – specifikovaný interface mezi komponentou a okolním světem.
- Závislost – komponenty mohou být na sobě závislé a to buď přímo nebo přes rozhraní.

6.7.9 Diagram nasazení

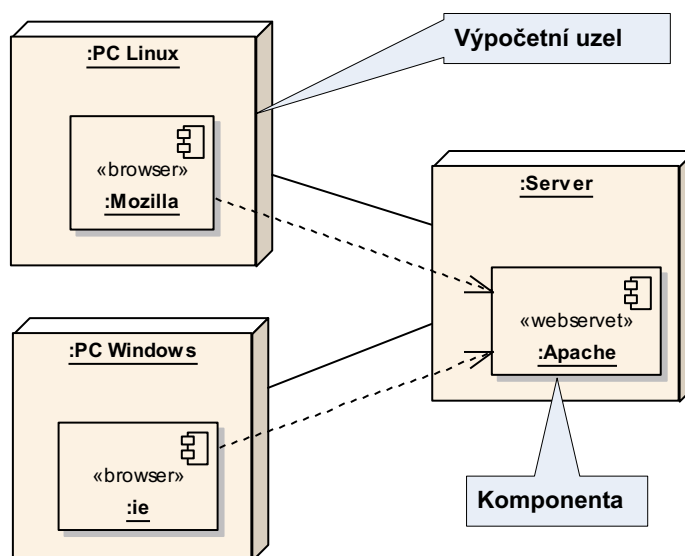
6.7.9.1 Použití

Diagram nasazení se typicky používá v mapování komponent na jednotlivé výpočetní uzly

6.7.9.2 Použité prvky

V diagramu nasazení se používají tyto prvky:

- uzel – představuje typicky fyzický výpočetní prostředek.
- komponenta – komponenta je přiřazena k uzlu.



obr. 52. Diagram nasazení

6.8 Metamodel UML

Jazyk UML je definován jako standard organizace Object Management Group (OMG). V současné době je aktuální standard verze 1.5 (více v [OMG 2003]). Jazyk UML je založen na obecné čtyřvrstvé metamodelovací architektuře. Napsat něco o MOF dalsich verzich

6.8.1 Čtyřvrstvá architektura

Standardy OMG (a tedy i UML) jsou založeny na čtyřvrstvé architektuře metamodelu. Tato architektura se skládá z těchto vrstev (více viz tabulka odkaz):

- meta-metamodel,
- metamodel,
- model,
- uživatelské objekty

Nejvyšší vrstva tj. meta-metamodel slouží k definování a tvorbě odvozených metamodelů. Existuje mnoho meta-metamodelovacích standardů (GOPRR, MOF, COMMA, CoCoA, COOM atd.). Všechny standardy OMG založené na metamodelu používají standard MOF (Meta Object Facility – viz [OMG 2002a]). Všechny metamodely založené na MOF mohou vyžít standardní formát pro výměnu dat XMI (XML Metadata Interchange).

Následující vrstvu tvoří metamodel. Tato vrstva definuje pravidla pro tvorbu modelu. Organizace OMG definuje mnoho standardů definovaných metamodelem – UML, CMW (Common Data Warehouse), MDA (Model Driven Architecture – viz [OMG 2002b]) atd.

Nejnižší dvě vrstvy tvoří model a data definovaná tímto modelem.

<i>Vrstva</i>	<i>Popis vrstvy</i>	Příklad
meta-metamodel	Definuje pravidla jazyka pro tvorbu metamodelu (viz nižší vrstva)	MetaClass, MetaAttribute
metamodel	Instancí meta-metamodelu je metamodel	Class, Attribute
model	Definuje pravidla pro tvorbu modelu Instance metamodelu je model. Definuje pravidla popisující informační doménu („typy“)	Osoba, Úvěr, Ručitel úvěru, Cenný papír, Rodné číslo, atd.
uživatelské objekty	Instancí modelu jsou uživatelská „data“	Jan Novák, Úvěr číslo 10215, 541212/2312

obr. 53. Standardní čtyřvrstvá architektura pro metamodelování

6.8.2 Struktura metamodelu UML

Metamodel UML je strukturován pomocí balíčků, anglicky package, takto:

- Foundation – základní infrastruktura specifikující statickou strukturu modelu
 - Core – balíček specifikuje základní koncepty metamodelu a definuje páteř pro přidávání dalších jazykových konstrukcí.
 - Extension Mechanism – specifikuje možnosti, jak rozšířit a přizpůsobit model novými prvky a novou sémantikou.
 - Data Types – specifikuje základní datové typy jazyka UML.

6.8.3 Jazyk OCL

Jazyk OCL (Object Constraint Language – více v [Warmer 1998]) je formální jazyk sloužící vyjádření omezení. Ty většinou specifikují invariantní podmínku, kterou musí splňovat modelovaný systém. OCL není programovací jazyk (nemá výrazy pro řízení toku programu) a umí pouze vyhodnocovat výrazy a zjišťovat jejich pravdivost. Slouží pro tyto účely:

- specifikaci invariantu třídy nebo typu v třídním diagramu,
- specifikaci invariantu v stereotypu,
- vysvětlení předběžných a následných podmínek operací a metod,
- jako jazyk pro navigaci,
- pro specifikaci omezení.

7 Příklady objektově orientovaných metodik

Na přelomu 80. a 90. let nastala v objektovém světě změna. Objektově orientované jazyky už existovali (Smalltalk od 70. let a C++ od první poloviny 80. let), ale dosud se na ně hledělo jako na něco velmi experimentálního. Příchod grafického rozhraní počítačů (GUI) tento pohled začínal měnit – vlastnosti objektových jazyků byly velmi výhodné pro tvorbu GUI (on vlastně Smalltalk původně vznikl jako jazyk pro snadnou komunikaci s uživatelem pomocí GUI v projektu Dynabook laboratoří Xeroxu v Parc Place). Pro návrh velkých informačních systémů se tehdy používali metody klasické (tj. strukturované) analýzy a návrhu (například Yourdonova strukturovaná metoda – viz [Yourdon 1989]). V té době objektově orientované jazyky dospěly do stadia, kdy je bylo možno použít pro větší projekty. Pro rozsáhlejší projekty jsou však nezbytné metodiky analýzy a návrhu – pro objektově orientované jazyky však v té době neexistovaly. Vznikla velká poptávka po takových metodikách. A kde je poptávka, tam se objeví i nabídka. Na konci 80. a první polovině 90. let začal boom objektově orientovaných metodik. V zemi nikoho, si každý snažil vykolíkovat svou výnosnou parcelu. Někteří vycházeli ze strukturovaných metodik (Coad-Yourdon, OMT), někteří šli objektovější cestou (Objectory). Nastala fáze expanze. Od poloviny 90. let se začínají objevovat snahy standardizovat alespoň grafický jazyk objektového modelování, když už ne metodiku. O standardizaci se nejvíce zasloužila firma Rational (pánové Booch, Rumbaugh a Jacobson – říká se jim Three Amigos), tvůrce prvních verzí standardu UML, a sdružení OMG, které má na starosti standardizaci a rozvoj UML v současné době.

Následující kapitola představuje několik objektově orientovaných metodik analýzy a návrhu vzniklých v první polovině 90. let..

7.1 Shlaer-Mellor

Metodika Object-Oriented Systems Analysis (OOSA) patří k prvním přístupům k objektově orientovaným metodologiím – byla prezentovaná již v roce 1988 v [Shlaer 1989].

Základní rámec metody tvoří identifikace domén, což jsou samostatné a nezávislé problémové oblasti. Domény mohou být dále rozděleny do subsystémů. Analýza začíná informačním modelem popisujícím objekty, atributy a jejich vztahy (chybí tedy metody, čímž tento model připomíná spíše datový model). Dále je vytvořen stavový diagram objektů. Diagram datových toků pak zobrazuje procesní model. Schlaer-Mellorova metodika zdůrazňuje prototypování a inkrementální vývoj.

Schlaer-Mellorova metodika nabízí následující proces pro vývoj informačního systému (i když kroky na sebe logicky navazují, stejně jako u jiných metod dochází k jejich překrývání a k iteracím v průběhu vývoje):

1. Rozdělit celý systém na domény.
2. Analyzovat aplikační doménu.
3. Potvrdit analýzu pomocí statické a dynamické verifikace (simulace).
4. Vybrat požadavky pro servisní domény.
5. Analyzovat servisní domény.
6. Specifikovat komponenty domény architektury.
7. Vytvořit komponenty architektury.
8. Přeložit modely každé domény za použití komponent architektury.

7.2 Coad – Yourdon

Tato metoda Object Oriented Analysis (OOA) pánů Coad a Yourdona (viz [Coad 1990]) je jednou z nejstarších objektových metodologií – vznikla v roce 1990. Autoři zdůrazňují úlohu objektové orientace zejména kvůli zlepšení komunikace mezi analytiky a experty na problémovou oblast, zvýšení konzistence mezi analýzou, designem a programováním, znovupoužitelnosti analýz, designu a programových produktů.

Analýza zahrnuje pět vrstev nazývaných SOSAS (podle počátečních písmen termínů reprezentujících každou fázi):

- **Subjekty** - jedná se o rozdělení problémové oblasti na části (z důvodu přehlednosti a lepší zvládnutelnosti). Tím vzniknou rozhraní mezi, které se musí dodržovat. Rozhraní má být jednoduché, složitost zůstává uvnitř subjektu.
- **Objekty** - specifikace tříd. Třídy mohou být konkrétní nebo abstraktní a mohou sloužit jako zdroj dědění.
- **Struktury** - jedná se o dva typy struktur, klasifikační struktury a složené struktury. Klasifikační struktury mají vztah ke vztahům dědičnosti v modelu tříd (opět se jedná o generalizaci/specializaci). Složené struktury definují ostatní typy vztahů mezi třídami.
- **Atributy** - atributy tříd a propojení instancí.
- **Služby** - jedná se o metody tříd a komunikační propojení. Pro definici služby je nutné určit, co se dělá (obsah služby), kdo to dělá (objekt) a kdo to potřebuje (jiný objekt). Pro odvození původce služby se doporučuje využít stavů objektu

Ve fázi designu je těchto pět vrstev transformováno do čtyř komponent:

- **Komponenta problémové oblasti (Problem Domain Component)** - třídy popisující problémovou oblast. Výsledek analýzy je třeba přizpůsobit implementačnímu prostředí a propojení s dalšími komponentami.
- **Komponenta lidské interakce (Human Interaction Component)** - uživatelské třídy jakožto interface pro komunikaci s uživatelem.
- **Komponenta správy úkolů (Task Management Component)** - třídy pro správu systému (zpracování chyb, zajištění bezpečnosti). Určení hardwarové architektury systému a operačního systému.
- **Komponenta správy dat (Data Management Component)** - třídy pro přístup k databázím. Určení místa uložení dat.

7.3 OMT

Metodiku Object Management Technique (OMT) popsal James Rumbaugh v roce 1991 (viz [Rumbaugh et al. 1991]). Obsahuje celou řadu myšlenek a přístupů, které jsou důležité pro analytiky a designéry. OMT byla velmi oblíbenou metodikou analýzy a návrhu – mnoho nových metodik jí bylo inspirováno. OMT zahrnuje jak notaci (jednotlivé použitelné diagramy), tak i popis objektově orientovaného vývoje informačního systému.

Vlastní analýza sestává ze 3 relativně samostatných modelů:

- objektový model (OM - Object Model),
- dynamický model (DM - Dynamic Model),

- Funkční model (FM - Functional Model).

Objektový model obsahuje definice tříd a jejich vztahů společně s atributy a metodami. Objektový model zachycuje statickou strukturu systému.

Dynamický model zachycuje dynamiku objektů a změny jejich stavů. Zabývá se chováním objektů v čase a tokem zpráv a kontroly mezi objekty. Dynamický model zahrnuje stavové diagramy (STD - State Transition Diagram) pro každou třídu nebo pro důležité části návrhu. Dále pak diagramy interakcí. Součástí DM je také celková mapa událostí (Event Trace Diagram) a model událostí (Event-flow Diagram).

Funkční model pak popisuje funkční závislosti systému, je podobný diagramu datových toků. Popisuje, co systém dělá (nezabývá se tím, jak to dělá).

OMT také popisuje fáze objektově orientovaného vývoje informačního systému. Jsou jimi analýza, systémový design, objektový design, implementace a testování. Ve fázi analýzy je nutné porozumět a namodelovat část reality, kterou bude systém obsahovat. V systémovém designu je potřeba určit celkovou architekturu systému. Objektový design se zabývá optimalizací modelů analýzy z hlediska konceptu implementace. V implementaci je pak programován vlastní počítačový systém. Testování probíhá průběžně ve fázích inkrementálního vývoje.

OMT dále detailněji popisuje kroky, které by měly být provedeny v jednotlivých fázích objektově orientovaného vývoje informačního systému. Ve fázi analýzy je třeba vytvořit objektový, dynamický a funkční model.

Tvorba objektového modelu by měla obsahovat následující kroky:

- Vytvořit slovní popisy modelovaného problému.
- Určit třídy objektů.
- Zrušit nepotřebné a chybné třídy.
- Připravit knihovnu znalostí
- Určit asociace mezi třídami.
- Zrušit nepotřebné a chybné asociace.
- Určit atributy tříd.
- Zrušit nepotřebné a chybné atributy.
- Určit vazby dědičnosti.
- Projít vše znova a určit nedostatky.

Tvorba dynamického modelu by měla obsahovat tyto kroky:

- Určit užité případy (use cases) a připravit scénáře typických interakčních sekvencí.
- Určit události mezi objekty a připravit mapu událostí pro každý scénář.
- Vytvořit diagram událostí systému.
- Vytvořit stavové diagramy pro třídy s významným dynamickým chováním.
- Zkontrolovat konzistenci a úplnost událostí sdílených mezi stavovými diagramy.

Tvorba funkčního modelu zahrnuje následující kroky:

- Určit vstupní a výstupní hodnoty.

- Vytvořit diagramy datových toků pro vyjádření funkčních závislostí.
- Popsat každou funkci, co dělá.
- Určit omezení.

7.4 OOAD – Booch

Metodologie OOAD (*Object-Oriented Analysis and Design with Applications* viz [Booch 1994] nebo [Booch 1996]) vznikla v roce 1994. Tato metodika se skládá ze čtyř hlavních aktivit a šesti notací. Pokrývá oblasti analýzy požadavků a analýzu reality, avšak hlavní důraz je kladen na design.

Booch nabízí různé pohledy pro popsání systému, a to fyzický a logický model a statický a dynamický model. Logický model je reprezentován strukturou tříd a objektů. Notace umožňuje vytváření kategorií tříd pro seskupování tříd. Každá kategorie je vlastním diagramem tříd. Objektový diagram zachycuje vztahy (komunikaci) mezi objekty.

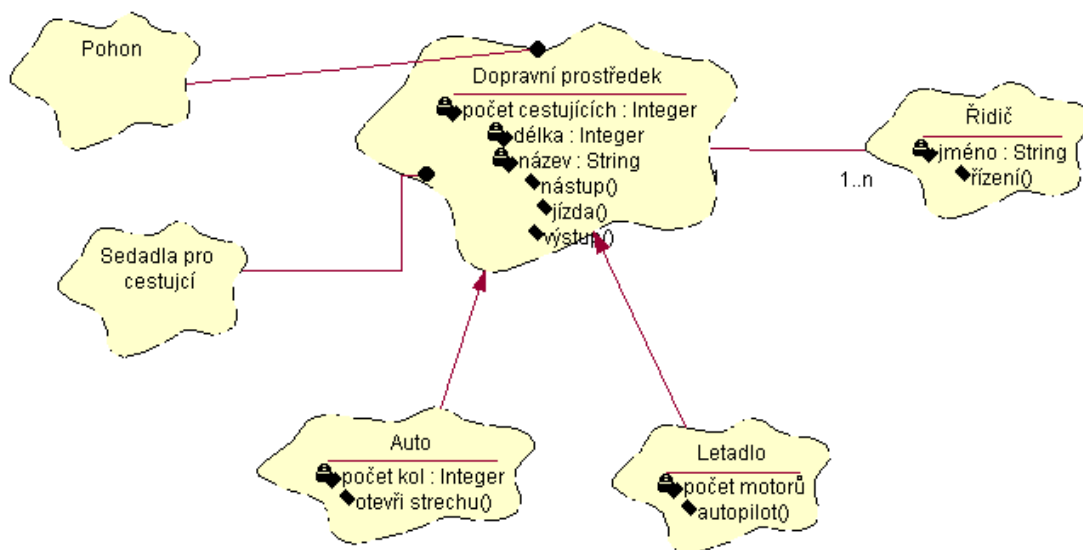
Boochova metodika definuje dva druhy procesů popisujících objektově orientovaný vývoj systému, a to makroproces a mikroproces. Mikroproces popisuje spíše každodenní aktivitu vývojářů. Zaměřuje se na zachycení obecně uznávaných nutných kroků pro vytvoření diagramu tříd a diagramu objektů. Makroproces je kostrou shrnující kroky nutné při inkrementálním vývoji systému. Systém jako celek je vytvářen v postupných krocích, následující krok inkrementálního vývoje vždy o něco rozšíří funkčnost a strukturu vytvořenou v předcházejícím kroku.

Kroky makroprocesu:

- určit základní požadavky (konceptuální část),
- vytvořit model požadovaného chování (analýza),
- vytvořit architekturu (design),
- implementovat,
- údržba.

Prvky mikroprocesu:

- určení tříd a objektů na dané úrovni abstrakce,
- určení sémantiky těchto tříd a objektů (zachycení jejich vlastností),
- určení vztahů mezi těmito třídami a objekty,
- určení rozhraní a implementace těchto tříd a objektů



obr. 55. Třídní diagram metodiky Booch

7.5 Objectory

Metodika Objectory Ivara Jacobsona vznikla v roce 1992 v knize [Jacobson at al. 1992]. Snaží se podpořit celý životní cyklus vývoje softwarového produktu. Jednodušší verzí Objectory je OOSE (Object-Oriented Software Engineering), kterou Jacobson vytvořil jakožto metodologii pro tvorbu aplikací.

Hlavním rozdílem od jiných metodologií té doby byl užitý případ (z této metodologie užitých případů převzalo UML). Jeho definice zahrnuje diagram zachycující interakce mezi aktory a systémem. Užitý případ je pak každý popis individuálního užití systému, aplikace nebo nějaké třídy, kdy aktoři přistupují k systému jako k černé skřínce. Zachycuje tedy aktorovy požadavky na systém.

Případ užití se vytváří jako první v modelu požadavků, poté se používá k vytvoření objektového modelu problémové oblasti. Podle autora metodiky hrají užití případy (use cases) dvě významné role. Jednak zachycují požadavky na funkcionalitu systému a jednak strukturují každý objektový model. Případy užití jsou nutné z důvodu rozdělení celého komplexního objektového diagramu na části, s kterými lze snadněji pracovat. Pro rozdělení se hodí právě užití případy. Pro každý užitý případ se vytvoří „samostatný“ objektový diagram pouze s těmi objekty, které se účastní tohoto užitého případu.

Při vývoji rozsáhlých informačních systémů doporučuje metodika Objectory rozdělit systém na podsystémy (z důvodů zvládnutelnosti řešení, údržby a dalších využití subsystémů) na každém podsystému bude pracovat jeden tým. Je tedy nutné předem určit interface (rozhraní) mezi podsystémy. Interface vytváří jeden nebo více kontraktů (contracts) mezi každou dvojicí komunikačně propojených podsystémů. Kontrakt je definicí služeb, které daný systém nabízí. Před tím, než je možné rozdělit vývoj systému na podsystémy, musí dojít k rozdělení požadavků a definici rozhraní. K tomu se doporučuje využít diagramy interakcí. Diagram interakcí popisuje, jak jsou užití případy systému distribuovány mezi podsystémy.

8 Unified Process

Protože neexistuje jednotná metodika analýzy a návrhu objektově orientovaného informačního systému, tak nám autoři jazyka UML (Jacobson, Booch, Rumbaugh) nabízejí metodiku, která aspiruje na to být standardem. Tato metodika se jmenuje Unified Process a následující kapitola o ní pojeňuje.

8.1 Unified Process - úvod

8.1.1 Historie UP

Metodika návrhu objektových informačních systémů Unified Software Development Process (USDP), známá spíše pod zkráceným názvem Unified Process (UP), je jednou z mnoha objektově-orientovaných metodik založených na jazyce UML. Tato metodika pochází přímo od autorů jazyka UML (Booch, Jacobson, Rumbought) a je (spolu se svými deriváty – např. RUP) v současnosti nejpoužívanější metodikou.

Metodika UP je založena na metodikách Ericson (Ericson approach), Rational (Rational Objectory Process), OMT a na dalších zdrojích vycházejících z nejlepších postupů.

Kořeny metodiky sahají do roku 1967, kdy vznikl Ericssonův model, který vychází z faktu, že se složité systémy mají modelovat jako množiny vzájemně propojených bloků. Malé bloky byly propojeny tak, že skládaly větší bloky, jež pak skládaly celý systém. Základem tohoto postupu byla zásada „rozděl a panuj“. Další inovací Ericssonova modelu byl způsob pomocí kterého byly tyto bloky hledány – tzv. „provozní případy“ (traffic cases), jež popisují způsob užití systému. Uvedené provozní případy se postupně transformovaly v dnešní v UML používané užití případy. Výsledkem tohoto procesu návrhu byla architektura, která popisovala nejen všechny stavební bloky, ale i jejich způsob vzájemné spolupráce. Kromě popisu požadavků (traffic cases) a architektury (statický model) obsahovala metodika Ericsson také dynamický pohled. Ten se skládal z sekvenčních diagramů, diagramů spolupráce a stavů. Všechny tyto diagramy dnes jsou dnes obsaženy v jazyce UML.

V roce 1987 zakládá Ivar Jacobson firmu Objectory AB. Ta vyvinula metodiku Objectory (více [Jacobson at al. 1992]). Pravděpodobně nejdůležitější inovací byla skutečnost, že metodika Objectory byla vyvinuta jako systém se svými náležitostmi a návaznostmi – celý průběh metodiky (požadavky, analýza, návrh, implementace, test) byl zachycen pomocí diagramů. Tato metoda umožňovala (a také vyžadovala – podobně jako UP) své přizpůsobení konkrétnímu uživateli a projektu. Když v roce 1995 firma Rational koupila firmu Objectory, začal se Jacobson zabývat postupným sjednocováním metodiky Objectory s dalšími metodikami firmy Rational. Výsledkem byla architektura 4+1 samostatných pohledů na systém. Tyto pohledy jsou – logický, procesní, fyzický a vývojový, které sjednocuje pohled případů užití. Do metody Rational Objectory Process (ROP) byly zahrnuty také zkušenosti (zejména s architekturou) Gradyho Boocha obsažené v metodě OOAS (viz [Booch 1994]). Současně se vznikem ROP vzniká ve firmě Rational první návrhy společného objektového modelovacího jazyka UML. Jazyk UML se stal jazykem metody ROP.

V roce 1998 firma Rational vytváří metodiku Rational Unified Process (RUP – více třeba [Kruchten 2000]). Ta obsahuje zejména zlepšení (oproti ROP) v oblastech zachycování požadavků, správa konfigurace, testování atd. V roce 1998 Jacobson publikuje *knihu Unified Software Development Process* (viz [Rumbaugh at al. 1998]), v níž je popsána metodika Unified Process (UP). Zatímco RUP je komerčním produktem firmy Rational, tak UP je otevřený standard od tvůrců jazyka UML.

8.1.2 UP a odvozené metody

Existuje několik variant metody UP. Nejrozšířenější je RUP, mezi další patří například Enterprise Unified Process od Amblera (viz [Amber 2005]). Vztah UP a jeho variant je takový, že odvozené metody předefinovávají (a upřesňují) jednotlivé části metodiky, nabízejí implementované nástroje, předvytvořené šablony atd.

8.1.3 Zásady použití UP

Metodika UP je obecnou metodikou tvorby softwaru. Obsahuje mnoho různých metod pro jednotlivé fáze životního cyklu projektu, pro jednu fázi může existovat i několik konkurenčních metod práce. Pro každý konkrétní projekt je nutno vytvořit nový výběr z použitých metod (vytvořit novou instanci metodiky). Mezi kritéria jaké metody z metodiky UP vybrat pro konkrétní projekt slouží například tato kritéria:

- typ projektu (databázová aplikace, řídicí systém atd.),
- rozsáhlost projektu,
- velikost týmu,
- použité technologie (programovací jazyky, databáze).

Více o výběru správné instance metodiky UP je například v [Rumbaugh at al. 1998].

8.1.4 Axiomy UP

Metodika UP je založena na třech základních axiomech. Jsou to:

- řízení případem užití a rizikem,
- zásada soustředění na architekturu
- zásada iterace a přírůstku.

Metodika UP je řízena požadavky. Ve všech fázích tvorby softwaru posuzuje další postup na základě analýzy rizik. Metodika UP je založena na návrhu a postupném vývoji robustní architektury systému. Architektura popisuje nejen aspekty rozkladu systému na komponenty, ale také popisuje způsob jakým se tyto komponenty ovlivňují.

Metodika UP je iterativní a přírůstková. Iterativní aspekt znamená, že rozklad projektu na menší podprojekty – iterace. Znamená to, že projekt tvoříme postupným upřesňováním a rozšiřováním funkcí systému.

8.1.5 Iterace v metodice UP

Iterace se v metodice UP skládá z těchto pěti pracovních postupů (částí):

- Požadavky – zachycují to, co by měl systém dělat.
- Analýza – vybroušení požadavků a jejich strukturování.
- Návrh – realizace požadavků v architektuře systému.
- Implementace – vlastní tvorba softwaru.

- Testování – ověření, zda jsme implementace funguje podle zadaných požadavků.

Každá iterace metodiky UP tvoří základní linii (baseline), což je soubor revidovaných a schválených výstupů dané iterace. Přírůstky (inkrementy) jsou rozdílem mezi jednou základní linií a jinou základní linií. Přírůstky jsou jednotlivými dílčími kroky směrem k finální verzi systému.

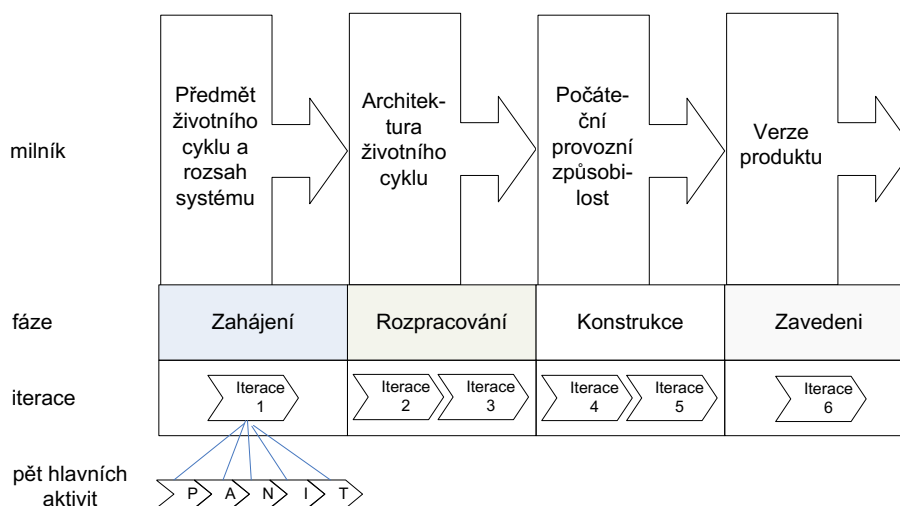
Jednotlivé práce na jednotlivých iteracích mohou probíhat paralelně (pokud to závislosti mezi nimi dovolují). To umožňuje zrychlit proces návrhu systému.

8.1.6 Struktura UP

Metodika UP se skládá ze čtyř po sobě následujících fází (viz obrázek 56). Každá tato fáze končí hlavním milníkem. Tyto fáze jsou:

- Začátek (inception) – období plánování,
- rozpracování (elaboration) – období architektury,
- konstrukce (construction) – počátky provozuschopnosti,
- zavedení (transition) – nasazení produktu do uživatelského prostředí.

Každá fáze má určitý cíl, na nějž jsou soustředěny aktivity jednoho nebo více pracovních postupů a jež je významným milníkem v životním cyklu projektu.



obr. 56. Struktura UP

8.1.6.1 Fáze začátek

Cíle fáze Začátek

Cílem fáze začátek je zahájení projektu. Tato fáze obsahuje:

- Tvorbu podmínek proveditelnosti – to může zahrnovat tvorbu prototypů;
- prvotní návrh obchodního (podnikatelského) případu – na něm má být možno ukázat kvantitativní obchodní přínos;
- zachycení podstatných požadavků – ty umožňují definovat rozsah systému;
- označení kritických rizik.

Hlavními pracovníky jsou v této fázi manažer projektu a systémový projektant.

Zaměření fáze začátek

Hlavní důraz je v této fázi kladen na pracovní postupy zabývající se specifikací požadavků a jejich analýzou. Do této fáze však mohou spadat rovněž určité návrhářské a implementační práce (například tvorba prototypu). V této fázi obvykle nedochází k testování, protože jedinými softwarovými výstupy jsou prototypy, které budou stejně zahozeny. Pracovní postupy etapy tedy jsou:

- požadavky – úvodní specifikace požadavků,
- analýza – analýza požadavků,
- návrh – případný návrh prototypů,
- implementace – případná implementace prototypů.

Milník: Předmět životního cyklu a rozsah systému

Milníkem²⁹ počáteční fáze je předmět životního cyklu a rozsah systému (Life Cycle Objectivities). Podmínky, které musí být splněny, aby byl milník považován za dosažený jsou v tabulce 57.

Hodnotící kritéria (míry)

Uživatelé a zainteresované osoby (zadavatel) souhlasí se záměry životního cyklu.

S uživateli a zainteresovanými osobami byl dohodnut rozsah projektu.

S uživateli a zainteresovanými osobami byly dohodnuty zachycené klíčové požadavky.

Uživatelé a zainteresované osoby schválili náklady a pracovní rozvrh.

Manažer projektu přednesl prvotní návrh obchodního případu.

Manažer projektu odhadl rizika.

Potvrzení proveditelnosti obsažené v technických studiích a prototypech.

Počáteční návrh architektury.

Je třeba dodat

Dokument o představě, jenž obsahuje hlavní požadavky na projekt, funkce a podmínky.

Počáteční případ užití (kompletní asi z 10 až 20 procent).

Slovník projektu.

Počáteční plán projektu.

Obchodní případ

Dokument obsahující odhad rizik.

Jeden nebo více zkušebních prototypů.

Počáteční dokument zachycující architekturu.

obr. 57. Hodnotící criteria fáze zahájení

8.1.6.2 Fáze Rozpracování

Cíle fáze Rozpracování

Hlavními výstupy fáze rozpracování jsou:

- Tvorba spustitelného architektonického základu,

²⁹ Zatímco se většina metodik návrhu softwaru zaměřuje zejména na tvorbu konečných artefaktů, metodika UP se zaměřuje spíše na konečné cíle. Každý milník nastavuje konečné cíle, kterých musí být dosaženo, aby bylo možné považovat milník za dosažený. Některé cíle mohou být výsledkem určitých artefaktů, některé nikoliv.

- vylepšení odhadu rizik,
- definice atributů kvality,
- zachycení případů užití pro 80% funkčních požadavků,
- tvorba přesného plánu konstrukční fáze,
- formulace nabídky, která obsahuje veškeré požadavky na čas, vybavení, personál a náklady.

Hlavním cílem je tvorba spustitelného architektonického základu. Je to skutečný, spustitelný systém, sestavený na základě specifikací architektury. Není to prototyp, který můžeme zahodit, ale je to první pokus o konečný systém.

Zaměření fáze Rozpracování

Ve fázi rozpracování je kladen důraz na tyto pracovní aktivity:

- požadavky – upřesnění rozsahu systému a požadavků na něj kladených,
- analýza – stanovení toho, co se bude tvořit,
- návrh – tvorba stabilní architektury,
- implementace – tvorba spustitelného architektonického základu,
- testování – testování architektonického základu.

Ve fázi rozpracování je kladen zejména důraz na pracovní postupy zabývající se požadavky, analýzou a návrhem. Implementace nabývá na důležitosti s blížícím se koncem fáze rozpracování při tvorbě spustitelného architektonického základu.

Milník: Architektura jako vodítko pro systém v jeho budoucím životě

Milníkem této fáze je architektura (Life Cycle Architecture). V tomto milníku ověřujeme zejména detaily systému a jeho rozsah, výběr architektury a výsledek analýzy rizik. Více v tabulce 58.

Hodnotící kritéria (míry)

Je třeba dodat

Byl vytvořen odolný a robustní spustitelný architektonický základ.

Spustitelný architektonický základ. Statický model UML.

Architektonický základ ukazuje, že byla rozpoznána a vyřešena důležitá rizika..

Dynamický model UML. Model případu užití.

Vize produkty byla stabilizována.

Dokument o vizi.

Odhad rizik byl revidován.

Aktualizovaný odhad rizik.

Obchodní případ byl revidován a odsouhlasen uživateli a zainteresovanými osobami

Aktualizovaný obchodní případ.

Projekt byl vytvořen do dostatečné hloubky, aby umožnil sestavení realistické nabídky zahrnující odhad času, peněz a prostředků pro nadcházející fáze. Uživatelé a zainteresované osoby souhlasí s plánem projektu

Aktualizovaný plán projektu.

Plán projektu byl porovnán s obchodním případem.

Obchodní případ a plán projektu

Bylo dosaženo dohody s uživateli a zainteresovanými osobami o pokračování projektu

Konečný dokument.

obr. 58. Hodnotící kritéria fáze rozpracování

8.1.6.3 Fáze Konstrukce

Cíle fáze Konstrukce

Cílem konstrukční fáze je splnit všechny požadavky analýzy a návrhu a vyvinout ze základu získaného z předchozí etapy konečnou verzi systému. Klíčovým zadáním konstrukční fáze je zachování integrity architektury vytvářeného systému.

Zaměření fáze Konstrukce

V této fázi je primární pracovní postup implementace. Tato fáze obsahuje opět tyto pracovní postupy:

- požadavky – odhalit veškeré požadavky, které byly v předchozích fázích přehlédnuty,
- analýza – dokončit analytický model,
- návrh – dokončit model návrhu,
- implementace – zajistit počáteční provozní způsobilost (Initial Operation Capability),
- testování – testovat počáteční funkční variantu.

Milník: počáteční provozní způsobilost

Milníkem této fáze je softwarový systém připravený k testování u uživatele (beta verze). Hodnotící kritéria tohoto milníku jsou uvedena v tabulce 59.

Hodnotící kritéria (míry)	Je třeba dodat
Softwarový produkt je dostatečně stabilní a na takové úrovni, aby jej bylo možno nasadit na počítače uživatele.	Softwarový produkt. Model UML. Testovací sadu
Uživatelé a zainteresované osoby souhlasí s nasazením softwaru do svého prostředí a jsou připraveni	Uživatelské příručky. Popis verze.
Poměr skutečných vůči plánovaným výdajům je přijatelný.	Plán projektu.

obr. 59. Hodnotící kritéria fáze konstrukce

8.1.6.4 Fáze Zavedení

Cíle fáze Zavedení

Fáze zavedení začíná v okamžiku, kdy je dokončeno testování a konečné zavedení systému. To zahrnuje opravu všech chyb nalezených v beta verzi. Cíle této etapy jsou takovéto:

- Oprava chyb,
- příprava uživatelského pracoviště na přijetí nového softwaru,
- přizpůsobení softwaru v případě vzniku problémů,
- tvorba manuálů a dokumentace,
- konzultace s uživateli,
- konečná revize.

Zaměření fáze Zavedení

V této fázi je kladen zejména důraz na implementaci a testování. V této fázi by už neměli vznikat nové požadavky a neměla by se provádět analýza. Pracovní postupy v této fázi jsou:

- návrh – úprava návrhu, jsou-li při testování nalezeny chyby,
- implementace – přizpůsobení softwaru pracovišti uživatele a oprava chyb,
- testování – beta testy a převjímací testy u uživatele..

Milník: Nasazení produktu

Je to poslední milník. Softwarový produkt je dokončen a připraven pro užívání zákazníkem. Podmínky dosažení tohoto milníku jsou v tabulce 60.

Hodnotící kritéria (míry)

Beta-testy jsou dokončeny, byly provedeny nezbytné změny a uživatel souhlasí, že byl systém úspěšně nasazen.

Pracovníci uživatele produkt aktivně využívají.

Strategie podpory produktu byla nejprve s uživateli dohodnuta a následně implementována.

Je třeba dodat

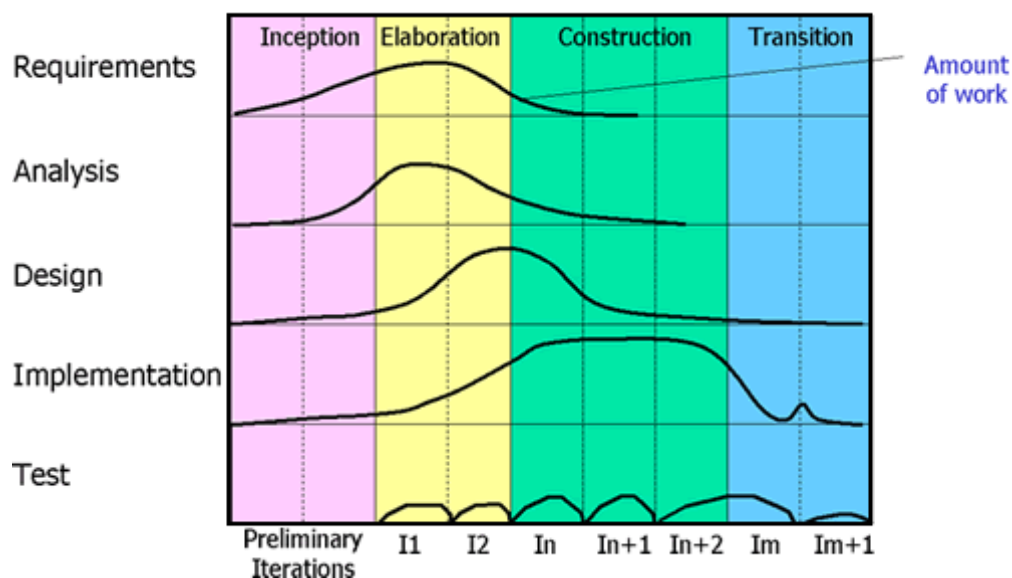
Softwarový produkt.

Plán uživatelské podpory. Uživatelské příručky.

obr. 60. Hodnotící kritéria fáze nasazení

8.2 Požadavky

Z obrázku 61 vyplývá, že hlavní práce při definici a specifikaci požadavků probíhá ve fázích začátek a rozpracování – tj. na začátku celého projektu. To je logické, protože hned na začátku potřebujeme mít alespoň rámcový přehled o tom, čeho chceme dosáhnout, jaký je význam požadavků a jejich specifikace. Musíme zjistit, co má systém dělat a dosáhnout o tom shodu se zadavatelem systému. Vše by mělo být popsáno v jazyku uživatelů budoucího systému. Tvoříme vlastně nejvyšší specifikaci toho, co má systém dělat. Práce s požadavky je označována jako inženýrství požadavků (requirements engineering).

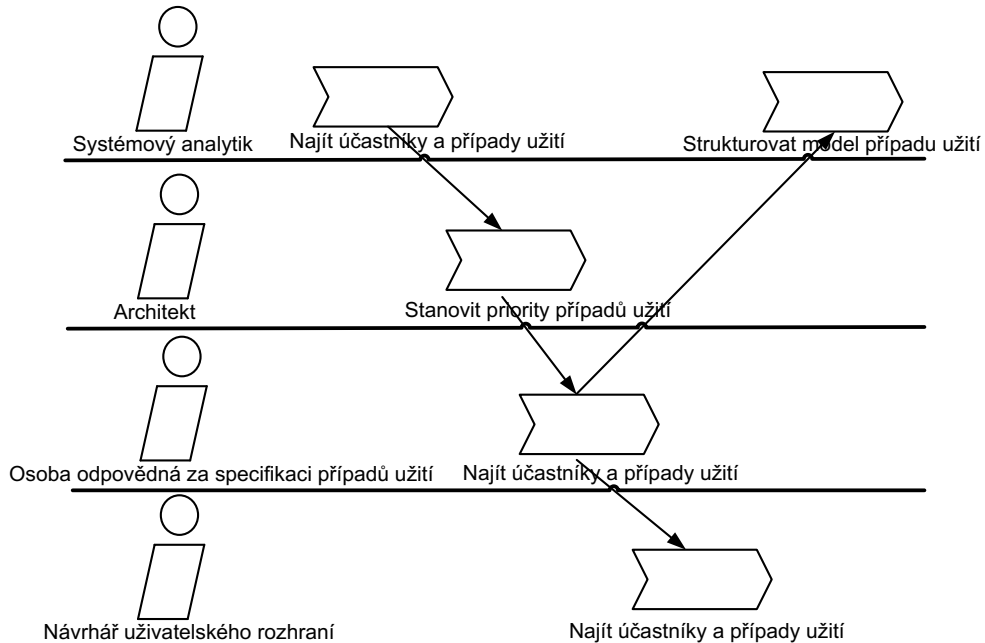


obr. 61. Staffing

8.2.1 Pracovní postup Požadavky

Na obrázku odkaz jsou znázorněny specifické úkoly pracovního postupu specifikace a definice požadavků v metodice UP. Tento systém diagramu je označován jako detail pracovního postupu (workflow detail).

Detaily pracovního postupu jsou modelovány pomocí jednotlivých úloh (symbol široká šipka) a jejich vzájemnými návaznostmi (šipky znázorňují tok prací). Za vykonání každé jednotlivé úlohy je zodpovědný člověk, který je na obrázku znázorněn svou rolí



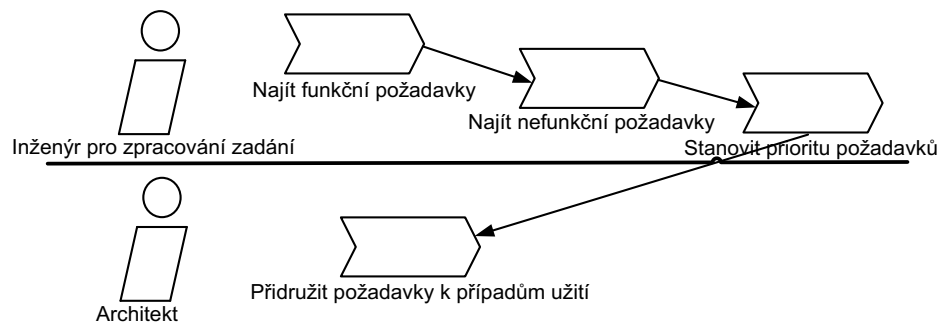
obr. 62. Pracovní postup - požadavky

Dále si podrobněji popíšeme tyto úkoly pracovního postupu požadavky:

- Vyhledání aktorů a případů užití,
- Detail případu užití,
- Strukturovat modelu případu užití.

Standardní postup UP se ve fázi požadavků zabývá zejména případy užití. Tento přístup má ovšem své nedostatky při modelování nefunkčních požadavků na systém. Proto je výhodné si rozšířit pracovního postupu definice a specifikace požadavků. Na obrázku 63 je navrženo rozšíření pracovního postupu o následující nové úlohy (podle [Arlow 2003]):

- Vyhledání funkčních požadavků,
- vyhledání nefunkčních požadavků,
- stanovení priorit jednotlivých požadavků,
- přidružení požadavků k případům užití.



obr. 63. Detail pracovního postupu

8.2.2 Definice požadavků

Požadavek lze podle [Rumbaugh at al 1998] definovat jako „specifikaci toho, co by mělo být implementováno“. Existuje mnoho typů požadavků, ale nejdůležitější jsou tyto dva druhy požadavků:

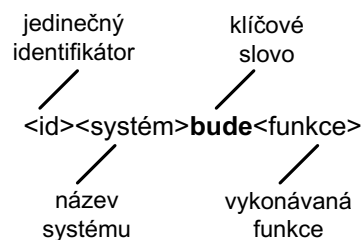
- Funkční požadavky, jež určují, jaké chování bude systém nabízet,
- Nefunkční požadavky, které specifikují vlastnosti nebo omezující podmínky daného systému.

Požadavky by měli být základem všech systémů. Jsou vyjádřením toho, co by měl systém dělat, nikoli toho, jak by to měl dělat.

8.2.2.1 Specifikace požadavků

Jazyk UML neposkytuje žádné doporučení, jak psát požadavky. S požadavky se vypořádává výhradně pomocí užitých činností. Diagramy užitých činností neumí zejména zachytit nefunkční požadavky na systém, proto se v [Arlow 2003] doporučuje používat tento jednoduchý zápis (viz obr 64) pro zachycení požadavků.

Každý požadavek má jedinečný identifikátor (typicky číslo), následně název systému, kterého se požadavek týká, potom klíčové slovo (bude, či anglicky shall) a nakonec funkci vykonávanou systémem.



obr. 64. Specifikace požadavku

8.2.3 Modelování případů užití

Modelování případů užití se skládá z následujících kroků:

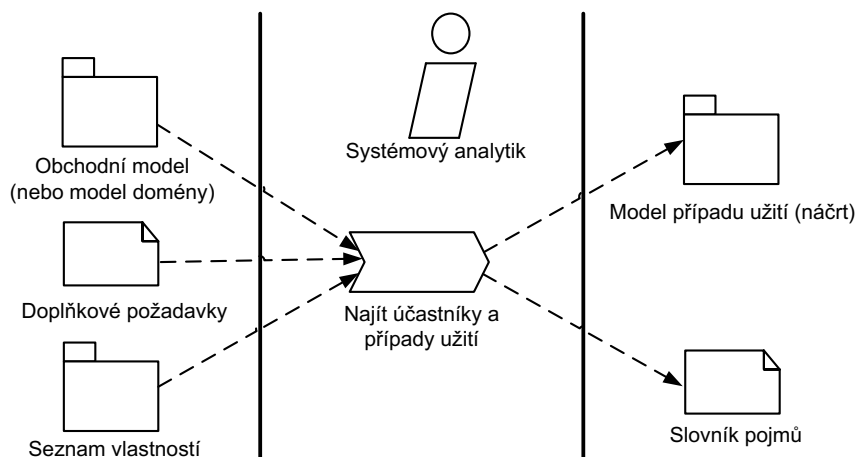
- Nalezení hranic systému,
- vyhledání aktorů,
- nalezení případů užití,

- specifikace případu užití,
- tvorba scénářů.

Výstupem uvedených aktivit je model případu užití.

8.2.4 Aktivita metodiky UP: Najít aktory a případy užití

Tato aktivita metodiky UP se zaměřuje na vytvoření první verze modelu případu užití a slovníčku pojmů. Podrobnosti jsou znázorněny na obrázku 65.



obr. 65. Aktivita metodiky UP: Najít aktory a případy užití

8.2.4.1 Hranice systému

První věcí při návrhu informačního systému je, že musíme určit jeho hranice. Jinými slovy musíme určit, co je součástí systému (uvnitř jeho hranic) a naopak co není jeho součástí (za hranicemi systému). Stanovení hranic systému má dopad na funkční požadavky na systém. Špatné stanovení (či v horším případě nestanovení) hranic systému vede častokrát k neúspěchu při jeho tvorbě.

Hranici systému definuje ten, kdo systém používá (aktoři), a to, co specifikuje přínos systému aktorům (čili případy užití).

8.2.4.2 Aktoři

Aktor specifikuje roli, kterou určitá externí entita přijímá v okamžiku, kdy začíná daný systém používat. Abychom mohli aktory systému najít, musíme zvážit kdo, nebo co³⁰ daný systém používá. K rolím, které osoby nebo předměty ve vztahu k systému hrají, můžeme dospět úvahami o konkrétních osobách a předmětech a následnou generalizací.

Při modelování aktorů je nutno pamatovat na následující body:

- Aktoři jsou vždy vůči systému externí – jsou tedy na systému nezávislí.
- Aktoři komunikují bezprostředně se systémem – to nám napomáhá k určení hranic systému.

³⁰ Aktorem nemusí být jen role, kterou hraje živý člověk, aktorem může být například i externí systém.

- Aktoři představují role, které vůči systému hrají – nejsou to konkrétní osoby nebo předměty.
- Jedna osoba nebo předmět může vůči systému vystupovat ve více rolích.
- Každý aktor musí mít krátký, výstižný název, který je srozumitelný z obchodního hlediska.
- Každý aktor musí mít svůj krátký popis, který ho popisuje z obchodního hlediska.
- Aktor může (ale není to typické) své atributy.

8.2.4.3 Případy užití

V knize *The Unified Modeling Language Reference Manual* (viz [Rumbaugh et al. 1998]) je případ užití definován jako „specifikace posloupností a chybových posloupností, které systém, podsystém nebo třída může vykonat prostřednictvím interakce s vnějšími aktory“.

- Případ užití je něco, co aktor od systému očekává. Je to „případ užití systému specifickým aktorem“:
- Případy užití jsou vždy iniciovány aktorem, případy užití jsou vždy napsány z hlediska aktora.

Případy užití obvykle považujeme za část systému. Případy užití můžeme použít i při modelování business procesů.

Při hledání případů užití lze postupovat tak, že se postupně procházíme seznamem aktorů a uvažujeme, jakým způsobem aktor interaguje se systémem. Výsledkem tohoto postupu je seznam případů užití. Název každého případu musí být slovesnou vazbou.

Hledání případů užití je iterativní proces – často jejich hledání nalezneme nové aktory. Pro ně opět nalezneme nové případy užití a pokračujeme takto stále dál, dokud nenalezneme všechny případy užití. Následně pokračujeme upřesňováním případů užití.

8.2.4.4 Slovník pojmů

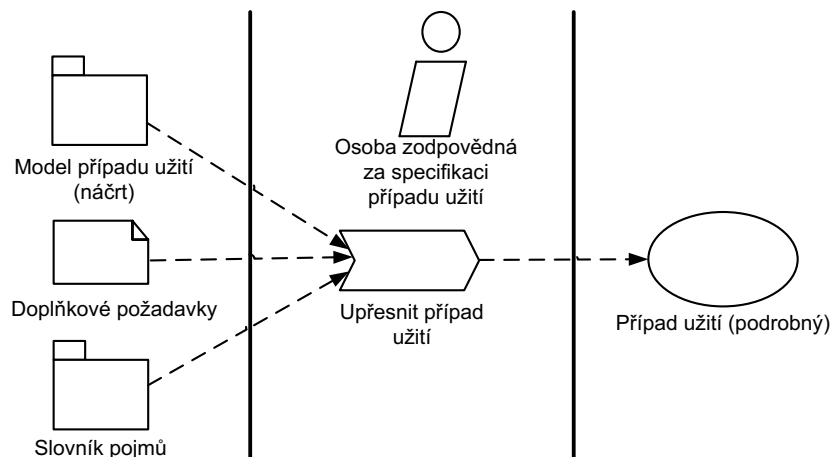
Slovník pojmů projektu (project glossary) může být jednou z nejdůležitějších částí projektu. Tvůrce systému typicky není odborník v oblasti obchodní domény systému – může dojít k omylům a nedorozumění díky existenci synonym (tj. více označení pro tu samou věc) a homonym (stejný termín pro několik rozdílných věcí).

Do slovníku pojmů by se měli zanést všechny důležité termíny³¹ a u každého termínu uvést jeho veškerá synonyma. Jazyk UML neobsahuje standard pro slovník pojmů. Typicky platí zásada – čím jednodušší, tím lepší. Typicky se používá prostý textový dokument, nebo dokument v jazyku HTML.

8.2.5 Aktivita metodiky UP: Detail případu užití

Po vytvoření diagramu případů užití je nutno jednotlivé případy užití podrobněji specifikovat. To je obsahem aktivity metody UP „Detail případu užití“ a ta je shrnuta na obrázku 66.

³¹ Hezké homonymum – termín jako datum (odevzdání) a termín jako pojem.



obr. 66. Aktivita metodiky UP: Detail případu užítí

Každý případ užítí má svůj název a specifikaci (viz obrázek 67). Specifikace se skládá z těchto součástí:

- Vstupní podmínky. Jsou to kriteria, které musí být splněna ještě předtím, než je možné spustit případ užítí.
- Tok událostí. Jsou to jednotlivé kroky v případě užítí.
- Následné podmínky. Jsou to kriteria, jež musí být splněna na konci případu užítí.

Případ užítí: PlatitDPH
ID: UC1
Aktoři: Čas Finanční úřad
Vstupní podmínky: 1. Je konec fiskálního čtvrtletí?
Tok událostí: 1. Případ užítí začíná na konci fiskálního čtvrtletí. 2. systém určuje výši DPH, kterou je třeba odvést státu 3. Systém odesílá elektronickou platbu finančnímu úřadu
Následné podmínky: 1. Finanční úřad přijímá DPH ve správné výši

obr. 67. Specifikace případu užítí

8.2.6 Kdy modelovat případy užítí

Vzhledem k tomu, že případy užítí zachycují funkci systému z hlediska účastníků (aktorů), jsou málo efektivní, má-li systém pouze jednoho uživatele, nebo dokonce nemá-li žádného. Případy užítí zachycují pouze funkční požadavky – tedy nejsou efektivní v systémech, kde převládají nefunkční požadavky. Pro modelování pomocí případů užítí jsou vhodné následující případy:

- Systémy, kde převažují funkční požadavky.
- Systémy s mnoha typy uživatelů, kterým systém poskytuje různé služby (systém má mnoho aktorů).
- Systémy s mnoha rozhraními (systém má mnoho aktorů).

Případy užití nejsou vhodné v případech:

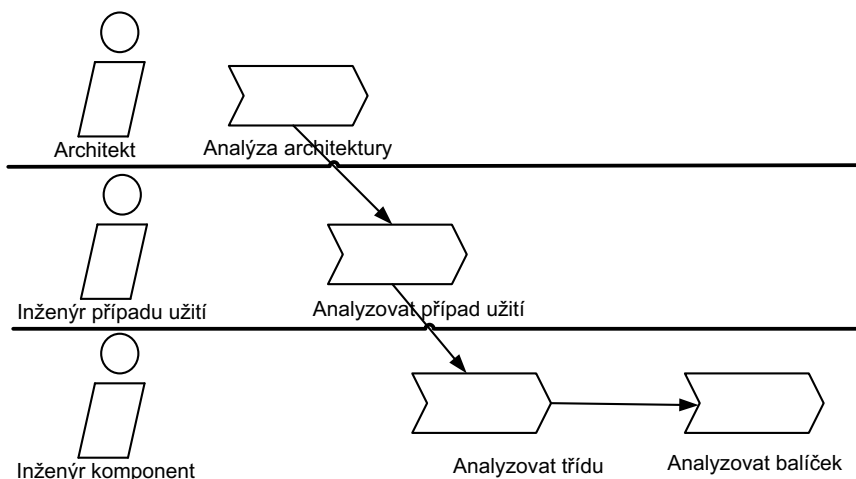
- V systému převládají nefunkční požadavky.
- System má málo uživatelů.
- System má málo rozhraní.

8.3 Analýza

Většina práce týkající se analýzy probíhá na konci fáze Zahájení a zejména ve fázi Rozpracování. Většina prací ve fázi rozpracování se týká tvorby analytických modelů, které zachycují požadované chování tvořeného systému.

8.3.1 Detail pracovního postupu Analýza

Na obrázku 68 je znázorněn pracovní postup analýzy v metodice UP.



obr. 68. Detail pracovního postupu Analýza

8.3.2 Analytický model

Každý systém je jiný. Tedy je nemožné vytvořit obecný analytický model. Přesto platí, že analytický model středně velkého systému se skládá z 50 až 100 analytických tříd. V analytickém modelu se je nutno omezit pouze na třídy, jež jsou součástí slovníku problémové domény. V analytickém modelu se nesmí vyskytnout návrhové třídy (např. třídy pro komunikaci s databází). Cílem je dosáhnout co nejstručnější a co nejjednodušší formulaci struktury a chování analytického modelu. Veškerá implementační rozhodnutí patří do pracovních postupů návrh a implementace.

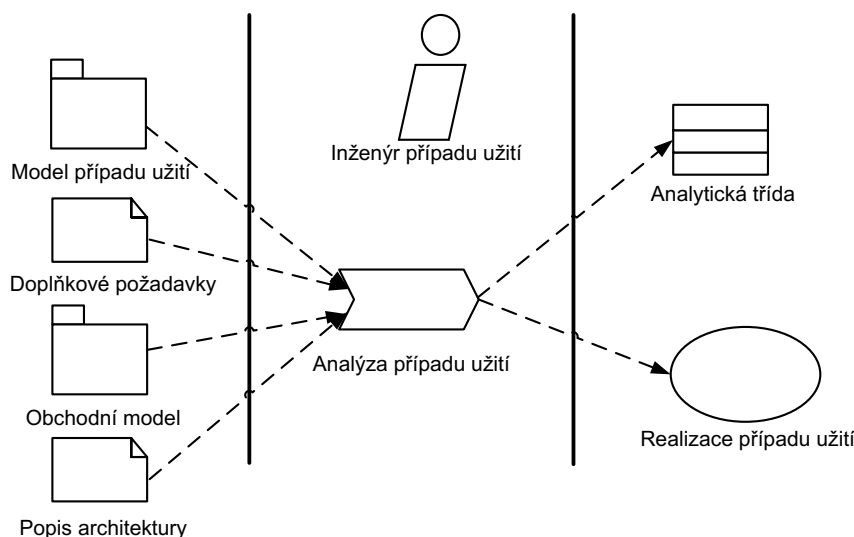
Pro zvýšení šance na vytvoření dobrého analytického modelu je třeba dodržovat následující pravidla:

- Analytický model je vždy tvořen v jazyku obchodní domény. Veškeré abstrakce v analytickém modelu by měly být zahrnuty v slovníku pojmů příslušné domény.
- Každý diagram by měl objasňovat nějakou důležitou část struktury či chování připravovaného systému. Když neobsahuje, je zbytečný.

- Analytické diagramy by měli být zachyceny ve správné perspektivě – neměly by být přesprávně podrobné.
- Je nutno rozlišovat mezi problémovou doménou (business požadavky) a doménou řešení (podrobné úvahy o návrhu a implementaci). Vždy se soustředit pouze na problémovou doménu.
- Minimalizovat vzájemné vazby.
- Pokud existují v modelu přirozené hierarchie, zauvažovat o generalizaci (ISA hierarchii). V analýze generalizaci nepoužíváme jako nástroj znovupoužitelnosti, zachycujeme jí to, že prvky jsou (velmi těsně) příbuzné.
- Snažit se, aby analytický byl užitečný jak pro uživatele, tak pro návrháře a vývojáře – hrozí ignorování modelu.
- Analytický model má být co nejjednodušší a nejsrozumitelnější, ale dostatečně složitý, aby postihnul všechny hlavní oblasti vyvíjeného systému.

8.3.3 Aktivita metodiky UP: Analýza případu užití

Výstupem aktivity „Analýza případu užití“ (viz obrázek 69) jsou analytické třídy a realizace případu užití.



obr. 69. Aktivita metodiky UP: Analýza případu užití

8.3.4 Analytické třídy

Analytické třídy:

- reprezentují abstrakci problémové domény,
- měli by mapovat pojmy skutečného světa.

Za problémovou považujeme tu doménu, z níž vyšel první podnět pro vznik daného systému, obvykle se jedná o specifickou obchodní doménu. Nejdůležitějším aspektem analytické třídy je skutečnost, že by měla jednoznačně mapovat určitý pojem z obchodního světa.

Analytické třídy by měly obsahovat pouze ty nejdůležitější informace. Jsou to název, nejdůležitější atributy (typicky bez typů a viditelnosti), nejdůležitější operace (vyjádřené pouze jako odpovědnosti třídy). Stereotypy a označené hodnoty mají být použity pouze pokud vylepšují model.

Dobrá analytická třída by měla mít tyto vlastnosti:

- Její název odráží její účel,
- je to hrubá abstrakce, která modeluje specifický element problémové domény,
- mapuje jasně identifikovatelnou vlastnost problémové domény,
- obsahuje malou, ale správně definovanou množinu odpovědností,
- je velmi souběžná,
- obsahuje minimum vazeb na ostatní třídy.

Během analýzy se obvykle snažíme přesně, avšak stručně modelovat jeden aspekt problémové domény. Analýza vychází z perspektivy celého systému.

8.3.5 Hledání analytických tříd

Bertrand Meyer v knize *Object Oriented Software Construction* (viz [Meyer 1997]) ukazuje, že neexistuje žádný jednoduchý algoritmus nalezení správných analytických tříd. Kdyby podobný algoritmus existoval, vedl by k postupu návrhu a tvorby objektově orientovaného softwaru, což je ovšem nepravděpodobné.

Přesto existuje několik s úspěchem používaných metod hledání analytických tříd, které si v následujících částech ukážeme.

8.3.5.1 Analýza podstatných jmen a sloves

Analýza podstatných jmen a sloves je jednoduchým způsobem analýzy textu, ve kterém chceme najít třídy, atributy a odpovědnosti. Podstatná jména a fráze v textu vyjádřené podstatnými jmény jsou v podstatě vyjádřením tříd a atributů, zatímco slovesa a slovesné fráze jsou vyjádřením odpovědností nebo operací třídy. Analýza podstatných jmen a sloves se dlouho a úspěšně v praxi používá. Je totiž založena na přímé analýze jazyka problémové domény. Tato metoda má se v určitých situacích může dostat do problémů. Jsou to:

- Možnost výskytu synonym a homonym, což může vést k vytvoření špatných analytických tříd.
- Je nutno problémovou doménu přesně pochopit a definovat.
- Nenalezení skrytých tříd – metoda vychází z analýzy textu a pokud některé třídy v textu nejsou zmíněny explicitně a je předpokládána jejich implicitní znalost, tak je tato metoda nenajde.

8.3.5.2 CRC karty

CRC karty (Class, Responsibilities & Collaborators; třída, odpovědnost a spolupracovníci – viz [Beck 1989]) je dobrou metodou jak zapojit do hledání tříd i uživatele. Lístečky (cca ve velikosti

samolepících štítků³²) mají mít políčka na svůj název, odpovědností (tj. co má třída na starosti) a spolupracovníky (tj. další třídy s kterými daná třída spolupracuje).

CRC karty vznikají na základě spontánní diskuse (tj. každý nápad je dobrý, analýza se provede až později) analytiků, zainteresovaných osob a odborníků v jednotlivých doménách. Každý předmět, pracující v patřičné doméně se zapíše na lísteček (je to kandidát na třídu nebo atribut). Následně se ke každému lístečku naleznou jeho odpovědnosti. Následně se k lístečkům doplní jejich spolupracovníci.

Další fází této metody je analýza získaných informací. Analytici a odborníci v dané doméně rozhodují o tom, zda daný lístek patří skutečně do dané domény, zda z lístečku vznikne třída či atribut, rozdělují vzniklé třídy, či je slučují atd.

8.3.5.3 Další zdroje tříd

Výše vedené metody nemusí být jedinými zdroji analytických tříd v modelu. Existuje i mnoho dalších zdrojů tříd, které by se měli vzít v úvahu. Vzhledem k tomu, že hledáme zobecněné pojmy dané domény, tak můžeme třídy hledat ve skutečném světě. Zdrojem tříd mohou být:

- Fyzické objekty (letadlo, lidé, hotely).
- Kancelář a kancelářské pomůcky (faktura, objednávka).
- Známá rozhraní k vnějšímu světu (obrazovky).
- Klíčové předměty vzhledem k chodu obchodní činnosti (například věrnostní program).

8.3.6 První verze analytického modelu

V tomto okamžiku již máme všechny informace k vytvoření první verze analytického modelu. Postupujeme takto:

- Vzájemně porovnáme všechny analytické třídy získané z různých zdrojů (metoda podstatných jmen a sloves, CRC karty atd.).
- Sloučíme analytické třídy a atributy získané z různých zdrojů a porovnáme je s slovníkem pojmů (sloučíme synonyma, dáme si pozor na homonyma).
- Spolupracovníci z CRC karet nám dají relace mezi třídami.
- Zrevidujeme vznikající analytický model a vylepšíme pojmenování tříd, atributů a odpovědností.

Výsledkem této aktivity by měla být množina analytických tříd, z nichž každá může mít určité atributy, avšak by měla mít 3 – 5 odpovědností. Tímto jsme získali první verzi analytického modelu.

8.3.7 Analytické balíčky

V jazyku UML je balíček (package) abstrakcí sdružování. Balíček je univerzálním mechanismem uspořádání elementů a diagramů do skupin. Pomocí něho lze realizovat následující úlohy:

³² Ne větší, třída by pak mohla být příliš mnoha odpovědnostmi. Pokud mi lísteček nestačí a potřebuji doplnit ještě nějakou odpovědnost objektu, pak patrně správné řešení je rozdělit tuto třídu na několik dalších.

- Seskupování sémanticky souvisejících element,
- definice „sémantické hranice“ uvnitř modelu,
- tvorbu jednotek správy konfigurace,
- v etapě návrhu poskytují balíčky jednotky pro souběžnou správu,
- balíčky definují jmenný prostor – v rámci balíčku musí být jména elementů jedinečná.

Každý modelovaný element je vlastněn jediným balíčkem. Balíčky utvářejí hierarchii.

Analytické balíčky mohou obsahovat:

- Případy užití,
- analytické třídy,
- realizaci případů užití.

8.3.7.1 Analýza architektury

V analýze architektury jsou všechny třídy uspořádány do množiny soudržných analytických balíčků jež jsou dále strukturovány v oddílech a vrstvách. Jedním z cílů analýzy architektury je dosáhnout minimalizaci vazeb uvnitř modelovaného systému. Toho lze dosáhnout těmito způsoby:

- minimalizací závislostí mezi balíčky,
- minimalizací počtu veřejných elementů v balíčcích,
- maximalizací soukromých členů v balíčcích.

Redukce propojení a vzájemných vazeb je jedním z nejdůležitějších úkolů analýzy architektury, protože systémy s vysokým stupněm propojení jsou natolik složité, že to ztěžuje jak implementaci, tak i následnou údržbu systému.

Analytické balíčky lze specifikovat hledáním skupin modelovaných elementů, jež se vyznačují silnou sémantickou soudržností. Zdrojem pro hledání analytických balíčků je zejména statický analytický model. Dalším zdrojem může být rovněž model případu užití.

Po nalezení několika kandidátů na analytické balíčky, je nutno pokusit se minimalizovat vazby mezi balíčky. To lze pomoci:

- Přesouvání tříd mezi balíčky,
- přidáváním nových balíčků,
- odstraněním balíčků.

Mezi balíčky nesmí vznikat vzájemné kruhové závislosti. Odstranění kruhových závislostí lze dosáhnout slučováním balíčků a případným novým rozdělením na balíčky.

Klíčem k úspěšné tvorbě dobré struktury balíčků je vysoký stupeň soudržnosti uvnitř balíčku a minimalizace vazeb mezi balíčky.

Model analytických balíčků by měl být co nejjednodušší. V praxi je nejlépe použít pět až deset analytických tříd na balíček.

8.3.8 Aktivita metodiky UP: Analýza případu užití

V předchozích částech jsme si ukázali, jak jsou vytvářeny analytické třídy, což je jeden z výstupních artefaktů aktivity analýza případu užití. Druhým výstupem z této aktivity jsou realizace případů užití.

Analytické třídy modelují statickou strukturu systému, zatímco realizace případů užití se zaměřují na vzájemnou spolupráci mezi objekty systému. Je to tedy dynamický pohled na systém.

8.3.9 Realizace případu užití

Realizace případů užití je proces upřesňování. Postupně se berou jednotlivé aspekty chování systému, jak jsou zachyceny v případech užití a přidružených doplňujících požadavcích, a následně se modeluje způsob, jak dané chování realizovat pomocí spolupráce a interakce jednotlivých instancí analytických tříd.

Nejdůležitějším výstupem realizace případu užití je zachycení, jakým vzájemným způsobem budou analytické třídy mezi sebou komunikovat. Prostředkem k tomu jsou diagramy interakce – tj. diagram spolupráce (collaboration diagram) a sekvenční diagram (sequence diagram).

8.3.9.1 Diagramy interakce

Vysvětlení diagramům spolupráce a sekvenčnímu diagramu jsme se podrobně v kapitolách odkaz a odkaz, proto se zde o nich pouze krátce zmíníme.

Tyto diagramy modelují spolupráci a interakce mezi objekty, které uskutečňují případ užití nebo jeho část. Ve skutečnosti existuje společný model objektových interakcí, který má dva pohledy:

- Diagramy spolupráce, které zdůrazňují strukturální relace mezi objekty a jsou zejména užitečné během analýzy, při tvorbě náčrtů spolupráce jednotlivých objektů.
- Sekvenční diagramy zdůrazňují časovou posloupnost zpráv předávaných mezi objekty. Tento pohled je výhodnější z hlediska uživatelů. Při složitějších interakcích se však sekvenční diagramy brzo přeplní.

Tyto diagramy jsou vzájemně izomorfní – přeci vyjadřují pohled na stejný model UML.

8.3.10 Diagramy aktivit

Diagramy aktivit jsou „objektově orientovanými diagramy toků“. Díky nim lze modelovat jakýkoliv proces jako kolekci aktivit a přechodů mezi nimi. Diagram aktivit lze připojit k libovolnému modelovanému elementu a umožní nám modelovat jeho chování. Diagramy aktivit jsou obvykle připojeny k:

- případům užití,
- třídám,
- rozhraním,
- komponentám,
- uzlům,
- spolupracím,

- operacím a metodám.

Pomocí diagramů aktivit se častokrát modelují business (podnikatelské a správní) procesy.

8.4 Návrh

Při vytváření návrhového modelu je třeba vzít skutečnost, že se návrhový model vytváří evolučně z analytického. Je třeba se rozhodnout, zda mají být oba modely uchovávány odděleně. Mohu zvolit několik strategií:

- Pokud analytický model přetransformuji do návrhového, tak ztratím analytický pohled
- Pokud analytický model přetransformuji do návrhového a pomocí CASE nástroje skryji návrhové podrobnosti, tak analytický pohled nemusí být dostačující.
- Pokud analytický model na určitém stupni vývoje prohlásím za finální, tak se mi oba pohledy časem rozsynchronizují.
- Pokud budu udržovat dva modely – analytický a návrhový odděleně (a odděleně je vyvíjet) a při změnách v nich provádět synchronizaci, tak sice budu mít dva synchronizované modely, ale jejich údržba bude složitá.

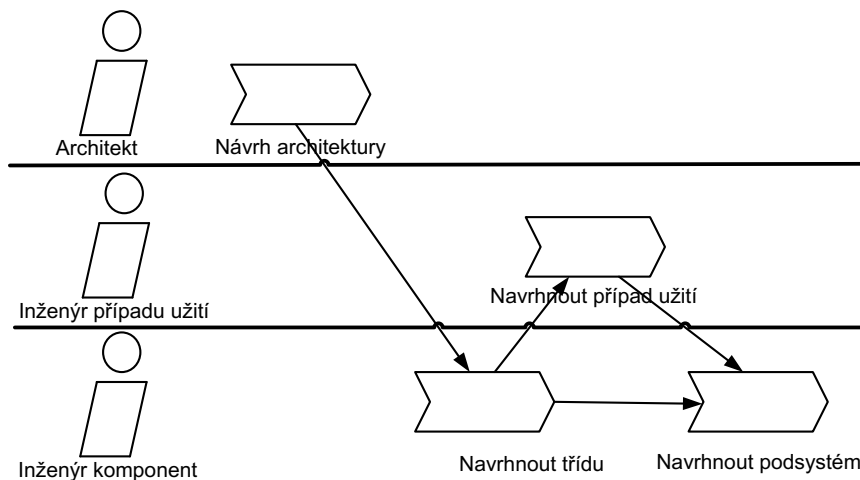
Strategii jakou použiji při transformaci analytického modelu na návrhový záleží v největší míře na složitosti projektu a zda je potřeba zachovat analytický pohled na systém. Analytický model má výhodu že je mnohem jednodušší a přehlednější než návrhový – analytický pohled může obsahovat jen 1 až 10 % tříd obsažených v podrobném návrhovém modelu. Zachování analytického modelu umožňuje lepší :

- Za pojení nových lidí do projektu,
- pochopení funkcí systému dlouhou dobu po implementaci,
- pochopení způsobu, kterým systém uspokojuje uživatelské požadavky,
- plánování údržby a rozšíření,
- pochopení logické struktury architektury systému,
- možnost externí tvorby systému.

Pokud se projektu týká některý z výše uvedených bodů, tak je dobré analytický model zachovat. Je-li systém malý (do cca 200 tříd) je možno analytický pohled nezachovávat a použít první nebo druhou strategii. Pokud se modelují komplexní systémy a systémy strategického významu je nutno analytický model zachovat.

8.4.1 Detail návrhu

Pracovní postup metodiky UP je znázorněn na obrázku 70. Hlavními účastníky jsou architekt, inženýr případu užití a inženýr komponent.



obr. 70. Detail návrhu

Návrhový model lze považovat za rozpracovanou formu analytického modelu, do něhož jsou přidány detaily a specifická technická řešení. Návrhový model obsahuje stejné typy předmětů jako analytický model, ale veškeré elementy jsou již plně vytvořeny a musí obsahovat i veškeré detaily. Návrhové modely se skládají:

- z návrhových podsystémů,
- z návrhových tříd,
- z rozhraní,
- z realizací případu užití – návrh,
- a z diagramu nasazení (deployment diagram).

V této fázi se tvoří první verze diagramu nasazení, ale většina prací na něm je až ve fázi implementace.

8.4.2 Návrhové třídy

Návrhové třídy jsou třídy, jejichž specifikace je na takovém stupni, že je lze implementovat.

Návrhové třídy lze získat ze dvou zdrojů:

- Z problémové domény prostřednictvím upřesňování analytických tříd – přidáním implementačních detailů, či přepracováním analytických tříd do více návrhových (například aplikací návrhových vzorů – více o návrhových vzorech viz [Gamma et al. 1995]).
- Z domény řešení – tj. použitím knihovnických tříd a znovupoužitelných komponent. Sem patří také třídy spadající do prostřední vrstvy třívrstvého schématu aplikace (tj. vrstvy aplikačního serveru), databáze, uživatelské rozhraní atd.

Analýza spočívá v modelování toho, co by systém měl dělat, návrh spočívá v tom, jakým způsobem má být toto chování implementováno.

8.4.2.1 Anatomie návrhové třídy

Návrhová třída musí obsahovat:

- Kompletní sadu atributů a jejich detailní specifikaci včetně názvu, typu, viditelnosti a nepovinně i implicitní hodnoty.
- Převod operací v analytické třídě na úplnou sadu metod (s viditelností, parametry, typem návratové hodnoty).

8.4.2.2 Správně formulované návrhové třídy

Návrhové třídy, aby byly správně navrženy, musí splňovat následující podmínky. Musí být:

- Úplné a dostačující – úplnost je podmíněna tím, zda třída poskytuje svým klientům vše, co se od ní žádá. Dostatečnost naopak slouží k ujištění, že metody třídy jsou zcela zaměřeny na realizaci zaměřeného úkolu. Říkáme tedy, že třída by měla dělat to, co se od ní očekává – nic míň a nic víc.
- Jednoduchá – metody by měly být konstruovány, aby poskytovaly jednoduchou nedělitelnou službu. Třída by neměla poskytovat víc způsobů vykonání jedné úlohy.
- Vysoce soudržná – každá třída by měla modelovat jeden jednoduchý abstraktní pojem a měla by obsahovat pouze metody, které odpovídají účelu třídy.
- Bez těsných vazeb – každá třída by měla být přidružená pouze k tolika dalším třídám, aby to umožnilo realizaci toho, za co je třída odpovědná.

8.4.2.3 Agregace nebo dědění

Při tvorbě návrhového modelu se musíme také rozhodnout jak implementovat vztah generalizace z analytického modelu. To jde pomocí agregace (skládání) nebo dědění. Co si vybrat záleží na okolnostech. Proti dědění mluví:

- Je to nejsilnější známá forma vazby mezi třídami.
- Zapouzdření je v rámci hierarchie velmi slabé – veškeré změny v nadtřídě se projeví i v podtřídách – to může mít velký dopad na systém (tzv. křehká nadtřída).
- Dědění je velmi nepružný typ relace. Vztah dědičnosti nelze za běhu v nejpoužívanějších objektově orientovaných jazycích měnit (je to statická vazba). Hierarchie založené na agregaci jsou mnohem pružnější.

8.4.3 Upřesnění analytických relací

Ve fázi návrhu musíme upřesnit analytické asociace mezi třídami. Všechny návrhové asociace musí mít:

- řiditelnost,
- násobnost na obou koncích,
- všechny návrhové asociace by měli obsahovat na obou koncích role.

Upřesnění analytických asociací do návrhových asociací se skládá z několika postupných kroků:

- Upřesnění vhodných asociací do agregace nebo kompozice,
- implementace asociačních tříd,
- implementace asociací typu 1:N,
- implementace asociací typu M:N,
- implementace obousměrných asociací.

8.4.4 Rozhraní a podsystémy

8.4.4.1 Rozhraní

Rozhraní (interface) specifikuje pojmenovanou množinu operací. Jeho podstata spočívá v oddělení specifikace funkčnosti (rozhraní) od fyzické implementace (např. pomocí třídy nebo podsystému). Pomocí rozhraní specifikujeme dohodu, kterou musí prvek implementující rozhraní dodržovat. Obvykle se říká, že rozhraní definuje služby nabízené danou třídou, podsystémem nebo komponentou. Rozhraní jsou klíčem k tvorbě komponentového softwaru

Rozhraní se z pohledu architektury stává nesmírně důležitým pojmem především v etapě návrhu, protože umožňuje definovat poskytuje „zásuvky a zástrčky“. Ty umožňují propojit návrhové podsystémy, ale nevyžadují propojení jednotlivých tříd v nich definovaných. Každá operace v rozhraní musí obsahovat:

- Úplnou signaturu operace (název, typy argumentů, návratový typ),
- sémantiku operace – tu lze zaznamenat jako text nebo pseudokód,
- nepovinně stereotyp a množinu omezení.

Rozhraní nesmí obsahovat:

- atributy,
- implementaci operací,
- vztahy říditelné z rozhraní.

Rozhraní je vlastně čistě abstraktní třída³³ (všechny operace jsou abstraktní) bez atributů.

8.4.4.2 Podsystémy

Podsystém je balíček s předdefinovaným stereotypem <<subsystem>>. Návrhové podsystémy obsahují:

- Návrhové třídy a návrhová rozhraní,
- realizace případů užití,
- další podsystémy,
- specifikační elementy jako případy užití.

³³ V C++ je rozhraní takto implementováno.

Podsystemy se používají:

- K osamostatnění návrhových okruhů,
- k vyjádření rozsáhlých komponent,
- k zapouzdření starších systémů.

Návrhové podsystemy jsou způsobem jak v modelu zavést komponenty.

8.4.5 Realizace případů užití – návrh

Realizace případu užití v etapě návrhu spočívá ve spolupráci návrhových objektů a tříd, které slouží k realizaci případu užití. Realizace případu užití v návrhu specifikuje implementační rozhodnutí a implementuje nefunkční požadavky. Skládá se z:

- Návrhových diagramů interakce (typicky se používá sekvenční diagram),
- diagramů tříd znázorňující zúčastněné návrhové třídy

V analýze jsme se při realizaci případů užití zaměřovali na to, co má třída dělat, v návrhu nás zajímá způsob, jakým to systém udělá. Musíme tedy specifikovat implementační detaily. Realizace případů užití ve fázi návrhu je tedy mnohem podrobnější a složitější.

8.4.6 Stavové diagramy

Stavové diagramy jsou důležitou pomůckou pro modelování dynamického chování reaktivních objektů. Výše zmíněné diagramy aktivit jsou speciálními případ stavových diagramů. Liší se zejména použitím a bohatostí syntaxe (stavový diagram má bohatší syntaxi). Diagramy aktivit se používají zejména při modelování business procesů, jichž se účastní několik objektů. Stavové diagramy se používají k modelování životního cyklu jednoho reaktivního objektu. Reaktivní objekty:

- reagují na vnější události,
- jejich životní cyklus je modelován jako řada stavů, přechodů a událostí,
- jejich chování je důsledkem předchozího chování.

Stavový diagram obsahuje právě jeden stavový automat pro jeden reaktivní objekt.

V objektově orientovaném modelování můžeme použít stavové automaty k modelování dynamického chování reaktivních objektů. Jsou to:

- třídy,
- případy užití,
- podsystemy,
- celé systémy.

Stavové automaty se nejčastěji používají pro modelování dynamického chování tříd.

8.5 Implementace

Pracovní postup Implementace začíná již ve fázi rozpracování, ale hlavní důraz je na něj kladen až ve fázi Konstrukce (viz 61).

Implementace spočívá v převodu návrhového modelu do spustitelného kódu. Z pohledu analytika nebo návrháře je smyslem implementace tvorba požadovaného implementačního modelu. Tento model zahrnuje rozdělení návrhových tříd do komponent.

Pracovní postup Implementace je zaměřen hlavně na tvorbu spustitelného kódu. Vedlejším produktem této aktivity může být implementační model, přestože tento model není typicky výsledkem explicitní modelovací aktivity. Modelování implementace je typicky necháno plně v rukou programátora. Existují však případy, kdy explicitní modelování implementace může být nesmírně důležité:

- Chceme-li generovat kód přímo z modelu, pak se musí v modelu explicitně definovat detaily, jako jsou zdrojové soubory a komponenty.
- Vytváříme-li komponentový software, je rozdělení na komponenty a umístění jednotlivých tříd a rozhraní věc strategicky důležitá a měl by ho vytvářet analytik a ne programátor.

8.5.1.1 Detail pracovního postupu Implementace

Z obrázku 70 vyplývá, že pracovní postup Implementace zahrnuje architekta, systémového integrátora a inženýra komponent.

Klíčovým artefaktem implementace je z pohledu analytika nebo návrháře implementační model. Tento model se skládá ze dvou typů diagramů:

- Diagramu komponent – tento diagram modeluje závislosti mezi softwarovými komponentami, jež utvářejí systém, a z
- diagramu nasazení – který modeluje fyzické výpočetní uzly, v nichž bude software nasazen, a relace mezi nimi.

8.5.1.2 Komponenty

Podle knihy *The Unified Modeling Language Reference Manual* (viz [Rumbaugh et al. 1998]) je komponenta „fyzickou nahraditelnou částí systému, která obsahuje realizaci množiny specifikovaných rozhraní“. Následující artefakty mohou být komponentami:

- Zdrojové soubory,
- implementační podsystémy,
- ovládací prvky ActiveX,
- • objekty standardu JavaBeans,
- • servlety jazyka Java
- • a jiné objekty založené na komponentových technologiích.

Každá komponenta může obsahovat mnoho tříd a implementovat mnoho rozhraní. Komponentový model ukazuje způsob, jakým jsou třídy a rozhraní přiřazovány ke komponentám.

8.5.1.3 Nasazení

Diagram nasazení (deployment diagram) ukazuje nejen fyzický hardware, na němž bude softwarový systém spuštěn, ale i způsob, jímž je software na tomto hardwaru nasazen.

Existují dvě verze diagramu nasazení:

- Diagram nasazení (descriptor form deployment diagram) – obsahuje komponenty, uzly a vztahy mezi jednotlivými uzly- Uzel reprezentuje typ hardwaru (např. PC) a komponenta typ softwaru.
- Diagram konkrétního nasazení (instance form deployment diagram) – obsahuje instance uzlů, relace mezi nimi a instance komponent. Instance uzlu zastupuje konkrétní hardware (například počítač v kanceláři 214) a instance komponenty zastupuje konkrétní (nainstalovanou) kopii softwaru. Častokrát (pokud nás nezajímá specifická instance) používáme anonymní instance.

Přestože o diagramu nasazení hovoříme jak o implementační aktivitě, vzniká první verze diagramu nasazení obvykle již v etapě návrhu – jako součást procesu týkajícího se návrhu hardwarové architektury. Tuto verzi diagramu později (při implementaci) upřesňujeme do jednoho nebo více diagramů konkrétního nasazení.

9 Metamodelování

V poslední době se objevilo nové „moderní“ slovo, nebo spíše předpona - meta. Provádíme operace nad metadaty. Metamodelujeme a tím vytváříme metamodel v meta-CASE nástroji, který je řízen metamodelem. Z metadat generujeme programy. Dokonce můžeme i metaprogramovat v metaprogramovacím jazyce.

Abychom pochopili význam slova je dobré se nejdříve podívat do encyklopedie a tam zjistíme, že meta- pochází z řečtiny a mimo jiné znamená za, po, vně, mimo. Další význam z hlediska informačních technologií najedeme na www.metamodel.com a říká nám, že meta- znamená o jeden stupeň abstrakce výše tj metadata jsou data popisující data, metamodel je model popisující model.

9.1 Architektura pro metamodelování

Klasický rámec pro metamodelování je podle [OMG 2002a] založen na čtyřvrstvé architektuře. Tyto vrstvy jsou obvykle popisovány takto:

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

Na obrázku č. 71 je ukázán klasický čtyřvrstvý rámec pro metamodelování. Příklad ukazuje metadata a metamodel pro jednoduchý příklad skládání referenta s automobilem. Další vrstva (metamodelu) obsahuje meta-metada a popisuje co je to *Trida*, *Skladani*, *Atribut* a jejich vztahy pomocí termínů z vrstvy meta-metamodelu (*MetaClass*, *MetaAssociation* aj.). Vrstva meta-metamodelu definuje metamodelovací konstrukce (*MetaClass* atd.).

9.2 Metamodelovací prostředky

Metamodelovací metody definují rámec pro metamodelování, který obvykle obsahuje definici meta-metamodelu a metamodelovacího jazyku, pomocí kterého je definován metamodel. Pro potřeby metamodelování v informačním a softwarovém inženýrství bylo vyvinuto mnoho přístupů pro tvorbu metamodelu - COMMA, GOPRR, MOF, OPRR, CoCoA, NIAM, COOM atd.. V dalších odstavcích jsou popsány tři metamodelovací rámce, které jsou vhodné pro definici metodik.

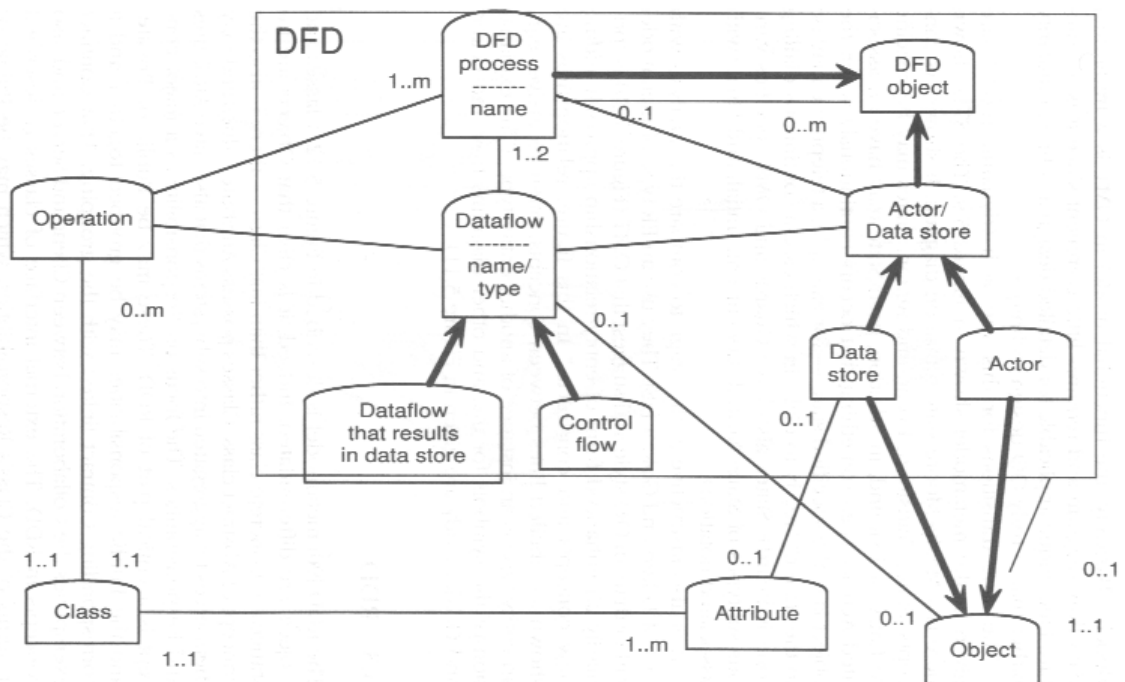
9.2.1 COMMA

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

COMMA používá tyto základní pojmy:

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

Hlavním výsledkem projektu COMMA je vytvoření velmi jednoduchého (ale mocného) objektově orientovaného metamodelovacího jazyka. Nevýhodou je, že tento projekt již skončil a neexistuje napojení na CASE nástroje.



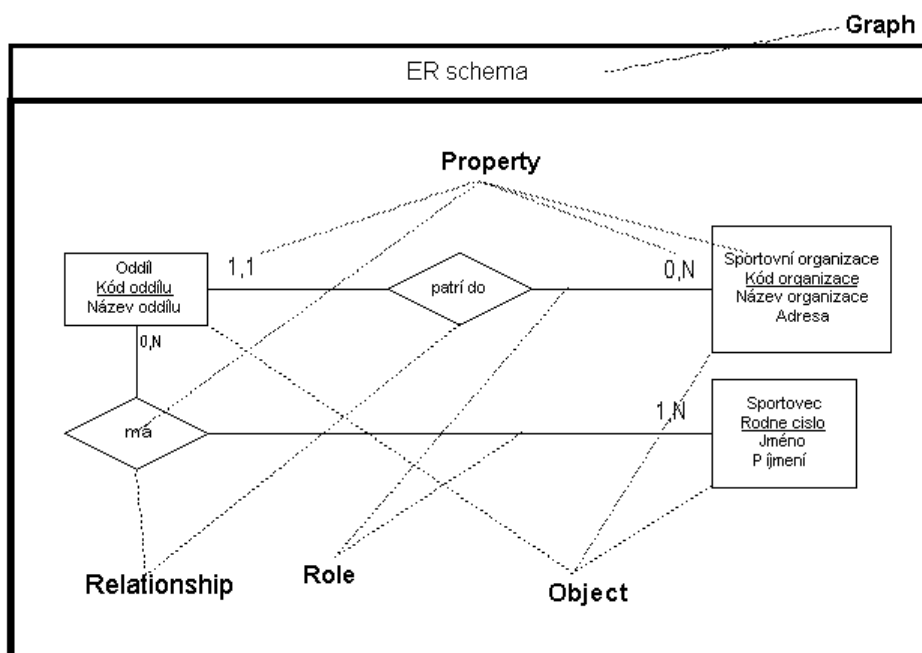
obr. 72. COMMA

9.2.2 GOPRR

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

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

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



obr. 73. GOPRR příklad - obrázek

9.2.3 Rodina standardů OMG

Standardy OMG (Object Management Group) jsou založeny na čtyřvrstvé architektuře popsané v odstavci 2. Vrstvu meta-metamodelu popisuje standard MOF (Meta Object Facility – viz [1]). OMG definuje několik metamodelů založených na MOF:

- metamodel UML (Unified Modeling Language) – standard pro objektový modelovací jazyk (více [2]),
- metamodel IDL (Interface Definition Language) – standard popisující objektová rozhraní tříd pro standard distribuovaných objektů CORBA, a jejich mapování do různých programovacích jazyků.
- metamodel CDW (Common Data Warehouse) – standard definující architekturu datových skladů.

Data mezi metamodely založenými na MOF mohou být vyměňována pomocí formátu XMI (XML Metadata Interchange).

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

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

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

9.3 Použití metamodelování v praxi

Metamodelování se využívá zejména pro popis a tvorbu nových metodologií, při implementaci metodologií v CASE nástrojích, pro strukturování repositářů, při integraci systémů, při generování programů z modelů, generování reportů, a kontrole modelů. Znalost metamodelu lze také použít pro flexibilní návrh informačního systému pomocí generických modelů.

9.3.1 Využití metamodelu při popisu metodologií

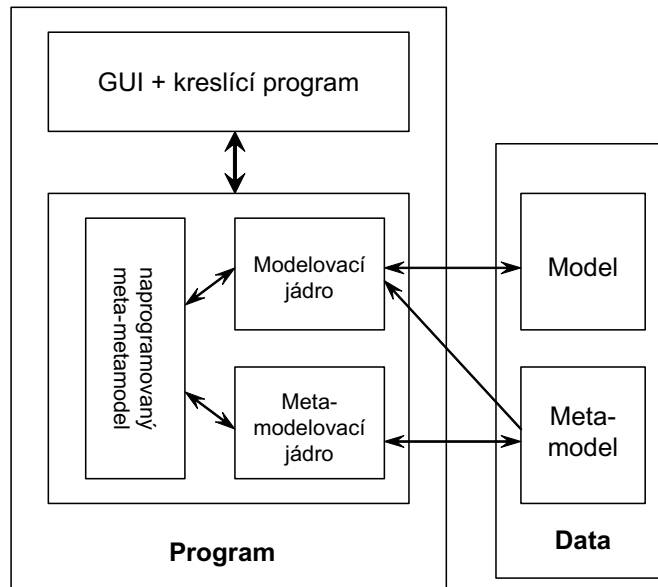
Pomocí metamodelu definujeme metodologie. Každá metodologie je definována pomocí svého metamodelu. Ten může být formálně nebo neformálně popsány. Neformální popisy metamodelů metodologií jsou např. uvedeny v učebnicích metodologií. Například se v každé učebnici UML dočteme, že třída se značí obdélníkem se třemi poli, v kterých jsou údaje jako jméno třídy, atributy třídy a její metody. Dále se dočteme, že asociace je vztah mezi třídami a jak se zakresluje plnou čarou. Pro hlubší pochopení a případnou počítačovou podporu modelování však potřebujeme formalizovaný popis metamodelu (třeba viz [OMG 2003]).

Metamodelování je základem vývoje nových metodologií (ME – Method Engineering). Poskytuje také prostředky k nalezení společných vlastností různých metodologií, jejich standardizaci a případnou možnost jejich vzájemné kooperace.

9.3.2 CASE a Meta-CASE nástroje

Jedny z prvních aplikací metamodelu byly CASE nástroje. Metodologie jsou definovány pomocí metamodelu a CASE nástroje je musí implementovat. Mezi další použití metamodelu u CASE nástrojů patří formáty na výměnu dat mezi CASE nástroji – starší CDIF a na XML založený XMI.

V pozdější době se objevily tzv. Meta-CASE (neboli také CAME- Computer Aided Method Engineering) nástroje (například MetaEdit+), které umožňují si definovat vlastní metamodel a tím definovat vlastní metodologii.



obr. 74. Schéma Meta-CASE nástroje

Obrázek 2 –schéma Meta-CASE nástroje

Na obrázku č. 74 je schéma Meta-CASE nástroje. Hlavní části jsou:

- Kreslicí jádro + GUI – stará se o vlastní kreslení a interakci s uživatelem.
- Naprogramovaný meta-metamodel – je to interpret jazyka meta-metamodelu.
- Modelovací jádro – je programováno pomocí metamodelu, vytváří vlastní model.
- Metamodelovací jádro – pomocí něho programujeme, v jazyce definovaném meta-metamodelem, metamodel. Toto jádro je hlavním rozdílem oproti klasickým CASE nástrojům, umožňuje nám programovat vlastní metodiky.
- Model – data modelu, typicky ukládaná do databáze.
- Metamodel – data meta-metamodelu, typicky ukládaná do databáze.

Meta-CASE nástroje někdy nazýváme metamodelem řízené programy, protože změnou metamodelu je můžeme přeprogramovat.

9.3.3 Metamodel při zpracování dat a metadat

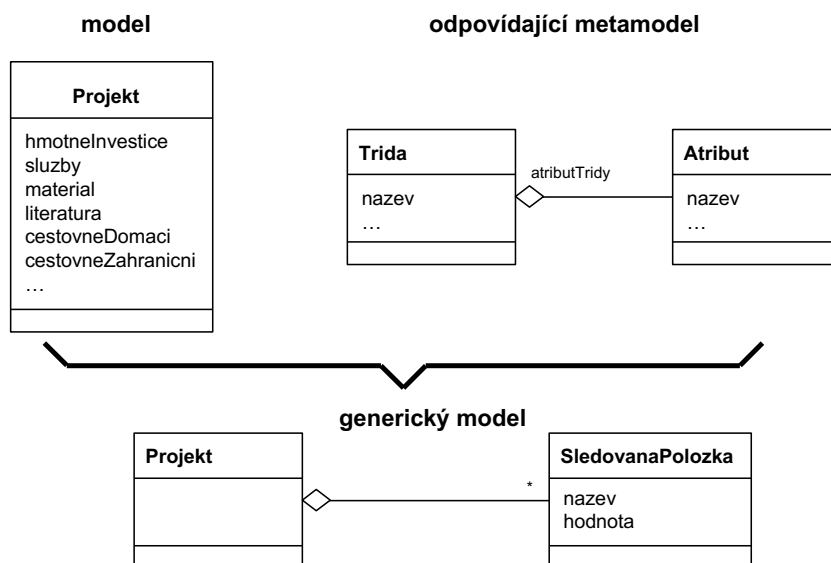
Znalostí metamodelu dat umožňuje přesně pochopit strukturu dat a následně s nimi pracovat. Typickým příkladem jsou různé repositáře metadat. Jedno z prvních použití metamodelu se týkalo repositářových standardů PCTE a ISO standard IRDS (Information Resource Dictionary Standard). Pokud máme metadata uložena v repositáři (databázi) je možno tyto data snadno dále zpracovávat. Typickým příkladem takového zpracování je zjišťování konzistence, měření metrik, generování reportů a programů.

Metamodelování dovoluje popsat jednotným způsobem odlišné datové struktury více systémů, což poskytuje možnost je skládat dohromady ve vyšší systém, který dokáže pracovat s daty systémů, ze

kterých je sestaven a tím pomoci k jejich integraci. A naopak metamodel poskytuje dobrou abstrakci systému a pomáhá nám k dobrému rozdělení systému na podsystémy a tím pomáhá zvládat komplexní projekty.

9.3.4 Generický model

Pro optimalizaci návrhu projektu je možno použít generický model (viz [Carda 2003]). To je takový model, který v sobě kombinuje vlastnosti modelu a metamodelu. Více na obrázku č. 75.



obr. 75. Generický model

Na obrázku je třída Projekt, která je součástí „standardního“ návrhu informačního systému. Instance této třídy mají mnoho atributů (sluzby, material, literatura, cestovne atd.) týkajících se vyúčtování peněz v projektu. Tento návrh je jistě funkční, ale problém nastane pokud původně navržené atributy přestanou pokrývat naše potřeby (např. potřebujeme evidovat zvlášť domácí a zahraniční cestovné). Další nevýhodou tohoto návrhu je, že obvykle spousta atributů bude nulová (tj. atributy nám mohou chybět a zároveň přebývat). Elegantním řešením je vytvoření třídy SledovanaPolozka a její skládání (1:n) s třídou Projekt. Tím jsme dostali model, který má i vlastnosti metamodelu. Struktura metamodelu tedy slouží jako příklad jak postupovat při modelování.

Předností generických modelů je v tom, že vedou ke konstrukci snadno modifikovatelného software, a to dokonce i takovým způsobem, který nebyl očekáván při analýze.

9.4 Metamodelování – shrnutí

V současné době se metamodelování používá pro stále větší počet úloh. Mezi jeho nejdůležitější úkoly patří zajištění interoperability metod, podpora vytváření nových metodologií (ME – Method Engineering) pomocí metaCASE nástrojů, integrace systémů a dat a generování zdrojových textů z modelu. Pro tyto úkoly má mnoho metamodelovacích prostředků, z nichž zejména standardy organizace OMG (tj. metamodel UML, MOF) jsou velmi perspektivní.

10 Použitá literatura

10.1 Publikace autorů

- [Carda et al. 2003] Carda A., Merunka V., Polák J.: *Umění systémového návrhu*, Grada 2003, ISBN 80-247-0424-2
- [Hall et al. 2004] Hall J. et al.: *Accounting Information Systems 3rd edition*, South-Western Publishing, 2004, ISBN 0538877960. (spoluautoři kapitoly 4. – System Development Activities)
- [Knott et al. 2000] Knott, Roger P.; Merunka, Vojtěch; Polák, Jiří: *Process Modeling for Object Oriented Analysis using BORM Object Behavioral Analysis*, in Proceedings of Fourth International Conference on Requirements Engineering ICRE 2000, Chicago 2000. IEEE Computer Society Press ISBN 0-7695-0565-1
- [Knott et al. 2003] Knott, Roger P.; Merunka, Vojtěch; Polák, Jiří: "The BORM methodology: a third-generation fully object-oriented methodology", Knowledge-Based Systems 3(10) 2003, Elsevier Science Publishing, New York.
- [Liu et al. 2005] Liu L., Roussev B. et al: *Management of the Object-Oriented Development Process*, Virgin Island 2005, ISBN 1-59140-605-6 (spoluautoři kapitoly 15. – The BORM Methodology)
- [Merunka et al. 2000] Merunka, V.; Polák, J.; Kofránek J.: *Úvod do metody BORM*, ve sborníku konference Objekty 2000, ISBN 80-213-0682-3
- [Merunka 2001] Merunka V.: *Zkušenosti s objektovou analýzou a návrhem*, ve sborníku konference Objekty 2001, ISBN 80-213-0829-X
- [Merunka 2002A] Merunka V.: *Objekty v databázových systémech*, skriptum ČZU Praha, Praha 2002. ISBN 80-213-0318-2
- [Merunka 2002B] Merunka V.: *Řídící a podpůrné procesy v objektově orientované tvorbě softwaru*, ve sborníku konference Objekty, listopad 2002 Praha, ISBN 80-213-0947-4
- [Merunka 2003] Merunka V.: *Objektový databázový systém Gemstone*, sborník konference OBJEKTY 2003. Ostrava 2003. ISBN 80-248-0274-0
- [Merunka et al. 2004] Merunka V., Pergl R., Pícka M., *Objektově orientovaná tvorba software*, ČZU Praha, 2004, ISBN 80-213-1159-2

10.2 Ostatní

- [Abadi 1996] Abadi M., Cardelli L.: *A Theory of Objects*, Springer 1996, ISBN 0-387-94775-2
- [Agha et al. 1994] Agha Gul, Hewitt Carl: *Actors - A Conceptual Foundation for Cocurrent Object-Oriented Programming*, pp 49-74, Research in OOP 1994, MIT Press ISBN 0-262-19264-0
- [Ambler 1997] Ambler Scott: *Building Object Applications That Work, Your Step-By-Step Handbook for Developing Robust Systems Using Object Technology*, Cambridge University Press/SIGS Books, 1997, ISBN 0521-64826-2
- [Ambler 1998] Ambler, Scott W.: *Process Patterns - Building Large-Scale Systems Using OO Technology*, Cambridge University Press - Managing Object Technology Series 1998, ISBN 0-521-64568-9
- [Ambler 1999] Ambler, Scott W.: *More Process Patterns - Delivering Large-Scale Systems Using OO Technology*, Cambridge University Press - Managing Object Technology Series 1999, ISBN 0-521-65262-6
- [Ambler et al. 2004] Ambler Scott, Beck Kent: *Object Orientation – Bringing data professionals and application developers together*, <http://www.agiledata.org/essays/objectOrientation101.html>, prosinec 2004
- [Ambler 2005] Scott W. Ambler, John Nalbone, and Michael Vizdos, *The Enterprise Unified Process*, Prentice Hall PTR, 2005, ISBN 0-13-191451-0
- [AP 2001] *AP manifesto*, <http://www.agilealliance.org>, December 2004
- [Arlow 2003] Jim Arlow and Ila Neustadt: *UML and the Unified Process: Practical Object-Oriented Analysis and Design*. Addison Wesley. 2003
- [Barry 1998] Barry D.: *The Object Database Handbook: How to Select, Implement, and Use Object-Oriented Databases*, ISBN 0471147184
- [Bartoška et al. 2004] Bartoška Jan, Adámková Milena, Hnátková Běla, *Tvorba softwaru nad biometrickou strukturou dat*, ve sborníku konference Objekty 2004
- [Beck 1989] Kent Beck and Ward Cunningham: *A Laboratory for Teaching Object-Oriented Thinking*. In *Proceeding of OOPSLA 89*. SIGPLAN Notices, Vol.24, No. 10, pp 1-6
- [Beck 1997] Beck K.: *Smalltalk Best Practice Patterns*, Prentice Hall 1997, ISBN 0-13-476904-X
- [Beck 2002] Beck K.: *Extrémní programování (český překlad)* Grada 2002, ISBN 80-247-0300-9
- [Beck 2003] Beck K.: *Agile Database Techniques - Effective Strategies for the Agile Software Developer*, John Wiley & Sons; 2003, ISBN 0471202835
- [Bellin et al. 1997] Bellin, David; Simone, Susan Suchman: *The CRC Card Book*, Addison-Wesley 1997 ISBN: 0201895358
- [Bertino et al. 1995] Bertino E., Martino L.: *Object-Oriented Database Systems - Concepts and Architectures*, Addison Wesley 1995, ISBN 0-201-62439-7
- [Biermann 1990] Biermann Alan W.: *Great Ideas in Computer Science*, MIT Press 1990, ISBN 0-262-02302-4
- [Blaha et al. 1995] Blaha M., Premerlani M.: *Object-Oriented Modeling and Design for Database Applications*, Prentice Hall 1995, ISBN 0-13-123829-9
- [Booch 1994] Booch Grady: *Object-Oriented Analysis and Design with Applications*, Second Edition, Benjamin Cummings 1994, ISBN 0-8053-5340-2
- [Booch 1996] Booch, Grady: *Object Solutions: Managing the Object-Oriented Project*, Addison-Wesley, 1996, ISBN 0805305947
- [Buchalceková 2005] Buchalceková A.: *Metodiky vývoje a údržby informačních systémů*, Grada 2005, ISBN 80-247-1075-7
- [Burlison 1999] Burlison D.: *Inside the Object Database Model*, CRC Press 1999, ISBN 0-8493-1807-6
- [Catell 1998] Catell R. G.: *The Object Data Standard: ODMG 3.0*, ODMG 1998, ISBN 1558606475

- [Choppy 2004] Choppy C.: *Improving Use-Case Based Requirements*, in proceedings of FASE – Fundamental Approaches to Software Engineering 2002, pp. 244-261, Springer ISSN 0302-9743
- [Coad 1990] Coad, P., Yourdon E., *Object Oriented Analysis*, Prentice-Hall, Englewood Cliffs, N.J., 1990.
- [Coad 1993] Peter Coad and Jill Nicola: *Object-Oriented Programming*, Yourdon Press, 1993
- [Coad 1995] Peter Coad , David North and Mark Mayfield: *Object Models: Strategies, Patterns and Applications*, Yourdon Press, 1995
- [Coleman et al. 1994] Coleman, D., Arnold, P., Bodoff, S., Dollin, C., Gilchrist, H., Hayes, F. Jeremaes, P., *Object-Oriented Development - The Fusion Method*, Englewood Cliffs, N.J.: Prentice-Hall, 1994.
- [Darnton 1997] Darnton, Geoffrey, Darnton, Moksha: *Business Process Analysis*, International Thomson Publishing 1997 ISBN: 1861520395
- [Davis 1993] Davis, A.: *Software Requirements - Objects, Functions and States* , Prentice Hall 1993
- [Derr 1995] Derr K.W.: *Applying OMT - A Practical Guide to Using the Object Modelling Technique*, Sigs Books 1995, ISBN 1-884842-10-0, Prentice Hall 1995 ISBN 0-13-231390-1
- [Doskočil 2004] Doskočil Tomáš, Merunka Vojtěch, *Zkušenosti z výuky s objektovým databázovým systémem Gemstone/S v předmětu DBII na KII PEF ČZU*, ve sborníku konference Informatika o výuce informatiky, Zlín 2004, ISBN 80-7302-066-1
- [Drbal 1996] Drbal Pavel, *Proudy v objektově orientovaných metodikách*, ve sborníku konference Tvorba softwaru 1996, Ostrava.
- [Drbal 2002] Drbal P.: *Extrémní programování a metodický přístup*, ve sborníku konference Tvorba softwaru 2002, ISBN 80-85988-74-7
- [Ewusi 2003] Ewusi K.: *Software Development Failures*, MIT Press 1993, ISBN 0-262-05072-2
- [Fowler 1999] Fowler M., Kendall S., *UML Distilled (2nd Edition)*, 1999, Addison-Wesley ISBN: 0-201-65783-X
- [Gamma et al. 1995] Gamma, E.; Helm, R.; Johnson, E.R; Vlissides, J.: *Design Patterns - Elements of Reusable Object Oriented Software*, Addison-Wesley, New York, USA, 1995. ISBN 0201633612
- [Gemstone 2004] *Gemstone Object Server* - documentation & non-commercial version download, <http://www.gemstone.com>, <http://smalltalk.gemstone.com>, prosinec 2004
- [Goldberg 1995] Goldberg Adele, Kenneth Rubin S.: *Succeeding with Objects - Decision Frameworks for Project Management*, Addison Wesley 1995, ISBN 0-201-62878-3
- [Graham et al. 2002] Graham, Ian; Simons, Anthony J H: *30 Things that Go Wrong in Object Modeling with UML 1.3*, University of Sheffield & IGA Ltd. <http://www.iga.co.uk>, květen 1992
- [Hammer et al. 1994] Hammer, M. – Stanton, A.: *The Reengineering Revolution*, Collins 1994
- [Henderson-Sellers 1994] Henderson-Sellers B, Edwards JM.: *MOSES - A Second Generation Object-Oriented Methodology*, pp 68-73, Object Magazine 4(3) 1994
- [Henderson-Sellers 1998] Henderson-Sellers, B.; Bulthuis, A.: *Object-Oriented Metamethods*, Springer-Verlag New York, 1998, ISBN 0-387-98257-4
- [Hopkins 1992] Hopkins T.: *Animating an Actor Programming Model*, Computer Science Dept. University of Manchester 1992
- [HOPL 1988] Wexelblat Richard L.: *History of Programming Languages*, Academic Press Pennsylvania 1988, ACM Monographic Series, from the ACM SIGPLAN History of Programming Languages Conference, June 1-3, 1988, ISBN 0-12-745040-8
- [Hruška 1995] Hruška T.: *Objektově orientované databázové technologie*, sborník konference Tvorba softwaru 1995, ISBN 80-901751-3-9

- [Jacobson et al. 1992] Jacobson, I., Christerson, M., Jonsson, P., Övergaard, G., *Object-Oriented Software Engineering - A Use Case Driven Approach*, 1992, Addison Wesley ISBN 0-201-54435-0
- [Jacobson 1985] I. Jacobson: *Concepts for Modelling Large Real-Time Systems*, Doctoral Dissertation, Department of Computer Systems, The Royal Institute of Technology, August 1985.
- [Jacobson et al. 1999] Jacobson, G. Booch, J. Rumbaugh: *The Unified Software Development Process*, Addison Wesley, 1999, ISBN 0-201-5-7169-2
- [Kelly 1997] Kelly, S.: *GOPRR Metamodel, appendix of doctor thesis „Towards a Comprehensive MetaCASE and CAME Environment: Conceptual, Architectural, Functional and Usability Advances in Metaedit+“*, Ph.D. Thesis, Jyväskylä University, 1997
- [Khodorkovsky 2002] Khodorkovsky V. V.: *On Normalization of Relations in Databases*, Programming and Computer Software 28 (1), 41-52, January 2002, Nauka Interperiodica.
- [Kotonya 1999] Kotonya G. – Sommerville I.: *Requirements Engineering - Processes and Techniques*, 1999, J. Wiley and Sons
- [Kroha 1995] Kroha P.: *Objects and Databases*, McGraw Hill, London 1995, ISBN 0-07-707790-3.
- [Kruchten 2000] Kruchten Philippe: *The Rational Unified Process – An Introduction*, Addison-Wesley, 2000, ISBN 0201707101
- [Lacko 1996] Lacko B.: *Komparativní analýza strukturovaného a objektově orientovaného přístupu*, sborník konference OBJEKTY 1996. ČZU Praha 1996 str.69-76
- [Larsson 2002] Larsson M., Crncovic I.: *Building Reliable Component-Based Software Systems*, Artech House 2002, ISBN 1-58053-327-2
- [Loomis 1994] Loomis M.: *ODBMS - Hitting the Relational Wall*, pp 56-60, JOOP January 1994
- [Loomis et al. 1994] Loomis M., Chaundri A.: *Object Databases in Practice*, Prentice Hall 1994, ISBN 013899725X
- [Martin et al. 1995] Martin James, Odell James J.: *Object-Oriented Methods - A Foundation*, Prentice Hall 1995, ISBN 0-13-63856-2
- [Meyer 1997] Bertrand Meyer: *Object Oriented Software Construction*, Prentice Hall, 1997, ISBN 0136291554
- [Molhanec 2004] Molhanec M.: *Několik poznámek k porozumění objektového paradigmatu*, ve sborníku konference Objekty 2004
- [Nierstrasz et al. 1990] Nierstrasz Oscar, Papathomas Michael: *Viewing Objects as Patterns of Communicating Agents*, pp 255-266 , in Object Management 1990, Centre Universitaire d' Informatique - Université de Geneve, also at <ftp://cui.unige.ch/OO-articles/viewingObjectsAsPatterns.ps.Z>
- [Nootenboom 2004] Nootenboom Henk Jan: *Nut's - a online column about software design*. <http://www.sum-it.nl/en200239.html>, prosinec 2004
- [OMG 2002a] Object Management Group (OMG). *Meta Object Facility (MOF) specification – version 1.4*. 2002. <http://www.omg.org>
- [OMG 2002b] Object Management Group (OMG). *Model Driven Architecture (MDA) specification – Version 1.4*. 2002. <http://www.omg.org>
- [OMG 2003] Object Management Group (OMG). *OMG Unified Modeling Language Specification - Version 1.5*. 2003. <http://www.omg.org>
- [Partridge 1996] Partridge C.: *Business Objects - Reengineering for Reuse*, Butterworth-Heinemann 1996, ISBN 07506-2082X
- [Rashid 2004] Rashid A.: *Aspect Oriented Database Model*, Springer 2004, ISBN 3-540-00948-5
- [Riecken 1994] Riecken D.: *Software Agents*, pp 18-147, Communications of ACM, 37 (1994)
- [Rubin et al. 1992] Rubin K.S., Goldberg A.: *Object Behavioural Analysis*, pp 48-62 Communications of the ACM 35(9) 1992

- [Rubin et al. 1993] Rubin K.S, Goldberg A.: *Object Behavior Analysis*, pp 48-57 Communications of the ACM, vol 35 no 9. 1993
- [Rubin et al. 1994] Rubin Kenneth S, McClaughry Patrick, Pellergini David.: *Modelling Rules using Object Behavior Analysis and Design*, pp 63-68, Object Magazine 4(3) 1994
- [Rumbaugh et al. 1991] Rumbaugh James, Blaha Michael, Premerlani William, Eddy Frederic, Lorensen William: *Object-Oriented Modelling and Design*, Prentice Hall 1991, ISBN 0-13-630054-5
- [Rumbaugh et al. 1998] Rumbaugh, I. Jacobson, G. Booch: *The Unified Modeling Language Reference Manual*, Addison-Wesley, 1998, ISBN 0-2013-0998-X
- [Rumbaugh 1996] James Rumbaugh: *OMT Insights*. SIGS Book. 1996.
- [Shiver et al. 1987] Shriver Bruce, Wegner Peter.: *Research Directions in OOP*, MIT Press 1987, ISBN 0-262-19264-0
- [Shlaer 1992] Mellor Stephen, Shlaer Sally: *Object Lifecycles: Modeling the World in States*, ISBN 0136299407
- [Shlaer 1989] Sally Shlaer and Stephen. P. Mellor: *Object-Oriented Systems Analysis: Modeling the World in Data*, Yourdon Press, 1989
- [Shoham 1993] Shoham Y.: *Agent Oriented Programming*, pp 51-92, Artificial Intelligence 60 (1993)
- [Silbershatz 2004] Silbershatz A et al.: *Database Systems Concepts – 4th Edition*, McGraw Hill 2004, ISBN 0-07-228363-7
- [Taylor 1995] Taylor, David A.: *Business Engineering with Object Technology*, John Wiley 1995 ISBN: 0471045217
- [UML 2004] *The Unified Modeling Language Documents*, <http://www.uml.org> , prosinec 2004
- [Vaniček 2004] Vaniček J.: *Měření a hodnocení jakosti informačních systémů*. Praha: ČZU PEF, 2004, ISBN 80-213-1206-8
- [Wai 1992] Wai Y. Mok, Yiu-Kai Ng and David W. Embley, *An Improved Nested Normal Form for Use in Object-Oriented Software Systems*. Proceedings of the 2nd International Computer Science Conference: Data and Knowledge Engineering: Theory and Applications, pp. 446-452, Hong Kong, December 1992.
- [Warmer 1998] Jos Warmer and Anneke Kleppe: *The Object Constraint Language: Precise modeling with UML*. Addison-Wesley, 1998.
- [Wilkie 1993] Wilkie G.: *Object-Oriented Software Engineering*, Addison Wesley 1993, ISBN 0-201-6276-1
- [Wirfs-Brock 1990] Rebeca Wirfs-Brock, Brian Wilkerson and Lauren Wiener: *Designing Object-Oriented Software*, Prentice Hall, 1990
- [Yonghui et al. 2001] Yonghui Wu, Zhou Aoying: *Research on Normalization Design for Complex Object Schemes*, Info-Tech and Info-Net, vol 5. 101-106, Proceedings of ICII 2001, Beijing.
- [Yourdon 1994] Yourdon E.: *Object-Oriented System Design - An Integrated Approach*, Prentice Hall 1994, ISBN 0-13-176892-1
- [Yourdon 1995] Yourdon E.: *Mainstream Objects - An Analysis and Design Approach for Business*, Prentice Hall 1995 ISBN 0-13-209156-9
- [Yourdon 1989] Yourdon E.: *Modern Structured Analysis*. Yourdon Press/Prentice-Hall. New York 1989.
- [Yourdon 1991a] Peter Coad and Ed Yourdon: *Object-Oriented Analysis*, Yourdon Press, 1991
- [Yourdon 1991b] Peter Coad and Ed Yourdon: *Object-Oriented Design*, Yourdon Press, 1991
- [Zahir et al. 1997] Tari Zahir, Stokes John, Spaccapietra Stefano: *Object Normal Forms and Dependency Constraints for Object-Oriented Schemata*, ACM Transactions on Database Systems 513-569, Vol 22 Issue 4, December 1997.

11 Přílohy

11.1 Slovníček pojmů

<i>Pojem</i>	<i>anglický překlad</i>	<i>vysvětlení</i>	<i>oblast použití</i>
agilní přístup	agile approach	Varianta \rightarrow evolučního životního cyklu tvorby softwaru. Pro agilní přístupy je typická orientace na vlastní tvorbu programu na úkor analytických a dokumentačních aktivit.	všechny fáze BORMu
agregace objektů	object aggregation	Zvláštní případ \rightarrow skládání objektů, kdy je existence a \rightarrow identita objektu závislá na jiném objektu, který je v něm skládán. (Změna skládaného objektu by v systému způsobila i změnu skládajícího objektu.)	CM, SM
aktivita	activity	Aktivita představují jednotlivé části chování \rightarrow business objektů tak, jak byly rozpoznány technikou \rightarrow OBA. V \rightarrow procesních diagramech se pomocí vybraných aktivit provádějí \rightarrow přechody mezi \rightarrow stavy objektů. Aktivita různých objektů mezi sebou komunikují (\rightarrow komunikace). Z aktivit se odvozují \rightarrow metody.	BM
aktor	actor	Aktor je specifický název objekt, který je schopen \rightarrow delegovat zprávy. (\rightarrow čistý objektový programovací jazyk)	CM, SM
aktorový model výpočtu	actor computational model	Varianta objektově orientovaného chování systému, kde dochází k \rightarrow delegování \rightarrow zpráv mezi objekty. (\rightarrow aktor, čistý objektový programovací jazyk)	CM, SM
analýza	analysis	Souhrnný název pro fáze tvorby systému, při kterých se zjišťuje zadání systému a modeluje se jeho požadovaná funkčnost. Po analýze následuje \rightarrow návrh. V případě \rightarrow spirálního způsobu tvorby softwaru se analýza kryje s \rightarrow expansí systému. V \rightarrow BORMu se analýza skládá z fáze \rightarrow strategické analýzy, \rightarrow úvodní analýzy a \rightarrow podrobné analýzy.	BM, CM
architektura systému	system architecture	Komplexní model systému, který se skládá z několika vrstev, které jsou samy o sobě modelem zaměřeným na jednu stránku systému. Je to například vrstva \rightarrow procesů, logický model (popis dat, funkcí a pravidel) a model komponent (např. softwarové aplikace nebo organizační struktury). Prvky z různých vrstev architektury jsou mezi sebou vzájemně provázány, a proto je vhodné při každé změně provést rozbor dopadů na prvky jiných úrovní. (\rightarrow konvergenční inženýrství, BPR)	všechny fáze BORMu
AS-IS	AS-IS	První fáze \rightarrow BPR. AS-IS znamená „tak, jak to je“.	BM, BPR
asociace	association	Vztah mezi objekty, který vyjadřuje, že jeden objekt v modelu potřebuje nebo používá nebo k němu jinak přísluší druhý objekt. Tento vztah se nesmí vykládat jako vazba mezi záznamy v tabulkách \rightarrow relačních databází a na rozdíl od jiných metod se v BORMu nepoužívá pro \rightarrow softwarové modelování. Z asociací lze odvodit vazbu \rightarrow skládání objektů.	BM, CM
asynchronní zpráva, paralelní zpráva	asynchronous message, parallel message, fork message	\rightarrow Zpráva, na jejíž výsledek \rightarrow objekt, který zprávu poslal, nečeká. Objekt tedy poslanou zprávou spustil vykonávání nějakých aktivit a současně dál pokračuje ve svých aktivitách. Tento typ zpráv je v \rightarrow čistých objektových jazycích základem paralelního programování a také umožňuje \rightarrow delegování a \rightarrow závislost mezi objekty. (\rightarrow synchronní zpráva)	CM, SM

atribut, vlastnost	attribute	Vlastnosti →objektu, kterými se objekt prezentuje svému okolí v systému. (Například „barva“, „váha“, „cena“ atd.) V →BORMu by atributy neměly být chápány jen jako data uvnitř objektů, protože atributy lze realizovat i jako →metody, které dané hodnoty poskytují. (→ <i>zapouzdření, protokol</i>)	všechny fáze BORMu
automat	automaton (mn.č. „automata“), finite state machine	Zařízení, které přijímá vstupní informace a jako odpovědi na ně vydává informace výstupní, které závisí jednoznačně na vnitřním stavu tohoto zařízení a na vstupní informaci. Během své činnosti automat vykonává →přechody čímž mění svoje →stavy. Teorie automatů je v →BORMu základem pro modelování chování →objektů v →procesech. (→ <i>ORD</i>).	všechny fáze BORMu
BM	BM	Zkratka pro →business modelování.	BM
BO	BO	Zkratka pro →business objekt.	BO
BORM	BORM	Zkratka pro „Business Object Relation Modeling“. BORM je metodika vyvinutá v rámci výzkumného projektu VAPPIENS Britské rady (British Council) ve spolupráci s Deloitte&Touche. Základní filosofií BORMu je plná podpora →objektového přístupu, využívání →procesního modelování pro →získávání požadavků a myšlenky →konvergenčního inženýrství.	všechny fáze BORMu
BPR – reinženýring business procesů	BPR – Business Process Reengineering	Přehodnocení a rekonstrukce →procesů za účelem jejich zdokonalení. Tato činnost je někdy důležitá pro nalezení správného zadání informačního systému. (→ <i>získávání zadání</i>) Provádí se ve dvou fázích: V první etapě →AS-IS se podrobně modeluje stávající stav procesů. Po vyhodnocení výsledků první etapy se přistupuje k druhé etapě →TO-BE. Zde se navrhují nové procesy a provádí se návrh nové organizační struktury. Pro podklady na BPR se používá technika →OBA.	BM, BPR
business inženýrství	business engineering	Inženýrský obor, který se zabývá analýzou, modelováním a návrhem organizačních a řídicích struktur. (→ <i>BPR</i>) Některými autory je považován za součást →informačního inženýrství. Podle BORMu jsou tyto aktivity potřebné pro získání správných požadavků na informační systém. (→ <i>získávání požadavků</i>)	BM, BPR
business modelování	business modeling	První etapa BORMu, kdy se vytváří model →procesů systému tak, aby došlo k rozpoznání zadání problému a případnému přepracování stávajících procesů a nebo organizační struktury za účelem lepší implementace softwaru. (→ <i>BPR</i>) Zahnuje fáze →strategické analýzy a →úvodní analýzy.	BM, BPR
business objekt	business object	Objekt reálného světa, který slouží k analýze a popisu zadání systému. (→ <i>participant</i>)	BM
business proces	business process	Business procesy jsou východiskem pro funkční popis podniku nebo organizace. (→ <i>proces, BPR</i>)	BM, BPR
C# („cis“)	C# („c-sharp“)	Jazyk z dílny Microsoftu. Velmi podobný jazyku →Java.	SM
C++ („cé plus plus“)	C++	→Smíšený programovací jazyk. Patří mezi složité programovací jazyky, ale je v něm možné dosáhnout efektivního kódu. Jeho první verze pocházejí z 80. let.	SM

CASE – nástroje pro podporu softwarového inženýrství	CASE – Computer Aided Software Engineering	Programy, které pomáhají analytikům a vývojářům postupovat podle nějaké metodiky. Správný CASE by měl mít podporu týmového vývoje, nástroje pro práci s →diagramy, možnost tvorby reportů a analýz nad projektovými daty a generovat kód. CASE nemusejí být pouze pro metody →softwarového inženýrství, ale i pro →business inženýrství. Pokud podporuje →metamodelování (např. Proforma Workbench, Metaedit), tak lze CASE uživatelsky „nastavovat“ a „vylepšovat“ jím podporovanou metodiku. (→ <i>metamodelování</i>)	všechny fáze BORMu
cíl	goal	V kontextu →BPR to je součást detailního popisu →aktivity. Cíle vymezují, kam má podnik směřovat. (Například získat dominantní postavení na trhu nebo ve vybraném tržním segmentu.) Jejich znalost je důležitá pro rozhodování nad souvislostmi zadání informačního systému, →podnikových procesů s dalšími atributy organizace. (→ <i>získávání zadání</i>)	BM, BPR
CM	CM	Zkratka pro →konceptuální modelování.	CM
CO	CO	Zkratka pro →konceptuální objekt.	CO
CORBA	CORBA	Standard pro práci s objekty v distribuovaných softwarových systémech. Je doporučován sdružením →OMG a je podporován v →objektových databázích. (→ <i>objektově relační databáze, Gemstone</i>)	SM
CSF	CSF	Zkratka pro „Critical Success Factor“ (→ <i>faktor</i>).	BM, BPR
činnost, pracovní činnost	real activity	V kontextu →BPR to je součást detailního popisu →aktivity. Je to konkrétní popis týkající se plnění jednoho pracovního úkolu. (Například „zajištění pracoviště“ nebo „oprava vozidla“). Jedna aktivita typicky obsahuje více činností. (→ <i>BPR</i>)	BM, BPR
čistý objektový programovací jazyk	Environment pure object-oriented programming language (EPOL)	Programovací jazyk, který důsledně podporuje vlastnosti objektového přístupu. Takový jazyk není úpravou nějakého neobjektového jazyka. Ve srovnání se →smíšenými jazyky je v nich objektové programování snadnější – například dovolují programovat aplikaci i za jejího chodu. Odhaduje se, že čistý jazyk potřebuje asi 2× až 5× méně příkazů na jeden →funkční bod. Mezi čisté jazyky patří například →Smalltalk.	SM
databáze, SŘBD – systém řízení báze dat	database, DBMS – database management system	Software, který dovoluje uchovávat a vydávat data pro potřeby uživatelů a nebo jiných systémů k němu připojených. Databáze je důležitou součástí většiny →informačních systémů. Podle principu činnosti se člení na síťové, →relační, →objektově relační a →objektové.	SM
datový tok	data flow	Data, která si objekty vyměňují při →komunikacích nebo posílání →zpráv. Rozlišují se →parametry zprávy a →návrátové hodnoty.	všechny fáze BORMu
dědičnost	inheritance	Vazba mezi →softwarovými objekty, pomocí které lze programovat nové objekty (potomky) s využitím již existujících objektů (předků). Nové objekty jsou potom v programu schopny využívat →metody svých předků. Ve →smíšených programovacích jazycích je dědičnost jediná možnost, jak zajistit →polymorfismus mezi objekty. (→ <i>hierarchie typů</i>)	SM

delegování	delegation	Zpracování → zprávy objektem tím způsobem, že objekt, který přijal zprávu, ji nebude zpracovávat a vyvolá přenesení této zprávy na jiný objekt, který ji již dokáže zpracovat. Výpočet se pak dál odehrává jen u delegovaného objektu a objekt, který zprávu od sebe delegoval se ho již neúčastní (ani mu nemusí být předán výsledek). V případě, že implementační prostředí tuto vazbu nepodporuje, musí se ve fázi → softwarového modelování nahrazovat. (→ <i>smíšený programovací jazyk</i>)	CM, SM
deliverable	deliverable	Synonymum pro → výrobek.	SM, BM
design	design	Synonymum pro → návrh.	CM, SM
diagram	diagram	Schéma, které popisuje pomocí značek a dalších dle standardu dohodnutých způsobů kreslení nějaký model. V → BORMu se diagramy označují jako → ORD.	všechny fáze BORMu
esenciální objekt	essential object	Synonymum pro → business objekt.	BM
evoluční způsob tvorby softwaru, evoluční model	evolutionary development lifecycle	Varianta životního cyklu tvorby softwaru. Je založen na myšlence postupného vývoje po malých částech, kdy může docházet k doplňování požadavků od uživatele. Tento způsob je vhodný k využití → objektového přístupu. (→ <i>sekvenční model, → agilní přístup</i>)	všechny fáze BORMu
expanse	expansion	První část cyklu → spirálního modelu. Zde dochází ke hromadění informací, potřebných pro pochopení zadání a vytvoření aplikace. Expanze končí dokončením → konceptuálního modelu, který na logické úrovni reprezentuje požadované zadání a formálně popisuje řešený problém. (→ <i>konsolidace, analýza</i>)	BM, CM
faktor, kritický faktor pro úspěch	CSF – critical success factor	V kontextu → BPR to je součást detailního popisu → aktivity. Faktor je činitel, který má zásadní vliv na podnikatelský úspěch. (Je to například „existence podnikové informační strategie“, „hodnocení na základě měření“ nebo „dodržování standardů“.) Jejich znalost je důležitá pro rozhodování nad souvislostmi zadání informačního systému, podnikových → procesů s dalšími atributy organizace. (→ <i>získávání zadání</i>)	BM, BPR
funkce systému	system function	Požadovaná funkce je nejjednodušším slovním popisem požadovaných → procesů v systému podle → OBA. Pro pozdější bezproblémovou komunikaci analytiků a zadavatelů je vhodné do tohoto seznamu zahrnout a zvlášť vyznačit i funkce, které popisují, co se modelovat nebude. Ze seznamu funkcí se odvozuje seznam → scénářů.	BM, OBA
funkční body	function points, feature points	→ Měření, které slouží k odhadování pracnosti vyvíjeného softwaru metodou výpočtu funkčních bodů, které se vypočítají z rozsahu požadavků na systém (např. počet datových struktur, uživatelských vstupů a výstupů, ...). Z výsledného počtu funkčních bodů lze odhadnout velikost budoucího programu, protože pro různé → programovací jazyky je na základě statistických měření známo, kolik příkazů je třeba na pokrytí jednoho funkčního bodu. Metoda „feature points“ je variantou této metody pro případ objektové tvorby softwaru.	BM, CM
Gemstone	Gemstone	Představitel → objektového databázového systému. Databázový systém Gemstone používá programovací jazyk → Smalltalk nebo → Java. Podporuje standard → CORBA.	SM
generalizace-specializace	generalization-specialization	Synonymum pro → hierarchii typů.	CM

generický model	generic model	Generický model je takový model, který v sobě kombinuje vlastnosti modelu s vlastnostmi →metamodelu, což může usnadnit návrh systému. (→ <i>metamodelování, znovupoužitelnost, návrhové vzory</i>)	všechny fáze BORMu
hierarchie typů	type hierarchy	Tato vazba vyjadřuje vztahy mezi →protokoly objektů. To znamená, že objekty nadtypu jsou →polymorfni s objekty podtypu. K upřesnění vazby lze uvést seznam →zpráv, které se polymorfismu týkají. Tato hierarchie objektů vzniká z hierarchie →JE-JAKO a později je z ní odvozována hierarchie →dědičnosti mezi objekty.	CM
identita objektů	object identity	Spolu s →polymorfismem a →zapouzdřením základní vlastnost →objektů, díky které jsou různé objekty v jednom systému nezaměnitelné ani tehdy, mají-li shodná data a nebo →metody. V některých konkrétních softwarových prostředích (→ <i>smíšené programovací jazyky</i>) je ale třeba identitu zvlášť zajišťovat. Například v →relačních databázích musí mít objekty pro vzájemné rozlišení navíc zvláštní údaj nazývaný primární klíč (neboli index). V →objektových databázích to není potřeba.	všechny fáze BORMu
implementace	implementation	V této fázi se vytváří (programuje, sestavuje či generuje z CASE nástroje), testuje a předává uživateli požadovaný software. Při →spirálním způsobu tvorby softwaru je součástí →konsolidace systému.	SM
informační inženýrství	information engineering	Inženýrský obor, který podle některých autorů zahrnuje →softwarové inženýrství a →business inženýrství. Jedná se o disciplínu, která nahlíží komplexně na organizační a řídicí struktury a →informační systémy. (→ <i>sociotechnický systém, konvergenční inženýrství</i>)	všechny fáze BORMu
informační systém	information system	Systém sloužící k poskytování informací a sestavený podle nějakého zadání složený z hardwarových, softwarových a organizačních prvků.	všechny fáze BORMu
instance	instance	Objekt, který má svoje vlastnosti (= strukturu dat a metody) popsané v nějaké →třídě. Všechny instance téže třídy tedy mají stejné metody a stejnou strukturu dat a liší se mezi sebou jen konkrétním obsahem těchto dat. Instance se odvozují z →business objektů.	CM, SM
iterativní způsob tvorby softwaru, iterativní model	iterative development lifecycle	Způsob vývoje softwaru, kdy se v případě potřeby navrací do počátečních fází vývoje. Pro upřesnění zadání se mohou vytvářet →prototypy. Jeho variantou je →spirální model. (→ <i>vodopádový model</i>)	všechny fáze BORMu
Java („džáva“)	Java	→Smíšený programovací jazyk. Jeho první verze pocházejí z 80. let, ale rozšířil se až ve druhé polovině 90. let s nástupem WWW. Je podobný →C++, ale je jednodušší a podporuje více objektových vlastností. Podle některých autorů je považován za →čistý objektový programovací jazyk.	SM
Je-Jako	IS-A	Vztah hierarchie objektů. Objekty na nižší úrovni této hierarchie jsou prvky domény, která je podmnožinou domény objektů vyšší úrovně. Tento vztah se v →BORMu nesmí vykládat jako →dědičnost mezi →softwarovými objekty, protože je na dědičnost postupně transformován přes →hierarchii typů.	BM
klient	client	Objekt, na kterém jsou →závislé jiné objekty (→ <i>server</i>).	CM, SM
klient	client	Softwarová aplikace, která využívá data nebo služby jiné aplikace na jiném počítači v síti. (→ <i>server</i>)	SM
kolekce, sada	collection	Synonymum pro →množinu objektů.	CM, SM

komunikace	communication	Rízení → aktivit → business objektů. Komunikace je abstrakce → zpráv mezi objekty. V pozdějších fázích modelování se zprávy z komunikací odvozují.	BM
konceptuální modelování	conceptual modeling	Druhá etapa → BORMu, která zahrnuje tvorbu modelu z → konceptuálních objektů. Takový model stojí napůl cesty mezi zadáním a řešením, kdy je problém považován za rozpoznáný a nalézá se v něm část k softwarovému řešení. V konceptuálních modelech se na systém nazírá jako na ideální svět počítačových objektů. Mezi modelované vazby patří například → asociace, → skládání, → závislost, → delegování a → hierarchie typů. Zahrnuje fáze → podrobné analýzy a → úvodního návrhu.	CM, UML
konceptuální objekt	conceptual object	Objekt, který se používá v → konceptuálních modelech k popisu první podoby modelu softwaru. Jsou to → instance, → třídy nebo → množiny. Na rozdíl od → softwarových objektů konceptuální objekty nejsou zatíženy detaily konkrétního implementačního prostředí. Konceptuální objekty se odvozují z → business objektů a samy slouží k popisu → softwarových objektů.	CM, UML
konsolidace	consolidation	Druhá část cyklu → spirálního způsobu tvorby softwaru. Model se zde postupně po předchozí myšlenkové "expanzi" stává fungujícím programem. V tomto stadiu může dojít k tomu, že některé z návrhů bude nutno vypustit vzhledem k časovým, kapacitním, implementačním nebo i intelektuálním omezením. (→ <i>expanse, návrh, implementace</i>)	CM, SM
kovergenční inženýrství	convergent engineering	Inženýrský obor, který neklade hranici mezi podnikovým systémem a informačním systémem. (→ <i>sociotechnický systém</i>). Je to směr → informačního inženýrství, který techniky a postupy používané v → softwarovém inženýrství aplikuje do → business inženýrství. Opírá se o → objektový přístup. Je jednou ze základních filosofí → BORMu.	BM, CM
logický objekt	logical object	Synonymum pro → konceptuální objekt.	CM
metamodel	metamodel	Metamodel je model systému, který sám obsahuje prvky k modelování něčeho jiného. Znalost metamodelu metody může pomoci při pochopení metody a také usnadnit návrh konkrétního řešení. (→ <i>metamodelování, generický model</i>)	všechny fáze BORMu
meta-modelování	metamodeling	Metamodelování znamená tvorbu modelu na vyšší úrovni abstrakce tak, že předmětem modelování je něco, co je samo o sobě modelem něčeho jiného. (→ <i>metamodel</i>). Dovoluje popsat jednotným způsobem odlišné datové struktury více systémů, což poskytuje možnost skládat je dohromady ve vyšší systém. Zajišťuje standardizaci metodologií a interoperabilitu → CASE nástrojů.	všechny fáze BORMu
metoda	method	Část programového kódu, kterou objekt spouští, pokud přijme příslušnou → zprávu. Metody mohou například měnit data uvnitř objektů (→ <i>skládání, zapouzdření</i>), posílat další zprávy dalším objektům nebo provádět → přechody mezi → stavů objektu. Metody se odvozují z → aktivit.	CM, SM
měření (v praxi používaný pojem je „metriky“)	measurement (metrics)	Nástroj pro podporu formálního měření kvality softwarových produktů nebo modelů. Aparátem pro softwarové měření je numerická matematika a statistický aparát. Jejich sběr a vyhodnocování je důležitou součástí řízení projektů → BPR nebo tvorby softwaru. (→ <i>funkční body</i>).	všechny fáze BORMu

množina objektů, sada, kolekce	collection	Skupina →objektů, které mají dohromady nějaký zvláštní význam pro model, kde se s nimi pracuje nejen po prvcích, ale i jako s celkem. Množiny do systému zavádíme když není vhodné skupinu objektů realizovat jako →třidu. Prvky množiny mohou být objekty různých tříd, pokud jsou mezi sebou →polymorfni. Množiny objektů se odvozují z →business objektů.	CM, SM
modelovací karta, modelová karta	modeling card	Modelovací karta je tabulka, která podrobněji popisuje →objekt. Je součástí techniky →OBA. Může obsahovat →aktivity objektu, s objektem spolupracující objekty a nebo také vlastnosti objektu, které v modelu potřebujeme. (→proces)	BM, OBA
nadtyp-podtyp	supertype-subtype	Synonymum pro →hierarchii typů.	CM
návratová hodnota	return value	Data přenášená jako odpověď na posláni →zprávy mezi objekty. Lze je považovat za →datový tok ve směru opačném k směru posláni zprávy.	CM, SM
návrh, design	design	Souhrnný název pro fáze tvorby systému, při kterých se na základě již provedené →analýzy model přetváří do podoby, kterou je již možné implementovat. (→implementace). V případě →spirálního způsobu tvorby softwaru je návrh součástí →konsolidace systému. V →BORMu se návrh skládá z fáze →úvodního návrhu a →podrobného návrhu.	CM, SM
návrhový vzor	design pattern	Návrhové vzory jsou znalosti z předchozích úspěšných projektů. Představují mechanismus k uchování znalostí o tvorbě modelů, které se týkají →znovupoužitelných řešení. Je to významný prostředek k předávání znalostí mezi tvůrci systému při →softwarovém modelování. Někteří autoři také poukazují na možnost využívání návrhových vzorů při →BPR.	SM, BPR
OBA – analýza objektů podle chování	OBA – Object Behavioral Analysis	Technika, která slouží k získávání strukturovaných podkladů ze zadání pro potřeby konstrukce prvotního objektového modelu. Má pět kroků. Výstupy z OBA je například seznam →funkcí systému, →scénářů systému, →modelovacích karet a →procesní diagramy.	BM, OBA
objekt	object	Základní prvek modelování objektově orientovaným způsobem. Objekt tvoří jeho data a operace (aktivity nebo metody), které vymezují jeho chování. (→zapouzdření, protokol, polymorfismus) Objekty mají v systému →identitu a jejich vlastnosti se mohou v čase měnit. (→automat) V →BORMu se nazírání na objekt mění podle fáze projektování. Proto se rozlišují →business objekty, →konceptuální objekty a →softwarové objekty.	všechny fáze BORMu
objektová databáze	object-oriented database, objectbase	→Databázový systém, který dovoluje uchovávat a také zpracovávat →softwarové objekty přímo v databázi. Takové systémy nepoužívají relační tabulky a pracují na jiném principu. Mezi objektové databáze patří například →Gemstone. (→objektově-relační databáze, relační databáze)	SM
objektově relační databáze	object-relational database, hybrid object database	→Databázový systém, který uchovává →softwarové objekty v relační databázi a podporuje některé vlastnosti →objektového přístupu. Principem jeho činnosti zůstává relační datový model. Mezi tyto systémy například patří →Oracle od verze 8.0 (→objektové databáze)	SM

objektový datový model (ODM)	object-oriented data model	Datový model, na jehož principu pracují → objektové databáze.	SM
objektový programovací jazyk	object-oriented programming language	Programovací jazyk, který dovoluje využívat ve svých příkazech → objektový přístup. Rozlišují se → smíšené programovací jazyky a → čisté objektové programovací jazyky.	SM
objektový přístup, objektově orientovaný přístup, paradigma	object-oriented paradigm, object-oriented approach, OOP	Souhrn myšlenek o způsobu nazírání na svět, chápání zadání a hledání způsobu jejich řešení. Podle objektového přístupu se systém modeluje jako soustava vzájemně komunikujících → objektů. (→ <i>konvergenční inženýrství, proces</i>) V užším slova smyslu to je jeden ze způsobů tvorby softwaru. (→ <i>objektový programovací jazyk, objektová databáze</i>)	všechny fáze BORMu
omezení podle chování	behavioral constraints	Sada pravidel, typicky zobrazovaná ve formě rozhodovacích tabulek, která popisují za jakých podmínek lze konkrétní vazby mezi → business objekty transformovat na vazby mezi → konceptuálními objekty.	BM, CM
OMG	OMG	Object Management Group. (http://www.omg.org) Je to odborné sdružení firem, univerzit a dalších institucí, které podporuje objektovou technologii. Vydává také publikace a standardy (→ <i>OQL, CORBA</i>).	SM
OOP	OOP	Zkratka pro Object-Oriented Paradigm. (→ <i>objektový přístup</i>)	všechny fáze BORMu
operace	operation	→ Metoda nebo → aktivita objektu.	CM, SM
OQL	OQL	Návrh rozšíření jazyka → SQL pro práci s → objekty. Je doporučován sdružením → OMG a je součástí standardu → CORBA.	SM
Oracle	Oracle	Představitel → relačního databázového systému. Nové verze Oraclu jsou → objektově relační. Používá jazyk → SQL.	SM
ORD – diagram vztahů mezi objekty	ORD – Object Relation Diagram	Diagram, který zobrazuje současně datové, funkční i časové vztahy a souvislosti mezi objekty. OR diagramů je v → BORMu víc druhů podle toho, jaké objekty zobrazují. Například pro fázi → business modelování to je → procesní diagram (nazývaný také procesní mapa) a pro pozdější fáze se používají upravené diagramy UML.	všechny fáze BORMu, UML
parametry zprávy	message parameters	Data přenášená jako součást → zpráv mezi objekty. Parametry lze považovat za → datový tok ve směru zprávy.	CM, SM
participant	participant	→ Objekt, který se účastní nějakého → procesu a je popsán v nějakém → scénáři a nebo → procesním diagramu.	BM
podnikový proces	business process	Synonymum pro → business proces.	BM, BPR
podrobná analýza,	advanced analysis	Vymezení softwarové domény uvnitř modelovaného problému a rozpracování analýzy do detailů jednotlivých druhů a vazeb → konceptuálních objektů. Při → spirálním způsobu tvorby softwaru je součástí → expanse systému. (→ <i>analýza</i>)	CM
podrobný návrh, podrobný design	advanced design	V této fázi dochází k přeměně prvků již existujícího → konceptuálního modelu do podoby, která je podřízena cílovému implementačnímu prostředí. (→ <i>softwarový model, softwarový objekt</i>) Při → spirálním způsobu tvorby softwaru je součástí → konsolidace systému. (→ <i>návrh</i>)	SM

polymorfismus	polymorphism	Schopnost objektů reagovat různou →metodou na stejnou →zprávu. Rozlišujeme polymorfismus 1) mezi různými objekty, 2) mezi různými →stavy téhož objektu a 3) v závislosti na objektu, který zprávu vyslal. Polymorfismus významně ulehčuje modelování a způsobuje →znovupoužitelnost objektů. Díky polymorfismu mohou v jednom systému pracovat a zastupovat se objekty s různou strukturou. (→ <i>protokol</i>) V některých →objektových programovacích jazycích je však implementován jen částečně. (→ <i>dědičnost</i>)	všechny fáze BORMu
pozdní vazba	late binding	Druh vazby mezi →zprávou a →metodou, kdy se výběr metody provádí až po poslání zprávy během chodu objektového programu, což je v souladu s filosofií →polymorfismu. Tento typ vazby je na rozdíl od →včasné vazby typický pro →čisté objektové programovací jazyky.	SM
pracovní pozice	job position	V kontextu →BPR to je součást detailního popisu →participantu. Je to konkrétní popis pracovního místa daného pracovníka s případným počtem konkrétních pracovníků na této pozici. Jeden participant typicky obsahuje více pracovních pozic. (→ <i>BPR</i>)	BM, BPR
problém, issue	issue	Problém je něco, co stojí v cestě při plnění nějakého cíle nebo úkolu a je třeba to vyřešit. Je to například „velké emise do životního prostředí“ nebo „státní regulace“ apod. Jejich znalost je důležitá pro rozhodování nad souvislostmi zadání informačního systému, podnikových →procesů (→ <i>business proces</i>) s dalšími atributy organizace. (→ <i>získávání zadání, BPR</i>)	BM, BPR
proces	process	Sled →aktivit objektů, které dohromady realizují nějakou funkci systému. Mezi procesy může být hierarchie. Procesy popisujeme →scénáři a →procesními diagramy. (→ <i>BPR, role objektu v procesu, business proces</i>)	BM, BPR
procesní diagram, procesní mapa	process diagram, process map	Tento →diagram představuje mapu všech možných průběhů →procesu pomocí současného zobrazení dvou dimenzí tohoto problému. První dimenzí jsou →role (= průběh aktivit jednoho objektu v procesu) participujících objektů jako →automaty se →stavy a →přechody. Druhou dimenzí je sled →komunikací mezi objekty, který představuje řídicí a datové toky mezi objekty v procesu.	BM, OBA
protokol objektu	object protocol	Množina →zpráv nebo →komunikací, které lze objektu poslat. Pomocí protokolu se u objektů popisuje, k čemu v modelu slouží a jaké operace mohou provádět. Má-li více objektů neprázdný průnik protokolů, jsou mezi sebou →polymorfni (→ <i>nadtyp-podtyp</i>). Znalost protokolu stačí k tomu, aby se s objektem mohlo v systému pracovat, protože pokud dokáže příslušné zprávy přijímat, tak již nezáleží na jeho vnitřní struktuře. (→ <i>zapouzdření, znovupoužitelnost</i>)	všechny fáze BORMu
prototyp	prototype	Program, který napodobuje funkčnost vytvářeného softwaru, ale byl sestaven s výrazně menšími náklady a nižšími nároky na robustnost. Slouží k upřesnění a ověření zadání. V →BORMu se používají i jako prostředky k demonstraci nově navržených →procesů. (→ <i>BPR</i>)	BM
přechod	transition	Jeden úkon →automatu, kterým automat změní svůj stávající →stav na nový stav na základě nějaké přijaté informace. V →BORMu se na →objekty v →procesech nahlíží jako na automaty, kde jejich →přechody jsou realizovány prováděním →aktivit. (→ <i>role objektu v procesu</i>)	všechny fáze BORMu

refaktoring	refactoring	Technika, při které se přeuspořádává vnitřní struktura programu za účelem větší srozumitelnosti, lepší údržby a pro snadnější potenciální znovupoužití v jiných aplikacích. Na rozdíl od ladění zde nejde o opravy a vylepšení funkčnosti programu. Po refaktoringu totiž program funguje z vnějšího pohledu stejně jako před refaktoringem.	SM
relační databáze	relational database, table-based database	→Databázový systém, který organizuje data do vzájemně propojených tabulek. Pokud dovoluje přímou práci s objekty a ne pouze s jednoduchými hodnotami např. typu text, číslo nebo datum, tak se hovoří o →objektově relačním systému.	SM
relační datový model (RDM)	relational data model	Datový model, na jehož principu pracují → relační databáze a →objektově relační databáze.	SM
relační vztah	relation	Znázorňuje spojení mezi záznamy v relačních tabulkách →relačních databází. Pro →objektové databáze se nepoužívá, protože tyto databáze pracují s objekty přímo propojenými.	SM
role objektu v procesu	object role in a process	Sled →stavů a →přechodů objektu, který se účastní nějakého procesu. Je to jedna ze dvou dimenzí →procesního diagramu. Na tento sled lze nazírat jako na dílčí →diagram, který popisuje →proces ze zorného úhlu diskutovaného objektu, a který lze výhodně použít ke kontrole správnosti a realizovatelnosti celkového modelovaného procesu. (→ <i>automat</i>)	všechny fáze BORMu
sada objektů	collection	Synonymum pro →množinu objektů.	CM, SM
scénář	scenario	Scénář systému je podrobnější popis →procesu v technice →OBA. Odvozují se z →funkcí systému. U scénáře zvlášť popisujeme 1) začátek procesu, 2) vlastní akce procesu, 3) seznam participantů a 4) konec procesu. Mezi scénáři může být hierarchie skládání procesů, hierarchie nadtypů a podtypů a časová návaznost procesů na sebe, pro které se používají stejné značky jako pro objektové diagramy (→ <i>UML, ORD</i>). Typické zobrazení scénářů jsou tabulky s uvedenými čtyřmi políčky.	BM, OBA
sekvenční způsob tvorby softwaru, vodopádový model	sequential development lifecycle, waterfall development lifecycle	Způsob vývoje softwaru, kdy se ke každé další fázi projektu přistupuje až po úplném dokončení předchozí fáze. Vracení se zpět není přípustné. Tento styl se dobře plánuje a řídí, ale není vhodný v případech, kde na začátku projektu není ještě přesně rozpoznané zadání. Jiné způsoby vývoje jsou →evoluční a →iterativní.	všechny fáze BORMu
server	server	Objekt, který je →závislý na jiném objektu (→ <i>klient</i>).	CM, SM
server	server	Softwarová aplikace, která poskytuje data nebo služby jiným aplikacím na jiných počítačích v síti. (→ <i>klient</i>) Může se jednat o →databázový systém.	SM
skládání objektů, kompozice objektů	object composition	Vazba mezi objekty, kdy jeden nebo více objektů představují datové složky jiného objektu. Skládání objektů se v →BORMu odvozuje z →asociací.	CM, SM
SM	SM	Zkratka pro →softwarové modelování.	SM
Smalltalk	Smalltalk	→Čistý objektový programovací jazyk. Jeho první verze byly vyvíjeny již v 70. letech. Smalltalk může být také použit pro práci s daty v →objektových databázích. (→ <i>Gemstone, OQL</i>)	SM

smíšený programovací jazyk, hybridní jazyk	hybrid object-oriented programming language	→Programovací jazyk, který má základ v neobjektovém programování a dovoluje využívat některé vlastnosti objektového přístupu, ale některé, jako např. →závislost nebo →delegování nepodporuje. (→čistý objektový jazyk). Mezi smíšené jazyky patří například →C++, →Java a →C#.	SM
SO	SO	Zkratka pro →softwarový objekt.	SO
sociotechnický systém	socio-technical system	→Informační systém včetně jeho podnikového prostředí tvořeného procesy uživatelů, organizační a řídicí strukturou popsány a modelovaný jako jeden související celek. (→informační inženýrství, konvergenční přístup)	všechny fáze BORMu
software	software	V kontextu →BPR to je součást detailního popisu →aktivity. Jsou to komponenty nebo moduly zamýšlených nebo již existujících →informačních systémů, které jsou potřeba pro vykonání dané aktivity v →procesu. Je to například „osobní evidence“ nebo „mzdy“ nebo „skladové hospodářství“ apod. (→architektura systému)	BM, BPR
softwarové inženýrství	software engineering	Inženýrský obor, který se zabývá analýzou, modelováním, návrhem a provozem softwarových systémů. Některými autory je považován za součást →informačního inženýrství. Podle →BORMu při tvorbě systému by měly aktivitám softwarového inženýrství předcházet aktivity →business inženýrství.	CM, SM
softwarové modelování	software modeling	Třetí a poslední etapa →BORMu, která zahrnuje tvorbu modelu →softwarových objektů. Jeho prvky a vazby vycházejí z modelu →konceptuálních objektů, ale na rozdíl od něj musejí zohlednit problémy implementačního prostředí. (→smíšený programovací jazyk, zděděný systém, objektově relační databáze) Skládá se z fáze →podrobného návrhu a →implementace.	SM, UML
softwarový objekt	software object	→Objekty odvozené z →konceptuálních objektů během →softwarového modelování. Na rozdíl od nich zohledňují omezení implementačního prostředí.	SM, UML
spirální způsob tvorby softwaru, spirální model	spiral development lifecycle	Varianta →iterativního způsobu tvorby softwaru. Fáze projektu se provádějí několikrát za sebou tak, aby při každém opakování došlo ke zpřesnění zadání na základě předchozí implementace. První část jednoho vývojového cyklu se nazývá →expance a druhá →konsolidace. Tento způsob je vhodný k využití →objektového přístupu. (→sekvenční model)	všechny fáze BORMu
SQL	SQL (“sequel”)	Structured Query Language. Programovací jazyk který je standardem pro práci s daty v →relačních databázích. Jeho první verze pochází z počátku 70. let. V současné době probíhá jeho rozšiřování tak, aby mohl pracovat i s objekty. (→OQL, Smalltalk)	SM
stav	state	Konkrétní konstelace →automatu v čase. Pokud automat přijme nějakou informaci, může to vyvolat přechod z jednoho jeho →stavu do druhého. V →BORMu se nahlíží na objekty v procesech jako na →automaty, které v různých stavech mohou provádět různé →aktivity. (→role objektu v procesu)	všechny fáze BORMu
strategická analýza	strategic analysis	První fáze →BORMu, kde dochází k vymezení samotného problému, je stanoveno jeho rozhraní, jsou rozpoznány základní procesy, které se v systému mají odehrávat. Při →spirálním způsobu tvorby softwaru je součástí →expance systému. (→získávání zadání, analýza)	BM

synchronní zpráva	synchronous message	→Zpráva, na jejíž výsledek musí →objekt, který zprávu poslal čekat. Objekt tedy pokračuje ve svých aktivitách až když jsou dokončeny všechny aktivity, které byly zprávou vyvolány. (→ <i>asynchronní zpráva</i>)	CM, SM
TO-BE	TO-BE	Druhá fáze →BPR. TO-BE znamená „tak, jak by to mělo být“.	BM, BPR
třída	class	V čistých →objektových programovacích jazycích to je zvláštní objekt, který jako svoje data obsahuje vlastnosti (= strukturu dat a metody) pro svoje instance. Ve →smíšených jazycích to je pouze pojem ze syntaxe programovacího jazyka, pomocí kterého se programují vlastnosti objektů. Třídy se v →BORMu odvozuji z →business objektů.	CM, SM
událost	event	Podnět z okolí k vykonání →operace nějakého →objektu. V →BORMu jsou zdroje událostí modelovány také jako objekty a jejich události jako →zprávy nebo →komunikace. (→ <i>závislost</i>)	BM, CM
úkol	target	V kontextu →BPR to je součást detailního popisu →aktivity. Úkoly popisují, čeho je třeba dosáhnout. Je to například „snížit surovinovou náročnost“ nebo „zkrátit průběh vyřízení objednávky“. Jejich znalost je důležitá pro rozhodování nad souvislostmi zadání informačního systému, →podnikových procesů s dalšími atributy organizace. (→ <i>získávání zadání</i>)	BM, BPR
UML – unifikovaný modelovací jazyk	UML – Unified Modeling Language	UML poprvé publikovali G. Booch, J. Rumbaugh a I. Jacobson v roce 1996. Je doporučovaným standardem pro notaci (způsob kreslení) objektových →diagramů a představuje sjednocení myšlenek původních metod svých autorů. Stále se ještě vyvíjí. Pro potřeby →BORMu je třeba UML doplnit. (→ <i>ORD</i>)	CM, SM
úvodní analýza	initial analysis	Fáze rozpracování samotného problému, jsou mapovány požadované →procesy v systému a vlastnosti základních →objektů, které se na diskutovaných procesech podílejí. Může při ní dojít k →BPR. Při →spirálním způsobu tvorby softwaru je součástí fáze → <i>expandse</i> . (→ <i>analýza</i>)	BM, BPR
úvodní návrh, úvodní design	initial design	Je to fáze →BORMu, ve které se →konceptuální model upravuje tak, aby byl schopen softwarové implementace. Z pohledu zadání by mělo již vše být hotovo a rozpoznáno. Úvodní návrh používá shodné nebo velmi podobné nástroje jako předchozí fáze →podrobné analýzy, ale liší se způsobem práce s nimi. Při →spirálním způsobu tvorby softwaru je součástí fáze → <i>konsolidace</i> . (→ <i>návrh</i>)	CM
včasná vazba	early binding	Druh vazby mezi →zprávou a →metodou, kdy již v době překladu kódu programu je ke zprávě překladačem jednoznačně přiřazena metoda, která se má provést, což může vést k omezení míry →polymorfismu mezi objekty. Tento typ vazby je na rozdíl od →pozdní vazby typický pro →smíšené objektové programovací jazyky.	SM
vrstva	layer	Jedna součást →architektury modelovaného systému.	všechny fáze BORMu, BPR
výrobek, deliverable	deliverable	Programový výstup z jednoho cyklu →spirálního vývoje softwaru. Označuje se takto nejen →prototyp, ale i produkt „posledního“ cyklu, protože i ten může posloužit jako nové zadání pro další vývojový cyklus. Může také sloužit pro tvorbu nové verze produktu - a to nejen skrze zkušenosti s ním, ale přímo i svým kódem jako výchozí model nové → <i>expandse</i> .	SM, BM

zapouzdření	encapsulation	Spolu s →polymorfismem a →identitou základní vlastnost →objektů. Díky zapouzdření může objekt obsahovat data nebo →metody, se kterými pracuje jen objekt sám, nejsou součástí jeho →protokolu, a neposkytuje je ostatním objektům v systému. Z vnějšího pohledu se potom objekt tváří, jako by tyto vlastnosti neměl.	všechny fáze BORMu
zařízení	device	Součást detailního popisu →aktivity v →BPR. Zařízení je konkrétní pracovní pomůcka nebo stroj (například „služební auto“ nebo „osobní počítač“ nebo „ochranný oděv“), který je potřeba k vykonání dané aktivity.	BM, BPR
závislost	dependency	Závislost vyjadřuje, že provedení nějaké →metody řídicího objektu (→klienta) nepřímo bez posláni →zprávy vyvolává provedení nějaké metody řízeného objektu (→serveru). V případě, že implementační prostředí tuto vazbu nepodporuje, musí se ve fázi →softwarového modelování nahrazovat. (→smíšený programovací jazyk)	CM, SM
zděděný systém	legacy system	Existující programy, data nebo organizační struktura či procesy, které nelze měnit nebo ignorovat, což komplikuje tvorbu nového systému ve fázi →softwarového modelování, protože se jim musí struktura nového systému přizpůsobit.	SM
získávání zadání, requirement engineering	requirement engineering	Inženýrský obor, který se zabývá rozpoznáváním, analýzou a verifikací zadání pro →informační systémy. (→informační inženýrství, business inženýrství, softwarové inženýrství)	BM, BPR
znovu-použitelnost	reusability	Schopnost →objektu sloužit v jiném systému jiným způsobem, než pro jaký byl objekt vytvořen. Objekty jsou znovupoužitelné díky →polymorfismu, protože jsou snáze mezi sebou zaměnitelné. (→protokol) Znovupoužitelnost je základem techniky →návrhových vzorů.	všechny fáze BORMu
zpráva	message	Žádost o provedení →metody →objektem, kterému je zpráva poslána. Na rozdíl od volání funkce v klasickém programování je v objektovém programování od sebe oddělená žádost o operaci a její vlastní provedení. (→polymorfismus, včasná vazba, pozdní vazba) Zprávy mohou přenášet data. Data ve směru posláni zprávy se nazývají →parametry zprávy, data posílaná opačným směrem jsou →návrátové hodnoty. Zprávy jsou →synchronní nebo →asynchronní.	CM, SM

11.2 Modelovací nástroj Craft.CASE

11.2.1 Popis programu

Craft.CASE[®] je první český původní modelovací a analytický nástroj podporující metodu BORM[®] vyvíjený firmou e-Fractal s.r.o. pro mezinárodní poradenskou a konzultační firmu Deloitte. Vedoucí vývoje ve firmě e-Fractal s.r.o. jsou Ladislav Lenárt a Petr Skála. Ve firmě Deloitte je hlavním zadavatelem Jiří Polák. Hlavním analytikem a konzultantem je Vojtěch Merunka. Craft.CASE je programován v prostředí VisualWorks/Smalltalk firmy Cincom Int.

11.2.2 Instalace a spuštění programu

Craft.CASE se instaluje z jediného .EXE souboru. Typické umístění programu je v adresáři „C:\Program_Files\CraftCASE“. V adresáři jsou další vnořené adresáře. „Templates“ obsahuje šablony projektů. „UHEs“ obsahuje soubory, které systém vytváří v případě chyby. Po nainstalování se na ploše objeví ikona, ze které lze program spustit:



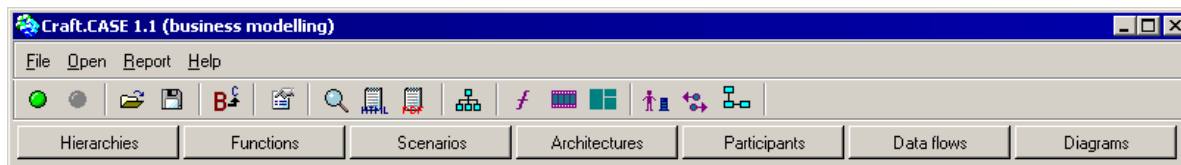
obr. 76. ikona programu

Program se spustí nastartováním souboru CraftCASE.exe. Je třeba, aby byl program nastartován v adresáři, kde je instalován, což je například pomocí ikony na ploše nebo ze startovacího menu. Není vhodné program startovat z jiného místa – například otevíráním datových souborů s uloženým projektem.

Pokud program nenalezl soubor s licencí (license.bl) a nebo licence vypršela, tak program poběží v DEMO režimu, který obsahuje následující omezení: Pokud projekt obsahuje více než jeden procesní diagram nebo více než pět participantů nebo více než jeden konceptuální diagram nebo více než pět tříd, tak nebude možné projekt uložit ani vytvořit žádný tiskový výstup.

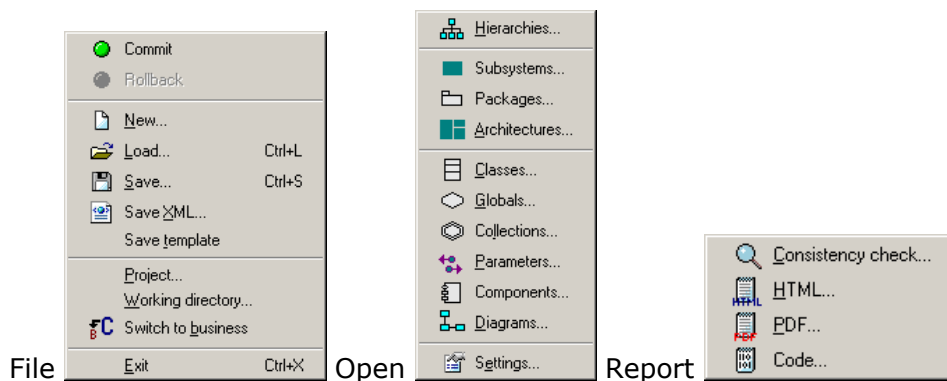
11.2.3 Hlavní okno – spouštěč (launcher)

Po nastartování se objeví hlavní okno aplikace.



obr. 77. hlavní okno (launcher)

Launcher je navržen tak, že většina funkcí z hlavního menu je také dostupná pomocí ikon nástrojové lišty a tlačítek pod nimi. Položky menu jsou následující:



obr. 78. volby hlavního menu

- Volba „Commit“ slouží k dočasnému uložení modelových dat na disk. K této záloze se lze kdykoliv vrátit pomocí volby „Rollback“. Jde tu o stejný princip, který se používá u databázových systémů pro řízení transakcí.
- „Working directory“ slouží k nastavení pracovního adresáře a pro zadání licenčního souboru.
- „Save template“ slouží k uložení modelu v podobě šablony. Pokud si analytik vytvoří šablony, tak je může znovupoužít při vytváření nového modelu.
- „HTML...“ vytvoří projektovou dokumentaci ve formátu HTML.
- „PDF...“ vytvoří projektovou dokumentaci ve formátu PDF.
- „Code...“ spouští generátory binárních nebo jiných zdrojových souborů (source files) do jiných programových systémů, které Craft.CASE podporuje. Pomocí této volby lze exportovat modelová data z Craft.CASE a dále je zpracovávat jinými nástroji.

Ostatní volby budou vysvětleny v následujícím textu.

11.2.4 Ovládání oken, menu a zobrazení hodnot v Craft.CASE

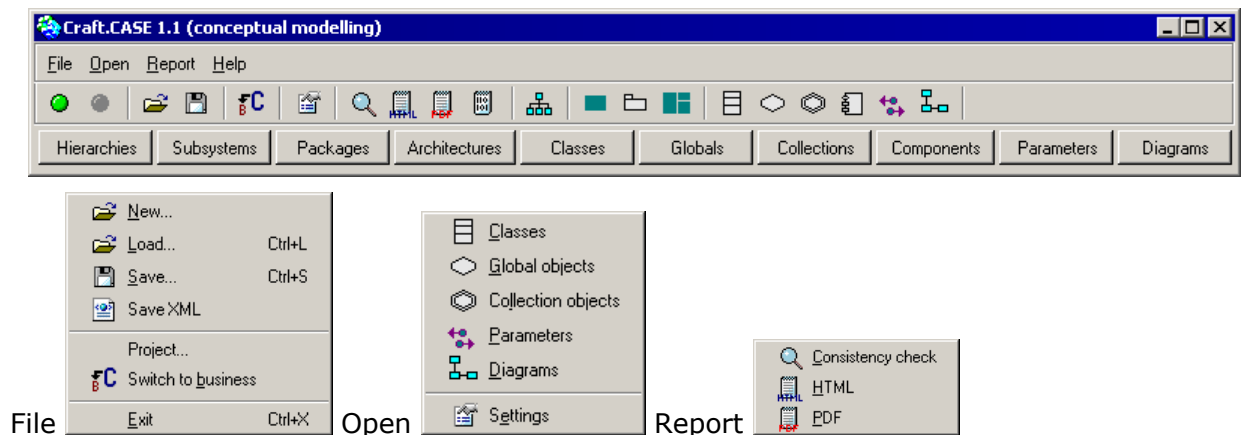
- Craft.CASE je navržen tak, že během práce s ním se otevírá několik samostatných oken na pracovní ploše obrazovky, mezi kterými si uživatel přepíná. Je třeba si dávat pozor na překrývání oken.
- Kromě viditelných tlačítek, menu a ikon jsou také přístupná kontextová menu na kliknutí pravým tlačítkem myši. Taková menu například slouží k přidávání nebo odebrání položek v různých seznamech a tabulkách.
- Některá políčka v tabulkách nejsou editovatelná. Klikne-li se ale na ně, vedle a nebo pod ně se ukáže jejich editor.
- Většina sloupců v tabulkách dokáže třídit hodnoty podle abecedy. Třídění se aktivuje kliknutím na titulek příslušného sloupce.

11.2.5 Business a konceptuální analýza v Craft.CASE, metoda BORM

Craft.CASE je sestaven tak, aby podporoval metodu BORM (Business and Object Relation Modeling). Podrobný výklad této metody lze nalézt v knize: Carda A., Merunka V., Polák J.: *Umění systémového návrhu - objektivně orientovaná tvorba informačních systémů pomocí původní metody*

BORM. Grada, Praha 2003. ISBN 80-247-0424-2 nebo v anglickém jazyce v knize Liu L., Roussev B. et al: *Management of the Object-Oriented Development Process*, Virgin Island 2005, ISBN 1-59140-605-6 a také v odborném článku Knott, Roger P.; Merunka, Vojtěch; Polák, Jiří: "*The BORM methodology: a third-generation fully object-oriented methodology*", *Knowledge-Based Systems* 3(10) 2003, Elsevier Science Publishing, New York.

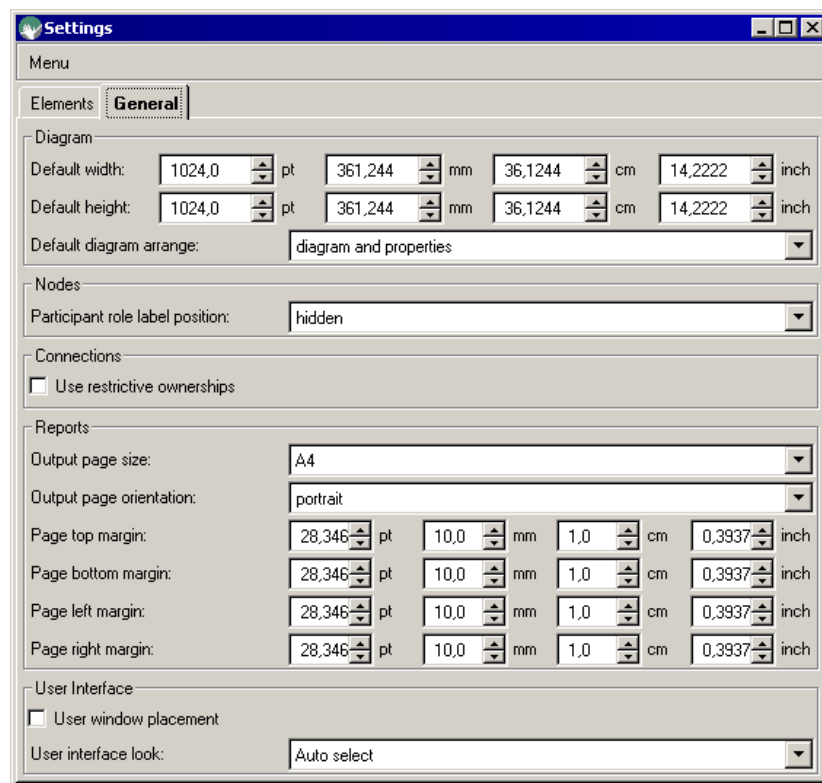
Craft.CASE verze 1.x podporuje nejen tu část metody BORM, která se týká analýzy procesů v systému, což je označováno jako „business“ analýza, ale také následnou a z předchozí vyplývající fází konceptuálního modelování softwarového systému, což je označováno jako „konceptuální“ analýza. Pro přechod mezi nimi slouží ikonky **B→C** a **B←C**. Po přepnutí do fáze konceptuální analýzy vypadá launcher následovně:



obr. 79. hlavní okno a menu pro konceptuální analýzu

11.2.6 Možnosti nastavení Craft.CASE

Položka Settings slouží k nastavení nástroje podle vlastních potřeb. Po spuštění se otevře následující okno:



obr. 80. nastavení programu

11.2.6.1 General properties

V záložce „General“ lze nastavit velikost stránky pro tisk a nastavení okrajů. Jsou tu ještě následující volby:

Default diagram arrange

Diagram and properties – V pravé části okna s diagramem se budou zobrazovat vlastnosti.

Diagram only – Okna s diagramy nebudou zobrazovat vlastnosti a bude je třeba vyvolávat ručně (například poklepáním myši).

Participant label position

Možnosti umístění textu s názvem uvnitř symbolů.

Use restrictive ownerships

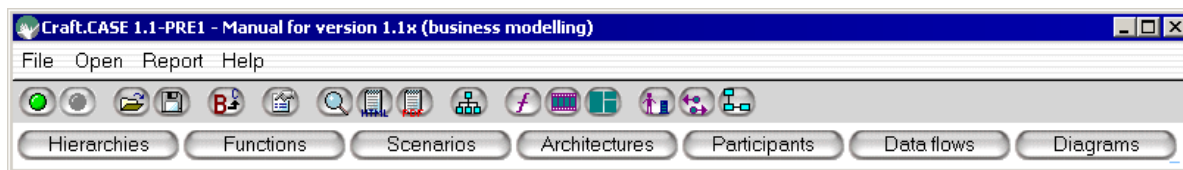
Pod pojmem „ownership“ se rozumí spojení mezi symboly. Je to například spojení mezi objektem a jeho aktivitou, kterou tento objekt provádí. Většina prvků v diagramech by neměla mít více než jeden takový vztah. Toto ale může komplikovat kreslení diagramů. Proto je možné pro snadnější grafické úpravy v diagramech toto pravidlo vypnout.

User window placement

Uživatelé zvyklí pracovat se softwarem v prostředí X-Window operačního systému UNIX mohou preferovat uživatelsky řízené otevírání oken a dialogů na obrazovce.

User interface look

Slouží k přepínání grafického vzhledu. Jsou podporovány vzhledy Windows Classic, Windows XP, MacOS X Aqua a Gnome. Na ukázce je launcher přepnutý na MacOS:

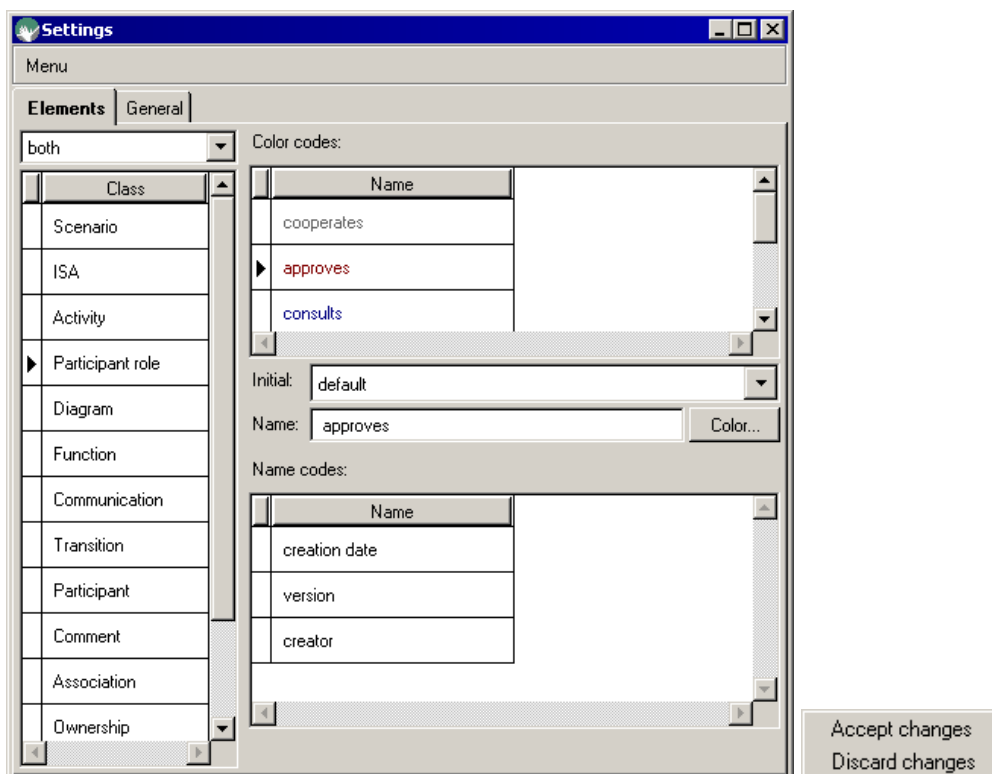


obr. 81. hlavní okno přepnuté do MacOS

11.2.6.2 Elements properties

V Craft.CASE mohou mít všechny používané pojmy uživatelsky nastavitelné vlastnosti. Pro větší uživatelský komfort lze k jednotlivým hodnotám takových vlastností přiřadit barvy. Je-li vlastnost nastavena barva, tak se při nastavení příslušné hodnoty změní barva pozadí nebo textu v diagramu.

Kromě toho lze uživatelsky definovat ke každému symbolu další atributy – označované jako „Name codes“. Tyto atributy slouží k ukládání dalších hodnot, které jsou v modelu potřeba. Všechny zde uvedené vlastnosti lze měnit i v již rozpracovaném modelu.



obr. 82. nastavení volitelných atributů

11.2.7 Metamodel Craft.CASE

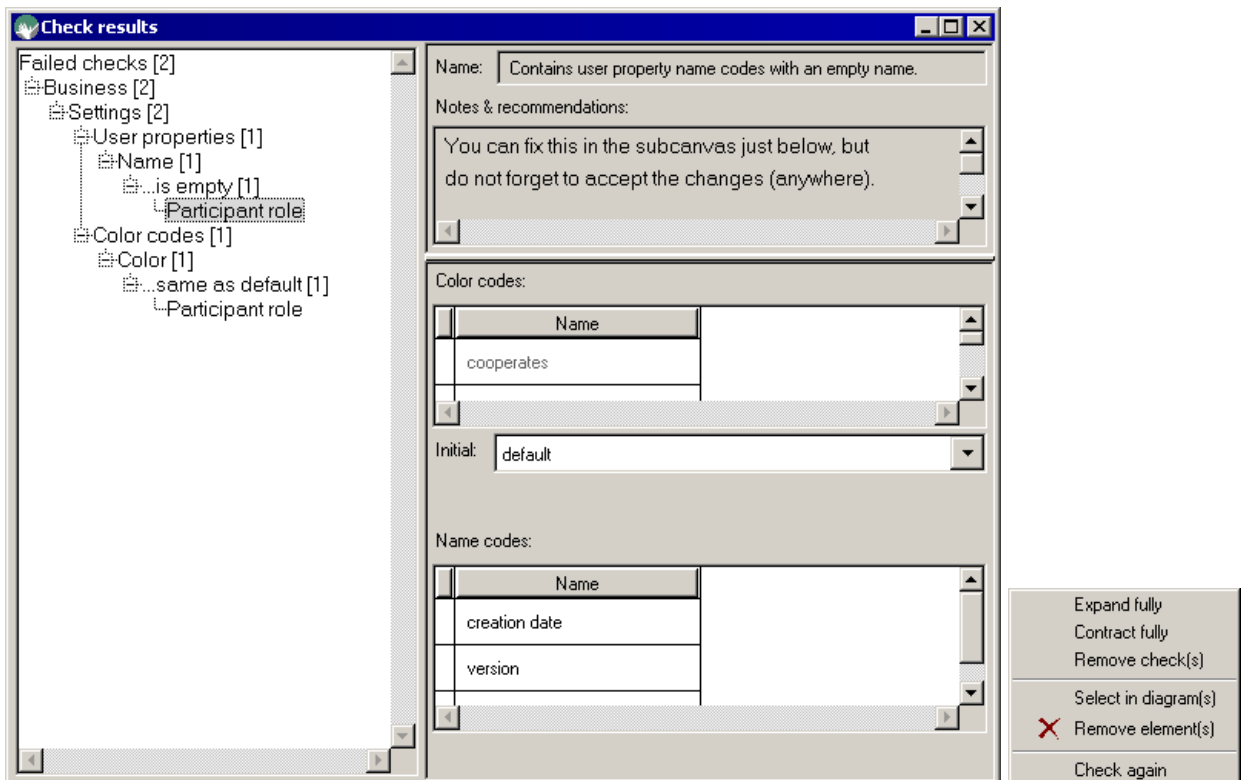
11.2.7.1 Uzly a spojení (Nodes and connections)

Všechny druhy diagramů i dalších dat v Craft.CASE jsou navrženy podle jednotné struktury. Prvky všech modelů jsou nazývány „nodes“ (uzly) a vazby mezi nimi „connections“ (spojení). Každý

uzel i spojení může mít různé proměnné. Uzly jsou například třídy objektů a spojení jsou například vazby skládání a dědění. Nebo podle jiného příkladu uzly jsou aktivity objektů a spojení jsou komunikace mezi aktivitami.

11.2.7.2 Kontrola úplnosti a správnosti modelu

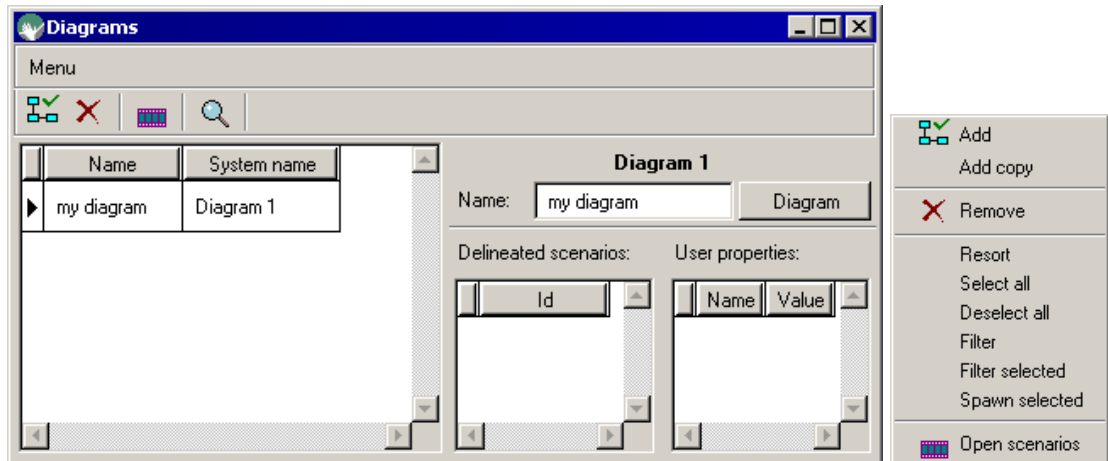
Craft.CASE při vytváření modelů dodržuje pravidla konkrétních uzlů a spojení a dovoluje modelovat jen takové údaje, které jsou pro příslušný typ uzlu a vazby přípustná. Kromě toho Craft.CASE obsahuje nástroj pro kontrolu konzistence a správnosti modelů, který hierarchickou formou zobrazuje nalezené nedostatky. Pro každou nalezenou chybu se zobrazí rada, jak ji lze napravit, dále editovatelné vlastnosti a pomocí menu pravého tlačítka myši lze najít nalezenou chybu přímo v diagramu:



obr. 83. kontrola úplnosti a správnosti modelu

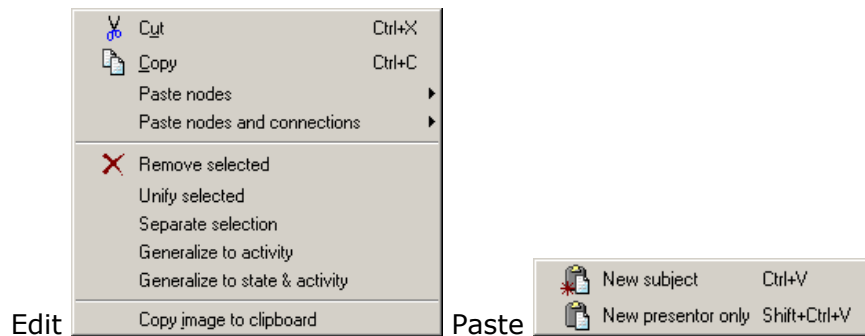
11.2.7.3 Identita objektů, presentory, kopírování objektů

- Diagramy lze kopírovat. Rovněž tak je možné kopírovat a vkládat objekty mezi diagramy.



obr. 84. editor diagramů

- Každý prvek (uzel i spojení) v databázi projektu má automaticky přiřazené jedinečné jméno označované jako „System name“, které se skládá z typu tohoto objektu a jeho pořadového čísla v databázi. Díky tomuto mechanismu je možné mít například v jednom modelu dva různé objekty se stejným jménem nebo mít přehled o nakreslených objektech, u kterých není ještě vyplněné jméno.
- Každý prvek se může v modelu vyskytovat vícekrát. V tomto případě se nejedná o více datových objektů se stejným jménem, ale pouze o více zobrazení stále stejného jednoho objektu v databázi. Různá zobrazení stejného objektu se nazývá „Presenter“. Takové zobrazení se vytvoří běžným kopírováním a vkládáním s volbou „Paste - presenter only“.
- Je-li v modelu více různých objektů nebo jen více presenterů, lze je spojit do jediného volbou „Unify selected“. Rovněž tak lze z více zobrazených presenterů jednoho objektu vytvořit samostatné objekty volbou „Separate selection“.



obr. 85. manipulace s presentory

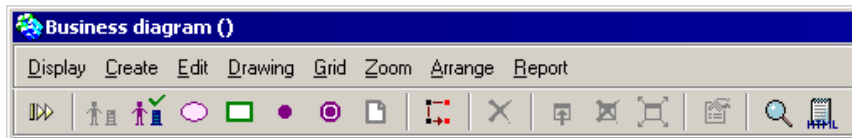
- Skupiny některých objektů (například aktivit v procesním diagramu) lze nahradit jediným objektem. (volba „Generalize“). Všechna spojení původních objektů jsou automaticky přepojena na nový objekt. Pokud se tato generalizace provede v kopii nějakého diagramu, tak v původním diagramu zůstávají původní objekty s původními spojeními. To znamená, že po provedené generalizaci je v diagramu nový objekt, jehož dekompozice zůstává v původním diagramu.

11.2.8 Práce s grafickým editorem

Grafický editor slouží ke tvorbě diagramů v Craft.CASE. Má dvě varianty: Pro kreslení diagramů business procesů a pro kreslení konceptuálních diagramů v následné fázi analýzy. Obě varianty však mají společný přístup ke tvorbě diagramů.

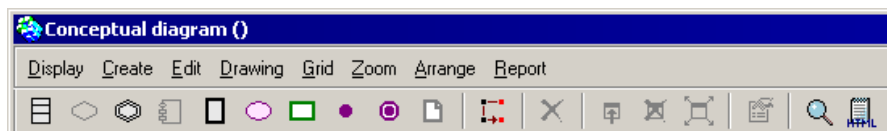
11.2.8.1 Menu

Grafický editor má následující menu pro variantu business procesů:



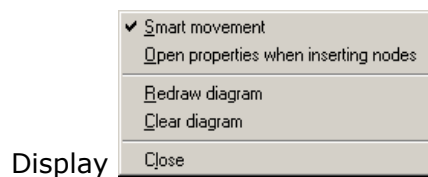
obr. 86. grafický editor business procesů

a pro konceptuální modelování:



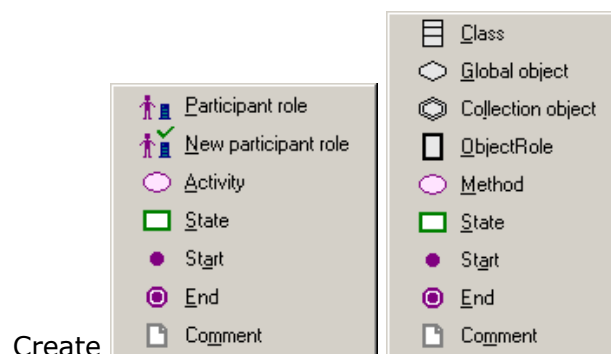
obr. 87. grafický editor konceptuálních diagramů

V této kapitole budou popsány funkce společné pro všechny varianty.



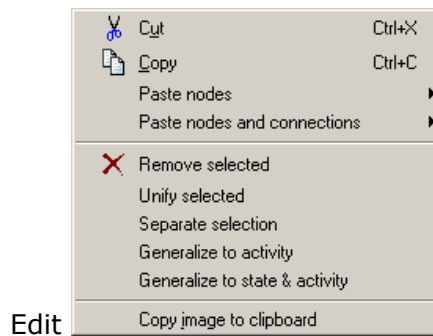
obr. 88. nastavení editoru diagramů

- **Smart movement.** Je-li tato volba aktivní, tak se při přesouvání objektu posunují i všechny objekty na něm závislé, aniž by je bylo třeba označovat. Toto se týká například přesouvání aktivit uvnitř symbolů stavů a rolí.
- **Open properties when inserting nodes.** Je-li tato volba aktivní, tak se při vkládání objektů automaticky otevírá nové dialogové okno s vlastnostmi tohoto objektu.



obr. 89. vkládání objektů do diagramu

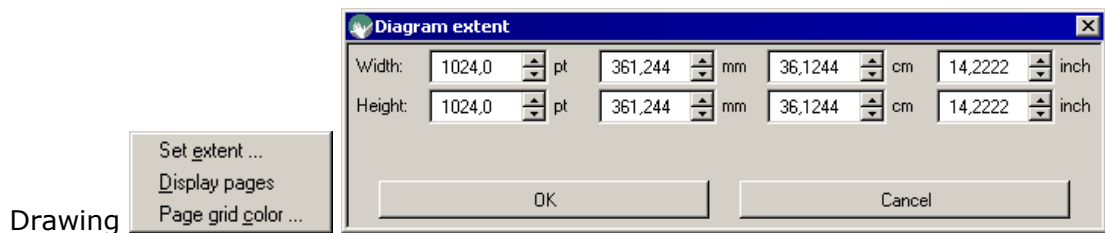
Tato menu slouží ke vkládání objektů do diagramu. Je třeba mít na paměti, že většina objektů musí být vytvořena předem a že jejich existence závisí na předchozích výsledcích analýzy. Tyto objekty se definují v seznamech, které se otevírají přímo z launcheru (kap. 4.)



Edit

obr. 90. editace objektů v diagramu

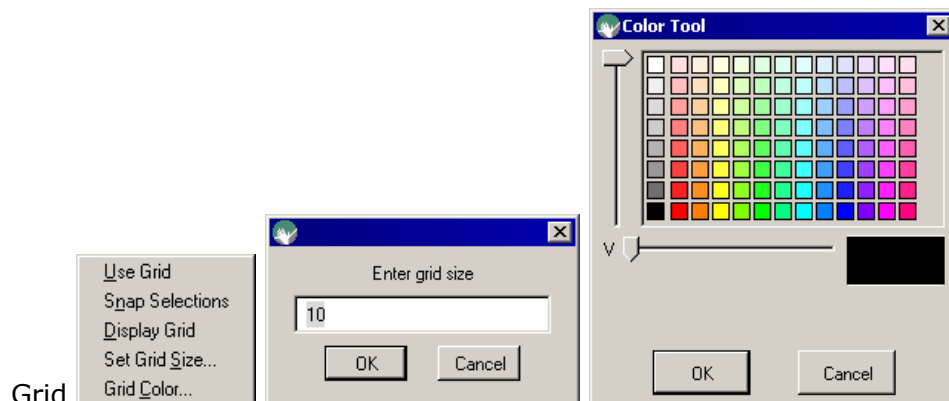
Toto menu obsahuje funkce popsané v kapitole 11.2.8.3. Poslední volbou lze vkládat diagram do schránky a vkládat do libovolného jiného programu ve Windows.



Drawing

obr. 91. nastavení možností kreslení

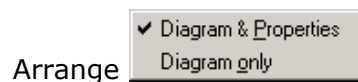
Nastavuje velikost diagramu a zobrazuje oddělovací čáry mezi stránkami papíru.



Grid

obr. 92. nastavení pomocné mřížky

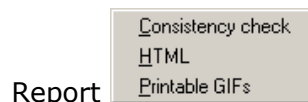
Nastavuje mřížku, na kterou se mohou zarovnávat grafické objekty. Volbou „**Snap Selections**“ se zarovnávají objekty, které byly nakresleny bez zarovnání.



Arrange

obr. 93. aktivace a deaktivace panely s vlastnostmi

Zde lze zvolit, jestli bude okno s grafickým editorem obsahovat pouze diagram a nebo zda se vedle diagramu v pravé části okna bude zobrazovat editor s vlastnostmi objektů.

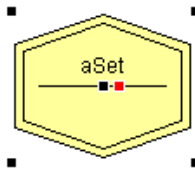


Report

Test konzistence je popsán v kapitole 11.2.7.2. Diagram lze také vypsat do HTML a GIF souborů.

11.2.8.2 Pomocné prvky při kreslení objektů - uzly

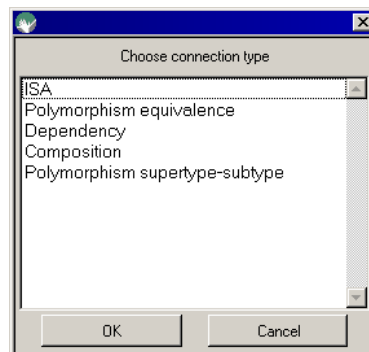
Každý grafický objekt má pět základních pomocných bodů. Vnější čtyři jsou určeny ke změně velikosti a polohy. Prostřední bod slouží k vytváření spojení (bude popsáno podrobněji v následující kapitole). Pokud jde o složitější objekt, tak jsou prostřední body dva – černý a červený. Například u symbolu množiny objektů se takto rozlišuje, zda se spojení týká celého objektu množiny a nebo jejích prvků.




obr. 95. pomocné body uzlů

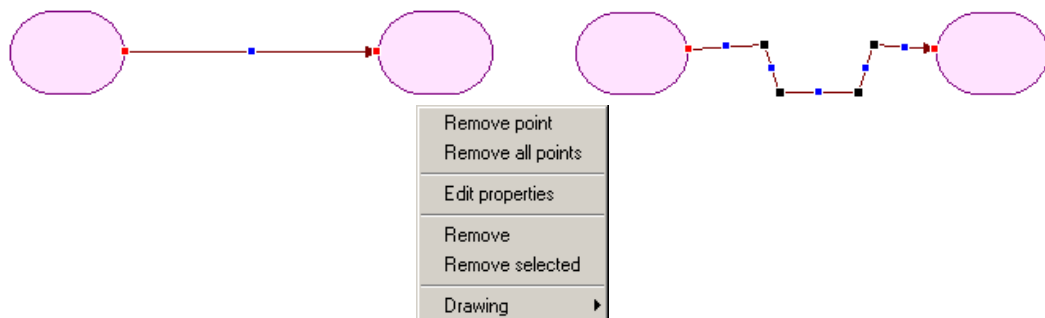
11.2.8.3 Pomocné prvky při kreslení objektů - spojení

Spojení mezi objekty se vytváří tak, že se myši propojí prostřední pomocné body objektů, mezi kterými se spojení vytváří. Pokud je k dispozici více variant vazeb, tak se otevře dialog, ve kterém je třeba požadovanou vazbu označit.



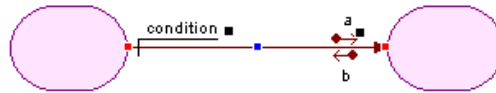
obr. 96. příklad dialogu

Zalomené spojení (vedené například okolo nějakého symboly) je možné nakreslit pomocí ikonky . Když je přímé nebo lomené spojení vytvořeno, je možné upravovat cestu, kudy vede. K tomu slouží modré body, ve kterých lze cestu členit na více úseček. Černé body slouží k přemístování. Pomocí červených bodů lze začátek a konec spojení přepojit k jinému objektu.



obr. 97. pomocné body spojení a menu

Na spojení se mohou také vytvářet další prvky. Jsou to například podmínky a datové toky na komunikacích. Tyto další prvky mají také černé pomocné body, pomocí kterých lze tyto prvky posunovat po celé cestě, kudy spojení vede.



obr. 98. další objekty na spojení

Mezi některými objekty je třeba vytvářet spojení, které vyjadřuje jejich příslušnost k sobě. Toto spojení se nazývá „ownership“. Craft.CASE toto spojení vytváří automaticky, pokud se symboly položí na sebe. Je to například spojení mezi stavy a aktivitami. Takové spojení se zobrazuje pouze, jsou-li objekty vedle sebe.

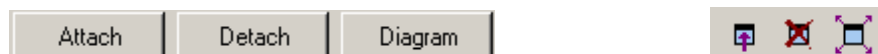


obr. 99. příslušnost objektů k sobě

11.2.8.4 Hierarchie v diagramech – dekompozice a generalizace

Dekompozice

Stavy a aktivity v business diagramu lze dekomponovat. To znamená, že na aktivitu nebo stav může připojit jiný diagram. Připojení diagramu, odpojení diagramu a otevření diagramu lze pomocí tlačítek ve vlastnostech aktivity a diagramu a také pomocí ikoněk na nástrojové liště.



obr. 100. nastavení dekompozice

Připojení diagramu k objektu je signalizováno malou ikonkou ve tvaru okénka v pravém dolním rohu příslušného objektu.

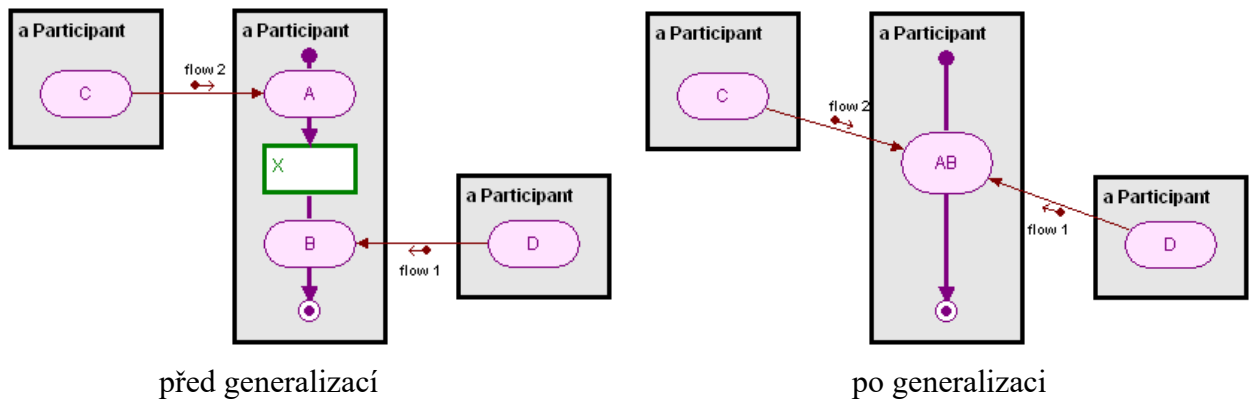


obr. 101. zobrazení dekompozice

Generalizace více stavů a aktivit

Jak již bylo uvedeno v kapitole 11.2.8.4, některé skupiny objektů je možné označit a nahradit jedním objektem. V business proces diagramu se tato vlastnost používá při úpravách sekvencí stavů a přechodů. Označenou sekvenci stavů a přechodů lze nahradit jedinou aktivitou a nebo jediným přechodem s aktivitou. Všechny komunikace, které vedly k nahrazeným aktivitám jsou zachovány a

vedou k nové aktivitě, která původní aktivity nahradila. Pokud si analytik vytvoří kopii diagramu, ve kterém chce provést generalizaci, tak také může nově vzniklou aktivitu propojit na kopii diagramu, kde zůstává původní „dekomponovaný“ model.



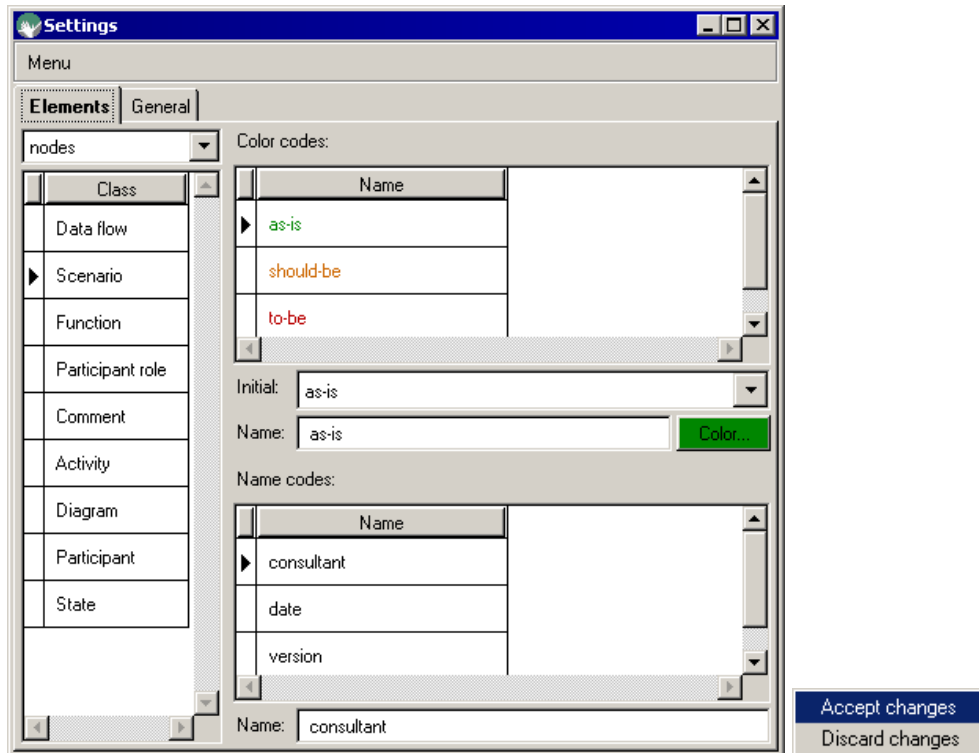
obr. 102. generalizace

11.2.9 Příprava k modelování

Na rozdíl od jiných nástrojů není v Craft.CASE v některých případech možné vkládat do diagramů objekty, které nebyly dříve definovány a začleněny do nějaké struktury. Metoda BORM je totiž založena na postupném odvozování nových pojmů z předchozích.

11.2.9.1 Nastavení parametrů

Pro konkrétní modelovaný problém je vhodné si nejprve rozmyslet, jaké atributy budou u jednotlivých objektů potřeba a nastavit je tak, jak se popisuje v kapitole 11.2.6. Například pro projekty zabývající se modelováním organizační a řídicí změny v nějaké organizaci je vhodné nastavit u scénářů atributy „as-is“, „should-be“ a „to-be“, které budou sloužit k rozlišení, zda se jedná o scénář popisující stávající proces, nebo zamýšlený proces nebo proces naplánovaný k implementaci. A do sady volitelných atributů přidat například „author“, „consultant“, „date“ a „version“.



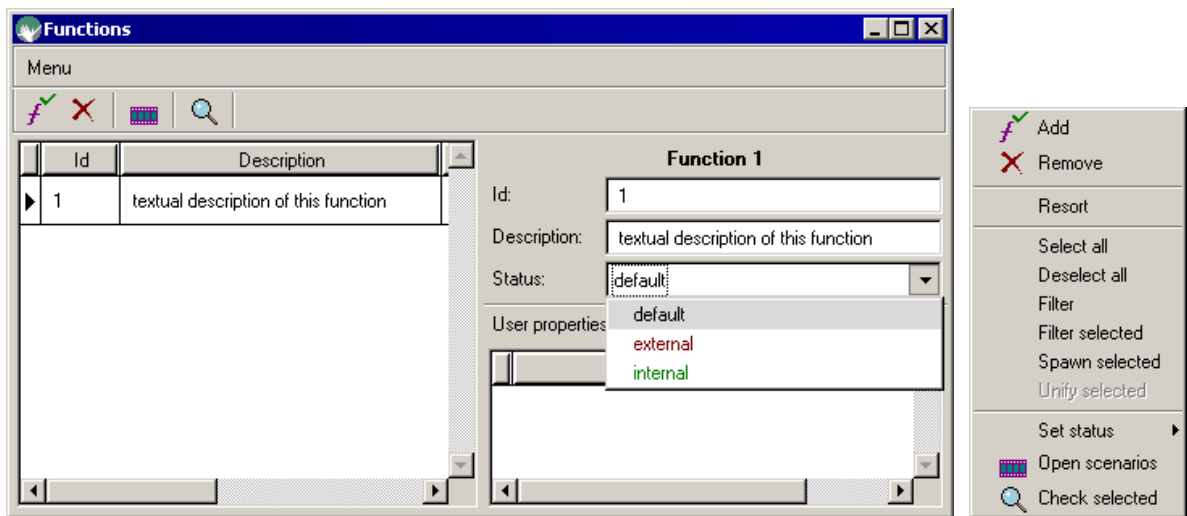
obr. 103. nastavení vlastností objektů před modelováním

11.2.10 Business modelování

První fází je fáze business modelování. Zde se analyzuje celý kontext modelovaného systému – především objekty a procesy v organizaci, pro kterou se systém analyzuje. Ve složitějších případech je třeba sestavit dvě sady modelů. První z nich je tzv. AS-IS model, který zobrazuje stávající stav a po jeho dokončení následuje tzv. TO-BE stav, který zobrazuje novou strukturu objektů a procesů po implementaci systému.

11.2.10.1 Požadované funkce

První popis procesů v systému jsou tzv. požadované funkce. Jejich seznam se spouští přímo z launcheru.

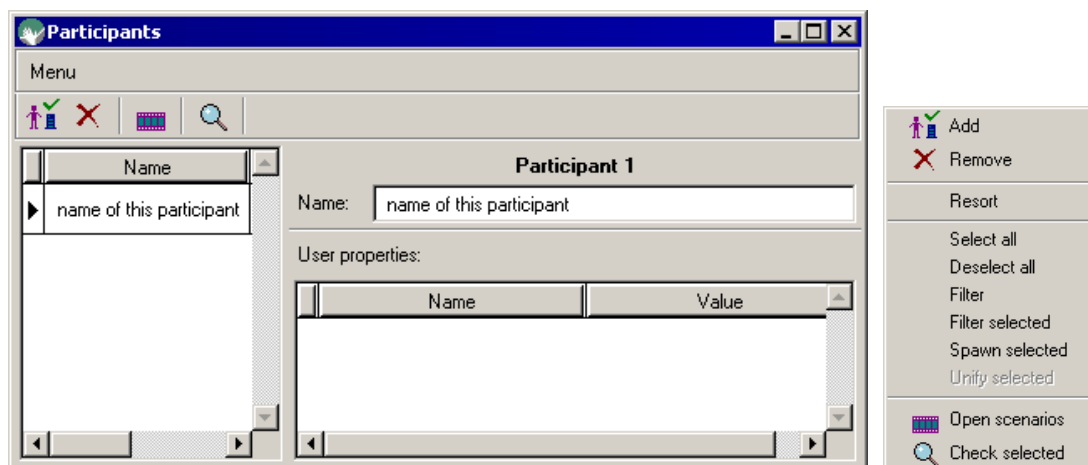


obr. 104. funkce systému

Předefinovaná vlastnost „internal“ a „external“ slouží k rozlišení, v jakém vztahu je daná funkce k systému, který je modelován. Podle metody BORM je totiž vhodné vyjmenovat i funkce, které jsou mimo okruh modelovaného zadání.

11.2.10.2 Participanti

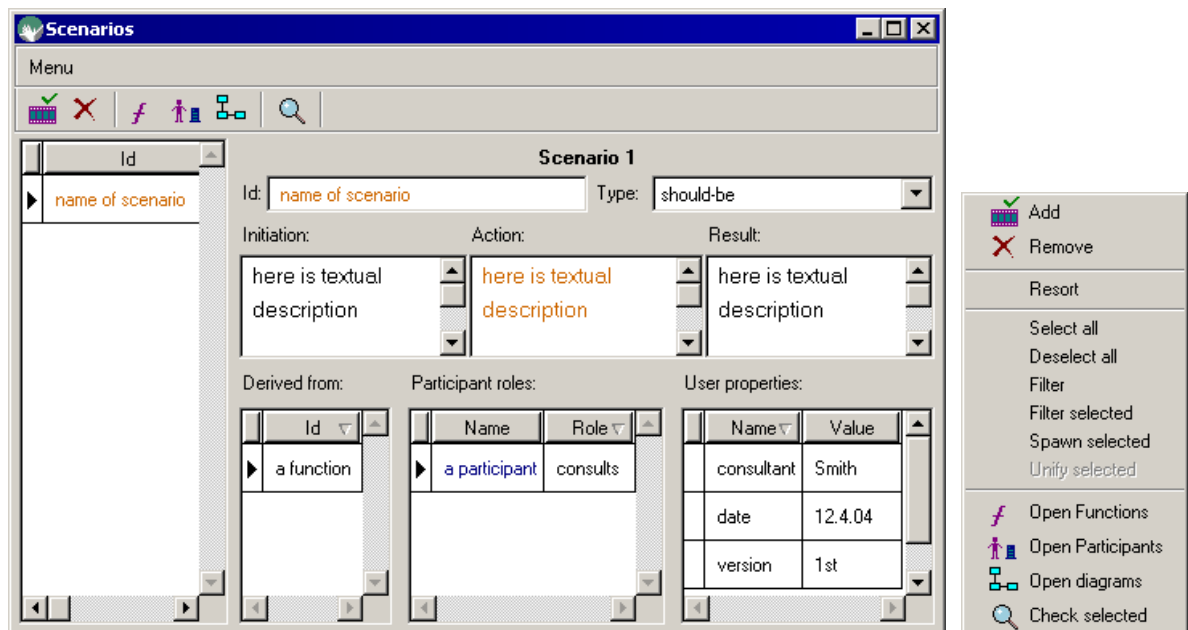
Participant je objekt, který se účastní procesů v systému. Mohou to být nejen živé bytosti, ale i stroje, informační systémy apod. Seznam participantů se spouští z launcheru.



obr. 105. participantů

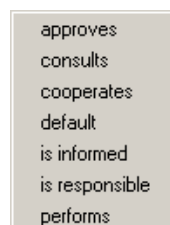
11.2.10.3 Scénáře

Scénář je podrobný popis procesu. Každý scénář by měl být odvozen z nějakého scénáře. Seznam scénářů se spouští z launcheru.



obr. 106. scénáře

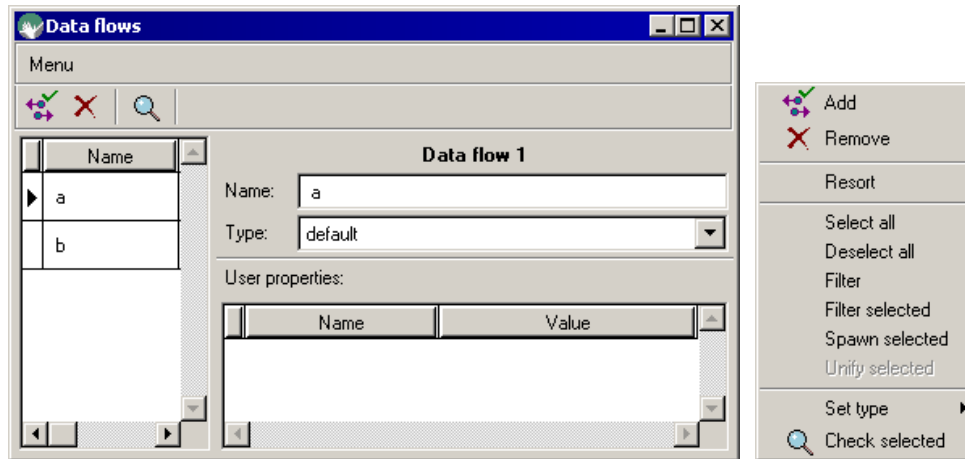
Součástí každého scénáře by měli být také participantů. U participantů lze nastavovat jejich různé role v modelovaném procesu. Předdefinovány jsou následující role, které mají vlastní barevné kódy:



obr. 107. předdefinované role participantů v procesech

11.2.10.4 Datové toky

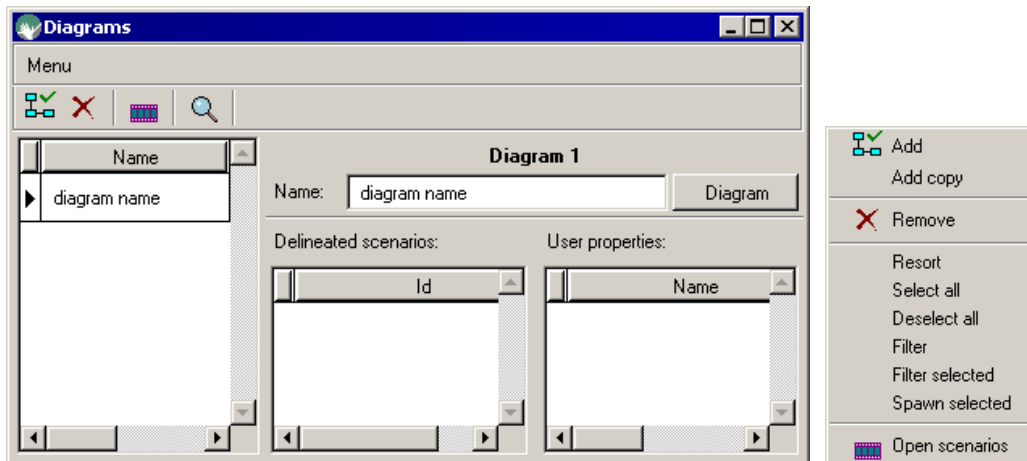
Datové toky jsou objekty, které si jiné objekty vyměňují při vzájemných komunikacích. Jsou to například informační, finanční nebo materiálové toky. Seznam datových toků se spouští z launcheru.



obr. 108. datové toky

11.2.10.5 Diagramy

K diagramům je vhodné přistoupit až po vyplnění scénářů (které byly odvozeny z funkcí), participantů a datových toků. Seznam diagramů se spouští z launcheru.



obr. 109. diagramy

Vlastní diagramy procesů mají následující syntaxi:

pojem	symbol	popis
role participantu	Obdélník se jménem zobrazeným uvnitř v levém horním rohu.	Představuje účastníka modelovaného procesu.
stav	Obdélník ohraničený zelenou barvou kreslený dovnitř symbolu pro roli participantu. (<i>Pro počáteční a koncový stav se používají symboly shodné s UML</i>)	Stavy vyjadřují postupné změny participantů v čase. Stavy lze dekomponovat na diagram.
asociace	Silná černá šipka s plným zakončením mezi rolmi participantů. U šipky se píše popis, který blíže specifikuje charakter vazby. Pojmy přirozeného jazyka zde mají přednost před programátorskými označeními typu "dědí", "skládá", ...	Asociace vyjadřují datově orientované vztahy mezi participanty (<i>Že se participanty z nějakého důvodu potřebují</i>). Asociace vyjadřují jednotným způsobem vztahy, které mohou být později upřesněny jako skládání, dědění nebo závislost objektů.
IS-A	Silná šedivá šipka s plným zakončením mezi rolmi participantů.	Vyjadřuje vztah nadtyp-podtyp mezi participanty.
aktivita	Ovál propojený čarou s participantem nebo jeho stavem. Ovály mohou být kresleny také dovnitř k nim příslušných objektů.	Aktivity reprezentují jednotlivé složky chování objektů. Aktivity lze dekomponovat na diagram.
komunikace	Šipka, která propojuje aktivity mezi sebou. Malé pojmenované šipky kreslené rovnoběžně k hlavní šipce komunikace vyjadřují datové toky.	Komunikace vyjadřují sled provádění a vzájemnou závislost aktivit různých objektů mezi sebou. datové toky mohou být vedeny oběma směry.
přechod	Šipka která propojuje aktivity a stavy jednoho objektu.	Součástí přechodu je také aktivita, ze které přechod vychází. Přechod s aktivitou představuje činnost, kterou je třeba vykonat, aby objekt změnil svůj stav.
podmínka	Přeškrtnutí s textovým popisem u komunikace nebo u propojení aktivity a objektu.	Podmínkou se vyjadřuje omezená platnost komunikace nebo aktivity.

obr. 110. syntaxe diagramu procesů

11.2.11 Konceptuální modelování

Tato fáze modelování v Craft.CASE je velmi podobná klasickým nástrojům CASE používající jazyk UML. Odlišnosti spočívají ve dvou věcech:

- Pojmy v této fázi modelování by měly vycházet z pojmů modelovaných v předchozí fázi. K tomuto je v Craft.CASE vyčleněna vazba „business origin“.
- Jazyk UML je zjednodušen a také doplněn o některé nové prvky za účelem lepší podpory objektově orientovaného konceptuálního modelování více nezávislého na implementačních prostředích smíšených programovacích jazyků (např. C++). Původní UML je totiž s těmito jazyky příliš těsně svázán. To nejen zbytečně komplikuje analýzu, ale také nedává dostatek výrazových prostředků pro implementaci v čistě objektově orientovaných prostředích a především objektových databázích.

11.2.11.1 Třídy

Podobně jako participanty je třeba třídy objektů definovat dříve, než jsou použity v diagramech. Seznam se otevírá z launcheru.

11.2.11.2 Globální objekty

Globální objekty jsou vyčleněné a pojmenované konkrétní instance tříd, které mají pro model takový význam, že pro ně nestačí symbol třídy, ve které jsou instancí. Seznam se otevírá z launcheru.

11.2.11.3 Sady objektů

Sady jsou skupiny objektů – například množiny instancí různých tříd. Množinami lze také vyjádřit různé skupiny objektů, které jsou instancemi stejné třídy. Jsou to například úložiště objektů v objektových databázích. Seznam se otevírá z launcheru.

11.2.11.4 Komponenty

Komponenty jsou skupiny objektů, které tvoří navenek jeden celek tak, že ho lze považovat za jediný objekt se svými metodami.

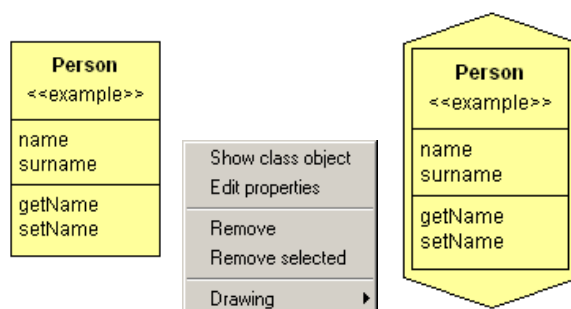
11.2.11.5 Parametry

Jedná se o datové toky na zprávách mezi metodami. Seznam se otevírá z launcheru.

11.2.11.6 Diagramy

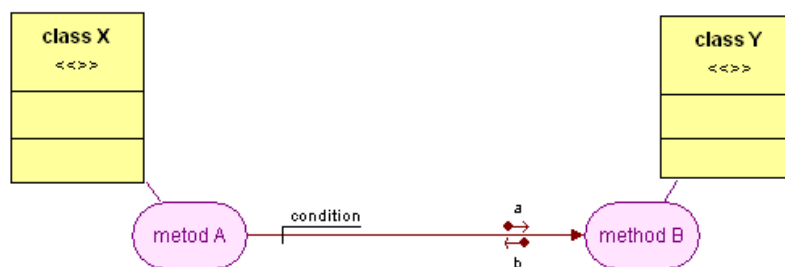
Craft.CASE používá v souladu se zásadami BORMu jen jeden typ konceptuálního objektového diagramu. Tento diagram vychází z diagramu tříd UML, ale dovoluje také zobrazovat zprávy mezi objekty, metody objektů a stavy objektů.

- Symbol třídy je shodný se symbolem podle UML. Navíc je ale možné k tomuto symbolu zobrazit tzv. „class object“, který reprezentuje třídu jako objekt. Spojení je potom možné vytvářet jak k obdélníku, tak i k šestihranu. Tím se rozlišuje, zda se vazba týká instancí třídy a nebo vlastního objektu třídy.



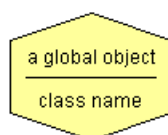
obr. 111. třída a objekt třídy

- Metody jsou analogické aktivitám z business modelování. Mezi metodami lze zobrazit zprávy s podmínkou, parametry i návratovou hodnotou.



obr. 112. metody objektů a zprávy mezi objekty

- Globální objekty mají symbol šestiúhelník.



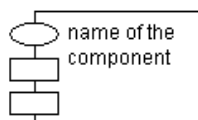
obr. 113. globální objekty

- Sady objektů mají také symbol šestiúhelník. Spojení vedená k vnitřnímu šestiúhelníku se týkají objektů-prvků uvnitř sady. Spojení vedená na vnější šestiúhelník se týkají vlastní sady jako celého objektu.




obr. 114. sady objektů

- Komponenty mají symbol dle standardu UML:



obr. 115. komponenty

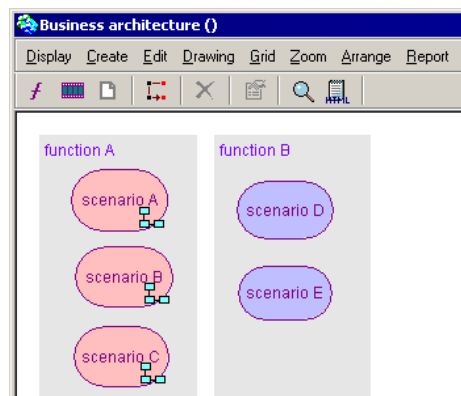
11.2.12 Architektury

V případě složitějších modelů může být sada business a konceptuálních diagramů příliš velká a tím pádem nepřehledná. Proto je možné jak business tak i konceptuální model zjednodušovat. Toto zjednodušení je zobrazitelné jako jeden či více diagramů, ze kterých lze pomocí dekompozice popsané v kapitole 11.2.8.4 otevírat celé business nebo konceptuální diagramy. Jedná se tedy o diagramy vyšší (nadřazené) úrovně abstrakce, než jsou obyčejné diagramy business procesů nebo konceptuální diagramy. Architektury se otevírají tlačítkem s nápisem „Architecture“ nebo ikonou  na hlavním panelu.

11.2.12.1 Business architektura

Business architektura je popsána jedním nebo více diagramy, ve kterém lze znázornit vztahy mezi funkcemi a scénáři v grafické podobě:

Ikonky v levém dolním rohu scénářů A, B a C znamenají, že tyto scénáře lze dekomponovat na diagramy business procesů.

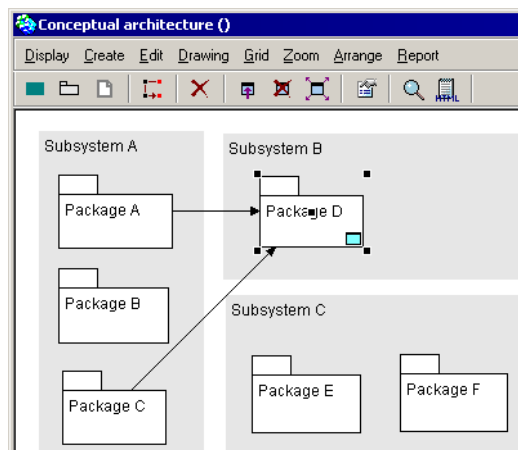


obr. 116. business architektura

11.2.12.2 Konceptuální architektura

Konceptuální architektura je popsána jedním nebo více diagramy, ve kterých lze znázornit vztahy mezi softwarovými subsystémy (Subsystems) a balíčky (Packages) v nich. Každý balíček lze poté dekomponovat na konceptuální diagram:

Na uvedeném příkladu jsou tři subsystémy a pět balíčků. Balíček „D“ je závislý na balíčcích „A“ a „B“ a obsahuje ještě dekompozici na konceptuální diagram, který popisuje jeho vnitřní strukturu.




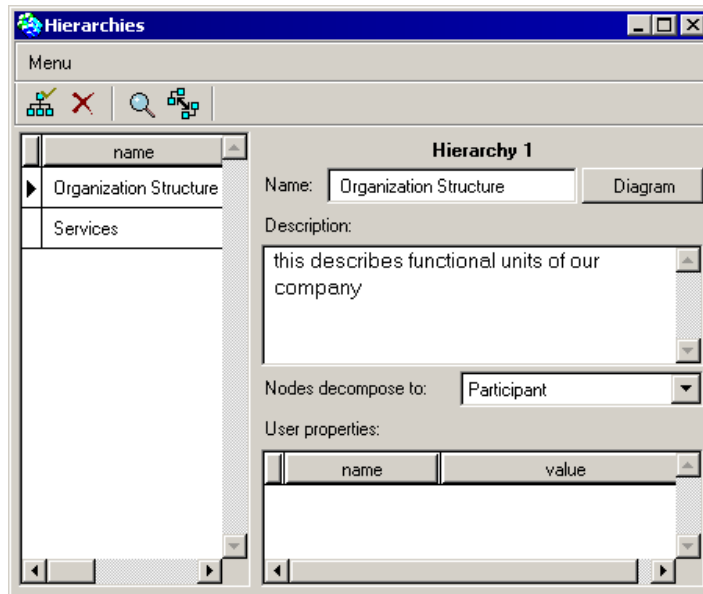
obr. 117. konceptuální architektura

11.2.13 Pomocné hierarchie

Při modelování procesů a požadavků na informační systémy je také potřeba tyto modely začlenit do celkového kontextu firmy nebo organizace, pro kterou se tento model vytváří a analyzuje. Craft.CASE umožňuje tento kontext modelovat ve formě uživatelsky volitelných hierarchií. Počet hierarchií není omezen. V následujícím příkladu budou jako příklad použity dvě hierarchie: Organizační struktura firmy a služby, která firma poskytuje.

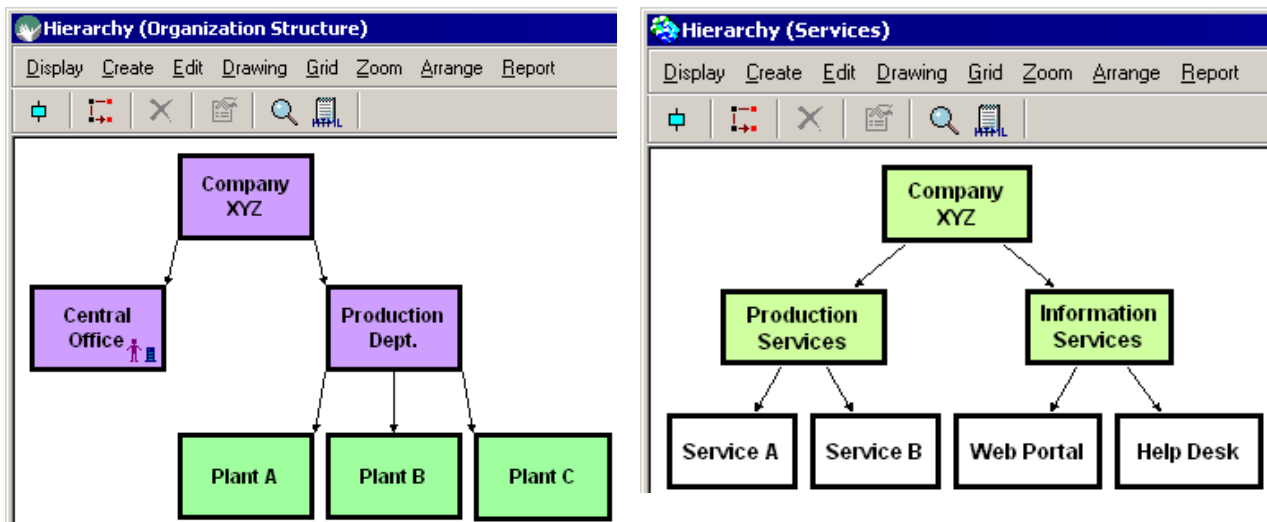
11.2.13.1 Vytvoření podpůrných hierarchií

Nástroj k vytváření hierarchií se otevírá z hlavního panelu tlačítkem „Hierarchies“ nebo kliknutím na ikonu .



obr. 118. vytváření hierarchií

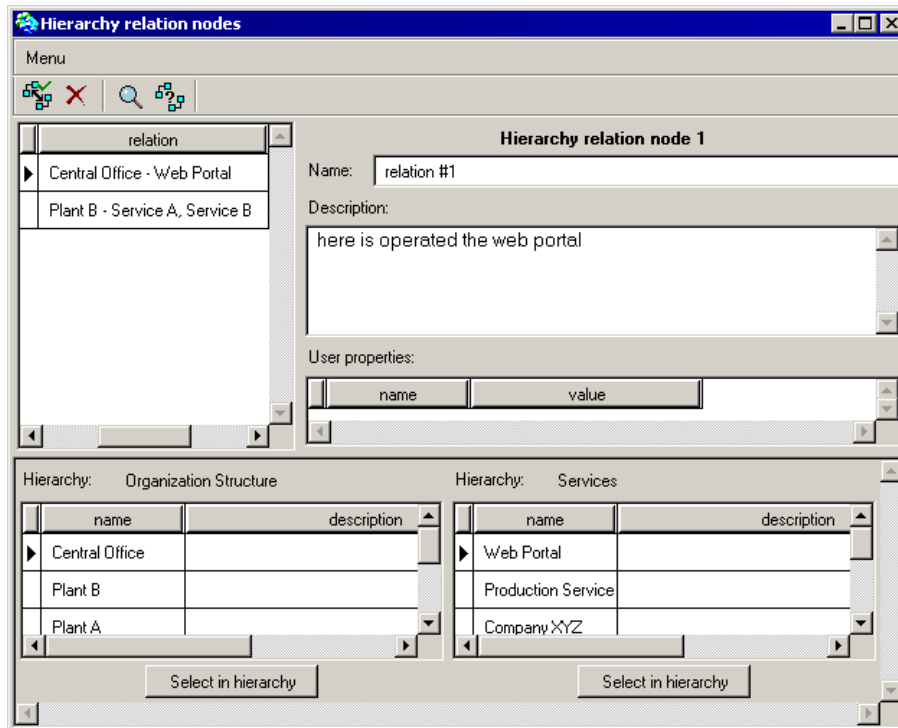
Byly vytvořeny dvě pomocné hierarchie: Organizační struktura a Služby. Pro elementy těchto hierarchií je možné definovat pomocné proměnné nebo barevné kódy stejným způsobem, který je popsán v kapitole 11.2.6.2. Každá vytvořená hierarchie je zobrazitelná diagramem:



obr. 119. příklady hierarchií – organizační struktura a služby podniku


11.2.13.2 Vazby mezi podpůrnými hierarchiemi

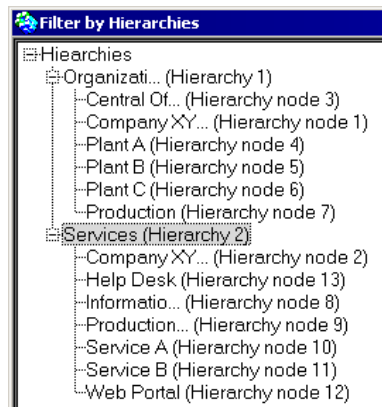
Elementy mezi hierarchiemi lze propojovat. K tomu slouží nástroj, který se spouští ikonkou



obr. 120. vazby mezi hierarchiemi

Pomocí tlačítek „Select in hierarchy“ se otevírají příslušné hierarchie s vyznačenými elementy, které jsou součástí propojení.

Ikonkou  se spouští vyhledávací dialog, ve kterém se všechny elementy zobrazují ve formě stromu. Tento dialog slouží k nalezení všech propojení, ve kterých se vyskytuje požadovaný element:





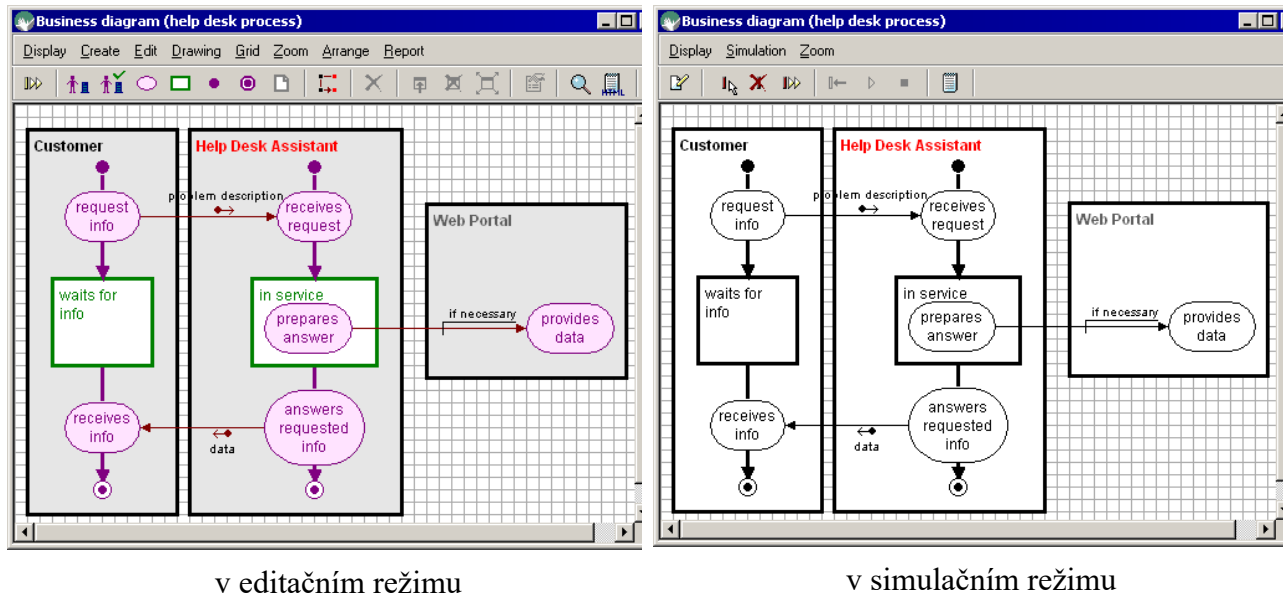
obr. 121. vyhledávání spojení mezi hierarchiemi

11.2.13.3 Propojení mezi podpůrnými hierarchiemi a modelem v Craft.CASE

Při vytváření podpůrné hierarchie jak je popsán v kapitole 11.2.13.1, je možné pro každou takovou hierarchii nastavit, do jakých prvků modelu je možné dekomponovat elementy této hierarchie. Tato vlastnost se nastavuje volbou „Node decompose to...“. Na příkladu z obr. 118 je propojení mezi elementy „Organization Structure“ a Participanty. (ikonka u elementu Central Office)




11.2.14 Simulátor

Business proces model je možné simulovat. Pro přechod do simulátoru slouží ikonka  editoru business diagramů. Ze simulátoru se vrací zpět do režimu editace diagramu ikonkou . Po zapnutí simulátoru se změní vzhled modelu i panel nástrojů:






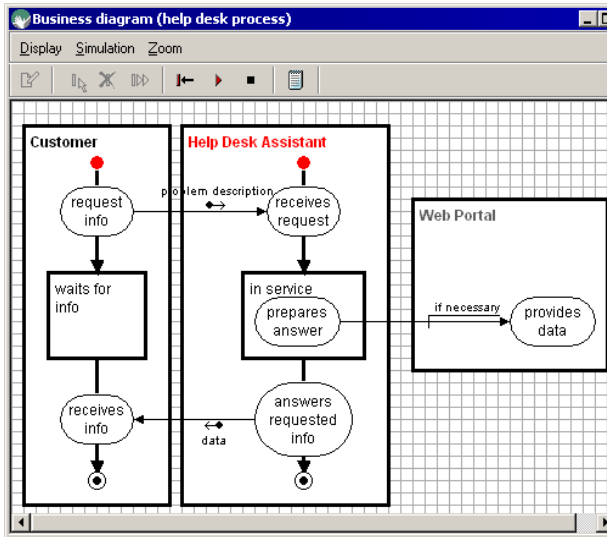
obr. 122. editační a simulační režim

11.2.14.1 Příprava simulace

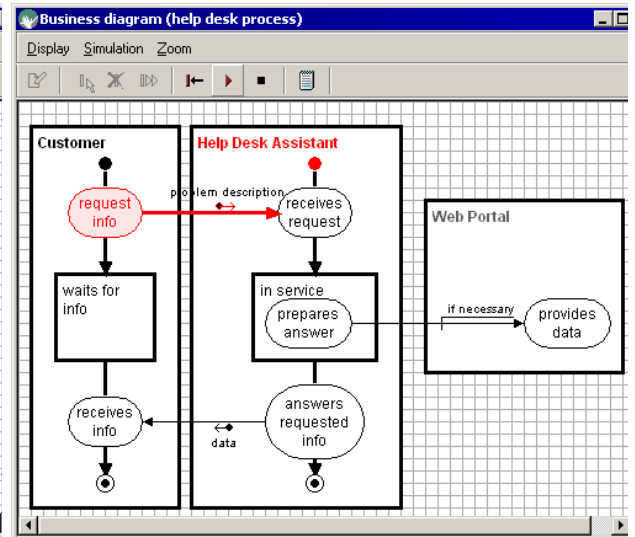
V simulačním režimu je třeba nejprve nastavit výchozí podmínky simulace. Ikonou  se zahajuje výběr startovacích míst pro simulaci, ikonou  se výběr startovacích míst ruší a ikonou  se zahajuje vlastní proces simulace.

11.2.14.2 Provedení simulace

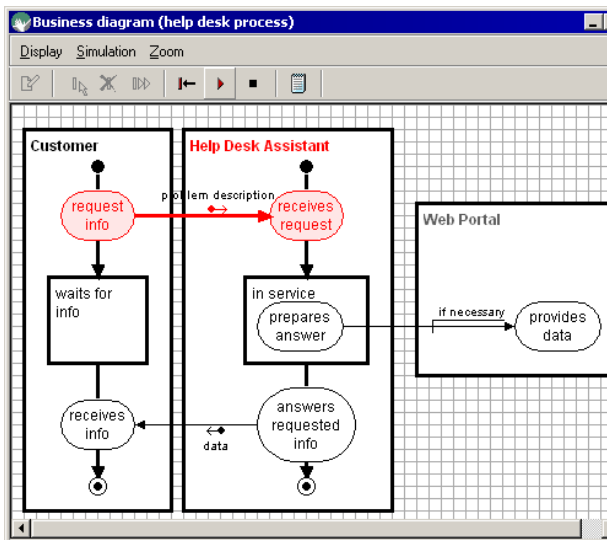
Je-li simulace připravena, je možné proces krokovat. Ikonou  se provádějí jednotlivé kroky, ikona  nastaví simulaci na začátek a ikona  simulaci ukončí. Jsou-li v procesu nějaké podmínky, simulátor spouští dotazy ve formě dialogových oken, na které musí uživatel odpovědět. Průběh simulace ukazuje následující sekvence:



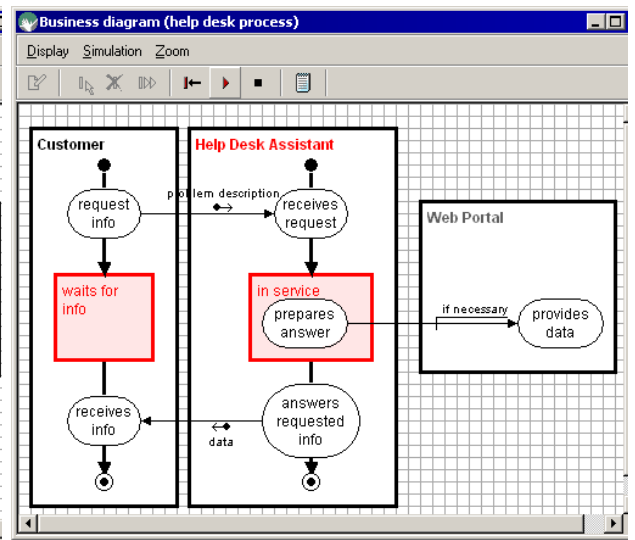
1.



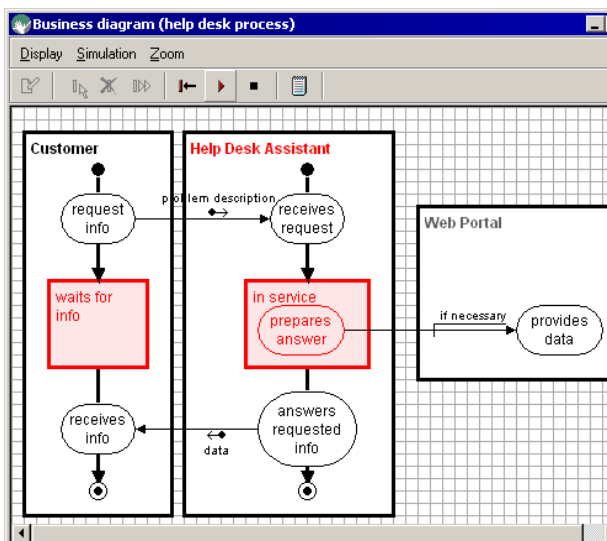
2.



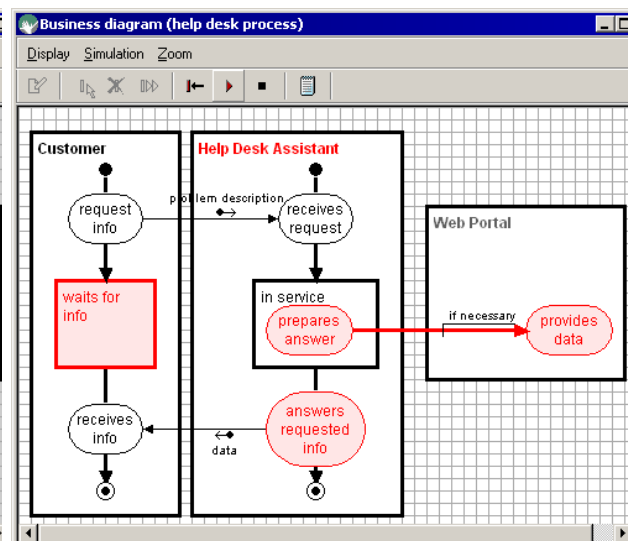
3.



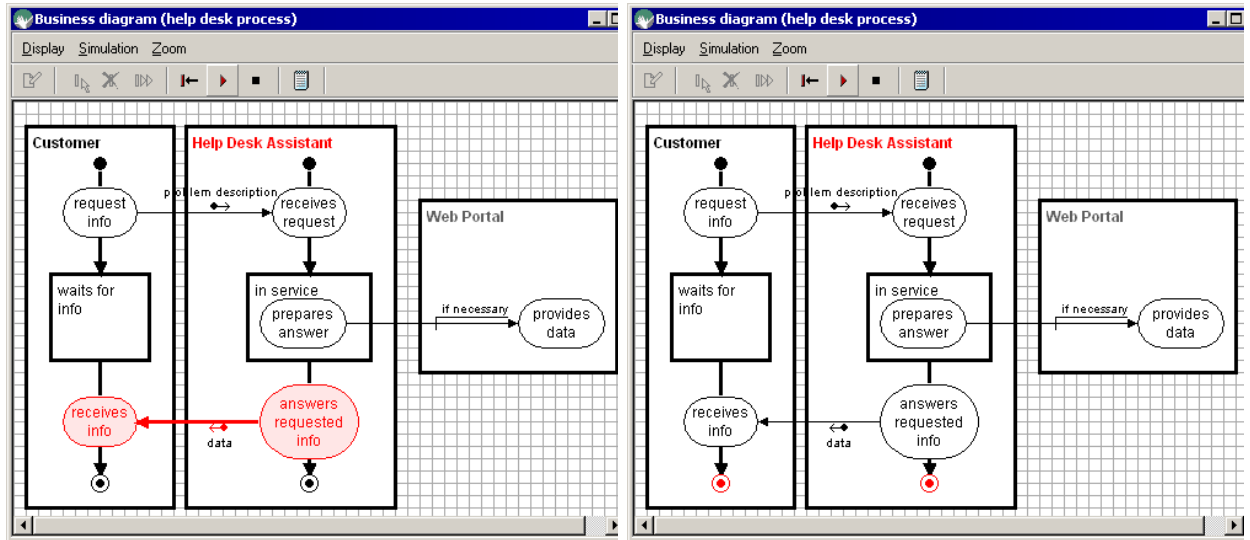
4.



5.



6.




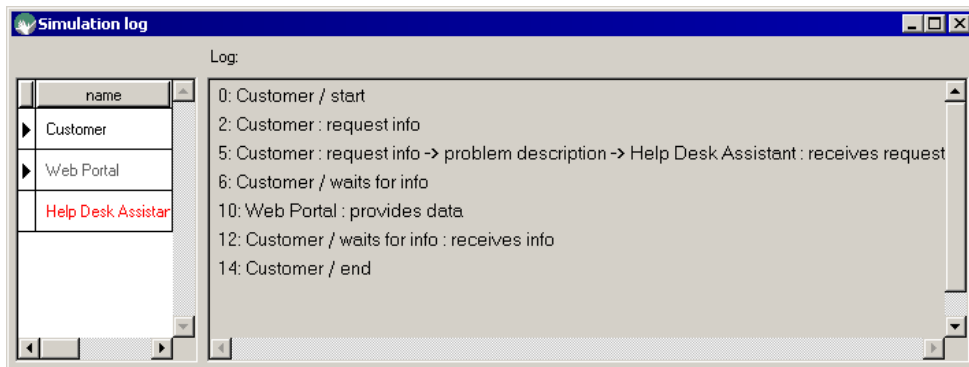
7.

8.

obr. 123. příklad simulace

11.2.14.3 Záznam simulace

Během simulace a nebo až po ukončení simulace je možné prohlížet simulační záznam (simulation log). Simulační záznam se spouští ikonou . Simulační záznam je možné kopírovat jako text.



obr. 124. záznam simulace

V tomto nástroji je možné označit více než jednoho participanta (nebo i všechny). To umožňuje prohlížet průběh vzájemné komunikace mezi participanty, jak probíhala při simulaci procesu.

11.2.15 Možné uživatelské problémy a jejich řešení

problém	řešení
Nejde vložit údaj do políčka v tabulce.	Po značení řádku v tabulce se pod tabulkou nebo vedle ní objeví editor. Některé údaje lze také vložit pomocí menu pravého tlačítka myši.
Nejde nakreslit vazbu mezi dvěma objekty.	Každá vazba je podle metody BORM orientovaná. Pokuste se ji nakreslit v opačném směru. Vazby se totiž vytvářejí od nadřazeného objektu k podřazenému.
Do diagramu nejde vložit objekt protože nabídkový dialog ho neobsahuje.	Objekt musí být nejprve vložen do databáze. Pouze stavy a aktivity (metody) je možné vkládat přímo.
V business proces diagramu se při vkládání nenabízí žádný participant, i když je zadán v databázi.	Je třeba, aby participant měl nějakou roli v nějakém ze scénářů, ze kterých diagram vychází.
V business procesu se musí při vkládání participanta označit i scénář.	Vkládaný participant nemá roli v žádném ze scénářů, ze kterých diagram vychází. (Ale vložit ho do diagramu je možné. Po jeho vložení do diagramu se také přidá jeho role do scénáře.)
Je možné nakreslit i vztahy, které porušují pravidla BORMu.	Některá pravidla, pokud by se kontrolovala již během kreslení, by mohla komplikovat postupnou tvorbu diagramu. Proto se kontrolují až při kontrole konzistence. Tam lze chyby jednoduše opravit.
Na pravém nebo dolním okraji diagramu se objekty zobrazují jen částečně nebo nefunguje jejich přesunování.	Objekty přesahují okraj kreslicí plochy. Je třeba zvětšit rozměry diagramu.
Kde jsou modelové karty participantů?	Modelové karty jsou součástí souhrnného HTML nebo PDF výstupu.
Při přesouvání objektů se zároveň přesunuje i jejich obsah, i když není označen.	Je zapnuta volba „smart movement“. (v menu „Diagram“)
Při přesouvání objektů jejich obsah zůstává na původním místě a posunuje se jen podklad.	Je vypnuta volba „smart movement“. (v menu „Diagram“)
Nelze nakreslit spojení mezi dvěma objekty, i když by tam podle pravidel mělo být.	Je zapnuta kontrola „restrictive ownerships“ a objekt už jednu vazbu má – například vazbu „ownership“ mezi rolí participanta a aktivitou. (Lze nastavit z launcheru Settings -> general).
I když je aktivní „use grid“, tak se nakreslený objekt nezarovnává na mřížku.	Objekt byl nakreslen dříve než byla zapnuta volba použití mřížky. Objekt je třeba označit a zvolit volbu „Snap selection“. Tím se zarovná.
Nejdou zadat „User properties“. Tabulka je prázdná.	Nejprve je potřeba v „Settings“ launcheru zadat, jaké proměnné jsou potřeba. V tabulce vlastností konkrétního objektu se potom objeví prázdná políčka k vyplnění.

obr. 125. možné chyby a jejich řešení

11.2.16 XML výstup

Craft.CASE umožňuje vypsát strukturu projektové databáze do XML souboru. V tomto souboru je uložena úplná informace o všech objektech a vazbách. Tyto údaje jsou využitelné ke tvorbě generátorů různých dalších výstupů, zdrojových kódů programovacích jazyků a nebo souborů pro přenos dat do jiných modelovacích nástrojů.

11.2.16.1 Syntaxe XML souboru

Craft.CASE využívá standard SIXX (Smalltalk Instance Exchange XML) Masashiho Umezawy. Standard SIXX patří mezi nástroje vývojového prostředí VisualWorks/Smalltalk firmy Cincom (<http://www.cincom.com>).

I když je SIXX primárně určen pro jazyk Smalltalk, tak jeho syntaxe je natolik jednoduchá, že XML data v tomto formátu mohou být snadno zpracovávána i v jiných programovacích jazycích.

Základní formát SIXX pro uložení objektu je na ukázce objektu třídy `ClassName` s instančními proměnnými `firstVariable`, `secondVariable` a `thirdVariable` s hodnotami 100, 200 a 300:

```
<sixx.object sixx.id="0" sixx.type="ClassName">
  <sixx.object sixx.id="1" sixx.name="firstVariable" sixx.type="SmallInteger">100</sixx.object>
  <sixx.object sixx.id="2" sixx.name="secondVariable"
sixx.type="SmallInteger">200</sixx.object>
  <sixx.object sixx.id="3" sixx.name="thirdVariable" sixx.type="SmallInteger">300</sixx.object>
</sixx.object>
```

Skalární objekty jako čísla a řetězce znaků se ukládají následovně:

```
<sixx.object sixx.id="0" sixx.type="SmallInteger">100</sixx.object>
<sixx.object sixx.id="0" sixx.type="String">this is a text</sixx.object>
```

Položka `id` zajišťuje identitu objektů uvnitř XML souboru. Je-li v SIXX souboru odkazován objekt, jehož hodnota již byla zavedena dříve, jeho odkaz má tvar:

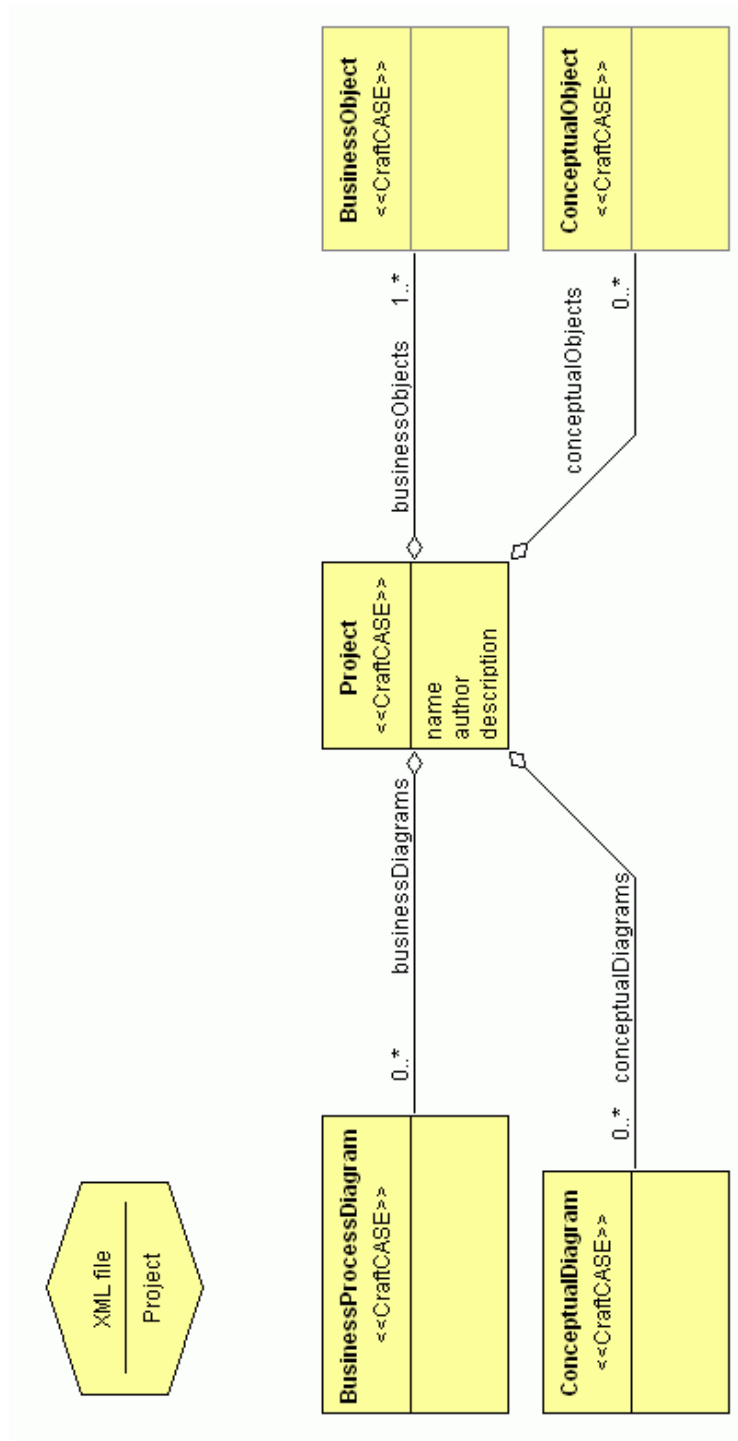
```
<sixx.object sixx.idref="123"></sixx.object>
```

Sady objektů se ukládají velmi obdobným způsobem jako objekty s instančními proměnnými (chybí pouze atribut `sixx.name`):

```
<sixx.object sixx.id="0" sixx.type="List">
  <sixx.object sixx.id="1" sixx.type="SmallInteger">100</sixx.object>
  <sixx.object sixx.id="2" sixx.type="SmallInteger">200</sixx.object>
  <sixx.object sixx.id="3" sixx.type="SmallInteger">300</sixx.object>
</sixx.object>
```

11.2.16.2 Datová struktura XML souboru

V XML souboru se celý projekt ukládá jako jediná instance třídy Project. Všechny objekty, vazby a diagramy jsou uloženy v jeho proměnných.



obr. 126. XML schéma

11.2.16.3 Hierarchie tříd XML objektů

Za jménem třídy jsou uvedeny instanční proměnné, které se dědí také všem podtřídám. Podtřídy jsou zobrazeny vždy pod danou nadtřídou odsazeně.

Project

name author description businessObjects businessDiagrams conceptualObjects conceptualDiagrams

ConceptAbstract name description systemName userProperties

BusinessComment

BusinessObject

Activity type attachedDiagram

BusinessStartState

BusinessState type attachedDiagram

BusinessStopState

DataFlow type

Function status

Participant

Scenario initiation action result functions participantRoles type

ConceptualComment

ConceptualObject businessOrigins

Class category type instanceVariables methods classObject

Collection className type ofCollectionObject

CollectionObject type className element

ConceptualStartState

ConceptualState type

ConceptualStopState

GlobalObject type className

Method hasDependents isDependent type

ObjectRole type

Parameter type

ConnectionAbstract from to

Association

ClassAssociation fromCardinality toCardinality

BusinessCommentConnection

BusinessISA

BusinessOwnership condition

BusinessTransition condition

Communications condition inputsFlows outputFlows

Composition type cardinality isAggregation

ConceptualCommentConnection

ConceptualISA

ConceptualOwnership condition

ConceptualTransition condition

Delegation

Dependency

ElementOf

Message condition parameters returnValue isAsynchronous

PolymorphismEq protocol

Polymorphism

DiagramAbstract objects connections

BusinessProcessDiagram scenarios

ConceptualDiagram businessOrigins

InstanceVariable

SimpleConceptAbstract

ClassObject instanceVariables methods ofClass

ParticipantRole participant role