VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA PODNIKATELSKÁ
ÚSTAV INFORMATIKY

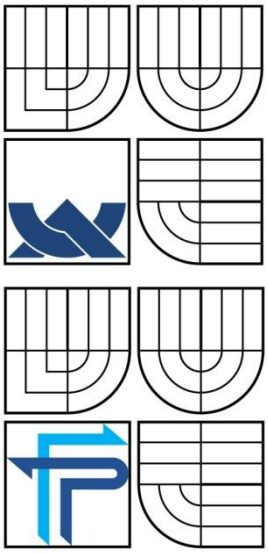
FACULTY OF BUSINESS MANAGEMENT
DEPARTMENT OF INFORMATICS

METODIKA NÁVRHU ARCHITEKTURY SW INFORMAČNÍHO SYSTÉMU

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. LUKÁŠ PLACHÝ
AUTHOR

BRNO 2008



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA PODNIKATELSKÁ
ÚSTAV INFORMATIKY

FACULTY OF BUSINESS MANAGEMENT
DEPARTMENT OF INFORMATICS

METODIKA NÁVRHU ARCHITEKTURY SW INFORMAČNÍHO SYSTÉMU

METHODOLOGY FOR DESIGN OF INFORMATION SYSTEM ARCHITECTURE

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. Lukáš Plachý

VEDOUCÍ PRÁCE
SUPERVISOR

doc. Ing. Miloš Koch, CSc.

BRNO 2008

Zadání

Abstrakt

Cílem práce je navrhnout takovou metodiku tvorby architektury SW aplikací, která umožňuje snadné, bezchybné a systematické přenesení reálných obchodních procesů do modelu vhodného jako zadání pro implementaci programátory. Cílem tohoto návrhu je taková metodika, která by výše uvedené umožnila na základě jednoduchých a snadno uchopitelných principů a která by byla k dispozici buďto jako fundament pro použití v rámci metodik které jsou zaměřené na řízení takovýchto projektů, nebo jako alternativa a metodikám které jsou příliš drahé, komplexní a/nebo určené pro velmi velké vývojové týmy.

Klíčová slova

Metodika, kostka, agilní metodiky, modelování, architektura aplikací, vývoj software.

Abstract

The aim of this work is to propose such a methodology for designing of SW applications architecture that allows effortless, exact and systematical transformation of the real business process into a model suitable as an assignment for implementation by programmers. The aim of this proposal is such a methodology that would allow to reach the above mentioned goals on the basis of easy-to-understand and simple principles and that would be available either as a fundament for its usage within methodologies that are focused on the management of such projects, or as an alternative to methodologies that are much to expensive, complex and/or designed for very large development teams.

Keywords

Methodics, cube, agile methodics, modeling, application architecture, software development.

Citace této práce

PLACHÝ, Lukáš Bc.: *Metodika návrhu architektury SW informačního systému*. Brno, 2008, diplomová práce, FP VUT v Brně.

Metodika návrhu architektury SW informačního systému

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením doc. Ing. Miloše Kocha CSc.

Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Lukáš Plachý
25. 5. 2008

Poděkování

Rád bych na tomto místě poděkoval všem osobám, které mě podpořily při tvorbě této práce, zejména vedoucímu své práce za jeho vyčerpávající pomoc a spolupráci a také své rodině a přátelům za jejich pochopení a toleranci během období, které jsem této práci věnoval.

Obsah

Obsah 4
1 Úvod 6
1.1 Hypotéza 12
1.2 Cíle práce 12
1.3 Co NENÍ cílem této práce 13
2 Úkoly metodiky a postup tvorby 14
2.1 Co je to metodika? 14
2.2 Pohledy na modelování systémů 19
2.3 Agilní versus rigidní 21
2.4 Co musí mít programátor k dispozici 23
2.4.1 Vize a cíl – co se vlastně dělá? 24
2.4.2 Struktura systému – jak to bude vypadat? 25
2.4.3 Struktura dat – s čím se tam bude pracovat… 26
2.4.4 Chování systému (algoritmus s ohledem na role) - … a jak? 28
2.4.5 Vzhled (uživatelského) rozhranní systému 29
2.4.6 (Kdo bude danou akci provádět) 30
3 Současné metodiky vývoje a postupy modelování 31
3.1 Modelovací metodiky 31
3.1.1 Přehled typů modelů 31
3.1.2 Datové modelování 39
3.1.3 Objektové a funkční modelování 42
3.2 Metodiky řízení životního cyklu 49
4 Metodika “Kostka“ 57
4.1 Modely 59
4.1.1 Model procesů aplikace 59
4.1.2 Model architektury (aplikace) 62
4.1.3 Model architektury aplikačních dat 66
4.1.4 Model rozhraní 69
4.1.5 Model use-case 71
4.2 Vstupní model 72
4.2.1 Jaké informace musí zachycovat 72

4.2.2 Jakou metodu / metodiku použít? ... 74
4.3 Souvislosti modelů v kostce ... 78
4.4 Postup tvorby a proces návrhu ... 83
4.4.1 Obecné poznámky ... 86
4.5 Řízení návrhu architektury ... 86
4.5.1 Kdo má vykonat které kroky? Součinnost ostatních rolí ... 88
4.5.2 Jak mají být kroky zasazeny do celého projektu ... 90
5 Závěr ... 92
6 Citovaná literatura ... 94
7 Slovník pojmů a zkratek ... 96
8 Přílohy ... 98

Bc. Lukáš Plachý (54219)
plachyl@std.fbm.vutbr.cz
20. 05. 2008

1 Úvod

Ačkoliv dnes je již možné debatovat o faktu zda a jak roste odvětví informačních technologií, je jednoznačné a neoddiskutovatelné že se tyto technologie staly součástí našeho života a že tedy dochází k jejich neustálému využívání a aktualizaci tak, aby vyhovovaly nejnovějším trendům a novinkám či změnám v jiných odvětvích. Jakkoliv se mění informační technologie, musí se měnit i jejich softwarová výbava. Jednou ze softwarových výbav, které podléhají nejvíce změnám nejen z jiných oborů ale i přáním a potřebám jejich provozovatelů, patří bezesporu informační systémy.

Tato rozsáhlá softwarová díla se neustále mění, vyvíjejí, některá zanikají, vznikají nová. Jejich společným jmenovatelem ale zůstává nutnost vyhovět obchodním procesům, které mají podporovat nebo dokonce suplovat. Tím je otevřené (a autor této práce by si troufal tvrdit, že se navíc stále rozšiřuje) pole působnosti pro IT specialisty, jejichž úkolem je tyto změny provádět.

Jak ovšem dokazují autorovy zkušenosti, otevírá se tak bohužel pole nejen pro profesionály nebo alespoň odborně zdatné osoby které pomocí svého zaměření na detail a tvrdé vyčerpávající mravenčí práce vždy nakonec dosahují (akceptovatelných) výsledků ale také pro dva druhy osob, které stojí na různých pozicích a jejichž práce rozhodně žádný přínos nepředstavuje, spíše je kontraproduktivní.

Prvními z nich jsou samotní programátoři, nebo – abychom použili korektnější termín – kodéři. Tito lidé jsou v jistém slova smyslu mistry svého díla. Disponují poměrně obsáhlými, takřka encyklopedickými, znalostmi své platformy,[1] pro kterou

[1] Platforma – zde jako soubor služeb ať již v podobě sady knihoven, operačního systému nebo vizualizačních nástrojů, které využívá programátor v podobě příkazů ve svém software. Nejzákladnější platformou pro programování PC typu IBM je obvykle BIOS a jakýkoliv další HW který na velmi nízké úrovni poskytuje různé služby. Pomocí různých úrovní abstrakce lze vytvořit například platformu operačních systémů která rovněž poskytuje služby pro zpřístupnění tohoto HW ale na vyšší úrovni a obvykle sama přidává některé další (například ošetření přerušení, chráněný režim a dynamické stránkování paměti atp.). Mezi nejznámější platformy které pak stojí NAD operačními systémy jsou pak například SUNJava nebo Microsoft .NET Framework, což jsou v obou případech nástavbové platformy které slouží výhradně jako rozhraní pro programátory ještě nad operačním systémem.

vyvíjejí a ev. dalších souvisejících jevů ale bohužel chybí jim (ač to je u programátorů s podivem) systematičnost a důslednost. Tito lidé tedy dovedou – a v tom tkví jejich „mistrovství“ – s minimem námahy a vynaloženého času napsat kód který se jeví funkčním a zdá se, že i plní požadované funkce. Bohužel při bližším pohledu se ukazuje, že se jedná o některé z typických nectností, ať už se jedná o spaghetti kód[2] nebo nepoužívání znovupoužitelných částí a využívání metod kopírování zdrojového kódu, nekorektních ošetření chyb nebo neúplných větví[3]. Jak je to možné? Odpověď je nasnadě – tito lidé nepoužívají cokoliv co by se dalo nazvat modely nebo metodiky. Na vině jsou právě jejich znalosti – každý reálný problém, který je jim zadáván, a nebo který mají řešit, si okamžitě a „z hlavy“ převádějí do podoby zdrojového kódu – dalo by se říci že ve zdrojovém kódu „myslí“. To je bezpochyby – zejména z ekonomického hlediska – v zásadě docela dobrý přístup, který nejen šetří čas, ale pokud nedojde k nějaké nepředvídané situaci nebo změně a navíc celý projekt díky své menší velikosti stojí na jediném člověku tak kupodivu také funkční postup. Bohužel v případě rozsáhlejších děl se jedná o postup naprosto nepoužitelný a vedoucí do jisté zkázy. Tou je zejména zjišťování dalších „překvapivých“ požadavků uživatelů, jejich vzájemná nesouvislost či dokonce protichůdnost, objevování neméně „překvapivých“ chyb a z těchto problémů díky tomu jak je celý kód pojatý a napsán neexistuje snadná cesta – ba právě naopak. Řešením jednoho problému vznikají další a z toho plynoucí důsledky mají (zejména ekonomický) vliv na celý projekt – jeho prodlužováním a jeho prodražováním.

Výše uvedený postup se tedy rozhodně nedá označit jako „analýza“ nebo „návrh software“ ačkoliv se mnozí z programátorů, kteří takto myslí, se s oblibou analytiky a návrháři nazývají. Jakkoliv je jejich práce ukázkou určité řemeslné zručnosti (tedy tam

[2] Tj. neprovedení dělení kódu na funkce a moduly, ale „naplácání“ veškeré funkcionality do jediné metody (procedury) která vykonává veškeré kroky nutné pro danou funkcionalitu sama.

[3] Každé větvení programového kódu, zpravidla pomocí klíčového slova „if“ by mělo mít vždy napsanou část „else“, tedy „co má kód dělat v případě že podmínka větvení není splněna“. Pokud není účelné nebo možné takovouto větev psát, pak by si měl programátor minimálně uvědomit, co tato větev znamená a vyjádřit to pomocí komentáře v kódu. To platí i pro ty možnosti běhu aplikace, které nejsou vyjádřitelné úplnými podmínkami (například zvážit možné kombinace vstupních hodnot atp.)

kde byla práce vůbec provedena) rozhodně se nejedná o práci systematickou a důslednou. Právě díky svým znalostem a zkušenostem totiž kodér, který ji prováděl, poměrně dobře věděl, kde ji může odbýt a „nedotáhnout" do konce. Ať už se tak stalo z jeho lenosti nebo z důvodu tlaku na termíny, výsledek se tím nijak nemění. Ale kodér sám není na vině. Za jeho práci odpovídá jeho nadřízený a ten by měl mít v arsenálu svých schopností i takové postupy metodiky, které by právě takovéto případy identifikovaly, nebo jim přímo předcházely.

Bohužel tito lidé právě příliš často patří do druhé zmiňované skupiny, kterou je operativní management. Zde dnes příliš často nacházíme lajdácké, nesystematické, sebezviditelňující a spíše rétoricky působící jedince, jejichž jediná schopnost tkví v náhodných samoúčelných výkřicích, které - ač zpravidla zcela správné a zdánlivě s věcí související - nemají s daným problémem ve skutečnosti nic společného nebo nejsou použitelné v celkovém kontextu, protože se soustředí pouze na daný problém, bez ohledu na okolí tohoto problému. Tito jedinci se zpravidla nacházejí na řídících funkcích, kde bez schopnosti citu pro detail úplnosti konceptu (tj. nikoliv pro detailní problémy ale pro detailní a úplné uchopení spektra činností které metodicky řídí činnosti exekutivního charakteru) pouze řeší problémy v lepším případě dle svého „citu" nebo pouze ad-hoc[4] v případě horším.

Nelze tuto situaci v krátkosti odbýt tvrzením, že se jedná o špatné osoby nebo o osoby na špatném místě. Tento problém je jiného charakteru – jedná se o rozložení odborných znalostí. Tyto znalosti mohou být v principu dvojího druhu – první z nich je rozložení znalostí nutných pro vlastní provádění práce (tedy dobře strukturované problémy). Je logickým očekáváním, že úroveň hloubky znalostí provádění té či oné činností z té či oné oblasti s rostoucí úrovní řízení postupně klesá, stejně tak jako množství času věnované této znalosti a jejímu udržování či používání.

[4] a zde v plném významu výrazu *ad-hoc* – tedy *dle konkrétní věci*, nebo *dle konkrétního případu*, volně přeloženo *případ od případu*.

Bc. Lukáš Plachý (54219)
plachyl@std.fbm.vutbr.cz

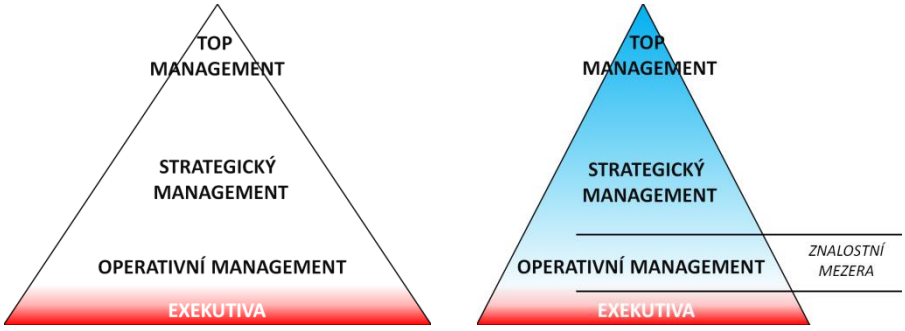
Obrázek 1: Množství odborných znalostí z daného oboru potřebných na jednotlivých úrovních firemní hierarchie – množství je vyjádřeno hustotou červené barvy

Naopak pokud se týká manažerských, tj. řídích odborností a schopností (tedy schopnosti potřebné pro rozhodování špatně strukturovaných problémů), úroveň i množství času používání těchto znalostí by měla růst, takže nakonec by mělo dojít k optimálnímu vytížení množství znalostí a jejich postupný přechod. (1)(3. díl, s. 30; 1. díl s. 15)

Obrázek 2: Množství znalostí a schopností managementu potřebných na jednotlivých úrovních firemní hierarchie – množství je vyjádřeno hustotou modré barvy

Bohužel v současné době dochází, jak bylo již výše uvedeno, k zvláštnímu trendu, který se týká zejména pozic operativního managementu. (2) Tito jedinci nemají potřebný rozsah ani hloubku znalostí pro pochopení celé složitosti principu mechanismu takovýchto systémů, zejména s ohledem na vzájemnou provázanost jejich součástí. Lze tedy říci, že drtivá většina osob netuší, že řeknou-li (myslí, nebo řeší či mění-li) A, musí na základě určitých vazeb říci (změnit) i B (a v případě softwaru mnohdy také C, D,

E…. až Z). Tím vzniká jistá mezera jedinců kteří disponují spíše manažerskými schopnosti, ale kterým chybí nutné odborné znalosti.



Obrázek 3: Reálný souhrn znalostí a jejich rozložení včetně vzniklé mezery

V případě programování se jedná o znalost jedinou – o schopnost programovat. Je tedy zcela normálním jevem, že přímým nadřízeným programátorů je neprogramátor (zde ve smyslu osoby, která o programování takřka nic neví). Tím ovšem dochází k nepříjemnému, ale pochopitelnému jevu, kterým je akumulace práce na programátorovi (toto tvrzení je zcela zjevné uvědomíme-li si výše uvedené rozložení – pokud jediným, kdo má odborné znalosti z oboru, je exekutivní pracovník, nemůže tuto práci také provádět nikdo jiný). Programátor – kodér se tak stává analytikem, aplikačním architektem, programátorem (obvykle všech vrstev aplikace s tím že dělba práce je na úrovni funkčních modulů), integrátorem a nezřídka i testerem, správcem a systémovou podporou. A v případě že takovýto programátor patří do první zmiňované skupiny, je již zaděláno na vážné problémy.

Jakkoliv by se akumulace práce v exekutivní úrovni mohla zdát přirozená a logická, není tomu tak. Podíváme-li se na úroveň operativního managementu, je to právě tato úroveň kde by mělo vznikat zadání, rozdělení a celkový koncept výsledné práce (3).

Tento problém obvykle řeší společnosti tak, že celé spektrum specializací vývojových pracovníků podřídí jediné osobě (zpravidla projektový manažer a podobné pozice) a tím odsouvají klíčová rozhodnutí mimo vlastní management k osobám, které k tomu nemusejí mít veškeré potřebné podklady, nebo komplexní obraz situace.

Další možností jak řešit tento problém je nasadit některou z metodik (viz kapitola 3.2). Bohužel to naráží mnohdy na problémy s pochopením a nasazením,

zejména z ekonomických a časových důvodů. Je to dáno relativně velkou složitostí těchto metodik, které se snaží být zbytečně komplexní. Jak jsme při tom uvedli již o několik řádků výše, základní problém tkví v pochopení systematického a důsledného rozboru aplikace – avšak takovým způsobem, aby bylo velmi jednoduše představitelné, proveditelné a uchopitelné.

A právě pro takovéto případy je určena tato práce. Jejím cílem je poskytnout managementu operativní úrovně jeden z nutných záchytných bodů pro vývoj informačních systémů. Jakkoliv by bylo bezpochyby záhodné a pro úspěšné odstranění uvedené znalostní mezery navíc nezbytné připravit celou projektovou metodiku, která by poskytovala taková vodítka pro práci manažera vedoucí vývojový tým, není to v rozsahu takovéto práce možné. Co více, jak vyplyne z následujících kapitol, takových metodik je k nalezení mnoho a lze z nich nejen čerpat, ale také je s různým stupněm rychlosti a flexibility uplatnit, a nebo dokonce i navzájem kombinovat. Zde se zaměříme pouze na jedinou část vývoje, která je ale z hlediska úspěchu jednou z nejklíčovějších – a tou je návrh architektury systému. Pozice navrhované metodiky (tedy přesněji části metodiky) je tedy tato:

Obrázek 4: Fáze životního cyklu projektu vývoje a kroky popisované touto metodikou

Je potřeba si ale již dopředu uvědomit, že zde popisovaná metodika rozhodně nemá poskytovat návod JAK navrhovat datové struktury, jak navrhovat vstupy a výstupy atd. Uváděná metodika říká „pouze“ CO musí být součástí těchto návrhů,

Bc. Lukáš Plachý (54219)
plachyl@std.fbm.vutbr.cz
20. 05. 2008

neboli JAKÉ INFORMACE jsou potřebné pro tvorbu architektury systému a JAK MAJÍ SOUVISET nebo-li v jakém pořadí je doporučené tyto zpracovávat a vytvořit. Cílem tedy je navrhnout metodiku, která bude představovat jakousi „minimální informační kostru“, tedy soubor informací nutných pro vývoj a doporučení pro jejich zachycení a zpracování.

Je pochopitelné, že pro vlastní provedení návrhu je nutné již mít detailní znalosti programovaných systémů, ale to již je doopravdy úkol exekutivních pracovníků. Jinými slovy, zde uváděná metodika specifikuje cíl, který je nutným milníkem při tvorbě informačních systémů a kterého je nutno dosáhnout aby bylo možné přistoupit k implementaci (zde ve smyslu kódování, testování a nasazení) systému v praxi.

Neméně důležitým autorovým cílem bylo sestavit tuto metodiku tak, aby byla snadno pochopitelná, uchopitelná a rychle aplikovatelná. Zmíněné informace, které metodika poskytuje, totiž poskytuje mnoho jiných metodik, jejichž nešvarem je ale právě příliš velká komplexnost, náročnost zavedení o náročnosti informace nastudovat a zpracovat ani nemluvě. Proto má navrhovaná metodika být jakýmsi „švýcarským nožem“ – lze ji „pohodlně nosit s sebou“, používat rychle a efektivně na mnoho rozdílných projektů, na svoje zavedení nevyžaduje téměř žádný čas.

1.1 Hypotéza

Je možné definovat určitou posloupnost kroků a jasně specifikovaných informací a jejich souvislostí, pomocí kterých je možné při známosti nějakého procesu tento proces rychle převést do podoby modelu softwaru. Tato posloupnost kroků je natolik snadná a jasně definovaná, že je vždy velmi rychle uplatnitelná a to za udržení zejména integrity návrhu.

1.2 Cíle práce

- Specifikovat požadované výstupní informace nutné jako zadání pro vlastní implementaci (tj. kdy a pro koho jsou určeny, kým by měly být definovány)
- Najít tyto specifikované výstupní informace v modelech (tj. kdy a kdo by měl najít jaké informace)

- Navzájem propojit tyto modely tak, aby zachovaly systematické souvislosti mezi jednotlivými prvky navrhovaného systému, včetně propojení na model reality (tj. kdo a jak by měl udržet tuto konzistenci modelů a na základě jakých informací)
- Navrhnout model reality nutný pro poskytnutí všech relevantních informací (tj. kdo a jakým způsobem by měl zachytit jaké informace).
- Specifikovat veškeré vstupní informace extrahované z reality, které jsou třeba pro tvorbu takového jejího modelu (tj. kdo a jakým způsobem by měl zachytit jaké informace)
- Zřetězit výše uvedené kroky do přesně dané posloupnosti a dát do souvislosti s rolemi pracovníků.

Poznámka: Výše uvedené cíle jsou záměrně specifikovány „odzadu“ tj. od výsledku (výstupu) ke vstupu. Vysvětlení k tomu podává blíže kapitola 4 Metodika “Kostka“ .

1.3 Co NENÍ cílem této práce[5]

- Specifikovat kompletní metodiku, která tvoří vodítka pro celý životní cyklus vývoje software informačních systémů (viz. Obrázek 4: Fáze životního cyklu projektu vývoje a kroky popisované touto metodikou)
- Specifikovat metodiku pro vývoj specializovaných aplikací jako jsou operační systémy, embedded systémy nebo jiné aplikace nepoužívající klasický model uživatel/klient/server, tím nejsou dotčeny jednovrstvé (aplikace nepoužívající databázové služby nebo databázové aplikace samotné) a vícevrstvé aplikace.
- Specifikovat kroky přechodu mezi realitou a modelem reality (jak je výše uvedeno specifikovány jsou pouze informace které musejí být v těchto krocích zpracovány)
- Specifikovat kroky přechodu mezi modelem software a softwarem (jak je výše uvedeno specifikovány jsou pouze informace, které musejí být v těchto krocích zpracovány)

[5] Jak by snad mohlo z úvodu či cílů mylně vyplývat.

2 Úkoly metodiky a postup tvorby

2.1 Co je to metodika?

Podívejme se nejprve na definici jednotlivých pojmů. K tomu si nejprve musíme uvědomit, kde se nacházíme. Pokud použijeme pojem „metodika IS“ vztahuje se podle (4) na všechny prvky IS, tedy:

- pracovníky
- organizační procedury a postupy
- data
- SW a HW
- organizační vlivy IS
- ekonomické otázky spojené s vývojem a provozem IS
- produkty a potřebné dokumenty v jednotlivých fázích životního cyklu IS
- použitelné metody, techniky a nástroje v jednotlivých fázích životního cyklu IS
- způsob řízení v jednotlivých fázích životního cyklu IS.

Co to tedy znamená v praxi? Pokud použijeme seznam všech částí (kroků nebo stavů) životního cyklu projektu[6], pak lze ve spojení s výše uvedenými požadavky definovat takovouto tabulku:

[6] a zde je důležité právě zmíněné slovo **projektu**. Musíme si uvědomit že se zde zabýváme pouze projektem tvorby a nasazení IS nikoliv jeho celého životního cyklu.

		Pracovníci	Org. postupy	Data	SW a HW	Org. vlivy	Ekonomické otázky	Produkty a potřebné dokumenty	Metody, techniky, nástroje	Způsob řízení
Analýza(+studie proveditelnosti)	**Cíle a strategie**	Všechny tyto položky musí kompletní metodika specifikovat								
	Testování klíč. techn.									
	Mapování požadavků									
	Mapování procesů									
Návrh	**Návrh procesů**									
	Návrh dat. struktur									
	Návrh I/O									
	Návrh. apl. arch.									
Implementace	**Návrh apl. procesů**									
	Coding									
Testování	**Návrh test-cases**									
	Provedení testů									
Nasazení	**Pilot**									
	Nasazení									
Vyhodnocení	**Vyhodnocení**									

Tabulka 1: Fáze a oblasti které musí popisovat metodika

Pokud se ovšem zaměříme na název této práce, což je „Metodika návrhu architektury SW aplikace“ vyplývá z ní jasně, že se nacházíme pouze ve výseku implementace SW produktu (jak bylo ostatně uváděno na začátku) a tedy zde chápeme IS jako „Aplikaci pro podporu IS“. Ta má sice v zásadě stejné fáze jako celý projekt IS, avšak některými otázkami se projekt softwarové aplikace nezabývá. Které to jsou, to znázorňuje uvedená tabulka:

		Pracovníci	Org. postupy	Data	SW a HW	Org. vlivy	Ekon. otázky	Produkty a potřebné dokum.	Metody, techniky, nástroje	Způsob řízení
Analýza(+studie proveditelnosti)	**Cíle**									
	Testování klíč. techn.									
	Mapování požadavků									
	Mapování procesů									
Návrh	**Návrh procesů**									
	Návrh dat. struktur									
	Návrh I/O									
	Návrh. apl. arch.									
Implementace	**Návrh apl. procesů**									
	Coding									
Testování	**Návrh test-cases**									
	Provedení testů									
Nasazení	**Pilot**									
	Nasazení									
Vyhodnocení	**Vyhodnocení**									

Význam barev:

Není popisovaná fáze životního cyklu

Jedná se o popis IS jako celku, netýká se popisu SW aplikace

Tato metodika popisuje nebo předepisuje postupy

Tato metodika se jimi zabývá se pouze částečně nebo pouze doporučuje postupy

Tabulka 2: Fáze a oblasti které popisuje metodika představená v tomto dokumentu

Jak bylo zmíněno dle (4), **metodiky** jsou naplňovány jednotlivými

- metodami,

s nimi souvisejícími

- technikami

a k tomu potřebnými

- nástroji

Bc. Lukáš Plachý (54219)
plachyl@std.fbm.vutbr.cz
20. 05. 2008

Metodika tvorby IS je tedy doporučený souhrn etap, přístupů, zásad, postupů, pravidel, dokumentů, řízení, metod, technik a nástrojů, pokrývající celý životní cyklus informačních systémů.

Určuje **kdo, kdy, co a proč** má dělat během vývoje a provozu IS.

Metodika by se měla vztahovat na veškeré prvky IS (pracovníky, organizační procedury, data, SW a HW a další), ekonomické otázky spojené s vývojem a provozem IS a doporučené dokumenty.

Metoda určuje **co** je třeba dělat v určité fázi nebo činnosti vývoje či provozu IS.

Metoda je vždy spojena s určitým přístupem, jako je funkční, datový, nebo například objektový přístup. S přihlédnutím k této charakteristice řeší každá metoda postup činností v určité části (jedné nebo n_kolika fázích) procesu vývoje systému, nebo pouze z některého úhlu pohledu na systém (data, funkce, SW, HW, atd.).

Příklady metod jsou: informační analýza, funkční analýza, analýza konceptuálních tříd a procesů, aj.

Technika určuje, **jak** (jakým postupem) se dobrat požadovaného výsledku. Zpravidla určuje přesný postup jednotlivých činností, způsob použití nástrojů, varianty rozhodnutí v určitých situacích a co z nich vyplývá, vymezuje obor své působnosti atd. Na rozdíl od metody je přesnější v závěrech a omezenější v okruhu použití.

Příklady: transakční analýza, normalizace datového modelu, analýza událostí aj.

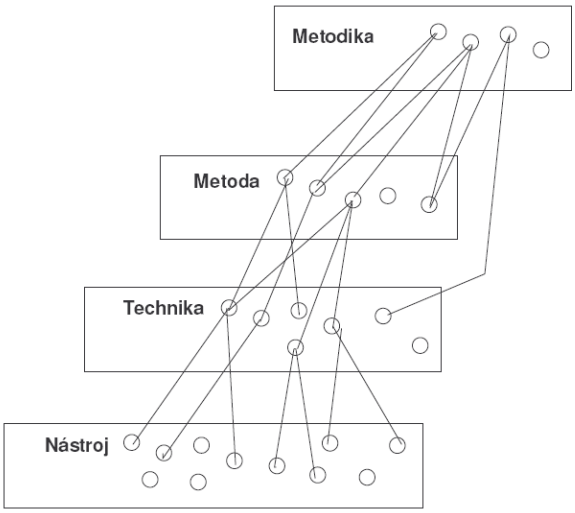
Nástroj je prostředkem k uskutečnění určité činnosti v procesu vývoje a provozu IS a prostředkem k vyjádření výsledku této činnosti. Nástroj je často svázán s konkrétní technikou. Nástroje vždy formalizují vyjádření, proto je možné a žádoucí, aby byly v maximální míře automatizovány.

Příklad: diagramy tříd, toku dat, stavový diagram, diagram procesů aj.

Není možné jednoduše prohlásit, že jednotlivé metody jednoznačně patří určitým metodikám, nebo že každá technika je tu výhradně ku podpoře nějaké své metody apod.

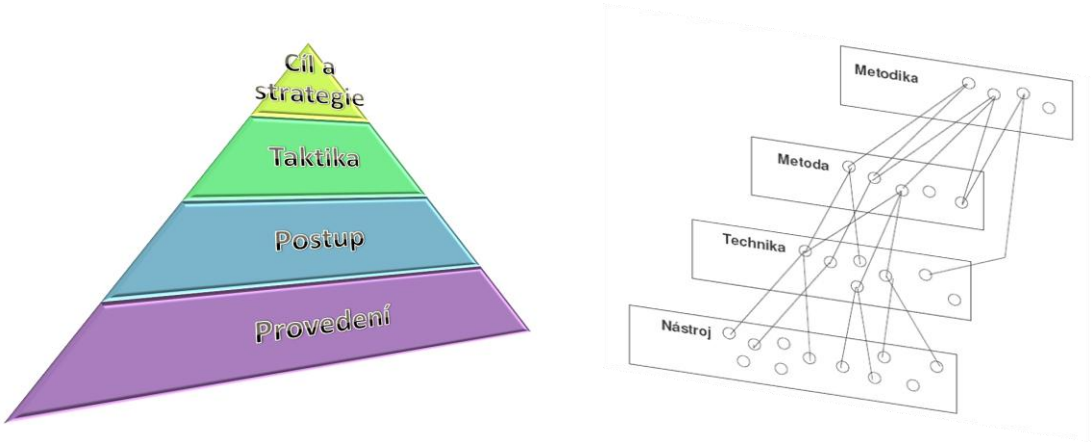
Vztahy mezi metodami, technikami a nástroji, i jejich přináležitosti k metodikám, mohou být rozmanité, jak je ilustrováno níže (Obrázek 5). Metodiky se odkazují na příslušné metody v relevantních místech životního cyklu IS, přičemž

některé metody jsou více specifické, jiné mají univerzálnější charakter (například na metody datového modelování se odkazují prakticky všechny metodiky). Obdobně i technika může patřit k určité metodě, nebo je společnou pro řadu různých metod (například techniky normalizace datových struktur, či analýza událostí). Technika buď vyžaduje specifický nástroj, který je s ní pevně svázán a bez ní nemá smysl, nebo používá obecněji použitelný nástroj (jako například zmiňovaný Diagram procesů, nebo ER diagram). Některé nástroje jsou natolik universální, že nemá smysl je vázat k nějakým technikám - jsou prostě nástroji, používanými různými metodami – takovým způsoben je například koncipován celý jazyk UML.



Obrázek 5: Souvislosti nástrojů, technik, metod a metodik (převzato z (4))

Nad tímto dělením a zejména definicí metodiky je nutno se pozastavit. Jakkoliv by bylo možné s ním souhlasit, podívejme se na diagram na Obrázek 5 pohledem manažerským a srovnejme jej s úrovněmi řízení a vztahem k cílům.



Obrázek 6: Porovnání hierarchie metodiky a hierarchie podnikových cílů

Co je tedy metodika z pohledu teorie managementu? Není ničím jiným než **strategickým cílem** – tedy specifikovat metodika jako nějaký soubor (či množinu nebo souhrn) prvků, které vedou k cíli je sice fakticky bezpochyby správné, z hlediska struktury takovéhoto uvažování ale ne zcela přesné. Metodika vyspecifikovaná napříč celou pyramidou (tj. všechny vertikální stupně specifikované v jejich plné šíři) je bezesporu úplnou metodikou. Ale jak vyplývá z výše uvedeného, jako metodiku lze považovat i takový soubor informací, který specifikuje cíle, kterých má být dosaženo, aniž by procházel až na „dno pyramidy“ nebo aby pokrýval veškeré její stupně v plné šíři. A to je právě (i z důvodu předpokládaného rozsahu) metodika „Kostka“.

2.2 Pohledy na modelování systémů

Při modelování systémů se v praxi používá vždy modelů, ať již se jedná o komplexní velké modely strojních zařízení, psychologické modely chování, modely vztahů. To platí i pro informační systémy stejně jako jejich aplikace. Definici modelu je možné najít v této práci v kapitole *3.1.1 Přehled typů modelů*. Podle (4) a podle tzv. P3A (principu tří architektur) lze tyto modely rozdělit z pohledu metodického na 3 základní typy, či spíše kroky jejich konstrukce:

- Model reality
- Technologické model
- Implementační model

Bc. Lukáš Plachý (54219)
plachyl@std.fbm.vutbr.cz
20. 05. 2008

Dle (4) je **model reality** prvním nutným krokem k zachycení aktuálního stavu. Jeho popis nemá nic společného s modelem popisovaného IT systému z pohledu SW aplikace. Je zde vytvořen zcela obecný, čistý obsahový model systému, nezatížený ani technologickou koncepcí řešení, ani jeho implementačními specifiky. Je zde abstrahováno od technologických a implementačních specifik řešení. Obsahový návrh určuje **co** je obsahem systému.

Dle (4) je v případě **technologického modelu** vytvořen model systému, zohledňující technologickou koncepci řešení, tj. ve strukturovaném pojetí koncepci organizace dat (technologie souborová, stromová, síťová, či relační databázová atd.) a technologickou koncepci jejich zpracování (jazyk 3. i 4. generace, technologické prostředky architektury klient - server atd.). Technologický model stále nesmí být zatížen implementačními specifiky řešení. Je zde tedy abstrahováno od implementačních specifik řešení, obsahové náležitosti jsou dány obsahovým modelem a zde se neřeší. Technologický návrh určuje **jak** bude obsah systému v dané technologii realizován (zohledňuje „logiku“ použití zvolené technologie).

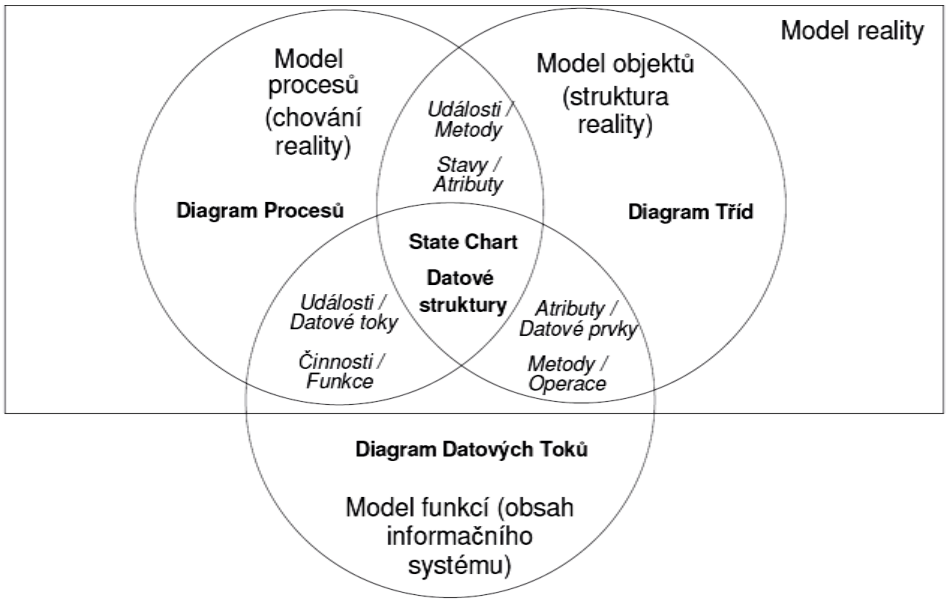
Dle (4) je teprve v **implementačním modelu** zachyceno řešení zohledňující implementační specifika použitého vývojového prostředí (konkrétního databázového systému, programovacího jazyka a dalších prostředků, jako například vývojového prostředí GUI atd.). Není zde abstrahováno od žádných specifik řešení, obsahové náležitosti jsou dány obsahovým modelem, technologie je dána technologickým řešením, implementační návrh se tedy týká pouze implementačně specifických rysů systému. Implementační návrh určuje **čím** je technologické řešení realizováno.

Bude bezpochyby účelné a vhodné během řešení cíle této práce výše uvedené dělení zachovat. Pokud to ale bude akceptovatelné a pro vlastní řešení možné, nebudou tyto jednotlivé typy modelů striktně odděleny. Logický význam ale zůstane zachován.

Modely mají ovšem ještě jedno specifikum, u kterého je vhodné se pozastavit. Jak uvádí (4) lze mezi nimi najít určité souvislosti. A protože právě na souvislostech a jejich zohlednění a metodické zobrazení tvoří nejdůležitější základ metodiky „Kostka“, je vhodné uvést některé fakty týkající se těchto souvislostí.

V předchozí části kapitoly byly uvedeny některé základní pohledy na modelovaný systém. Je jasné, že pro konečnou implementaci je potřebné mít

k dispozici implementační model systému. Ten ovšem navazuje na vlastní realitu a to tak jak je uvedeno na obrázku Obrázek 7.



Obrázek 7: Různé pohledy na systém a souvislost jejich modelů (převzato z (4))

A právě výše uvedené abstraktní součásti metodika kostka konkretizuje a předepisuje pro praxi, jak je uvedeno v kapitole *4 Metodika "Kostka"*.

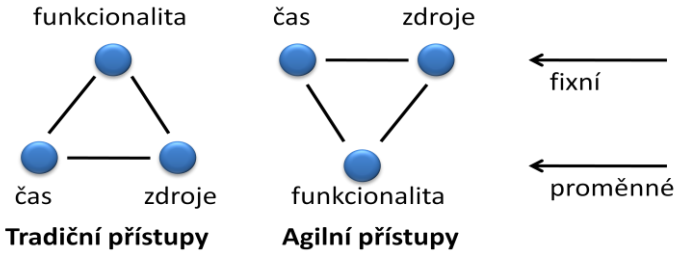
2.3 Agilní versus rigidní

Pokud hovoříme o metodikách vývoje softwaru, dostáváme se vždy na pole charakteru pole minového, zvláště tehdy pokud hovoříme o těchto přístupech v kontextu dnešního managementu. Ten se – jak již bylo řešeno v úvodu – orientuje spíše na rétoriku a upřednostňování komunikační stránky řešení (slangově známé např. jako „okecávání" atp.) a v takto pojatém světě nemají zpravidla přísná pravidla, normy a zejména systematičnost metodické práce své místo. Jak je tomu tedy doopravdy?

Pokud se budeme zabývat metodikami vývoje, nalezneme mezi nimi skupiny metodik označované jako „agilní metodiky". Tyto metodiky lze zkráceně popsat (například dle (5)) jako metodiky které – na rozdíl od tradičních metodik – umožňují (přímo očekávají) změnu vlastního zadání, tedy toho **co** má být dodáno[7]. Zatímco

[7] Jedná se o tzv. „trojimperativ projektu". Vychází z toho, že každý projekt má definovány 3 základní vektory omezení – prostředky (jak), čas (kdy) a požadovanou funkcionalitu (co).

původní metodiky předpokládají, že je potřeba dosáhnout požadované funkcionality s tím, že se může měnit termín nebo prostředky (obvykle tedy rozpočet), tedy „specifikované dodat za každou cenu“ agilní metodiky naproti tomu podřizují dodávanou funkcionalitu termínům a rozpočtu (tedy „dodat za všech okolností, otázka zní co“). Jak z uvedeného vyplývá, oba přístupy mají svoje klady i zápory. Tuto situaci nejlépe ilustruje následující obrázek (dle (5)).



Obrázek 8: Rozdíly agilních a tradičních metodik (dle (5))

Základním pilířem tohoto přístupu je manifest z roku 2001 (The Agile manifesto). Pokud se zaměříme na jeho českou verzi, říká toto (6):

Objevujeme lepší způsoby vývoje softwaru tím, že jej vytváříme a pomáháme v tom ostatním. Během této práce jsme se naučili upřednostňovat:

- ***lidi a jejich vztahy** před procesy a nástroji,*
- ***funkční software** před vyčerpávající dokumentací,*
- ***spolupráci se zákazníkem** před jednáním o smlouvě,*
- ***zohlednění změny** před dodržením plánu.*

To znamená, že ačkoliv uznáváme důležitost položek na pravé straně, přikládáme větší význam položkám na levé straně.

Velmi zajímavé shrnutí lze najít na (7) kde se říká: *„Kuchařka je dobrá věc, ale není lepší než jídlo.“*

Jak ovšem můžeme vidět, je mnohdy tento přístup zneužíván. Dobrou ilustraci k tomu poskytuje Michal Smrž (v (8)) v článku „Proč mám alergii na agile development“. v jedné z jeho části se říká:

„(...)Cílem je poukázat na velmi časté nepochopení agilního vývoje, kdy se ve skutečnosti praktikuje tzv. “Cowboy coding“. Tento jev se, dle mého názoru, projevuje ve zvýšené míře hlavně u méně zkušených, zato ale velmi kreativních developerů, kteří

si takříkajíc z agilního přístupu vybírají jen to, co se jim hodí. Druhou skupinou, která je neméně častá, jsou zkušení programátoři, ovšem od přírody "bastlíři". Každý z nich má spoustu, někdy velmi humorných výmluv(...)"

A autor dále uvádí příklady, jak jsou vize agilního vývoje zneužívány jako výmluvy pro různé obcházení a nedodržování některých základních metod práce vývojáře SW jako jsou například: neukládání zdrojových kódu do CVS, SVN nebo jiného systému pro správu verzí; nepoužívání patterns, protože jsou zbytečně obecné, je to prostě spousta kódu navíc, který nikdo nepotřebuje; tvrzení „tohle dělat nebudeme, protože to zákazník nechce" (ačkoliv to z logiky aplikace bude chtít sám za měsíc provozu, pouze si to doposud sám neuvědomil); dále tvrzení například „analýzu nemáme, zákazník nebo "lidi z businessu" ti stejně neřeknou, co vlastně potřebují", důvěra ve zkušenosti vývojářů před korektní architekturou a návrhem atp.

Je tedy v dnešním světě místo pro takové metodiky, které vyžadují přesnou specifikaci a té se pokud je to možné a účelné drží? Jak již vyplynulo z úvodu této práce, je jich potřeba. Pokud zohledníme principy agilního vývoje, neznamená to opuštění metodiky, pouze její přípravu na možnost rychlé akceptace změny a její uvedení do praxe. Jak zjistíme v kapitole *4 Metodika "Kostka"* nejedná se o klíčový prvek, protože metodika Kostka se snaží odstranit některé ze zdrojů nutnosti změny, a ačkoliv samozřejmě neřeší všechny, snižuje počet nutných změn z důvodů chyb, přehlédnutí atp.

2.4 Co musí mít programátor k dispozici

Protože se u metodiky popisované v této práci v konečném důsledku zajímáme (jak ostatně vyplývá z úvodu k této práci) jak dospět k realizaci software na základě určitých vstupů, stává se klíčovou znalostí znalost cíle. Ten bude definován právě v této kapitole.

Její obsah je vytvořen souhrnem podstat obsahu modelovacích nástrojů i technik a zejména osobních zkušeností autora práce z praktického vývoje (9)(10).

Co tedy potřebuje znát programátor-kodér, aby byl schopen začít již (v principu bez dalšího uvažování – tedy bez dalších analýz a návrhů architektury včetně například vývojových diagramů) přímo psát kód?

Nejprve si je nutné uvědomit, v jaké situaci se dnes takovýto programátor nachází. Ačkoliv v úvodu práce bylo uvedeno, že klíčovým problémem bývá – z pohledu chyb managementu – nedostatečně pečlivá (tj. systematická, důsledná a precizní) specifikace práce programátora a tím se veškerá rozhodnutí a analýzy včetně návrhu zpravidla odsouvají až na úroveň kodérů (jak je ostatně ukazováno i v kapitole *2.3 Agilní versus rigidní*) je zřejmé že není možné ani účelné specifikovat jejich práci až do posledního detailu. Dokumentace (ani dokumentace návrhu), která popisuje softwarovou aplikaci na úrovni vlastního zdrojového kódu, není rozhodně účelným řešením, uvědomíme-li si že se jedná již o vlastní aplikaci, a že tedy analytik / aplikační architekt si mohl celou aplikaci místo vytváření dokumentace napsat sám. Nyní se tedy pokusíme definovat takový rozsah „úrovně detailu“ zadání, které je ještě realizovatelné, udržitelné a upravovatelné na jedné straně a zároveň dostatečně detailní a jemné na straně druhé.

K následujícím kapitolám je nutné poznamenat, že jsou uvedeny rovněž i ve stejném pořadí jak o jejich obsahu obvykle bývá uvažováno a jak by s nimi také měli být programátoři seznamováni.

2.4.1 Vize a cíl – co se vlastně dělá?

Je známou a běžně užívanou manažerskou praktikou že zaměstnanci musejí mít pro svou konkrétní práci svůj konkrétní **cíl**. Tento cíl by měl být uzpůsoben jejich konkrétní pozici (1) a jejich konkrétní pozici v rámci obchodního procesu organizace. Stejně tak i kodér informovaný o své práci musí znát její obecný cíl. Kromě toho že tento prvek má významnou motivační roli, pomáhá rovněž v pochopení účelnosti ostatních kroků a prvků, ze kterých se vlastní pracovní úkol skládá.

Ačkoliv je tento krok naprosto nezbytný a esenciální, je mnohdy managementem špatně chápán a vykládán a to zpravidla ve dvou extrémních případech:

- Manažer pro detailní a vizionářské popisy cílů (obvykle pak z valné části nesouvisejících s úkolem) spotřebuje více času a prostoru než pro popis vlastních pracovních úkolů řešení.
- Manažer se obává, že pochopí-li zaměstnanec jak jeho práce zapadá do celku mohl by tuto znalost použít proti manažerovi (například požadavkem na vyšší plat, odhalením manažerových chyb atp.) a proto

naopak cíle zamlžuje, dynamicky mění, odbývá tvrzeními jako „O to se nestarej..“ atp.

Ani jeden z těchto přístupů samozřejmě není korektní a rozhodně neplní účel. Samotné definování cíle je otázkou několika mála jednoduchých stručných vět a v takovéto formě se i autorovi osvědčilo. V případě potřeby je možné – pokud je to u daného zaměstnance vhodné a účelné – doplnit například informací týkající se rozpočtu, který je na jeho práci k dispozici nebo odůvodnění některých nestandardních kroků (například ukázat podepsanou žádost zákazníka o nějakou změnu (na základě výjimky) která nemusí nutně splňovat nějaké normy atp.).

Jakkoliv je tento bod signifikantní, nejedná se o žádnou vědu – obecný cíl může být rovněž specifikován i ukázáním nadřazené funkce na vyšší úrovni modelu atp. a může být opravdu proveden pomocí poznámky takřka neformální.

2.4.2 Struktura systému – jak to bude vypadat?

Tento krok obvykle zkušenější a schopnější manažeři popisovaní v úvodu práce ještě zvládají – majíce dobrou představu o architekturách systémů jsou schopni definovat obecné informace jako „bude tam klient a databáze…“ atp.

Bohužel ačkoliv je tento bod opravdu nejvhodnější jako počátek zadání pro kodéra, kterému pomáhá zejména později při pochopení chování systému, obvykle trpí dvěma základními neduhy:

- Není dostatečně detailní – končí na úrovni celých aplikačních systémů, v lepším případě modulů.
- Stane se posledním a jediným popisem systému.

Navíc paradoxně může trpět neduhem třetím:

- Je příliš detailní – řeší něco co už by řešit neměl, mnohdy ne zcela korektně a znemožňuje tak řešení problémů.

Tento popis je při tom pouze prvním nástinem řešení. Je pravda, že uvažování v jeho dimenzích je pro většinu zúčastněných velmi zábavné a příjemné a připomíná „panovnické“ navrhování pomocí gest rukou ve smyslu „město veliké, jehož sláva hvězd se bude dotýkat“, ale - použijeme-li toto srovnání - určení urbanistického plánu ulic, pozice hradu a katedrály či vedení kanalizace atp. již ušlo nejen kněžně Libuši, ale bohužel uchází i současnému managementu. Co je potřeba v tomto okamžiku uvést

navíc a do jakého detailu se je potřeba spustit ilustruje více kapitola *4 Metodika "Kostka"*.

Výše uvedené hovoří pouze o detailu architektury systému jako takového. Ovšem pro vlastní programování je naprosto nedostačující. Následující dvě kapitoly popisují nejklíčovější součásti, které musejí být provedeny extrémně důsledně a systematicky a to s náležitou (a konstantní) úrovní detailu.

2.4.3 Struktura dat – s čím se tam bude pracovat…

Je naprosto neomluvitelné když náhle hlavní analytik při revizi práce programátora (a to se stává pouze analytikům a architektů, neboť tester (je-li nějaký) vychází zpravidla z jejich zadání) přijde k naprogramovanému modulu nebo jiné součásti systému a prohlásí „ale tady musí být ještě možnost zadat atribut X". Ještě horší situace nastává, když samotný programátor přijde za architektem či analytikem práce s otázkou typicky začínající „když tady má být zadána informace že X nebo Y jak tedy software pozná, zda má nabídnout X nebo Y?"

Jakkoliv se samozřejmě může stát, že k tomu došlo na základě působení „lidského faktoru", obvyklejším důvodem bývá spíše nedůsledná a chaotická práce. Zmínění se obvykle brání proti těmto situacím způsoby „na to jste měl přijít sám, to vás to nenapadlo" nebo „to je přece jasné, to Vám nemusím vysvětlovat" – obecně tedy právě v úvodu práce uváděným přenášením zodpovědnosti, které zde navíc naprosto není na místě.

Je odpovědností analytika či architekta specifikovat **data a jejich strukturu** související s kontextem plánované práce a to způsobem systematickým, důsledným a především **vyčerpávajícím.** (Právní terminologie zná rovněž termín „taxativní", tedy „právě a pouze položky v seznamu, žádné nesmějí chybět a žádné jiné navíc".)

Jednou z často opomínaných informací která z těchto informací ovšem často vypadává, je informace o **souvislostech** dat mezi sebou – tedy v jakém vztahu se ta která entita nachází vůči entitě jiné. A již je jedno jestli se jedná o strukturu dat v paměti realizované jako struktury s ukazateli či jako jiné druhy struktur (podle úrovně na které se vývoj nachází - například zásobníky) nebo o relační databázové schéma. Ve všech případech musejí být zcela a vyčerpávajícím způsobem popsány:

- Obecné entity (logická seskupení dat – například „vozidla autoservisu a jejich zákazníci“)
- Detailní struktura až na „atomické“ entity (například „vozidlo“, „motor“, „zákazník“)
- Pro každou atomickou entitu **všechny**[8] její atributy, pokud možno včetně technologických (například pro entitu „vozidlo“ by to bylo „ID vozidla“) včetně datových typů (dle úrovně modelů obecné nebo už závislé na konkrétní technologii) a jejich omezení (množiny povolených hodnot).
- Je naprosto nezbytné si uvědomit, že výše uvedené platí nejen pro data persistentního charakteru ale také pro **data „běhu aplikace“**[9] (tedy struktury užívané pouze pro předání dat například v paměti nebo pomocí sítě, včetně typů návratových hodnot atp.).

Tyto zmíněná data nemusejí být jen data charakteru databázového (ve smyslu služeb relačních databázových systémů spravovaných zpravidla pomocí jazyků SQL a DDL) ale jakýchkoliv dat obecně, například různých formátů souborů (ať již textových (znakových) nebo binárních), formátu a významu dat nacházejících se na datových linkách, paměťových rourách atp.

Pouze nad takto specifikovaným datovým modelem bude kodér nejen schopen vytvořit smysluplnou aplikaci, ale tento model může pomoci například i při odhadu náročnosti aplikace – pokud jsou známy řádová množství zpracovávaných datových vět (záznamů) je možné již v tomto raném stádiu odhadnout technologické nároky na hardware, jeho výkonnost, přenosové kapacity atp.

---

[8] A toto je právě jeden z naprosto zanedbávaných kroků – neexistuje snad manažer či aplikační architekt či analytik na obdobné pozici (alespoň autor práce se s ním ještě nesetkal) který by se věnoval tomuto kroku. Ale je to právě při specifikaci jednotlivých detailních datových položek, kdy se obvykle objevují ty nejneočekávanější problémy, které mají nezřídka dopad napříč celým systémem, avšak pokud se na ně přijde až při implementaci, mají také dopad na celý projekt, jeho harmonogram i rozpočet.

[9] Druhý - takřka neznámý – krok. Při tom tento krok je naprosto esenciální zejména tam, kde se stýká práce většího množství týmů, protože ty musejí naprosto přesně vědět jaká data, odkud a jakým způsobem dostanou a nebo naopak jaká data (přesně – tj. jaká struktura a význam, názvy datových položek, jakým protokolem aplikačním i transportním atp.) budou vracet.

2.4.4 Chování systému (algoritmus s ohledem na role) - … a jak?

Posledním krokem zadání programátorovi jsou informace o tom „jak by to mělo fungovat“. Tento popis by se ale neměl (jak bývá nevhodným zvykem) omezovat pouze na obecný popis funkcí („zaúčtovat fakturu“, „smazat účetní informace z toho roku“) nebo na funkce v souvislosti s uživatelským rozhraním („když se sem klikne tak…“). Tento popis by měl vycházet ze zvolené úrovně detailu architektury systému v kapitole *2.4.2 Struktura systému – jak to bude vypadat?*. Pokud jsme tedy nadefinovali třídy pro práci s nějakými daty, pak musíme těmto třídám specifikovat jejich funkce, a co mají vykonávat a zajistit především jejich vzájemnou korektní souvislost v kontextu (zadat informaci že třída A má ze třídy B získat nějaká data když v prototypu třídy B žádnou funkci, která by taková data vracela, nenajdeme je jednou z dalších neospravedlnitelných chyb).

Další z typických chyb je nedefinování konkrétních metod – pokud třída B poskytuje dvě metody s podobnými výsledky, je nutné explicitně uvést, kdy má být která použita.

Pomocí jakých modelů je vhodné toto provést a jaké informace musejí obsahovat je rovněž popisováno v další kapitole. Je ale podstatné aby programátor měl – a zde se uplatňuje procesní přístup vstup/výstup – následující informace:

- Na základě jakého vstupu (odkaz na data),
- která z částí aplikace (odkaz na architekturu)
- má dávat jaký výstup (odkaz na data)
- ev. jaký je algoritmus dosažení výstupu ze vstupu

Všimněme si, že vlastní algoritmus stojí jakoby „v pozadí“ a důraze je kladen na vstupně/výstupní interoperabilitu.

2.4.5 Vzhled (uživatelského) rozhranní systému

Protože obvykle žádný programový prvek nestojí zcela sám o sobě[10], má každý nějaké vstupy a výstupy. Tím se neníní data, která ukládá a předává sám v rámci sebe sama, ale jakákoliv rozhraní, kterými komunikuje s jinými prvky systému. Zde je nutné rozlišit mezi dvěma základními typy rozhraní – prvkem systému totiž může být nejen jiný program, ale také člověk-uživatel:

- Rozhraní programové
- Rozhraní uživatelské

V případě programového rozhraní je naprosto nezbytné, aby byla tato součást specifikována v rámci analýzy a návrhu architektury. Tento důležitý fakt jsme ale zmiňovali již už v kapitole 2.4.3. Jak souvisí datový model s modelem rozhraní? Zatímco datový model popisuje vlastní logiku těchto dat (tedy CO ve vstupně/ výstupních rozhraních budě) při popisu vlastních rozhraní se kromě odkazu na datový model popisuje i přesný způsob transportu, protokol kterým bude provedeno atp.

Pokud se jedná o uživatelské rozhraní, to není zpravidla úkolem programátora-kodéra. Jeho úkolem bývá spíše implementace vlastní programové logiky, zatímco návrhem uživatelského rozhraní se zabývají specialisté na design, či celé týmy zpracovávající rozhraní na základě výzkumů lidského chování atp. Ačkoliv tato informace jak uvádí kapitola *4 Metodika "Kostka"* musí být nedílnou součástí návrhu (a v konečném důsledku musí skončit u programátora – má-li jím zpracovávaný modul aplikace uživatelské rozhraní) není vyloženě nezbytně nutnou informací sloužící jako zadání pro programátora. V rámci návrhu aplikace ale své místo rozhodně má. Zde je nutno podotknout, že do této specifikace spadají kromě vlastních uživatelských rozhraní určených pro zpravidla grafická uživatelská rozhraní platformy, pro kterou je systém vyvíjen také:

- tiskové sestavy,

[10] A v IT praxi navíc existují pouze dva prvky, které mají pouze výstup a nebo pouze vstup – prvním z nich je generátor náhodných čísel (i když z technologického hlediska vzato má vstupy i on) a druhým je speciální zařízení používané v UN*Xových systémech, zvané /dev/null což je jakási „černá díra" do které lze zapisovat nekonečně mnoho dat, avšak při čtení okamžitě vrací konec souboru.

- **konfigurační vstupy** (soubory s konfigurací atp.),
- různé indikační systémy, které nejsou součástí standardního rozhraní daného výpočetního systému atp.,
- výstupy v podobě například opracovávaných prvků a další.

Výjimkou jsou samozřejmě systémy autonomní bez uživatelských rozhraní (ale i zde se bezpochyby najde aplikační modul, který slouží k jejich ovládání a ten rozhodně uživatelským rozhraním disponuje).

2.4.6 (Kdo bude danou akci provádět)

Tato informace je spíše doplňkovou informací k *2.4.4 Chování systému (algoritmus s ohledem na role) - ... a jak?* a používá se zejména tam, kde to může programátorovi například na základě jeho zkušeností s problematikou nebo na základě jeho vlastního uvažování při práci pomoci. Vzhledem k tomu, že u rozsáhlejších systémů by ale chování v závislosti na uživatelské roli mělo být součástí přímo návrhu, není tato informace nezbytná (pouze v kontextu programátora!) a obvykle je tedy spíše volitelným doplňkem k informaci o procesu.

3 Současné metodiky vývoje a postupy modelování

Ačkoliv mnohdy právě při opětovné tvorbě něčeho, co bylo již vytvořeno, dochází k nejvýznamnějším objevům a pokroku, nebylo pochopitelně záměrem autora této práce vytvářet dílo s nulovou přidanou hodnotou. Za účelem nalezení metodiky, která je ukazována v kapitole *4 Metodika "Kostka"* nebo metodiky jí podobné a za účelem nalezení některých informací které metodika obsahuje a které autor v době před tvorbou metodiky potřeboval, nebyly k dispozici, autor prostudoval širokou škálu dostupných metodiky a nástrojů které jsou v současnosti dostupné. S jejich pomocí pak identifikoval jejich prolínající se vlastnosti a naopak nedostatky a s doplněním klíčových informací ze znalostí programování mohla být vytvořena metodika uvedená v kapitole *4 Metodika "Kostka"*.

3.1 Modelovací metodiky

3.1.1 Přehled typů modelů

Pokud použijeme definici Websterova naučného slovníku (11), pak definice slova model nejlépe použitelná pro případ modelů IT systémů:

„a description or analogy used to help visualize something (as an atom) that cannot be directly observed"

tedy

„popis nebo analogie pomáhající zobrazit (vizualizovat) něco (jako například atom) co nemůže být přímo pozorováno"

Poněkud rozumnější a pochopitelnější definici nabízí známá Wikipedie *(12)*:

„Model je reprezentace určitého objektu nebo systému, pojatá z určitého úhlu pohledu."

Pokud tedy sloučíme obě definice, dostaneme informaci, že modely tedy slouží – pokud se budeme držet kontextu IT – k jedinému účelu:

„poskytují zjednodušené, ale pokud možno úplné schematické znázornění systému nebo jeho části který nebo kterou znázorňují."

(13) Z hlediska zpracování informací v informačních systémech můžeme tedy rozlišovat mezi dvěma základními pohledy:

- jaké informace zpracováváme,
- jak s nimi pracujeme.

Tyto dva pohledy jsou velmi důležité a je nutno se na ně zaměřit, protože budou využity jako jeden z klíčových podkladů v kapitole *4 Metodika "Kostka"*. Tyto dva pohledy jsou reprezentovány dvěma základními typy modelů:

- datové modely (jaké informace zpracováváme),
- funkční modely (jak s informacemi pracujeme).

V rámci těchto typů modelů se používají následující rodiny modelů:

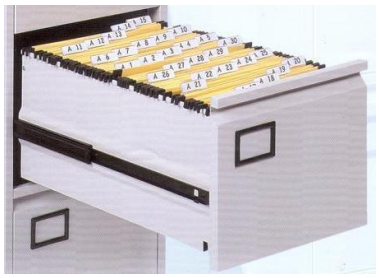
- datové modely (jaké informace zpracováváme) rozlišujeme na:
- lineární,
- hierarchický,
- síťový,
- relační,
- objektový.
- funkční modely (jak s informacemi pracujeme):
- slovní popis,
- dekompozice úloh,
- procesní diagramy,
- stavové diagramy,
- diagram toku dat (DFD),
- vývojové diagramy,
- rozhodovací tabulky.

K čemu jsou určeny tyto jednotlivé modely a jak jsou použitelné?

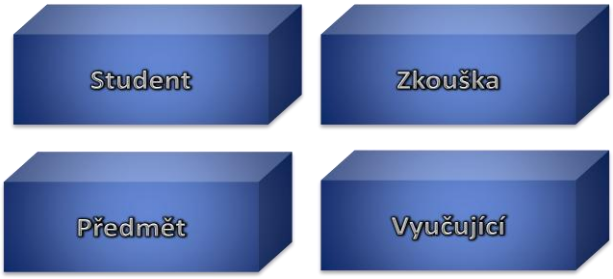
3.1.1.1 Datový lineární model (13)

V lineárních datových modelech není žádná vazba mezi jednotlivými skupinami objektů (tabulkami). Nemůžeme tedy stanovit, která

data jsou nějak logicky vázána na data jiná. Je to ovšem jediný datový model, který můžeme implementovat na libovolném datovém médiu – příkladem může být například kartotéka pacientů, kde jednotlivé karty s údaji o pacientech jsou seřazeny v krabici. Každá karta představuje jednu datovou větu databázového systému, ale mezi kartami (větami) není žádný jiný vztah než „následovník“ a předchůdce.



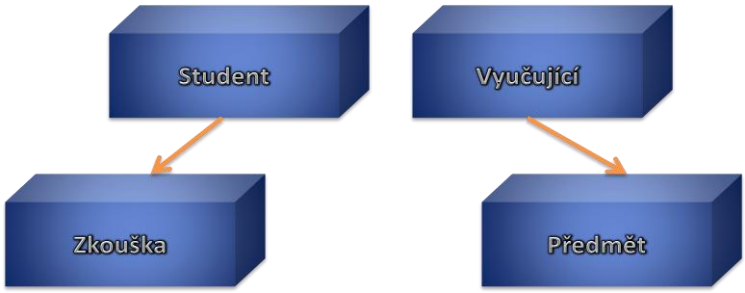
Obrázek 9: Lineární datový model a jeho ukázka



Obrázek 10: Schéma lineárního datového modelu

3.1.1.2 Datový hierarchický model (13)

Hierarchický datový model je tvořen rodičovským segmentem (větou) ze které vedou vazby na jemu podřízené segmenty (děti), což jsou segmenty (věty) jiné struktury a obsahu. Jestliže nadřízený objekt bude představovat například větu s údaji o studentovi, pak podřízené segmenty budou představovat například větu s údaji o jeho uskutečněných zkouškách. Vazby na jednotlivé segmenty jsou v implementační praxi vedeny tzv. pointery (ukazateli) které vytváří databázový systém na kterém je model implementován. Jak předpokládá toto paradigma, na segmenty podřízené rodiči se nelze dostat jinak než přes rodiče, tedy například nelze číst údaje o všech zkouškách studentů.

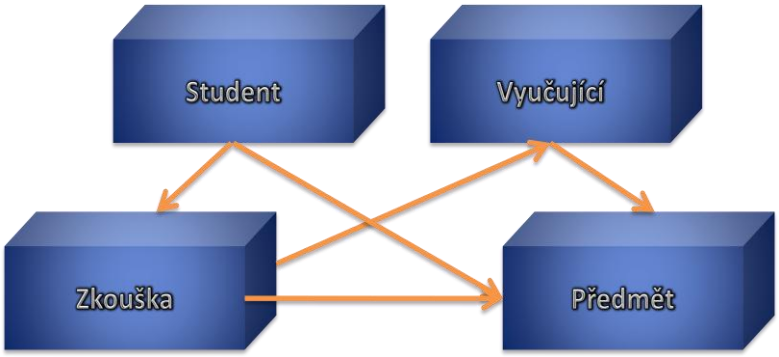


Obrázek 11: Hierarchický datový model

3.1.1.3 Datový síťový model (13)

Síťový model je obdobou modelu hierarchického avšak ukazatele vedou nejen z rodičovského segmentu na jeho dětské segmenty, ale mezi segmenty obecně. Výhody i nevýhody jsou zhruba stejné jako u 3.1.1.2 Datový hierarchický model avšak mezi další výhody patří například možnost libovolného propojení požadovaných segmentů a tedy i rychlý přístup k datům.

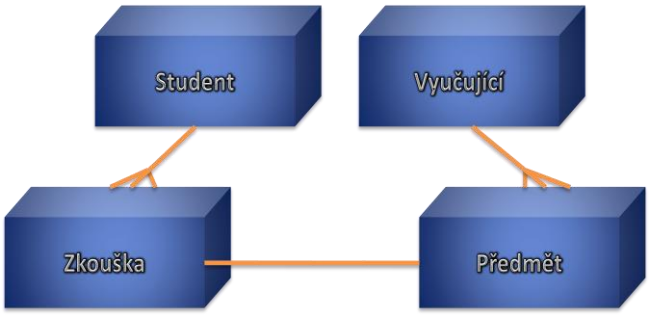
Oba tyto modely bývaly implementovány na počítačích typu mainframe – například v systémech DBS 25 nebo IDMS.



Obrázek 12: Síťový relační model

3.1.1.4 Datový relační model (13)

K nejpoužívanějším typům datových modelů v současnosti patří relační datový model. Vzniká z několika lineárních modelů spojených dohromady pomocí položky (položek) kterým říkáme **relační klíč**. Toto spojení není trvalé jako u předchozích modelů ale vzniká v okamžiku kdy potřebujeme mát společně k dispozici data ze všech spojených tabulek a zaniká, když práci s modelem ukončíme. Jednotlivé lineární modely lze pochopitelně využívat i samostatně.



Obrázek 13: Relační datový model

3.1.1.5 Datový objektový model (13), (14)

(13) Nejnovějším datovým modelem je objektový datový model. Objektové datové modely jsou vystavěny na základním prvku – objektu (odpovídá přibližně pojmu „věta“), kde tento objekt má kromě svých atributů (které má stejně jako věta v předchozích modelech) navíc i definované metody které určují chování objektu. Je-li takovým objektem například „zkouška studenta“ a atributy objektu například datum zkoušky, číslo studenta, známka a předmět, pak takovýto objekt může mít definovány metody jako „vytvoř záznam o zkoušce“ nebo „přiřaď studenta“.

(14) Vznik těchto metod byl motivován skutečností, že bylo (a stále je) potřebou vývojářů najít prostředek, který by umožňoval vytvořit reprezentaci skutečného světa, aniž by bylo potřeba měnit povahu věcí a jevů. Prvním použitím objektové technologie bylo využití v programu „Simula“[11] – tento postup byl v 60tých letech kdy byl poprvé použit revoluční: popište problém, nikoliv řešení. Objekty vypadali jako herci na virtuálním jevišti s vlastními znalostmi a stavem (daty – dnes již atributy) a schopnostmi (algoritmy, dnes již metodami). Tento postup vynikajícím způsobem demonstruje například dnes již legendární série her „Sims“. Jednotlivé objekty mají schopnost pomocí svých metod (schopností) ovlivňovat své atributy (své znalosti a stav) ale také atributy jiných objektů.

Takovéto objekty lze dělit na vlastní definici typů (tedy například můžeme definovat, že existuje objekt typu „zkouška“ který má datové atributy například „datum zkoušky“, „známka“, „předmět“ a „student“) a jejich konkrétní instance. Instancí je již

[11] Vyvinutý návrháři Dahl, Myhrhaug a Nygaard v Norském výpočetním středisku (Norwegian Computing Center).

existence záznamu **konkrétního** objektu, tedy **konkrétní zkoušky,** kterou s konkrétními údaji (například „1.2.2005 13:25“, „A“, „Datové a funčkní modelování“, „*Lukáš Plachý*“) konal určitý student. Samotný atribut studenta se může odkazovat na instanci objektu zcela jiného typu – tím bude instance objektu student („Jméno“, „Příjmenní“, „UČO“) která je „Lukáš“, „Plachý“, „54219“.

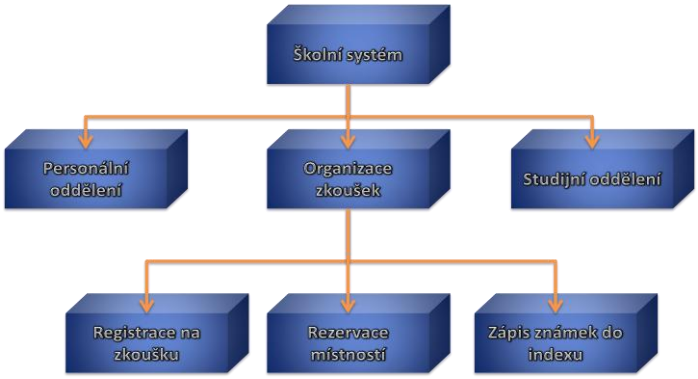
Nad obecnou definicí typu objektu můžeme definovat například metodu „ZadejHodnocení(Známka)“. Pokud takováto metoda bude zavolána nad naší konkrétní instancí třeba takto: ZadejHodnocení(„C“) změní se naše instance na „1.2.2005 13:25“, „**C**“, „Datové a funkční modelování“, „*Lukáš Plachý*“).

3.1.1.6 Funkční model slovním popisem (13)

Tato metoda patří k nejpoužívanějším při řešení úloh menšího rozsahu a pro komunikaci uvnitř pracovního analytického týmu. Z důvodů menší přehlednosti se nepoužívá při dokumentaci informačních systému, ačkoliv jak ukazuje například COCKBURN (15) lze s její pomocí modelovat informační systémy zejména ve stádiu sběru požadavků.

3.1.1.7 Funkční model dekompozicí úloh (13)

Tento model je spíše podpůrného charakteru s tím, že umožňuje pouze rozdělení systému na menší celky, zpravidla takové granularity, které jsou již předmětem samostatného řešení.



Obrázek 14: Funkční dekompozice

Bc. Lukáš Plachý (54219)
plachyl@std.fbm.vutbr.cz

3.1.1.8 Funkční model procesním diagramem (13)

V tomto modelu jsou na levé straně kresleny události, které proces ovlivňují, na pravé straně jednotlivé automatizované i neautomatizované činnosti. Diagram můžeme kreslit v různých stupních podrobnosti podle potřeby. Zpravidla začínáme na nejvyšším stupni zobecnění a podle potřeby diagram převádíme na dílčí řešení jednotlivých činností.

3.1.1.9 Funkční model stavovým diagramem (13)

V této metodě zkoumáme možné stavy objektů, které mohou nastat, a snažíme se popsat co tyto změny přechodu mezi stavy vyvolá. U tohoto typu diagramu si musíme uvědomit, že jej vždy kreslíme pro určitou entitu – objekt, jehož stavy a přechody mezi nimi sledujeme.

3.1.1.10 Funkční model diagramem toku dat (13)

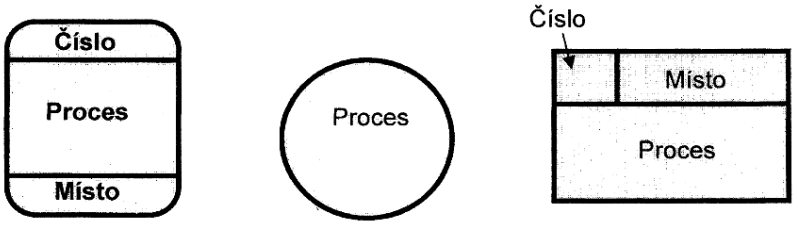
Diagram toku dat je jedna z nejpoužívanějších metod funkčního modelování. Můžeme z něj vyčíst návaznost jednotlivých činností v rámci úlohy, jaké datový vstupy a výstupy se v úloze objevují (tedy s jakými soubory a doklady se pracuje) a kdo jednotlivé činnosti provádí.

DFD diagramy můžeme kreslit na různé rozlišovací úrovni. Obvykle začínáme zachycením systému jako celku a postupně rozpracováváme jednotlivé funkce až na úroveň jedné úlohy.

Součástí DFD diagramů jsou následující entity:

- Proces (funkce, funkcionality, funkční krok)
- Externí entita (externí zdroj dat)
- Uložení dat, datová paměť (data)
- Pojmenovaný datový tok

Pro tyto entity existují různé symboly dle použité notace:

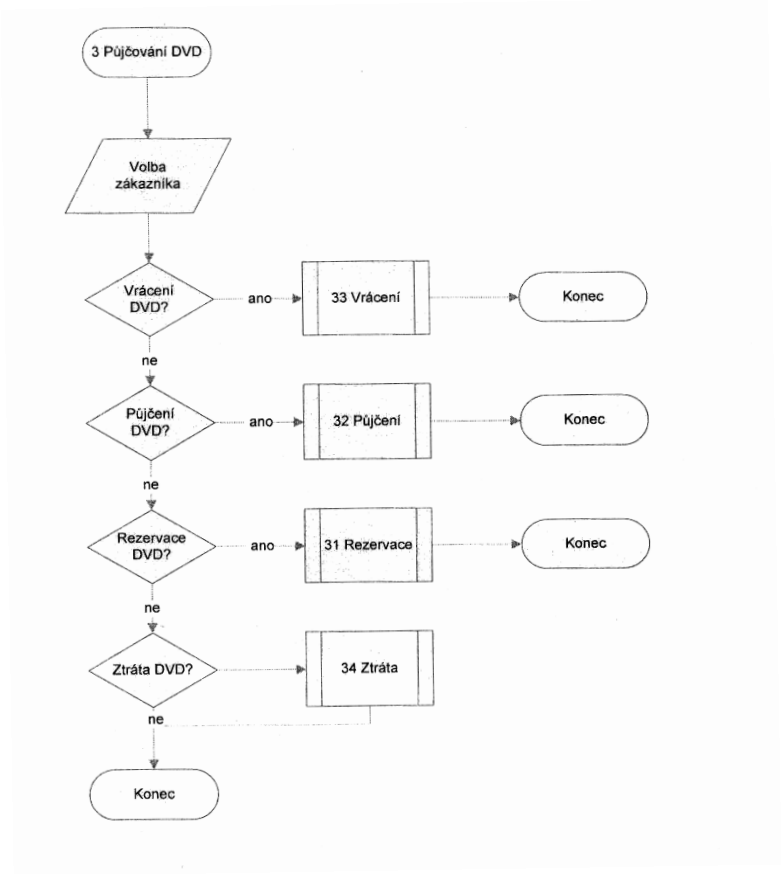


(Gane and Sarson) *(Yourdon and Coad)* *(SSDAM)*

Obrázek 15: Příklad různého zobrazení entity procesu (převzato ze (13))

3.1.1.11 Funkční model vývojovým diagramem (13)

Vývojový diagram patří společně s DFD k nejpoužívanějším – jeho hlavní výhodou je možnost zachytit velmi dobře větvení a smyčky zpracování podle splnění či nesplnění požadovaných podmínek.



Obrázek 16: Ukázka vývojového diagramu (převzato ze (13))

3.1.1.12 Funkční model rozhodovací tabulkou (13)

Poslední metodou v seznamu je rozhodovací tabulka. Tato metoda se používá k popisu výběru řešení v závislosti na parametrizaci rozhodovacího procesu. Rozhodovací tabulka má 4 kvadranty – v prvním, vlevo nahoře, jsou formulovány podmínky (dotazy). v druhém vpravo nahoře jsou uvedeny možné varianty odpovědí na tyto dotazy. Třetí kvadrant vlevo dole určuje množinu přípustných řešení a čtvrtý vpravo dole přiřazuje tato řešení k jednotlivým možným variantám hodnot parametrů odpovědi (z druhého kvadrantu).

3.1.2 Datové modelování

Veškeré tyto obecné typy diagramů jsou provedeny pomocí různých notací, které tyto typy nejen rozvíjí, ale také navzájem kombinují ev. doplňují. Je nutno také rozlišovat mezi způsobem modelování – tj. principem diagramu a jeho způsobem grafické reprezentace, tzv. notací.

- **ER Diagramy (Peter Chen)** (16)

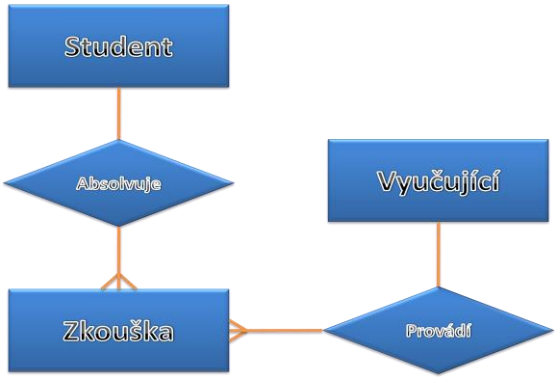
Pro návrh a zápis vztahů mezi jednotlivými entitami databáze byl vytvořen model E-R diagramů (Entity Relationship Diagrams). Typový E-R diagram je obvykle obrázek podobný klasickému vývojovému diagramu, v němž jsou entity znázorněny obdélníky, vztahy kosočtverci a mezi nimi vedou čáry, aby se poznalo, co k čemu patří.

Stejně jako bylo uvedeno datových relačních modelů (viz. kapitola 3.1.1.4) i zde se rozlišuje kardinalita vazby (v našem příkladu je metodika modelování ER diagramů doplněna notací „muří nohy“ (v originále „Crow’s foot“) použité poprvé v notaci R. Barkera – viz. níže):

1:1 - do vztahu vstupuje nejvýše jedna hodnota obou zúčastněných entit

1:N - nejvýše jedna hodnota jedné z entit a neomezený počet hodnot druhé entity

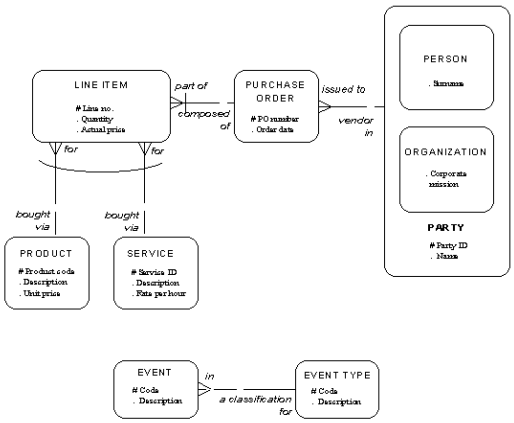
M:N - do vztahu vstupuje neomezeny počet hodnot obou zúčastněných entit



Obrázek 17: Ilustrace relačního diagramu

- **Richard Barker** (17)

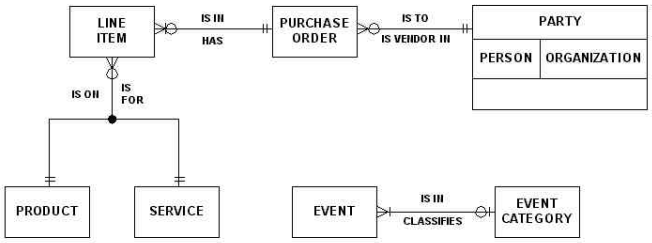
Tato notace byla původně vyvinuta v British consulting company a proslavena Richardem Barkerem (v publikaci: *CASE*Method: Entity Relationship Modeling*). Jedná se modelování entity (jako obdélníčky s kulatými rohy) s volitelnou možností zobrazení jejich atributů (uvnitř obdélníčku) a jejich vztahů. Tato metoda je dodnes používána ve společnosti Oracle.



Obrázek 18: Ukázka metody Richard Barker

- **Information engeneering** (17)

Metoda vyvinutá původně v roce 1970, Clivem Finkelsteinem a Jamesem Martinem, pracuje s datovými entitami. Protože každý z autorů vydává později svoji vlastní korekci metody, jsou k dispozici dvě různé metody – ovšem obě se stejnými diagramy, avšak zatímco Finkelsteinovy schémata zobrazují spíše datové entity, Martinova schémata popisují spíše více „entity o kterých ukládáme informace“.



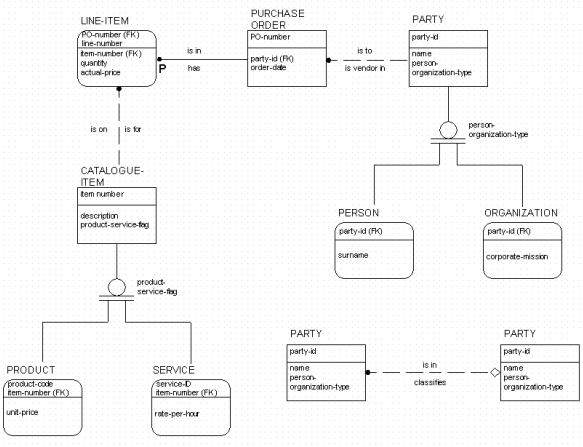
Obrázek 19: Ukázka metody IE

- **Object role (ORM)** (17)

Jedná se o analytický přístup, kdy se více než entity popisují zejména jejich vztahy. Je poměrně dost komplikované zobrazovat v této metodice entity bez jejich vztahů.

- **IDEF1X** (17)

Technika datového modelování používaná například zejména Federální správy Spojených států amerických. Jedná se o metodiku modelování entit a jejich atributů, stejně jako jejich vazeb. Entity jsou zobrazovány pomocí obdélníčků s kulatými rohy, uvnitř kterých se nacházejí i jejich atributy.



Obrázek 20: Ukázka modelu metodiky IDEF1X

- **XML** (18)

Jedná se rovněž o jeden ze způsobů jak modelovat data a to včetně jejich hierarchie. Cílem existence značkovacího jazyka je zápis datových (zejména textových) struktur, s určitými vlastnostmi, **čitelný jak člověkem, tak strojem**. Princip značkovacích jazyků spočívá v definici **klíčových slov** a speciálních znaků, sloužící jako „příkazy“ pro vykreslování vlastních datových částí dokumentu.

Bc. Lukáš Plachý (54219)
plachyl@std.fbm.vutbr.cz

Těmto klíčovým slovům se říká **„tagy“** a jsou uvozeny právě speciálně definovanými znaky. Příkladem tagu může být například:

```
<tečka/>
```

Vidíme klíčové slovo „tečka“. Na speciální význam tohoto slova upozorňují obě závorky (běžně používané ve smyslu „je větší než“ a „je menší než“) „<>“. Zde je složitější příklad:

```
<text>Vypisovaný text s <dulezite>důležitým</dulezite> slovem</text>
```

Zde vidíme tzv. párové tagy – tedy klíčová slova (obvykle stejná) z nichž jedno má význam otvírající, následuje datová část, a druhé má význam uzavírací. Uzavírací význam bývá specifikován pomocí speciálního znaku (zde lomítko „/“). Díky existenci takovýchto tagů jsme pak schopni pomocí značkovacích jazyků zapisovat i strukturu, například strukturu dat.

Díky existenci jazyka šablon (XSD), tedy jazyka, který definuje možné (povolené či dokonce vyžadované) tagy a jejich atributy, stejně jako datové typy jejich hodnot, lze definovat entity, jejich atributy a jejich relace, což může být druhem datového modelu. Stejně tak je možné použít XML nebo XSD jako standart pro zápis modelů jiných.

3.1.3 Objektové a funkční modelování

- **Jacobson** (19)

Jacobsonova metodologie (nazývaná Objectory) byla publikována v knize Object-Oriented Systems Engineering, (Addison-Wesley, 1992). Snaží se podpořit celý životní cyklus vývoje softwarového produktu. Jednodušší verzí Objectory je OOSE (Object-Oriented Software Engineering), kterou Jacobson vytvořil jako metodologii pro tvorbu aplikací.

Jejím hlavním přínosem oproti metodikám její doby byl use-case diagram. Podle Ivara Jacobsona hrají use cases (případy užití) dvě významné role. Zachycují totiž požadavky na funkcionalitu systému a navíc každý objektový model strukturují. Jacobson dále obhajuje použití use cases. O objektových modelech prohlašuje, že samy o sobě jsou velmi rozsáhlé a těžko zpracovatelné mají-li komplexně zachytit celou

Bc. Lukáš Plachý (54219)
plachyl@std.fbm.vutbr.cz
20. 05. 2008

modelovanou realitu. Proto je podle něj vhodné nějakým způsobem onen velký a komplexní objektový diagram rozdělit. K tomu se nabízí využití use cases.

Jak se seznámíme v kapitole *4 Metodika "Kostka"* používá jako jednu z dílčích metod i navrhovaná metodika.

- **Coad and Yourdon** (19)

Autoři této metodiky zdůrazňují její objektovou orientaci zejména z důvodů zlepšení komunikace mezi analytiky a experty na oblast problému, dále zvýšení konzistence mezi analýzou, designem a programováním, znovupoužitelnosti analýz, designu a programových výsledků atd.

Analýza zahrnuje pět vrstev nazyvanych 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 částmi, které se musí dodržovat. Rozhraní má být jednoduché, složitost zůstává uvnitř subjektu.

Objekty - specifikace tříd. Třídy mohou být abstraktní a mohou také sloužit jako zdroj dědění. Mohou byt natolik obecné, že jejich instance nemají v modelu použití. Na druhou stranu existují třídy, jejichž instance v modelu žijí (reálně existují a mají nějakou dynamiku).

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) (čtenář si zde může poskytnout vazby na kapitolu *4.2 Vstupní model*). Pro odvození původce služby se doporučuje využít stavů objektu. Pro kontrolu úplnosti služeb se simuluje chování objektů a přitom se sleduje, zda model dělá to, co má.

- **Shlaer-Mellor** (převzato dle (19))

Metodika Shlaera a Mellora patří k prvním přístupům k objektově orientovanym metodologiím. Byla prezentovaná již v roce 1988 v jejich knize Object-Oriented Systems Analysis - Modeling the World in Data and Object Lifecycles: Modeling the World in States (Prentice-Hall, 1988).

V prvním kroku je systém nejprve rozdělen do odlišných nezávislých domén. Existují čtyři typy domén:

- aplikační domény (prvky, funkce, systému, které bude využívat uživatel systému),
- servisní domény (uživatelské rozhraní),
- doména architektury (organizace dat, kontroly a řízení v programu),
- implementační domény (např. operační systém a programovací jazyk).

Ve druhém kroku při analýze aplikační domény jsou vytvářeny další modely, a to objektový informační model (Object Information Model), modely stavů (State Models) a specifikace akcí (Action Specifications). Objektový informační model definuje objekty domény, vztahy mezi objekty (metodicky správně by se mělo říkat třídy). Tento model je založen na relačním modelu dat (ERD - Entity Relationship Diagram). Model stavů popisují životní cyklus každého objektu, který má nějakou dynamiku. Specifikace akcí popisuje procesy potřebné v stavových diagramech. Každý stav objektu obsahuje specifikace akcí v něm probíhajících. Specifikace akcí mohou byt znázorněny pomocí diagramu datových toků s akcemi (Action Data Flow Diagrams - ADFD, obdoba klasických DFD) nebo dobře navrženým jazykem pro popis akcí.

Vedle těchto tří základních modelů existuje ještě několik odvozených modelů.

Jak se seznámíme v kapitole *4 Metodika "Kostka"* tyto pohledy byly jako idea převzaty do navrhované metodiky (spolu s pohledy metodiky ARIS – viz. níže), pouze mírně poupraveny tak, aby přesně vyhovovaly dnešním potřebám.

- **Fusion** (převzato dle (19))

Metodika Fusion byla vyvinuta u Hewlett-Packard v roce 1990 Derekem Colemanem, který vedl tým ve Velké Británii. Cílem bylo vytvořit jednoduchou, přitom však komplexní metodu návrhu informačního systému.

Bc. Lukáš Plachý (54219)
plachyl@std.fbm.vutbr.cz
20. 05. 2008

Tato metodika vznikla kombinací myšlenek jiných metod, a to Booche, Jacobsona, Rumbaugha a dalších. Zapracovány byly hlavně ty myšlenky, které měly největší využití v praxi. Metodika Fusion je zaměřena na OOA a OOD. Kombinuje několik vlastností předchozích metod. Poslední varianty této metody využívají jazyka UML (např. iterativní EvoFusion), na rozdíl od původní metody (ClassicFusion).

Fáze analýzy spočívá ve vytvoření těchto modelů (diagramů):

- Objektový model podle metodiky Fusion (drobné úpravy klasického objektového modelu).
- Diagram systémové interakce a mapa událostí.
- Model rozhraní.

Design pak využívá těchto modelů:

- Podrobné diagramy interakce (Interaction Graphs).
- Model dědičnosti podle metodiky Fusion.
- Důležitou součástí této fáze je i vytvoření konzistentního Dictionary.

Za výhody bývají označovány jednoduchost a možnost automatického testování.

- **OMT** (převzato dle (19))

Metodiku popsal James Rumbaugh v knize Object-Oriented Modeling and Design (Prentice-Hall, 1991). Obsahuje celou řadu myšlenek a přístupů, které jsou důležité pro analytiky a designery. OMT byla a je velmi oblíbenou metodikou analýzy a návrhu, dokazuje to i fakt, že mnoho nových metodik vychází právě také z ní.

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 (proto by byl asi lepší překlad názvu modelu jako model tříd, název objektový model může být terminologicky zavádějící - pozn. autora). Objektový model zachycuje statickou strukturu systému.

Bc. Lukáš Plachý (54219)
plachyl@std.fbm.vutbr.cz
20. 05. 2008

Dynamicky 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 diagram událostí (model událostí podle Drbal, P. a spol.: Objektově orientované metodiky a technologie - 1. a 2. díl, v angličtině se nazývá Event Schema nebo i 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).

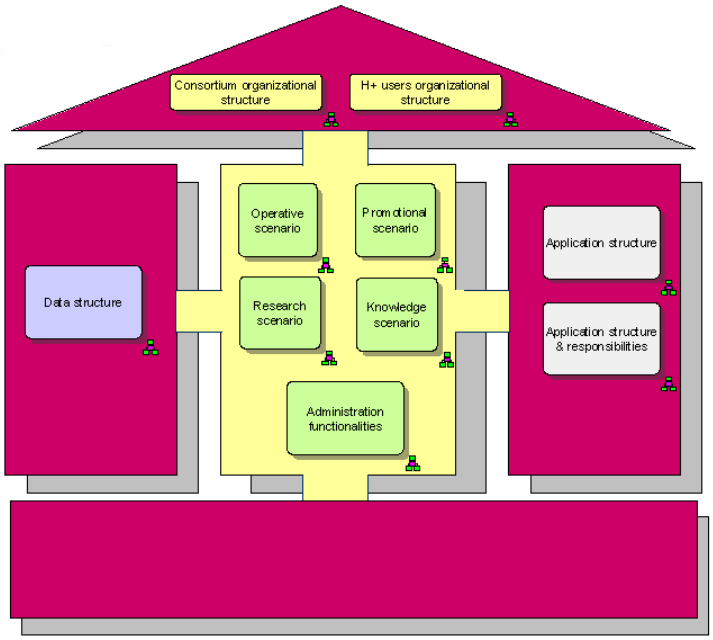
Každý model popisuje pouze některé aspekty systému. Obsahuje však odkazy a vztahy s ostatními modely. Pokusím se v krátkosti nastínit základní vztahy modelů (jsou velmi důležité, díky nim je tato metodika velmi kompaktní). Objektový model znázorňuje objektovou strukturu, na které jsou prováděny operace znázorněné v dynamickém a funkčním modelu. Dynamický model zase ukazuje akce, které závisí na hodnotách objektů (lépe řečeno jejich atributech). Ty pak způsobují změny hodnot objektů a vyvolávají funkce. Funkční model popisuje funkce vyvolané operacemi (metodami) v objektovém modelu a akcemi v dynamickém modelu. Funkce zpracovávají hodnoty dat definovaných v objektovém modelu.

- **ARIS** (20)

Toto je nejen sada nástrojů, ale také ucelená metodika, která ovšem není metodikou řízení vývoje nebo modelování přímo softwarových aplikací. Jedná se o metodiku vyvinutou v osmdesátých letech profesorem Augustem-Wilhelmem Scheerem, který ji do praxe uvedl s pomocí založení své společnosti IDS Scheer. Úkolem společnosti (a tedy i metodiky) je zejména procesní inženýrství. To spočívá samozřejmě zejména na modelování různých obchodních procesů. Metodika ARIS spočívá ve 4 základních pohledech na proces. Má tedy celkem 5 typů modelů, které jsou sdruženy do tzv. ARIS house. ARIS house je speciální přehledový model, v jehož centru se nacházejí modely procesů – tedy diagramy, které pomocí určitých objektů modelují průběh procesů v organizaci s tím že ke každému kroku procesu (funkci) přiřazují pomocí dynamických vazeb různé další objekty, jako jsou například role, aplikační systémy, odborné termíny (data) atp. Každý objekt může v procesním

diagramu být v jiné vazbě na daný krok (funkci, operaci) procesu. Například objekt role „účetní“ může být k objektu „zaúčtování faktury“ ve vazbě „provádí“ nebo třeba také jen „kontroluje“.

Strany domu ARIS jsou pak strukturujícími modely, které reprezentují strukturu a vztahy těchto objektů (entit), které jsou vázány na procesy. Vlevo se jedná o strukturu odborných termínů (v pojetí aplikace se jedná o data), vpravo o strukturu aplikací (tedy typicky softwarových systémů jejich součástí), ve střeše se nachází diagramy organizačních struktury (tedy struktura rolí a ev. osob v rámci organizace nějakého procesu) a v základech domu je struktura výrobků, tedy jak jednotlivé výrobky (produkty, služby) vycházející z procesu souvisejí spolu navzájem.



Obrázek 21: Ukázka strukturovacího modelu ARIS house

Metodika „Kostka“ ovšem čerpala náměty mimo jiné i z této metodiky, ačkoliv její prvky jako takové nepoužívá.

- **UML** (21)

Unified Modeling Language je celá sada modelů a jejich objektů určená pro modelování různých typů informací. Těmito diagramy jsou:

- Activity diagram (diagram činností) – soužící ke znázornění průběhu algoritmu s ohledem na vstupně/výstupní události (signály), datové

entity které vznikají/jsou měněny při tomto procesu a kontexty (swim lanes) ve kterém jsou dané akce vykonávány

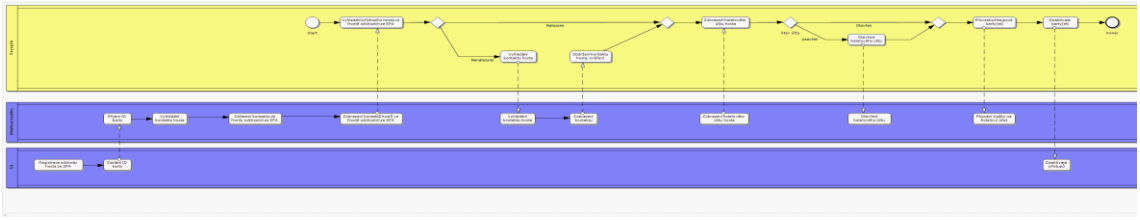
- Sequence diagram (diagram sekvence) je model který slouží ke znázornění průběhu nějakého děje, ovšem s důrazem na paralelizaci a ev. synchronnost/nesynchronnost těchto opreací.
- Use-case diagram – je diagramem který znázorňuje souvislost určitých funkcí či funkčních celků s aktéry (tedy rolemi v systému).
- State diagram – stavový diagram – je diagramem pro specifikaci stavů objektu v závislosti na časové linii, tedy specifikace stavů při průběhu v čase.
- Structuring diagram – diagram struktury znázorňuje prvky systému a jejich vzájemné souvislosti.
- Deployment diagram – je diagramem který reprezentuje strukturu aplikačního systému na úrovni konkrétních prostředků a nasazení, tedy jak je dané aplikace strukturována na úrovni nejvyšší (jednotlivých aplikací, služeb) a jak je zasazena do skutečného systému.
- Cooperation diagram - Diagram spolupráce zobrazuje interakci (důraz je tedy kladen na rozhraní) mezi objekty a jejich vzájemné vztahy.

- **BPNM** (22)

Tato metodika se specializuje na modelování obchodních procesů (podobně jako třeba ARIS). Jejím hlavním cílem je poskytnutí podkladu srozumitelného pro všechny zúčastněné (zainteresované). Na rozdíl třeba právě od metodiky ARIS se ale nesnaží modelovat různé pohledy na systém, ale používá pouze jeden pohled – pohled funkční. Z toho také vyplývá seznam objektů, které poskytuje: Start a konec události, Úkol (krok, funkce) a proces (odkaz na jiný diagram), bránu, rozdvojení a rozhodnutí.

Objekty souvisejícího toku jsou modelovány do tzv. swim lanes (podobně jako swim lanes aktivity diagramu UML, který ostatně z této metodiky do značné míry vychází). Do seznamu objektů také patří tzv. artefakty, což jsou Data, skupiny a anotace (poznámky).

Bohužel metodika je příliš otevřená pro dialekty (tj. různé implementace) v různých nástrojích, vlastní rozmístění objektů v modelu není standardizováno, objekty neznají svůj stav atp.



Obrázek 22: Ukázka modelu BPMN

3.2 Metodiky řízení životního cyklu

Ačkoliv zde navrhovaná metodika není ucelenou metodikou řízení projektu vývoje IS, má přesto (jak již bylo výše uvedeno) některé charakteristiky které předepisují či doporučují posloupnosti kroků a přenosu jejich znalostí, navíc zasahuje různé části metodiky – tedy jedná se o část metodiky řízení. Je tedy vhodné seznámit se i s metodikami řízení projektů.

- **„Velký třesk“, programuj a opravuj** (5)

Jedná se o metodiku, kdy se prostě napíše určitá aplikace, předpokládá se že vyhovuje požadavkům, tak jak je pochopili vývojáři (tedy přesněji přímo kodéři) a následně je software metodou velkého třesku (tj. „ze dne na den“) aplikován. Následně se na základě zjištěných chyb aktualizuje a mění dle potřeb a těchto chyb.

Vývoj je nestrukturovaný, jednotlivé etapy i projekt sám nemá přesně specifikovaný začátek nebo konec atp.

- **Vodopád (kaskáda), fontána, spirála** (5)

Toto jsou metodiky, které v zásadě popisují životní cyklus projektu, tedy dělí projekt na etapy a umožňují určité návraty.

Dle (5) se v případě vodopádu (někdy nazývané „kaskáda“) jedná o postup, kdy se celý projekt dělí do etap „Definice problém“ po které následuje „Specifikace požadavků“, „Návrh“, „Implementace“, „Integrace a testování“, „Údržba“. Etapy jsou

postupně prováděny tak jak jsou předepsány, po dokončení každé z nich se předpokládá její správnost a v případě problémů je nutno pro tuto opravu zahájit nový projekt.

V případě fontány se předpokládají podobné kroky jako u vodopádu, avšak mezi etapami se lze vracet vždy k předchozí etapě, pokud v té následné něco selže.

U spirálového modelu se jedná o komplexnější pojetí, kdy celý proces vývoje je rozdělen na 4 základní fáze, které se neustále opakují. Tím je „Stanovení cílů“, „Analýza rizik“, „Vývoj, testování a ověřování“ a „Plánování“. V každé této iteraci se celý projekt posunuje do další fáze, hlavní práce se nacházejí ve „Vývoji, testování a ověřování“ s tím, že podle stavu projektu se zde buďto provádí „vytvoření konceptu“, nebo (v další iteraci, které prošla všemi hlavními fázemi) „specifikace požadavků“, potom „návrh architektury“ a konečně „podrobný design“, „implementace“, „testování“ a „nasazení“.

- **Extrémní programování** (5)

Jedná se o metodiku řízení projektu spadající do agilních metodik (agilní metodiky viz. kapitola 2.3). Vychází se základních premis, že v každé lidské činnosti se osvědčuje jednoduchost, tedy vývoje softwaru v extrémě krátkých iteracích, kdy každý den mohou být měněny návrhy i jejich implementace, stejně jako se průběžně mění architektura a je průběžně testováno. Jako projekt pracuje XP (eXtreme Programming) se 4 projektovými proměnnými: kvalita (funkcionalita); čas; náklady; šíře zadání. Jedním z axiomů XP je že zákazníci a manažeři vybírají hodnotu tři z nich, čtvrtou hodnotu určí vývojový tým.

- **SCRUM** (5)

Cílem této metodiky je především zvýšení efektivity při vývoji softwaru. Rovněž SCRUM využívá výhod iterativního a inkrementálního přístupu. Základní motivací je vyrovnat se lépe a flexibilněji s neustále se měnícími požadavky.

SCRUM vychází z objektového přístupu, kdy každý programátor odpovídá za množinu jasně definovaných objektů s určitým pevně daným rozhraním. Vývoj pomocí metodiky probíhá v rámci tří až osmi posloupností pevně daných časových intervalů, tzv. Sprints (sprinty). Každý sprint trvá obvykle měsíc. V jejich rámci nepředpokládá

SCRUM žádné konkrétní procesy, ale pouze denní *scrum meetings* ze kterých vyplyne určení denních činností.

Metodika předpokládá, že není možné naplánovat dopředu detailnější průběh sprintu ani celého vývoje a celý svůj průběh staví právě na rozhodnutích učiněných v denních meetinzích.

- **Lean Development** (5)

Jedná se o metodiku (upravenou pro svět vývoje software z originálního *Lean Manufacturing*) která spočívá v systematickém přístupu k identifikování a eliminaci možných zdrojů plýtvání v průběhu celého vývojového procesu. Nejedná se o přesné pokyny jak řídit vývojový tým, ale metodika je zaměřena strategičtěji – snaží se definovat postupy a cíle k tomu, aby byl celý vývoj software efektivnější. Drží se následujících kroků:

1) Odstranit vše, co je zbytečné
2) Minimalizace zásob (meziproduktů)
3) Maximalizovat tok (zkrátit čas potřebný pro vývoj)
4) Vývoj je tažen poptávkou (rozhodnutí se dělají co nejpozději je možné)
5) Pracovníci mají pravomoci rozhodovat
6) Hlavním cílem je uspokojovat nynější ale i budoucí požadavky zákazníků
7) Zavést zpětnou vazbu
8) Odstranit lokální optimalizaci (neustálé optimalizace postrádají smysl!)
9) Vybudovat partnerství s dodavateli (využití subdodávek a předpřipravených komponent)
10) Vybudovat kulturu všeobecného zlepšování.

Bezesporu zajímavá metodika, avšak bohužel neříká nic o postupu a potřebných informacích nutných pro vývoj aplikace.

- **Test Driven Development** (5)

Metodika je založena na testování – testování tedy není jen jednou její částí, ale metodika jej prohlašuje za svůj základ. Její princip tkví v postupu, kdy je pro každou drobnou součást zdrojového kódu **před** jejím vlastním vývojem navržen a napsán test.

Pak je teprve implementován kód s tím, že se píše přesně takové množství kódu (ani méně ale ani ne více) který je schopen projít testem.

Metodika má následující cyklus: Přidání nového testu, Spuštění testů, realizace drobné změny, spuštění testů. Je-li test neúspěšný, vrací se metodika zpátky. Jinak po projití cyklu zahajuje cyklus nový. Protože zmíněné testy jsou vypracovány na detailní úrovni nejdrobnějších funkcí, délka jednotlivých cyklů je minimální.

- **Adaptive Software Development** (23)

Tato metodika nahrazuje tradiční vodopádový cyklus projektu opakující se sérií cyklů „spekuluj“, „spolupracuj“ a „uč se“. Tyto dynamické cykly poskytují průběžné učení se a adaptaci projektu do finálního stavu projektu. Charakteristiky jednotlivých cyklů jsou takové, že každý je zaměřený na konkrétní úkol (misi), založené na jednotlivých funkcionalitách, iterativní, ohraničené časem, řízené riziky a tolerantní ke změnám.

Slovo „spekulovat“ se odkazuje na tzv. paradox plánování – tedy předpokládá, že každý koho se dotýká aplikační systém se v nějakém aspektu tohoto systému mýlí, tedy nechápe jej korektně ve vztahu k jeho cíli (misi). „Spolupráce“ znamená snahy o vyvážení práce založené na předvídatelných částech prostředí (plánování a jejich vedení) a adaptaci okolního mixu změn způsobeného různými faktory – technologiemi, požadavky, prodejci softwaru, uživateli atp. „Učení se“ reprezentuje stav, kdy se během krátkých vývojových cyklů jednotliví účastníci učí ze svých chyb a chybných odhadů, což vede k jejich větší zkušenosti a eventuálně zvládnutí problémové domény.

- **Metodika Risk and Requirement Based Testing** (24)

Jedná se o metodu která nesnaží nestavět testování na jeho základních principech, kterým jsou dle autora zpravidla 4 premisy – testovat lze pouze specifikované požadavky, software musí splňovat všechny tyto požadavky, kde každý tetovací případ musí být trasovatelný k jeho požadavku, které musejí být specifikované v testovatelných výrazech.

Metoda tvrdí, že je ovšem potřeba jít dále, neboť systém není specifikován je svými požadavky, ale mnohem širší množinou potřeb. A právě testování by mělo být

schopno testovat i tyto nevyslovené požadavky. Je nevhodné chápat software jako soubor požadavků, které byly buďto splněny nebo ne. Mnohem lepším přístupem je chápat tyto požadavky jako hodnotu rizika. které vyplývá z jejich nesplnění. Jednou z dobrých otázek správného vývojáře na této cestě je: „Co jsou důležité problémy v tomto produktu?“

Zajímavým příkladem je rozčilená testerka, která začala uvažovat nad testovacím případem že „tento ovládací objekt musí zareagovat do 300 milisekund“. Její uvažování začalo přemýšlením o speciálním softwarovém nástroji, který si bude muset koupit a jak různé sub-procesy ve Windows mohou ovlivnit toto měření. Pak zjistila, že běžný uživatel s trochou pozornosti umí odhadnout tento čas s přesností +/- 50 milisekund, což se jí zdálo dostačující. Až když kontaktovala hlavního návrháře, zjistila, že původní požadavek byl „aby to nebylo tak otravně pomalé jako současná verze produktu“.

Tyto poznatky lze shrnout do metodických pokynů zmíněné metodiky:

1) Naše schopnosti rozpoznat problém jsou ovlivňovány naším chápáním problému. Seznam požadavků je jedním z možných zdrojů. Existují ale i jiné.
2) Účelem testování je ukázat ve světle rizika softwaru (produktu) nejen demonstrovat jeho konformitu s požadavky.
3) Zejména ve vysoce rizikových situacích lze mít mnohem účinnější testy, pokud umíme ospravedlnit návaznost jejich strategie na definice kvality.
4) Testovací proces bude mnohem efektivnější, pokud jsou požadavky specifikovány tak, aby komunikovaly esenci jejich přání, rizik a přínosů v souvislosti s relativní důležitostí každého požadavku.

Metodika se týká spíše testování než vývoje jako takového.

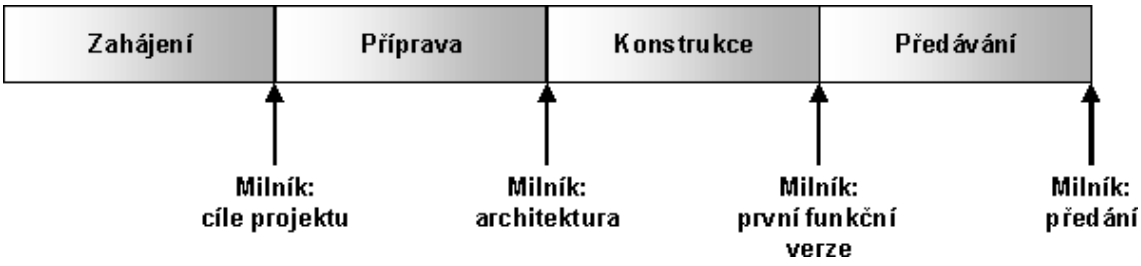
- **RUP** (citováno z (25))

Základní filosofií metodiky Rational Unified Process (komerční verze metodiky UP – Unified process) je šest základních pravidel, tzv. “nejlepších praktik” používaných při vývoji software:

- Iterativní vývoj software
- Správa požadavků
- Architektura založená na komponentách

- Vizuální modelování
- Ověřování kvality software
- Řízení změn software

Životní cyklus projektu je v metodice Rational Unified Process rozdělen do čtyř základních fází (viz obr. 23), z nichž každá je zakončena tzv. milníkem. Po dokončení každé fáze se provede zhodnocení, zda byly splněny požadované cíle. Další fázi projektu je možné zahájit pouze v případě splnění všech požadovaných kritérií.



Obrázek 23: Fáze metodiky RUP (převzato (25))

Přestože názvy a pořadí prvních šesti disciplin mohou vyvolávat určité asociace s "vodopádovým" přístupem k vývoji software, je třeba mít na paměti určitou odlišnost: v metodice RUP se tato posloupnost v průběhu projektu aplikuje vícekrát, podíl jednotlivých činností však závisí na stádiu, v němž se produkt nachází.

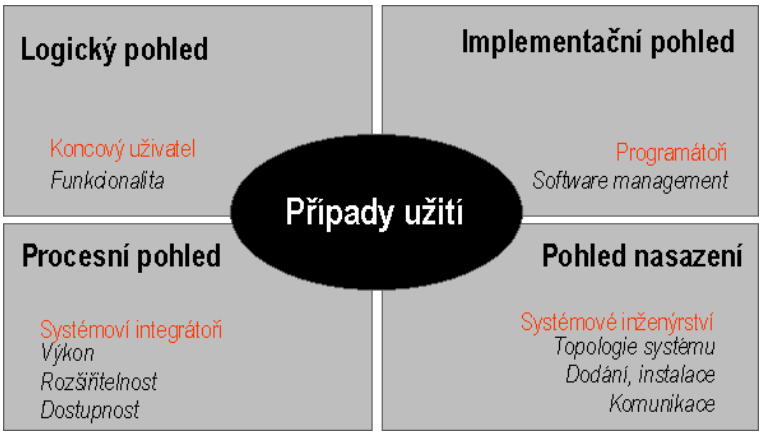
RUP rozlišuje mezi tzv. "hlavními" a "podpůrnými" disciplinami. Hlavní disciplíny zajišťují vlastní práci na projektu, podpůrné disciplíny slouží k jejich řízení a koordinaci. Jedná se o následující:

Hlavní disciplíny

- Tvorba podnikového modelu
- Správa požadavků
- Analýza a návrh
- Implementace
- Testování
- Nasazení

Podpůrné disciplíny

- Řízení projektu
- Řízení změn a konfigurace
- Správa prostředí



Obrázek 24: Pohledy na systém (převzato z (25))

Jak vidíme na obr. 24, Rational Unified Process používá pro znázornění architektury systému pět pohledů. Jednotlivé pohledy řeší různé aspekty fungování systému, nejsou na sobě však nezávislé a do určité míry se překrývají.

Všechny pohledy a modely zde uváděné nejsou samozřejmě "povinné" pro všechny projekty.

Jak můžeme vidět v uvedeném schématu, RUP používá určité pevně dané „pohledy" na problematiku zpracovávaného software. Podobné pohledy a souvislosti nalezneme i v metodice „Kostka" (ovšem jedná se o jiné pohledy, které jsou – dle autorova názvu – praktičtější).

Další aspekty metodiky RUP (Role, Projektová dokumentace, Zadání, Vize, konfigurace RUP, Plány iterací atd.) zde nemohou být z důvodů rozsahu práce uvedeny.

Co je na metodice RUP zajímavé je její komplexnost. Pokud budeme dostatečně hluboce studovat její dokumentaci a šablony dokumentů, zjistíme, že obsahuje (přesněji její konkrétní nasazení v konkrétním týmu může obsahovat) veškeré prvky, které jsou nezbytně potřeba. Bohužel metodika to neuvádí v dostatečně přehledném a rychle pochopitelném provedení a ani to není jejím účelem. Je velmi komplexní a to se podepisuje také na její použitelnosti. Rozhodně se nedá shrnout na jedinou stránku.

- **OOSD**

Je metodikou, která obsahuje jak objektově orientovanou analýzu, tak objektově orientovaný design. Jedná se o praktickou metodu vývoje SW, která se soustředí napříč

celým vývojem na objekty problému. Většina objektově orientovaných metod, které dnes soupeří o pozornost softwarových vývojářů, obvykle aplikují tradiční strukturovanou analýzu (funkční dekompozici) a teprve pomocí heuristických algoritmů transformují tyto dekompozice na objektové řešení.

- **Defect Driven Design (D^3, vyslov "D Cubed")** (zdroj nedostupný)

Tato metodika je poněkud nestandardním postupem, avšak její pohled na problematiku je bezesporu natolik zajímavý, že je dobré jej brát v úvahu. Tato metodika, pokud by měla být dotažena do absurdního konce, předpokládá, že celý projekt aplikace začne konstatováním „Program je hotov“. Jakmile se uživatel zeptá: „A kde se program spouští?“ je zaznamenána první chyba (defekt): „Program nelze spustit“. Chyba je předána implementátorovi a ten ji napraví – vytvoří spustitelný program, který se spustí a skončí. Tento produkt je předveden uživateli, který se zeptá: „A kde se budu přihlašovat?“ Je zaznamenána druhá chyba….
Tento přístup samozřejmě není úplně jednoduše realizovatelný a asi by nebyl ani vhodný pro praktické provedení, avšak brilance této metodologie tkví v její schopnosti odhadu. Můžeme s naprostou přesností předvídat časovou náročnost implementační fáze projektu: 0 dní. Ostatní je údržba – přesněji celý projekt spočívá v údržbě, což je realistické, protože skutečný informační systém nikdy není zcela hotov.
Co je na této metodice pozoruhodné je její teoretický iterační cyklus – metodika se totiž sama skládá z extrémně krátkých iterací nad celým životním cyklem, dalo by se říci, že metodika je sama iterací.

Uvedené metody jsou pouze výsekem různých metodik. Jedná se zejména o ty, které měly na metodiku uvedenou v kapitole *4 Metodika “Kostka“* největší vliv. Bylo by možné ale najít mnoho dalších metodik jako například JDS, JSP, Feature driven development, Crystal, Dynamic software development method atp.
Nejvýznamnějším přínosem těchto metodik je vedle nástrojů modelování zejména uvedení různých pohledů na sytém jako takový, z nichž metodika „Kostka“ dále čerpá.

4 Metodika "Kostka"

Poznámka: Kapitola vlastního řešení staví kromě vstupních rešerší zejména na informacích uvedených v kapitole *2.4 Co musí mít programátor k dispozici* která tvoří její nedílnou součást.

V předchozích kapitolách jsme viděli, že současné metodiky nejsou vždy zcela optimální a mnohdy samy sebe omezují pro vhodnost použití, ať už se jedná o úzké zaměření, přílišnou složitost atp.

Cílem této kapitoly je navrhnout metodiku, která bude představovat jakousi „minimální informační kostru", tedy soubor informací nutných pro vývoj a doporučení pro jejich zachycení a zpracování.

Jak jsme uvedli, námi popisovaný výsek metodiky, metodika „Kostka", má své pevné místo v životním cyklu ať už projektu aplikace informačního systému nebo informačního systému jako takového. Rekapitulujme znovu tuto pozici:

		Pracovníci	Org. postupy	Data	SW a HW	Org. vlivy	Ekon. otázky	Produkty a potřebné dokum.	Metody, techniky, nástroje	Způsob řízení
Analýza(+studie proveditelnosti)	**Cíle**									
	Testování klíč. techn.									
	Mapování požadavků									
	Mapování procesů									
Návrh	**Návrh procesů**									
	Návrh dat. struktur									
	Návrh I/O									
	Návrh. apl. arch.									
Implementace	**Návrh apl. procesů**									
	Coding									

		Pracovníci	Org. postupy	Data	SW a HW	Org. vlivy	Ekon. otázky	Produkty a potřebné dokum.	Metody, techniky, nástroje	Způsob řízení
Testování	**Návrh test-cases**									
	Provedení testů									
Nasazení	**Pilot**									
	Nasazení									
Vyhodnocení	**Vyhodnocení**									

Význam barev:

Není popisovaná fáze životního cyklu	Jedná se o popis IS jako celku, netýká se popisu SW aplikace
Tato metodika popisuje nebo předepisuje postupy	Tato metodika se jimi zabývá se pouze částečně nebo pouze doporučuje postupy

Tabulka 2: Fáze a oblasti, které popisuje metodika představená v tomto dokumentu

Kde tedy tato metodika začíná (tj. jaký je její vstup) a kde končí (tj. co má být jejím výstupem)? Jak je zřejmé z uvedené tabulky, vstupem mají být systematickým způsobem zmapované požadavky kladené na vyvíjenou aplikaci. Výstupem pak je zadání pro programátory, tj. takové modely systému, které již umožňují vlastní implementaci požadovaných funkcionalit.

Protože každé správné řešení je navrhováno „odshora dolů“, tj. od cílů po dekompozici všech jednotlivých činností a vstupů, které jsou potřeba dosažení těchto cílů, bude i tato kapitola (a tedy celá „Kostka“) vycházet z požadavků na své výstupy. Těmi jsou (dle *2.4 Co musí mít programátor k dispozici*):

- Vize a cíl – co se vlastně dělá?
- Struktura systému – jak to bude vypadat?
- Struktura dat – s čím se tam bude pracovat
- Chování systému (algoritmus s ohledem na role) - … a jak?
- Kdo bude danou akci provádět
- Vzhled (uživatelského) rozhranní systému

Doposud jsme se ovšem zabývali pouze programátorem, pro kterého v rámci implementace zadáváme práci. Existuje ovšem ještě jedna role v rámci implementace

systému, která musí mít dostatek informací. Touto rolí je aplikační administrátor. Musíme si uvědomit, že architektura systému lze specifikovat nejen na úrovni vlastní aplikace, ale také na úrovni jejího nasazení. Jedná se tedy nejen o vztahy mezi aplikačními moduly či dokonce jejich jednotlivými komponentami a třídami, ale – ze zcela opačného konce spektra – celými aplikačními celky a jejich vzájemnou souvislostí jako je například spojení klientů s databází, zapojení automatizovaných agentů atp.

Jak tedy budeme specifikovat jednotlivé modely a očekávané výsledky implementačních prací? Projděme nyní jednotlivé potřebné specifikace a s použitím informací a znalostí z kapitoly 3.1 sestavme přehled potřebných modelů a jejich souvislostí.

4.1 Modely

4.1.1 Model procesů aplikace

Konečnou a nejdůležitější informací pro programátora je informace o procesu (algoritmu) který plní. Takovýto model by měl být schopen zachytit:

- Na základě jakých vstupů (nebo akcí) mají
- které moduly provádět jaké akce,
- nad jakými daty
- a s jakým výstupem.

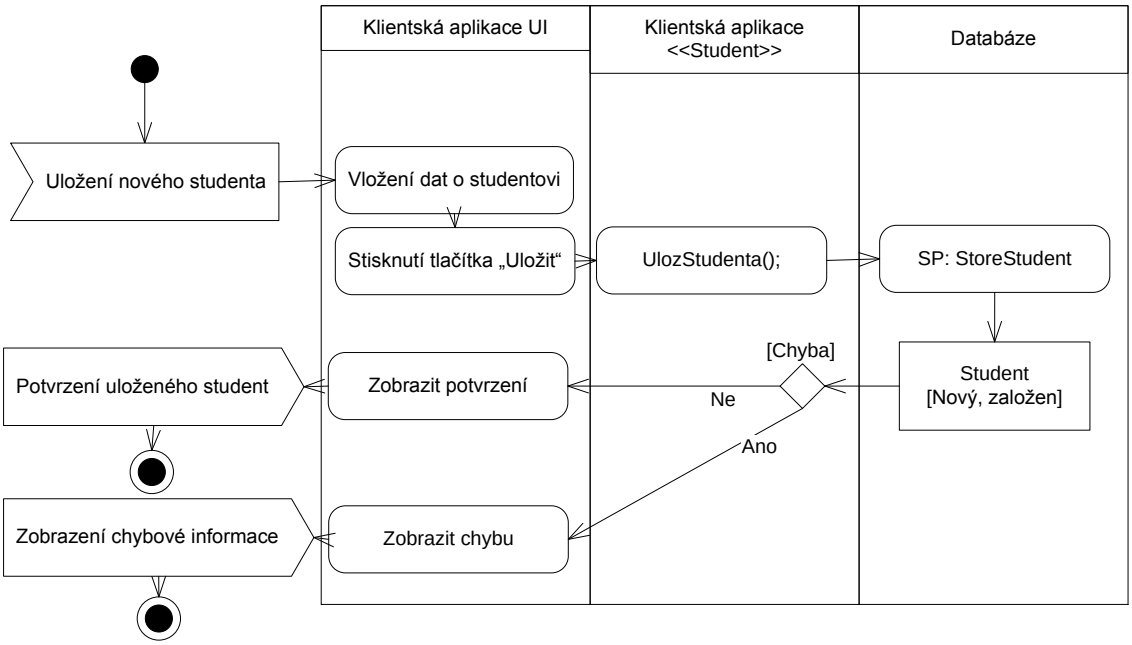
To znamená, dle různých úrovní detailu:

- Jaké operace se provádějí na základě jakých akcí mezi kterými aplikacemi/aplikačními moduly (implementační typ modelu).
- Jaké operace/funkce se provádějí na základě jakých akcí mezi kterými třídami (implementační typ modelu).
- Jaké konkrétní algoritmy představují jednotlivé operace (implementační typ modelu).

Výše uvedené lze aplikovat dle potřebného zaměření, ať již se jedná například o databázovou aplikaci (kde jednotlivými operacemi mohou být od uložených procedur, přes triggery či pohledy až po konkrétní SQL příkazy), klientskou aplikaci tohoto databázového systému (kde jednotlivými operacemi mohou být buďto funkční celky jednotlivých modulů či tříd nebo dokonce i konkrétní funkce těchto tříd).

Pro zachycení tohoto děje existují (budeme-li hovořit o UML) v zásadě dva vhodné diagramy: activity diagram a sequence diagram. Oba tyto diagramy lze použít pro zachycení zmiňovaných dějů s tím, že sequence diagram sice umožňuje větvení avšak pouze graficky poměrně těžko čitelným způsobem, na druhou stranu poskytujte neocenitelnou pomůcky v rámci OOP – znázornění existence objektů.

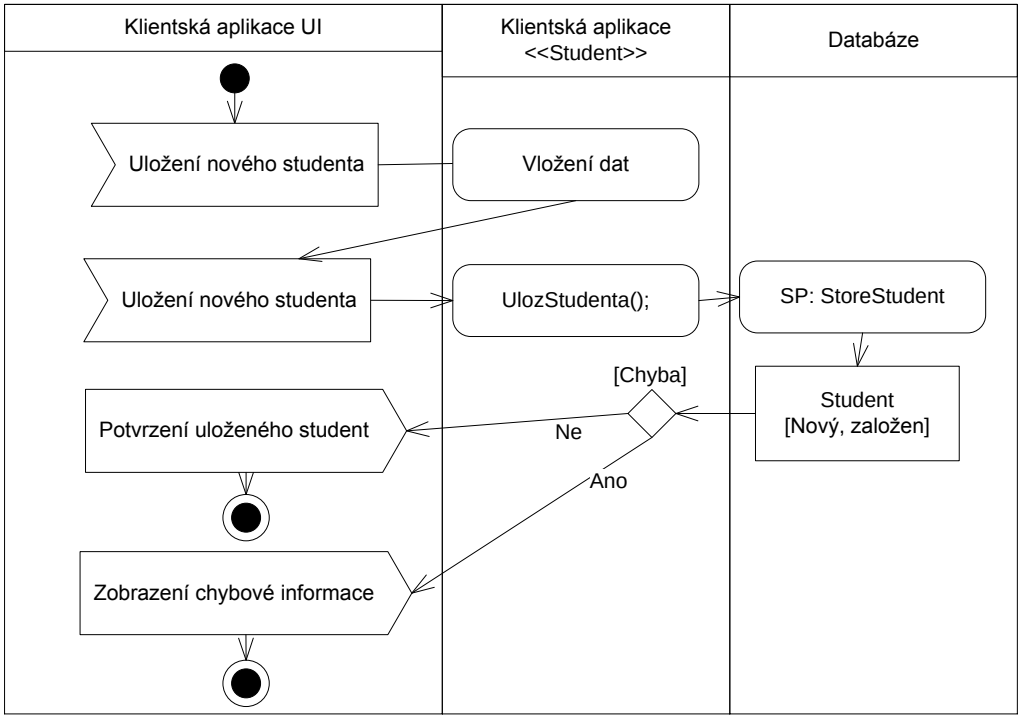
V praxi autor používá diagram který ideově vychází z aktivity diagramu, ale zavádí do něj navíc 3 prvky – jsou to jednak právě události, jednak informace o zpracovávaných datech a specifické vstupy/výstupy. Jedná se o identický model doporučený jako vstupní informace pro tuto metodiku. Jak souvisí vstupní model procesů s modelem výstupním a jaká je jeho metodika, o tom více v kapitole *4.2 Vstupní model*



Obrázek 25: Příklad toho jak by mohl vypadat sequence diagram v hypotetickém případě

V našem příkladu vidíme již velmi detailní popis funkce klientské aplikace a její volání uložené procedury (poznámka: jedná se opravdu jen o příklad – struktura aplikace, syntaxe procedur stejně jako zvolená úroveň detailu). Jak vyplývá z nákresu, jedná se zcela zjevně o procesní diagram (viz. *3.1.1.8 Funkční model procesním diagramem*). Jedná se o stav, kdy se pro uložení studenta je volána příslušná funkce, která volá službu databáze, která vytvoří záznam v datech (Student) a podle toho jak

toto volání skončilo, vrací chybu. Celý scénář by samozřejmě mohl vypadat i jinak, například takto:



Obrázek 26: Jiný příklad toho jak by mohl vypadat sequence diagram v hypotetickém případě

Zde se jedná o stejný proces, avšak uchopení funkcionalit se trochu liší – rozhodně poněkud korektnější, než předchozí příklad, uživatelské rozhraní je opravdu rozhraním a tak spíše než jej popisovat jako entitu, která zpracovává funkce je vhodnější na něj nahlížet jako na entitu která přijímá či vysílá signály. Na druhou stranu se trošku ztrácí korektnost pojetí procesního diagramu a striktního oddělení událostí a činností – na to je ale UML dobře připraven.

Podstatnou součástí modelu je jeho popis. Ten v případě tohoto procesního modelu obvykle není vhodné vytvářet. Popis procesního modelu je totiž samoúčelný – model sám má větší vypovídací hodnotu. V případě potřeby si lze navíc pomoci objekty typu „komentář“ nebo „poznámka“ které většina metodik nabízí (a pokud nenabízí, neznamená to ostatně, že se v modelu nemohou objevit).

Jediným objektem, který si obvykle zaslouží popis, jsou právě jednotlivé operace (či lépe funkce) které se na modelu vyskytují. Ty je ale vhodnější popisovat až v kontextu celých modul/tříd ve kterých se vyskytují.

Celý proces probíhající v rámci nějakého modulu je tak pouze klíčovým orientačním vodítkem pro vývojáře, jak jejich práce souvisí ať už mezi svými jednotlivými kroky, nebo jak navazuje na práci jiných. Jedná se tedy o model který ukazuje zejména souvislosti. Ale popis a vlastní dokumentace součástí systémů se týká především modelů následujících.

Jak je vidět na výše uvedených příkladech, náš model potřebuje následující informace:

- Informaci o struktuře aplikace (swim lanes),
- informaci o struktuře dat,
- informaci o rozhraních, která se procesu účastní,
- informaci o vstupně/výstupních událostech.

Všechny tyto informace získává z modelů ostatních, jak je uvedeno v následujících kapitolách.

4.1.2 Model architektury (aplikace)

Jednou z informací které model uvedený v kapitole 4.1.1 používá jako své swim lanes jsou jednotlivé aplikační moduly. Ty mají vazbu na aplikační architekturu, která je předmětem dalšího z důležitých modelů. Ten by měl obsahovat informace o:

- Jaké součásti aplikace,
- jakým způsobem souvisejí mezi sebou,
- a jak jsou ve vztahu k uživatelům.

I s ohledem na úrovně detailu lze tento model rozdělit na dva základní celky:

- na modely systému jako takového – tedy modely na rozhraní modelů reality a technologických, které nepředjímají nic o konkrétní implementaci (třebaže by ji měly zohledňovat) a
- na modely vnitřního schématu aplikace (tedy modely implementační).

Obě skupiny modelů jsou ovšem zpravidla velice podobné – třídy problémů které jsou řešeny a modelovány v modelech systému jako takového totiž model vnitřního schématu aplikace zpravidla přebírá a pouze rozšiřuje a doplňuje o nezbytné technologické nuance. Autorovi této práce se zde osvědčilo vynechat první třídu modelů a přejít rovnou k modelům aplikačním, přesněji chápat od počátku modely jako modely aplikace samotné a už během návrhu je rozšířit o nutné technologické metody a

atributy. Jakkoliv tento postup může v realitě způsobovat jisté problémy, má také některé výhody:

- zkušený architekt a programátor je schopen účelně zkombinovat logiku návrhu aplikace s rozumnou mírou její složitosti po programátorské stránce a je tak schopen vytvořit velice ekonomický návrh,
- zmenšuje se prostor chyby,
- urychluje se vývoj a snižuje se množství nutné komunikace,
- technologické problémy mohou být odhaleny zavčas (což by vytvoření návrhu struktury systému jako takového neumožňovalo, neboť to se – jak bylo uvedeno – abstrahuje od konkrétní implementace).

Mezi potencionální rizika této metody ovšem patří:

- „převážení“ role architekta na jednu ze stran – buďto uvažuje příliš jako programátor nebo příliš jako systémový architekt (či procesní inženýr),
- ztracení se v problému snahou kombinovat oba přístupy.

Protože metodika „Kostka“ má v postupu svého provedení zabudovány iterace, které tyto potencionální hrozby odstraňují, je zřejmé, že převáží spíše výhody.

Pokud tedy ztotožníme modely tříd systému s modelem tříd aplikace s tím, že se jedná v zásadě o jeden a tentýž model, jaké jsou jeho možné úrovně detailu? Jsou to tyto:

- Schéma základního principu aplikace – aplikační celky (celé aplikace) a vztahy mezi nimi [12], včetně vztahu ke konkrétním uživatelským rolím nebo jejich skupinám. (technologický typ modelu) V případě že se pouze jedná o domény problematik konkrétního systému jedná se o model systému jako takového, v případě že tyto (zpravidla ale zcela stejné) celky popisujeme jako aplikační moduly se jedná již o implementační model. Například pro systém knihovny:

Model systému - domény	**Model aplikace – moduly**
Výpůjčky (popis třídy funkčností týkající se výpůjček)	Výpůjčky (popis objektů aplikace s funkcemi pro funkčností týkající

[12] Tento model spolu s úrovní detailu „moduly aplikace“ mimochodem můžeme často nalézt v různých propagačních příručkách jako jednu se základních informací o celém SW produktu.

	se výpůjček)
Čtenáři (popis třídy funkčností pracující se záznamy čtenářů)	Čtenáři (popis objektů aplikace s funkcemi pro funkčností pracující se záznamy čtenářů)
Knihy (popis třídy funkčností pracující se záznamy o knihách)	Knihy (popis objektů aplikace s funkcemi pro funkčností pracující se záznamy o knihách)
Zaměstnanci (popis třídy funkčností evidence zaměstnanců)	Zaměstnanci (popis objektů aplikace s funkcemi pro funkčností evidence zaměstnanců)

- Diagram nasazení – konkrétní instance aplikačních celků z předchozí úrovně detailu, zpravidla modelovaná na konkrétní síťové topologii (implementační typ modelu) Toto je jediná úroveň detailu kde se model sytému liší od implementačního. Zatímco model aplikace zde už pracuje s konkrétními softwarovými celky aplikovanými obvykle i na konkrétní hardware, model sytému zde svoji reprezentaci nemá, nebo se jedná spíše o modely, které do modelování informačních systémů nejsou příliš často zařazovány – například plány podlaží a rozmístění kanceláří či provozoven atp.
- Diagram modulů jednotlivé aplikace a vztahy mezi nimi. (technologický typ modelu)
- Diagram tříd (rozhraní atp.) [13] a jejich vztahy, ev. atributy. (technologický typ modelu)
- Modely konkrétní třídy (rozhraní) a vztahy mezi jeho metodami a atributy (obvykle se nepoužívá). (implementační typ modelu)

[13] Zde se nemusí jednat jen o koncepty OOP, ale třídou se může rozumět například i konkrétní DB modul atp.

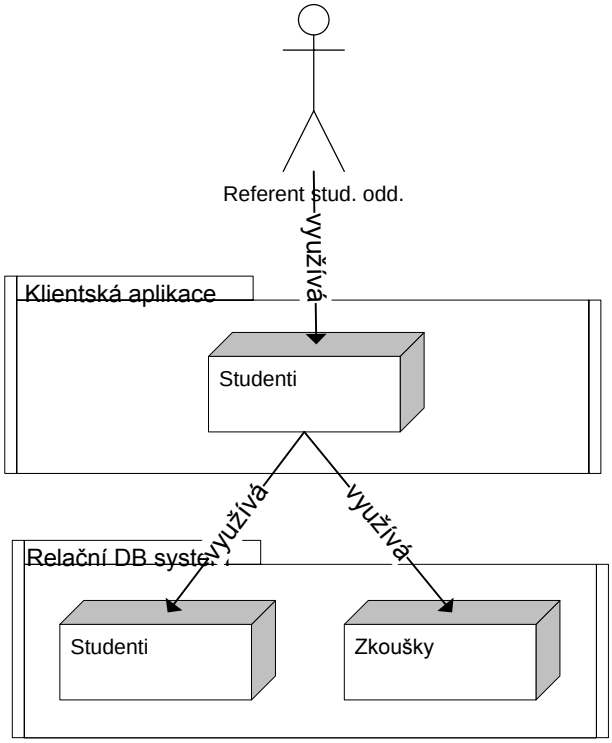
Bc. Lukáš Plachý (54219)
plachyl@std.fbm.vutbr.cz

Celý model není úplnou informací, která by měla vyplynout z této části metodiky. Model aplikačních modulů je nutné také zdokumentovat. Pokud se týká úrovně detailu schématu aplikace a jejích modulů je tento dokument spíše přehledový a svůj význam a uplatnění najde až u rozsáhlejších systémů nebo v případě modelů detailnější úrovně (diagramy tříd), ve kterých je naopak neopominutelný. Pro každý diagram by tedy měl existovat dokument, který popisuje:

- Význam jednotlivých tříd (modulů) v rámci aplikace (zejména s ohledem na jejich pozici v procesu, který má být aplikací podporován).
- Význam a přístupnost jejich atributů a jejich význam v rámci třídy (modulu) a v rámci celé aplikace a jejich ev. závislost na jiných třídách (modulech).
- Význam, přístupnost, vstupní a výstupní hodnoty a fungování jednotlivých metod daných tříd (funkcí modulů).

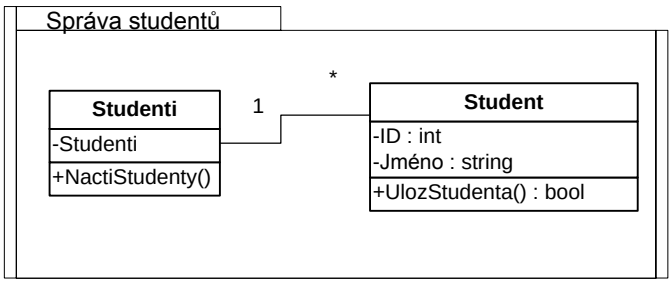
Pro tyto modely lze použít libovolné schéma umožňující zakreslit objekty a jejich relace, například class diagram UML.

U těchto modelů se na chvíli pozastavíme, pokud se týká příkladů. První úrovní detailu je diagram modulů a principu aplikace. Pro náš příklad může vypadat jeho výsek například takto:



Obrázek 27: Příklad diagramu struktury

Pokud se zaměříme například na objekt Klientská aplikace::Studenti můžeme jej detailně rozkreslit například takto:



Obrázek 28: Detailnější pohled na strukturu konkrétního balíčku

Na obrázku můžeme vidět přiklad tříd z minulého příkladu procesu aplikace. Vystupuje zde opět třída student, která poskytuje potřebné atributy a nástroje (metody) k manipulaci s konkrétním studentem.

4.1.3 Model architektury aplikačních dat

Jednou z informací které model uvedený v kapitole 4.1.1 používá jako doplňkovou informaci jsou jednotlivá data (datové entity a jejich atributy). Jejich struktura je zachycena na jednom z mála modelů, který je k nalezení v drtivé většině projektů a tím je model struktury dat. Bohužel jeho použití se zpravidla omezuje pouze

na strukturu dat a pohledů na ně v databázi ev. s nějakým popisem triggerů či uložených procedur (v lepším případě) nebo jen na samotnou strukturu dat, tedy entit a jejich atributů a vazeb (v případě horším). Tento model by tedy mel obsahovat následující informace:

- Datové entity (tj. nejen konkrétní data ale veškeré pohledy na ně atp.) ev. datové věty v případě lineárních médií,
- jejich atributy,
- a vztahy.

Připomeňme na tomto místě ještě jednou, že tato zmíněná data nemusejí být jen data charakteru databázového, ale může se jednat o jakákoliv data obecně, například různých formátů souborů (ať již textových (znakových) nebo binárních), formátu a významu dat nacházejících se na datových linkách, paměťových rourách atp.

Specifikem datového modelu obvykle bývá, že má pouze jednu úroveň detailu, tj. úroveň plně vyčerpávající a zobrazující všechny relevantní informace, což obvykle bývá technicky realizováno buďto pomocí výhradně digitální formy modelu, nebo v případě papírové podoby pomocí velkoformátových „plachet“ nebo naopak mnohostránkových modelů s referenčními odkazy mezi jednotlivými stránkami[14] (tedy jedná se o implementační typ modelu). To ale nemusí být vždy pravda a není vždy úplně dobrý řešením. I datové modely lze „generalizovat“ (tj. učinit technologickým modelem). Tak například pro tabulky „motory“, „karosérie“, „podvozky“, „majitel“ lze definovat entity „vozidla“ a „majitelé“ které jsou ve vztahu „vlastní“ nebo „je vlastněno“ (pro vztahy „vozidla“-„majitelé“) a ev. vztah „patří dohromady“ (pro jednotlivé součásti vozidla mezi sebou). Tyto generalizační modely obvykle nezobrazují jednotlivé atributy, o to důležitější jsou ale vztahy mezi nimi. Tyto modely mají ovšem obvykle jediný účel – prezentaci problematiky osobám, které buďto stojí mimo vývoj

[14] Pro praktické použití ať už se týká transportu, hledání nutných informací nebo i pro každodenní rutinní činnost je lepší papírová forma vícestránkového bookletu s odkazy tam, kde vazby přecházejí mezi stránkami. Pro celkové analýzy a provádění návrhu se více hodí velkoformátové výtisky. Pozn. autora.

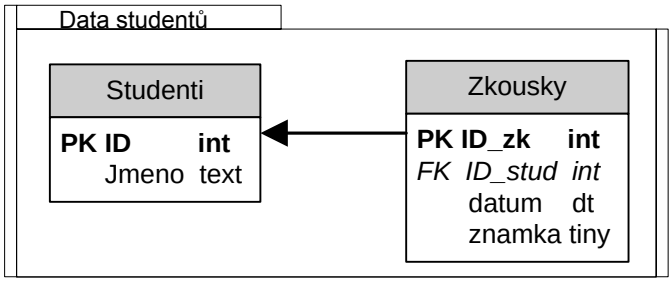
jako takový (manažeři, zákazníci) nebo teprve do vývoje pronikají (nováčci v týmu). Klíčovým modelem zůstává detailní model celého systému.

Tento model ale není úplnou informací. U datového modelu je více než u jakéhokoliv jiného modelu naprosto nezbytná dokumentace v podobě **popisu** tohoto modelu. Protože kdo si za dva roky úspěšného provozu systému vzpomene, co znamená položka „NAKL_50PRAC_VAR“ a jak (a že vůbec) souvisí s tabulkou „skladNAKLADY“ a co tato tabulka vlastně zaznamenává? A co více, který vývojář si na to vzpomene po půl roce práce na zcela jiných modulech? Proto musí existovat přesný popis jednotlivých datových položek, který obsahuje zejména:

- Jednotlivé celky dat a jejich význam (obvykle vázané na moduly aplikace)
- Jednotlivé datové entity (obvykle tabulky, pohledy, triggery, uložené procedury) a jejich význam. Zde obvykle bývá vhodné přidat i ukázku některých typických záznamů, pokud mají jednotlivé datové věty nějaké specifické významy podle různě vyplněných dat, zejména u tabulek s tzv. „řídkými daty“.
- Význam, datový typ, vazby a omezení jednotlivých atributů (a atributem zde může být například nejen položka XML souboru, položka hlavičky http, sloupec v databázi či CSV souboru, ale také vazba v relační databázi, omezení, identita atd.). Pokud má veličina nabývat nějakých konkrétních předdefinovaných hodnot se specifickým významem, který nemusí být na první pohled zřejmý (například oblíbené položky typu „integer“ které jsou ve skutečnosti polem bitových příznaků, kdy každý bit v položce má jiný význam).
- Pokud se týká moderních relačních databázových systémů je obvykle nutné u funkčních prvků (triggery, uložené procedury ev. pohledy) připojit informaci o očekávaných vstupech, vracených výstupech a informace o funkcionalitě ev, jejím významu.
- Nepovinnou položkou je pak u funkčních prvků (entit) závislost na jiných entitách (jiné funkční entity, tabulky atp.).

Naprosto jednoduchým a dokonalým, obvykle plně dostačujícím nástrojem na modelování těchto modelů je class relationship diagram, protože datový model (ačkoliv je v metodice používán a v této práci zmiňován odděleně) je pouze speciálním případem modelu tříd aplikace. Metodika jej používá jako oddělení protože data, architektura systému a funkcionalita jsou tři oddělené stránky aplikace (pohledy na ni) které je nutné brát v potaz nejen v souvislostech, ale i samostatně. Jak jsme navíc uvedli výše, lze použít také některý z ER diagramů, třebaže tento výraz není vhodný pro diagramy databází (nepopisuje „smysl“ vazby), avšak běžně se používá.

Mimoto existují i specializované SW, které se na tyto návrhy specializují, jak už bylo uvedeno, UML je zde používán jako jazyk vhodný k ilustraci metodiky, avšak metodika na něj není nijak vázána. V případě následujícího diagramu byla vybrána kombinace entit pro návrh struktury databáze programu Microsoft Visio v kombinaci se symbolem balíčku UML.



Obrázek 29: Diagram struktury aplikačních dat

Na obrázku vidíme data studentů (každý student má kromě svého studijního ID jen jméno) a jak jsou tyto údaje použity cizím klíčem ID_stud v údajích o zkouškách v relačním datovém modelu.

4.1.4 Model rozhraní

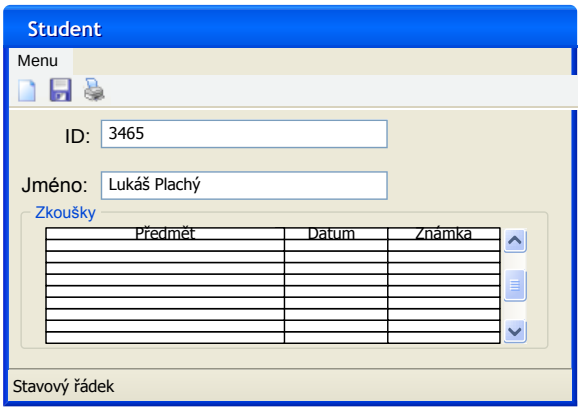
Jak již bylo uvedeno, v rámci jednotlivých funkcionalit může existovat celá řada vstupů a výstupů do a z aplikace. Tyto vstupy a výstupy je možné (mnohdy dokonce vhodné či přímo nutné) zobrazit v procesních diagramech. Ale i v případě kdy tomu tak není, musí existovat jejich detailní popis. Jak již bylo uvedeno v kapitole *2.4.5 Vzhled (uživatelského) rozhranní systému*, tato rozhraní mají generelně 2 typy: rozhraní uživatelská (pro interakci systému s člověkem) a rozhraní aplikační, kdy s námi navrhovaným a popisovaným systémem interaguje jiný takovýto systém (ať už se jedná

například o služby operačních systémů, služby API jiné aplikace, služby síťového charakteru atp.). Stejně tak se jedná o popis jakýchkoliv dalších výstupů.

V tomto okamžiku je nutné si uvědomit vazbu tohoto modelu na model *4.1.2 Model architektury (aplikace*). Představíme-li typické rozhraní, kterým je již zmíněné UI (user interface) je toto rozhraní (stejně jako třeba například rozhraní webové služby v rámci SOA) obvykle zařazeno do modelu architektury aplikace – tedy jedná se o nějaký další modul nebo třídu. V určitém kontextu je tedy model rozhraní jednou z úrovní detailu modelu architektury aplikace. Je proto (jak vyplyne v kapitole *4.3 Souvislosti modelů v kostce*) přímo na protější straně.

To ale neplatí vždy. Pro případ například tiskových sestav se jedná o nezávislý model.

V ilustraci můžeme vidět dialog uživatelského rozhraní, které jsme uváděli jako příklad v kapitole *4.1.1 Model procesů aplikace*. Jak jsme již uvedli, jedná se o velmi implementačně závislý model, zde tedy vidíme použitou abstrakci uživatelského rozhraní MS Windows:



Obrázek 30: Příklad návrhu uživatelského rozhraní - zde pro platformu Windows

Tento model je zpravidla grafického charakteru a nesleduje konkrétní metodiku modelování, spíše sleduje konkrétní platformu, na které je aplikace (a tím i rozhraní) implementováno (jedná se tedy o typ modelu implementačního). Přesto by měl zachycovat obecně tyto informace:

- Typy a rozvržení prvků rozhraní a to v úplném provedení, zejména ve vztahu k datovým entitám a jejich atributům navrženým v modelu.
- Jejich vazby na funkcionality procesního modelu aplikace, ev. události uživatele.

V dokumentaci, která provází tento model, by se pak mělo objevit:

- Popis jednotlivých funkcionalit, ev. speciálních druhů chování, které jsou zpravidla pod rozlišovací schopností modelů procesů.

4.1.5 Model use-case

Poslední z informací, které jsou uvedeny na procesním modelu, jsou události zahájení ev. ukončení nějaké akce. Tyto události jsou v přímé návaznosti na modely užití, zpravidla jsou s nimi 1:1 ev. je rozšiřují.

Tento model je důležitým nejen pro pochopení vazeb mezi uživatelskými rolemi ale také v souvislosti s návrhem a nastavením například uživatelských oprávnění.

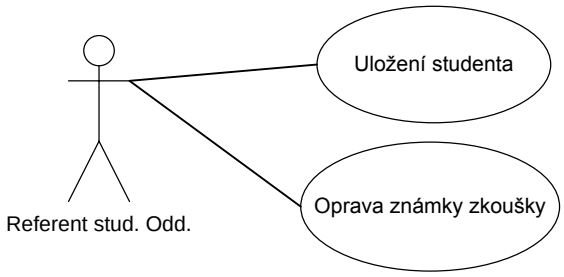
Pro jeho znázornění je potřeba běžný diagram use-case jaký poskytuje například modelovací jazyk UML.

V tomto modelu musejí být zachycené informace:

- Jaký uživatel (jaká role) vykonává
- jaké činnosti.

Doprovázející informací v dokumentaci může být, ale také nemusí, informace s popisem jednotlivých činností. Jedná se zcela rozhodně o model technologický, bylo by možná k diskuzi, zda se nejedná dokonce o model reality.

V našich příkladech pracujeme s hypotetickou evidencí studentů a jinak tomu nebude ani v případě této ukázky – zde se zobrazují objekty, které poskytují informace o tom kdo a co dělá.



Obrázek 31: Výsek z diagramu případů užití (Use-case) pro zde popisovaný hypotetický příklad

Jak lze vidět v kapitolách 4.1.2 až 4.1.5 jsou potřeba informace o struktuře dat, uživatelských funkcích a událostech, o rozhraních, architektuře aplikace a vlastních procesech které nad nimi probíhají. Odkud ale tyto údaje získat? Veškeré podklady jsou

skryty v základním podkladu, který tato metodika „Kostka“ očekává jako vstup. Jsou to informace, které zpravidla poskytují různé metodiky procesních analýz. Jak doporučuje metodika „Kostka“ provést modelování procesů tak, aby to posloužilo našemu výsledku, to je vysvětleno v následující kapitole.

4.2 Vstupní model

Nyní již známe veškeré modely, které jsou potřebné pro poskytnutí informací nutných k vývoji celé aplikace a jejich obsah, který mají poskytovat. Tyto výše uvedené modely ovšem spadají – jak ukazují kapitoly 4.1.1 až 4.1.5 – do kategorie modelů technologických a implementačních (podle jejich typu a úrovně detailu). Jak ovšem víme z kapitoly *2.2 Pohledy na modelování systémů* existuje třetí úroveň modelu – a to je model reality, který je vlastně úplně tím prvním a vstupním modelem. A protože nyní již víme, jaké výstupy má „Kostka“ obsahovat můžeme přistoupit ke specifikaci, co má být jejím vstupem.

4.2.1 Jaké informace musí zachycovat

Rekapitulujme ještě jednou požadované výstupní informace a zaměřme se na vstupy, které nám je mohou poskytnout. Při tom si musíme uvědomit, že ne všechny výstupní informace, které jsme uvažovali ve výše popsaných kapitolách budeme mít automaticky k dispozici jako vstup. Zde právě přicházejí ke slovu obvykle nealgoritmizovatelné metody kreativního charakteru, které na základě reálných procesů mohou vytvořit model aplikace. Neboli jinými slovy **teprve zde** se jedná o proces, který je tak známý všem programátorům i analytikům, tedy proces tvorby aplikace.

Požadovaný VÝSTUP	**Co je potřeba provést pro TRANSFORMACI vstupu na výstup**	**Potřebný VSTUP**
Use-case (**kdo** dělá **co**)	Zjistit uživatelské role a jejich odpovídající strukturu.	**Kdo** co dělá v procesu
Struktura aplikace (**co** se skládá z **čeho**)	Zjistit funkce patřící dohromady ke konkrétnímu procesu a uživatelské roli.	
Struktura dat (**data** a **atributy**)	Provést technologický a implementační návrh datových struktur na základě zpracovávaných informací a jejich atributů.	**Informace** zpracovávané v procesu systému
Definice rozhraní (jaká **data** jak **vstupují** / vystupují - v rámci struktury aplikace)	Provést návrh rozhraní (uživatelských nebo programových) na základě zpracovávaných informací a struktury aplikace a…	
	…na základě požadovaných funkčností.	**Jak** (a čím) – tj. proces – jsou informace zpracovávány (kdo **co dělá**)
Proces aplikace (co jak **funguje**)	Návrh jednotlivých funkcí (na takové úrovni aby byly tyto kroky implementovatelné) na základě požadovaných funkčností,	
	a které uživatelské vstupy je vyvolávají …	Use-case (**kdo** dělá **co**)
	… na kterých součástech aplikace…	Struktura aplikace
	… a s kterými daty.	Struktura dat

Kdo co dělá je v obou dvou

Tabulka 3: Předpis transformací vstupních informací na výstupní

Zaměříme-li se na pravou stranu tabulky (vynechejme nyní šedě znázorněné položky, jejichž přítomnost na obou stranách tabulky odůvodníme níže) objevíme tyto klíčové informace které požadujeme:

- **Kdo** pracuje
- **Jak** zpracovává
- **Co** zpracovává

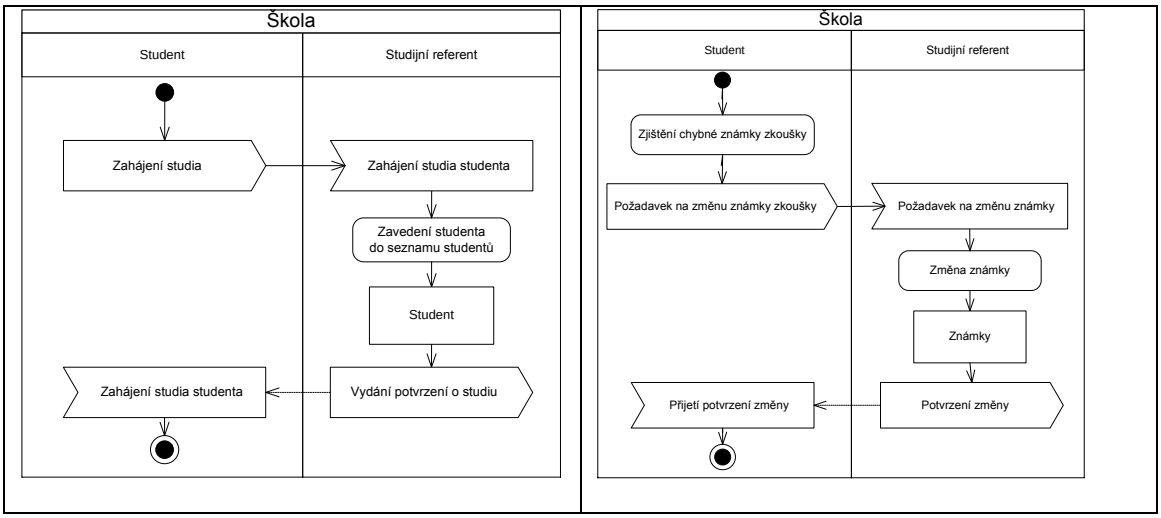
Zde se dostáváme již do poměrně známých vod procesního inženýrství a je také dlužno podotknout (v souvislosti s úvodem práce) že tyto otázky (si) již umí klást většina osob zúčastněných v procesu vývoje aplikací. Co ale obvykle chybí, jsou jednak právě informace (a modely) na levé straně (VÝSTUPY analýzy a návrhu aplikace) a jednak jejich souvislosti uvedené ve sloupci TRANSFORMACE.

Jakým způsobem ale zachytit informace KDO, JAK a CO? Ačkoliv z kapitoly *3.1 Modelovací metodiky* bychom mohli najít mnohé výborné metody či metodiky

(například BPMN či ARIS), nejlepší (a za účelem dodržení jednotné konvence této práce nejpochopitelnější) volbou je opět activity diagram UML.

Tento diagram obsahuje informaci o událostech, které vstupují a vystupují do systému, procesech složených z jednotlivých funkcí a úkonu v procesu a ty jsou řazeny podle rolí jednotlivých účastníků.

V našem hypotetickém příkladě (studijního oddělení pracující se studenty) tedy byly na vstupu pravděpodobně takovéto informace:



Vidíme, že se jedná o běžný activity diagram (diagram činností) zde ale nepopisující chování aplikace, ale chování osob v obchodním procesu. Tento diagram poměrně vhodně nahrazuje například i metodiku modelování BPMN (a to včetně její notace).

4.2.2 Jakou metodu / metodiku použít?

Uvedli jsme transformace, které musejí prodělat jednotlivé informace, aby se z nich stal kompletní softwarový produkt. Podívejme se tedy na tyto transformace blíže:

- **Zjistit uživatelské role a jejich odpovídající strukturu.** Dříve než začneme probírat tento bod, uvědomme si jeden důležitý fakt – nyní již máme práci s (budoucím) uživatelem za sebou. Tato práce se nezabývá sběrem požadavků – tedy konzultacemi s uživateli a jejich převodem do formalizované formy jako takovým. Jde nám o jasnou definici cíle, kterého má být dosaženo, ne o konkrétní metody (viz. *2.1 Co je to metodika?*)

Prvním krokem tedy zcela nezbytně musí být identifikace toho, s kým máme co do činění. Touto informací jsou v informačním systému role. Nelze totiž specifikovat konkrétní posty nebo dokonce osoby, neboť tím se připravíme o možná řešení a vůbec o schopnost zpracovávat situace kdy například osoba na nějakém postu zaskakuje za někoho zcela jiného, a tedy se kompletně mění sada jejích pravomocí a obecně procesů kterých se účastní. Pokud se aplikace, kterou implementujeme, týká konkrétní organizační struktury (nebo jejího výseku) je vhodné strukturovat role do skupin a celků které této organizační struktuře odpovídají, avšak nejjemnějším dělením celého stromu musejí být právě role. A ty jsou také jediným prvkem, který by měl pokračovat do dalšího zpracování.

- **Zjistit funkce patřící dohromady ke konkrétnímu procesu a uživatelské roli.** Toto je jedna z nejošidnějších fází. Znamená, že musíme být schopni definovat sadu (třídu) takových funkcí, která pracuje s konkrétními informacemi (daty) a nad nimi plní konkrétní úkoly. Zde je tedy vhodné spojovat do jednotlivých tříd ty funkcionality, které pracují:
  - se stejnými informacemi (daty)
  - plní stejné či podobné funkce
  - plní úkol pro stejnou roli

  Tím vzniká diagram tříd – tedy struktura systému, nebo (jak je uváděno v kapitola *4.1.2 Model architektury (aplikace)*) aplikace.
- **Provést technologický a implementační návrh datových struktur na základě zpracovávaných informací a jejich atributů**.

  Zde se zabýváme výhradně strukturou datových informací a jejich souvislostmi. Samostatnost tohoto pohledu je možná trochu diskutabilní uvážíme-li, že datové entity jsou pouze jinou entitou tříd aplikace z OOP, stejně tak jejich atributy a vazby. Vyčlenění pohledu na data jako samostatného modelu má všem zcela specifický a neopominutelný účel. Na rozdíl od modelu aplikace jako takové nejsou data nositeli funkcí, nejsou původci děje a jejich „funkčnost“ tkví pouze v jejich „smyslu“.

Zde se sice lze dostat do konfliktu s dnešním pojetím relačních databázových systémů, které na sobě umožňují implementovat i aplikační logiku procedurálních jazyků (například jazyk T-SQL). Jakkoliv z hlediska korektnosti a „čistoty“ přístupu by datové dotazovací jazyky měly zůstat opravdu pouze čistě dotazovacími a neměly by zavádět funkčnosti (jen o několik řádků výše jsme se přece uvedli, že data nejsou původci děje ani nositeli funkčnosti – z gramatického hlediska tedy jsou vždy předmětem a nikdy ne podmětem ve větě) je jejich přínos do vývoje aplikací natolik význačný, že pohled na data (zejména ta která jsou skladována v relačních databázových systémech) jako na zcela specifický prvek systému se ukazuje zcela oprávněný.

Postup modelování architektury dat vychází ze struktury problémových domén, kterými se systém zabývá (nebo modulů systéme – podle toho jak tyto „nadtřídy“ chápeme, viz.: *4.1.2 Model architektury (aplikace)*). Tyto třídy ovšem nyní zpracováváme pouze z hlediska informací a jejich atributů, které jsou v rámci těchto domén zpracovávány. Významné jsou i jejich vazby, tedy takové případy kdy součástí (atributem) jedné informace je celá jiná doména (entita) (například v případě knihovny jsou součástí (atributem) výpůjčky informace z domény čtenářů).

Při zpracování modelu struktury dat ovšem nezapomínáme na ta data, která se účastní technologické výměny informací v rámci softwarové aplikace. Metodika kostka nepředpokládá, že tato data jsou specifikována při prvním průchodu, ale při některé z dalších iterací.

- **Provést návrh rozhraní (uživatelských nebo programových) na základě zpracovávaných informací a struktury aplikace a na základě požadovaných funkčností.**

  Opět se dostáváme do kolize s modelem aplikační architektury nebo architektury systému. Musíme si ale uvědomit, že rozhraní jsou implementována často velmi specifickým způsobem. V tomto kroku se totiž zabýváme nejen předpokládanou interakcí uživatele a systému, ale systému se systémy jinými. Opouštíme tedy úroveň obecně systémovou (technologickou) a vstupujeme do úrovně ryze implementační.

Jak jsme ale uvedli již výše, je tento pohled součástí modelu aplikační struktury, avšak byl vyčleněn právě pro svou specifičnost. Zatímco při návrhu tříd aplikace je úroveň detailu, který provedeme více méně závislá na analytikovi / architektovi který ji provádí, specifikace rozhraní je informace která je pro naši aplikaci naprosto nutná, a tudíž musí být provedena, a právě proto byla vyčleněna jako specifický pohled na aplikaci, tedy jako specifický model.

Zde se totiž musíme zabývat jednotlivými funkcemi a jejich návazností na další prvky mimo systém. Princip návrhu spočívá v tom, že jsou analyzovány veškeré funkčnosti z use-case (pro rozhraní GUI) a veškeré funkce mající výstup / vyžadující vstup mimo systém (pro rozhraní programové) a tyto jsou zakresleny jako schéma. V případě vstupně/výstupních informací programového charakteru nezapomínáme ani na jejich datovou reprezentaci v modelu dat, kam strukturu dat která daným rozhraním procházejí, doplňujeme. Pokud se týká rozhraní GUI, jeho datové struktury obvykle vycházejí již z existujících modelů dat, neboť obvykle jsou určeny pro konkrétní funkcionality z konkrétní domény problémů našeho systému (modulu aplikace).

- **Návrh jednotlivých funkcí (na takové úrovni aby byly tyto kroky implementovatelné) na základě požadovaných funkčností, a které uživatelské vstupy je vyvolávají … na kterých součástech aplikace a s kterými daty.** Toto je poslední krok, kterým spojujeme funkčnosti, moduly aplikace a informace o zpracovávaných datech do celých procesů, které již ale probíhají uvnitř aplikace jako takové. Zjednodušeně řečeno, specifikujeme „jak to bude fungovat“.

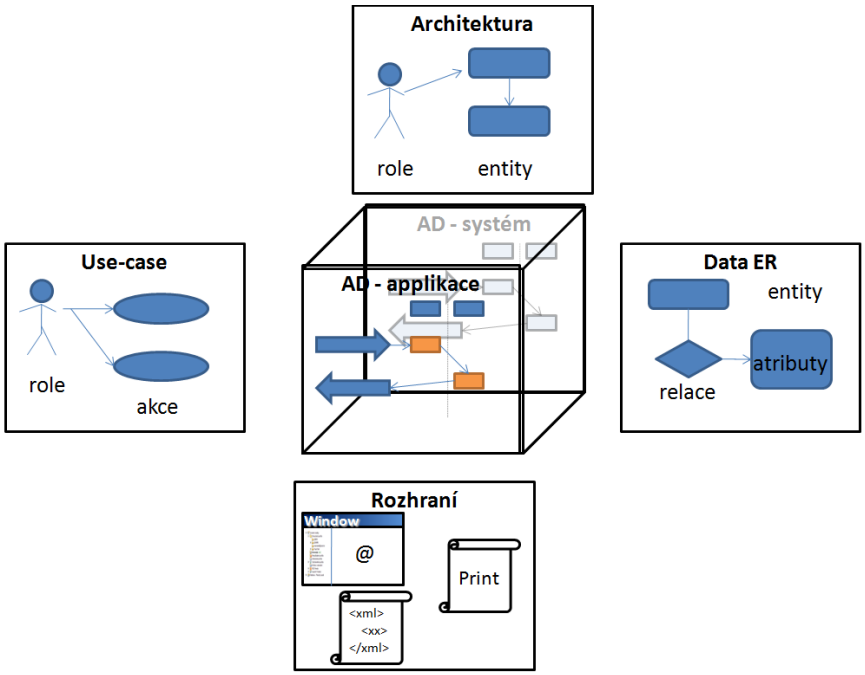
  Postup tvorby je takový, že na základě jednotlivých funkčností (z use-case) vytváříme procesy takové, aby se jich účastnily ty funkce modulů (tříd – dle úrovně detailu) které s nimi souvisí a které pro ně byly navrženy. Při tom obvykle pracují s daty, která do dané domény (modulu) spadají.

4.3 Souvislosti modelů v kostce

Jak souvisejí jednotlivé modely spolu se sebou? Pozorný čtenář již vysledoval, že každý model obsahuje některé z informace převzaté (nebo transformované) z ostatních modelů, zvláště patrné je to v tabulce 3. Sdílené informace vypadají takto:

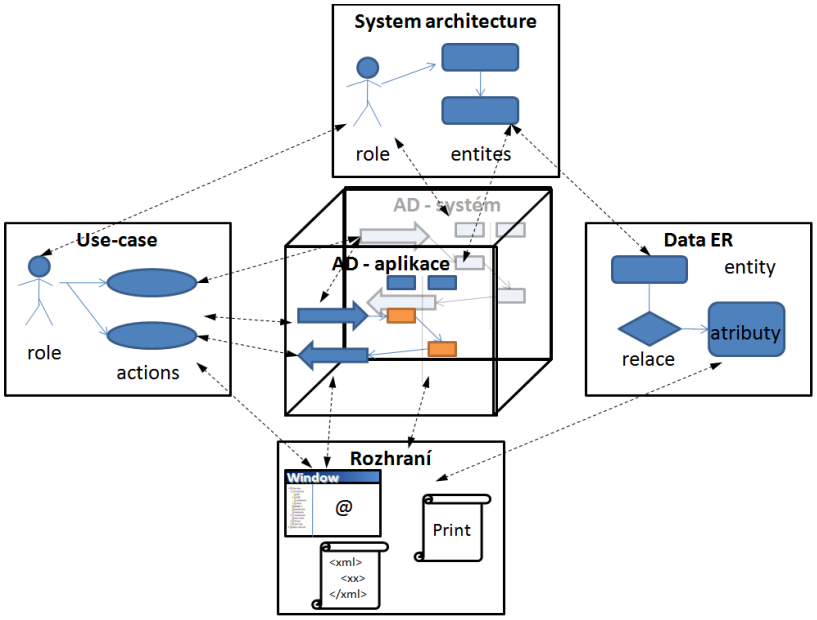
- Procesní diagram celého systému dle reality
  {AKCE – ROLE kdo co dělá **– TOK DAT}**
- Use-case diagram
  {AKCE – ROLE kdo co používá**}**
- Vstupy výstupy
  {forma vstupů / výstupů – dle struktury dat, AKCE (funkce)**}**
- Architektura systému
  {ENTITY SYSTÉMU – JEJICH VZTAHY - ROLE}
- Architektura dat (obecně ER diagram / ER diagram DB)
  {ENTITY DAT – JEJICH VZTAHY}
- Procesní diagram celého systému
  {ENTITY SYSTÉMU – AKCE – FUNKCE – TOK DAT (– entity dat)**}**

Jak pozorný čtenář bezpochyby rovněž zaregistroval, klíčovým pojmem systému je pojem „model“. Neboli veškeré informace jsou reprezentovány pomocí modelů, kterých je právě 6. Souvislosti těchto modelů lze graficky reprezentovat jako jednotlivé strany krychle. Lze tedy schematicky znázornit takto:



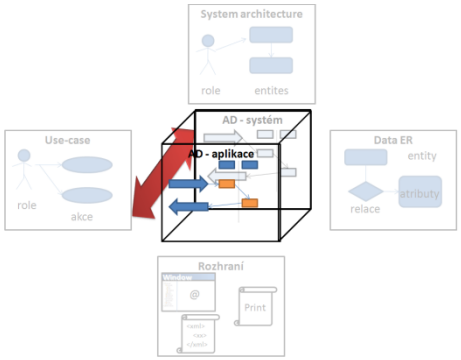
Obrázek 32: Kostka metodiky „Kostka“ - boční, horní a spodní strany byly pro lepší přehlednost znázorněny mimo perspektivu

Jejich souvislosti (zmíněné v kapitole *4.1 Modely* a *4.2 Vstupní model*) pak představují ty objekty které jsou modelům společné a které spolu sdílejí takříkajíc „přes hrany“ krychle.

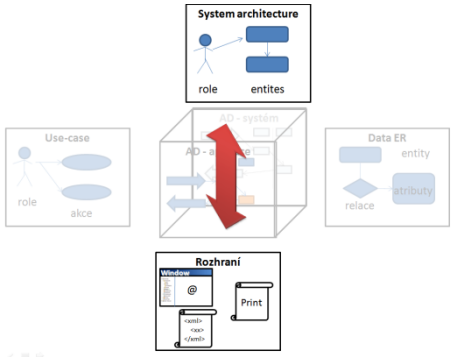


Obrázek 33: Kostka metodiky „Kostka“ a objekty které se prolínají mezi modely

Jsou zde navíc dvě vazby, které pro přehlednost nebyly znázorněny – ale již byly zmíněny. Jsou to vazby mezi dvěma páry hlavních modelů systému, a to jsou oba aktivity diagramy a architektura systému vůči rozhraním.



Obrázek 34: Souvislost obou procesních modelů



Obrázek 35: Souvislost modelu struktury aplikace a modelu rozhraní

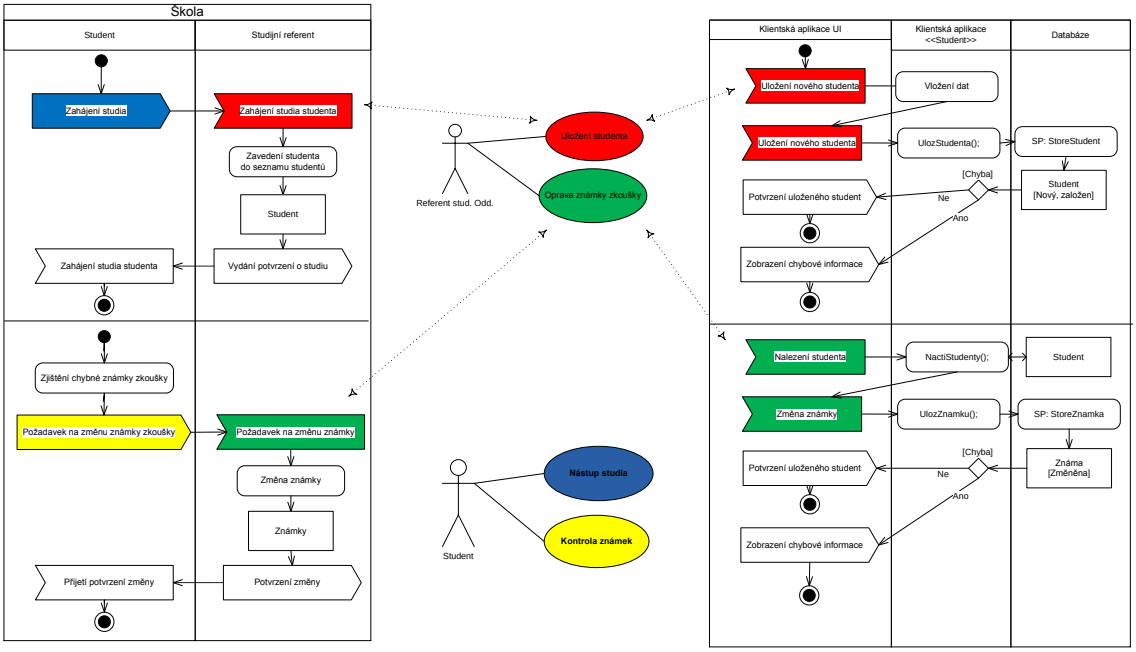
V prvním případě se jedná o takřka totožné diagramy, avšak v prvním případě se jedná o model procesu systému, podruhé o model procesu aplikace, avšak jejich procesy jsou takřka totožné, pouze v prvním případě se jedná o procesy reálného světa, v druhém o procesy probíhající v systému softwarové aplikace. Jejich vzájemná transformace není triviální záležitostí, ale je předmětem celé metodiky – této. Díváme se totiž – jak už je asi zjevné – na její vstup a výstup.

Druhý případ souvislostí již byl zmíněn. Jedná se o souvislost mezi systémovou architekturou a rozhraními systému. Již jsme řekli, že systémová architektura softwarové aplikace obsahuje jako jednu z úrovně detailu i návrh rozhraní, avšak my jsme tuto úroveň detailu zde vyčlenili do speciálního pohledu, protože zatímco úroveň detailu modelování je do jisté míry závislá na potřebách aktuálního pohledu a rozhodnutí architekta, model rozhraní by měl být v systému přítomen vždy. Navíc

uživatelské rozhraní nemusí být nutně vždy zařaditelné jako konkrétní úroveň detailu a jeho pozice může být nejasná.

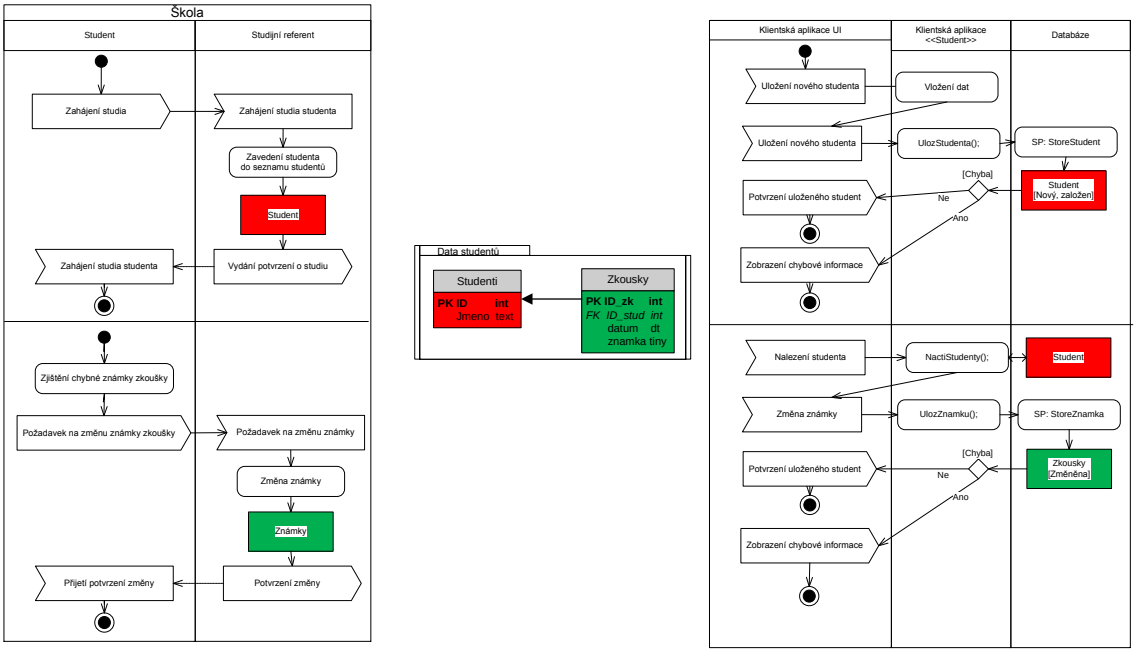
Poslední fakt, který si musíme uvědomit je fakt, že modely, které jsou zde reprezentovány, nejsou kompletní informací, ale vyžadují doplňkovou dokumentaci, které detailně popisuje ty informace, které není možné zachytit v modelu (ať už z důvodu přehlednosti nebo neefektivnosti takového postupu).

Když se tedy zaměříme na předchozí příklady, uvidíme následující souvislosti:



Obrázek 36: Souvislost obou procesních modelů s přechodem přes jednu z bočních stěn - diagram use-case. Ty objekty které jsou totožné (pouze jinak formálně zaznamenány) jsou barevně odlišeny.

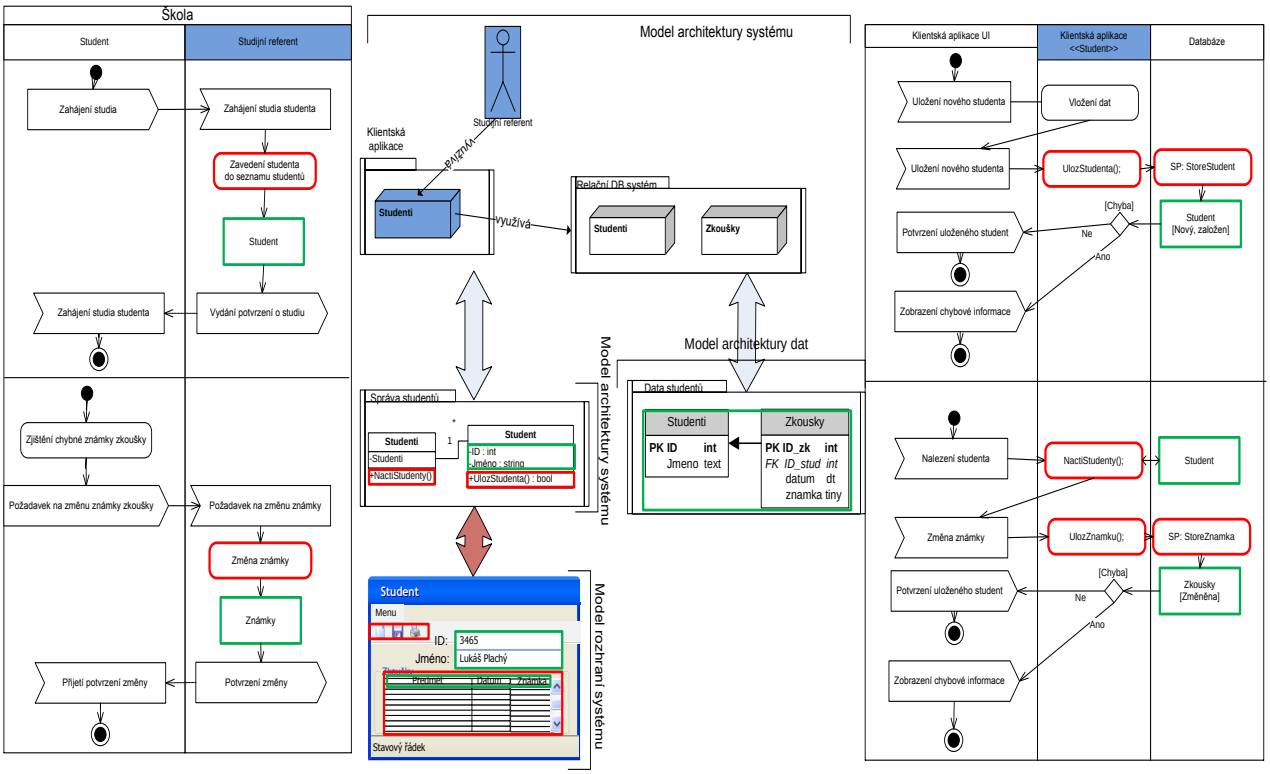
Zde je znázorněna souvislost mezi událostmi obchodního procesu, use-case diagramem a jeho užitím v diagramu procesů aplikace. Vidíme stejné události (funkcionality – tj. případy užití) které přecházejí z diagramu obchodního procesu do procesu aplikace. Je to jedna z vazeb.



Obrázek 37: Souvislost obou procesních modelů s přechodem přes druhou z bočních stěn - diagram modelu dat. Ty objekty, které jsou totožné (pouze jinak formálně zaznamenány) jsou barevně odlišeny.

Zde se nachází druhá vazba napříč diagramy – jedná se o data, zatímco v levé polovině se jedná o informace, které byly identifikovány jako odborný termíny, informace v obchodním procesu. V návrhu dat jsou tyto informace již zobrazeny a zpracovány jako formální struktura dat a objekty této struktury jsou použity v pravé části diagramu v procesu vlastní aplikace.

Další vazbou je vazba procesů na strukturu systému:



Obrázek 38: Znázornění souvislosti obou procesních modelů s přechodem přes modely na horní i spodní straně krychle. To jak spolu tyto modely souvisejí (viz. obrázek 33) je i zde jasně viditelné (vertikální plné šipky)

Zde je možné sledovat, jak se jednotlivé funkcionality přenášejí do funkcí (červené ohraničení) a jak se postupně přenášejí (a také mění) informace na data a datové atributy (zelené ohraničení) a to včetně jejich promítnutí do modelu rozhraní systému. Červená šipka - - odpovídá červené šipce v obrázku (Obrázek 35) v této kapitole.

To jsou tedy souvislosti těchto modelů – pokud budou promítnuty do kompletního modelu ve formě krychle, vyvstanou veškeré souvislosti mezi všemi modely.

4.4 Postup tvorby a proces návrhu

Jak tedy je nutno postupovat v případě kdy chceme provést kroky nutné k dosažení korektního návrhu aplikace?

Bc. Lukáš Plachý (54219)
plachyl@std.fbm.vutbr.cz
20. 05. 2008

Nejprve tedy musíme znovu shrnout vstupní informace. Pro každou specifikovanou funkci (požadavek) na vstupu musíme vědět:

- **Kdo** pracuje
- **Jak** zpracovává
- **Co** zpracovává

Tyto informace tedy potřebujeme mít specifikovaný pro každou funkcionalitu. Protože obvykle jsou tyto poznatky získávány od konkrétních osob, které spíše řeknou co dělají (a díky tomu jsme schopni i říci v jaké roli), ovšem informace o tom nad jakými přesně informacemi se to děje se z nich páčí již poněkud hůře skládá se toto zjišťování ze dvou samostatných kroků. Třetím krokem je pak zjišťování kde jednotlivé pozice na sebe v procesu navazují, je potřeba tady od každé role znát pro její činnosti i její vstupy a výstupy.

Jakkoliv tato metodika „Kostka“ nemá za cíl specifikovat postupy transformace požadavků uživatelů či specifikace jejich práce na analytické výstupy, její první 3 kroky se tímto zabývají – v principu vzato se jedná o ověření existence vstupních informací modelu.

1. Kdo co dělá
2. S jakými daty pracuje
3. Jejich vstupy a výstupy (vůči danému uživateli)

Máme-li tyto informace (obvykle jako podklad modelovaní use-case – ovšem písemně) můžeme přistoupit ke čtvrtému kroku. Tím je tvorba modelu use-cases, což jsou pro nás klíčové informace ohledně funkčností které má aplikace poskytovat.

4. Use-case model (dokumentací modelu jsou přímo poznámky z 1-3)

Use-case model nám tedy poskytl první pomocnou ruku pro sestavení modelu, který tyto funkcionality reprezentuje, tedy model reálného procesu. V něm už bychom měli zohlednit jednotlivé role a jejich činnosti, které nám v activity modelu reality reprezentují události, zatímco role jsou jednotlivými swim lanes modelu.

5. Activity model reality (dokumentací modelu jsou přímo poznámky z 1-3)

Nyní přichází první iterace. Samotný model reality totiž obvykle nemá dostatek informací a obvykle se zjistí, na co vše se při jeho modelování zapomnělo.

6. Iterovat 1-5 dokud není model korektní a funkční

Bc. Lukáš Plachý (54219)
plachyl@std.fbm.vutbr.cz
20. 05. 2008

V tomto okamžiku je již možné přistoupit k vlastnímu návrhu. Ten se skládá z kroků, tak jak jsou popsány v tabulce 3 a jsou to tedy tyto:

7. Struktura aplikace
8. Struktura dat
9. Struktura rozhraní

Tím jsme teoreticky dokončili rozvržení statických entit a aplikaci by bylo možné považovat za navrženou. To ale rozhodně není všechno, následujícím krokem je další kontrolní iterace, která má zajistit, aby návrh aplikace sledoval to rozvržení, které jsme rozlišili v modelu procesu reality. To znamená aby se například v diagramu struktury aplikace vyskytovaly právě ty role které jsme definovali v use-case diagramu a později použili jako swim lanes v aktivity diagramu modelu reality, aby se v návrhu aplikace vyskytovaly jen ty entity, které jsou v modelu reality jednotlivými skupinami případů užití atp. Stejně tak i nyní můžeme dojít k závěru, že nemáme dostatek informací nebo je něco nejasné a pak se musíme vrátit podruhé až na začátek a informací doplnit.

10. Iterovat 7-9 dokud nebude v souladu s 1-5, ev. opakovat 1-9 pokud není něco v pořádku, jasné, nebo něco chybí.

Nyní když již máme dobrou představu o tom jak má celý proces fungovat a jak má fungovat vlastní aplikace, musíme tuto funkčnost nadefinovat. To provedeme tvorbou posledního, nejnáročnějšího diagramu, kterým je activity diagram funkčních procesů vlastní softwarové aplikace.

11. Proces aplikace tak aby zajistil požadované funkčnosti navržených rozhraní nad požadovanými daty.

Při tvorbě tohoto diagramu se ovšem zcela zákonitě dostaneme do problémů – opět buďto s chybami v návrhu aplikace (nejtypičtější je stav kdy například předpokládáme, že jedna aplikace bude volat druhou a druhá bude mít od ní k dispozici například nějaký datový atribut, ale ten my v datech volání vůbec nemáme) nebo opět nejasnostmi v zadání. Proto přicházejí dvě poslední kontrolní iterace, po jejichž úspěšném absolvování se již dostaneme k výsledku naší práce.

12. Opakovat 7-11 pokud proces selže, nebo nevykazuje integritu
13. Opakovat 1-11 pokud není možné integritu udržet bez doplnění procesu reality.

Co je tedy přesně výsledkem naší práce? Vzpomeňme si na kapitolu *2.4 Co musí mít programátor k dispozici* a uvědomme si, že vše co jsme nyní navrhnuli, představuje právě informace potřebné proto, aby se dalo pokračovat dále. Dalším krokem je již implementace, které už se ale, jak vyplývá z úvodu této práce, nachází momentálně mimo náš rozsah.

4.4.1 Obecné poznámky

K výše uvedenému je třeba poznamenat, že tyto kroky se snaží odstranit zejména některé z necností některých metodik. Jako příklad uveďme snahu o striktní oddělování návrhů v metodice UP (ev. RUP) kdy se lze dočíst (například v (21) s. 129) doporučení, že v případě psaní systémů užití se „soustřeďte na co, nikoli na jak“. To je bezesporu dobré poučení pro zabránění kombinování modelů (ačkoliv prvky z jednoho modelu, jak v metodice ukazujeme, přecházejí do jiného, neznamená to, že bychom měli modely libovolně prolínat!) rozhodně to ale přispívá ke ztrátě informace. Pokud si totiž náš klient (nebo obecně jakýkoliv budoucí uživatel) například přeje v nějakém okamžiku procesu kliknout na tlačítko „OK“ musíme tuto skutečnost zaznamenat. Stejně tak ji zaznamenáme, pokud jsme tuto možnost s klientem byť i jen diskutovali.

Jak jsme totiž uvedli, přimodelování by architekt měl mít na paměti souvislosti, a tedy být schopen v případě potřeby i odstranit nebo zavčas vhodně upravit či nasměrovat ty požadavky, které by mohly způsobit zbytečné komplikace (například na technologické úrovni při vlastní implementaci) nebo neefektivní plýtvání, ačkoliv se celá věc dá řešit zcela jinak a lépe.

Proto kostka zavádí iterace již přímo do návrhu, a ačkoliv se tím nevylučuje iterace celého cyklu projektu nebo IS a tedy i změny, sám cyklus přímo v návrhu architektury umožňuje toto množství změn minimalizovat a šetřit tak ekonomické prostředky.

4.5 Řízení návrhu architektury

Nyní již víme, jaké kroky přesně musíme provést, co musejí jednotlivé modely a jejich dokumentace obsahovat, je však potřeba několik poznámek k řízení tohoto postupu jako takového. Vraťme se ještě jednou k naší tabulce, která celou metodiku lokalizuje v postupu prací na informačním systému:

		Pracovníci	Org. postupy	Data	SW a HW	Org. vlivy	Ekon. otázky	Produkty a potřebné dokum.	Metody, techniky, nástroje	Způsob řízení
Analýza(+studie proveditelnosti)	**Cíle**									
	Testování klíč. techn.									
	Mapování požadavků									
	Mapování procesů									
Návrh	**Návrh procesů**									
	Návrh dat. struktur									
	Návrh I/O									
	Návrh. apl. arch.									
Implementace	**Návrh apl. procesů**									
	Coding									
Testování	**Návrh test-cases**									
	Provedení testů									
Nasazení	**Pilot**									
	Nasazení									
Vyhodnocení	**Vyhodnocení**									

Význam barev:

Není popisovaná fáze životního cyklu	Jedná se o popis IS jako celku, netýká se popisu SW aplikace
Tato metodika již popsala	Tato metodika se jimi zabývá se pouze částečně nebo pouze doporučuje postupy.

Tabulka 4: Co ještě chybí popsat?

Jak tedy vyplývá z výše uvedené tabulky, zeleně označené části byly již popsány ve výše uvedených kapitolách, chybí pouze informace o provedení po stránce organizační. To předpokládá informace o:

- Kdo má být odpovědný za které kroky
- Jak mají být tyto kroky zasazeny do celku
- Jaká má být účast ostatních rolí

Bc. Lukáš Plachý (54219)
plachyl@std.fbm.vutbr.cz
20. 05. 2008

Výše uvedených bodů se ovšem v této práci dotkneme jen letmo. Důvodem je že se jednak jedná o záležitost, která je silně závislá na definici prostředí, ve kterém se celý projekt provádí a díky tomu nelze tyto postupy generalizovat, jak se mnohdy pokoušejí různé metodiky. Ty totiž například pro otázku „kdo má být odpovědný za…“ zavádějí nepřeberné množství rolí s tím, že dávají prostor pro jejich takřka libovolnou asociaci na konkrétních jedincích či pracovních postech. Abychom se vyhnuli takovéto generalizaci a současně nesklouzli do diktátu postupů, které by se v praxi mohly ukázat nerealizovatelné, doplníme odpovědi na tyto otázky spíše jako doplňující doporučení tak, jak se osvědčily autorovi této práce.

4.5.1 Kdo má vykonat které kroky? Součinnost ostatních rolí

Zrekapitulujeme-li tedy kroky metodiky a podívejme se na jednotlivé role, které v nich figurují. Jak již bylo uvedeno, jedná se spíše o doporučení. Aktivní roli by v tomto případě měl sehrát spíše vedoucí týmu, třebaže v případě rozsáhlejších projektů se jeho vlastí práce omezuje spíše na kontrolu a motivaci. Co je ale naprosto nevhodné je fakt aby se jednotlivé kroky dělily mezi různé osoby. Autor v tomto pojetí upřednostňuje spíše zodpovědnost „za modul“ než zodpovědnost „za činnost“. Zatímco u programátorů je to možné a závisí to na kvalitě zpracování právě jejich zadání (takže lze mít specialistu přes databáze, specialistu přes uživatelská rozhraní, nebo dokonce specialisty pro konkrétní problematiky podnikových procesů (HR, účetnictví, CRM….)) u analýzy je to holý nesmysly. Ačkoliv mnoho metodik se toto snaží doporučovat a vlastně se tak tváří že jsou natolik brilantní, že následující články v řetězci nemusejí mít „informace z první ruky“ a že plně dostačují jen informace z metodiky, není tomu tak. Právě metodiky (a to včetně této) zavádějí do systémů abstrakce, které sice mají sloužit právě převedení reálného toku informací do aplikačního toku dat, avšak pro porozumění nejsou vždy to nejlepší. První osoba, která bude závislá pouze na dokumentaci a bude se řídit výhradně jí a nebude v přímém kontaktu s uživatelem nebo alespoň jeho požadavky (a proto jsou uvedeny jako dokumentace k activity diagramu reality) začne do systému zanášet chyby. Nelze tvrdit, že takováto osoba je vyloženě na škodu. Může totiž vidět problémy, o kterých jiní účastníci ani netuší (tj. pro stromy nevidí les) a může být schopna pomoci.

Takovýmto požadavkem (maximální agregace jednotlivých kroků na jedné osobě) se ovšem dostáváme do problémů v případě rozsáhlejších projektů. Pokud by měly být všechny kroky vykonány jednou osobou, nemusel by být takový projekt zvládnutelný v rozumném čase. Zde se ale na pomoc dostává různá úroveň detailu a do popředí vystupuje právě role vedoucího celé činnosti. Celý proces, který jsme zde popsali, se totiž odehrává určitě několikrát, vždy s rostoucími detaily. Jak totiž analytik postupně proniká do detailů hierarchie společnosti a jednotlivých procesů tak se postupně neustále zpřesňují jeho informace a roste detail a také časová náročnost modelování. Také jsou už ale známější jednotlivé moduly a ty lze právě rozdělit mezi jednotlivé týmy, avšak právě vedoucí zde musí být za účelem synchronizace na úrovni těchto modulů (zejména pokud se týká předávaných dat).

Pokud se ale týká role jedné osoby, tedy analytika, ta se může postupně měnit. Což je mnohem důležitější, neboť jednotlivé role determinují určité chování analytika.

	Analytik			Programátor	Vedoucí
	Analýza požadavků	Návrh architektury	Návrh aplikace		
1. Kdo co dělá	■				
2. S jakými daty pracuje	■				
3. Jejich vstupy a výstupy (vůči danému uživateli)	■				
4. Use-case model (dokumentací modelu jsou přímo poznámky z 1-3)	■				
5. Activity model reality (dokumentací modelu jsou přímo poznámky z 1-3)	■				■
6. Iterovat 1-5 dokud není model korektní a funkční	■				
7. Struktura aplikace		■			■
8. Struktura dat		■		■	
9. Struktura rozhraní		■	■	■	■
10. Iterovat 7-9 dokud nebude v souladu s 1-5, ev. opakovat 1-9 pokud není něco v pořádku, jasné, nebo něco chybí.		■	■		
11. Proces aplikace tak aby zajistil požadované			■	■	

	Analytik			Programátor	Vedoucí
	Analýza požadavků	Návrh architektury	Návrh aplikace		
funkčnosti navržených rozhraní nad požadovanými daty.					
12. Opakovat 7-11 pokud proces selže, nebo nevykazuje integritu					
13. Opakovat 1-11 pokud není možné integritu udržet bez doplnění procesu reality.					

Význam barev:

Neúčastní se
Provádí
Aktivně se účastní
Může se účastnit / pomoci uvažovat.

Tabulka 5: Role a jejich aktivita v jednotlivých krocích metodiky

4.5.2 Jak mají být kroky zasazeny do celého projektu

Tato kapitola je poměrně jednoduchá a stručná – důvodem je fakt, že celá „Kostka“ je vlastně klíčovou činností (činností díky svému rozsahu a významu zcela jednoznačně na kritické cestě) která je potřeba vykonat aby se celý projekt z fáze analýzy přesunul do fáze implementace.

Zatímco uvnitř vlastního provedení je možné paralelizovat jednotlivé činnosti na úrovni modulů, tak jak je si je metodika sama rozdělí, co více jednotlivé moduly mohou být teoreticky předány do fáze implementace ještě před tím než skončí analýza a návrh ostatních (ačkoliv z toho vyplývá nebezpečí, že v případě změn v modulech kde již probíhá implementace bude nutno doposud udělanou práci zahodit).

Metodika nepopírá ani životní cyklus vodopádu, fontány či spirály, je plně schopna je reflektovat. Je nutné si navíc uvědomit, že činnosti zde popsané jsou činnosti, které **musejí** být (dříve či později) provedeny aby bylo možné vůbec implementovat (a to že je neprovedeme a s odkazem na agilní metodiky, a nebo „je zameteme pod koberec“, tj. donutíme nebohého programátora aby je provedl jaksi „mimo plán“ nám rozhodně nepomůže). Jak je možné navíc vidět, metodika sama v sobě implementuje kombinaci fontány (kroky 4, 6, 10 a 12) a spirály (krok 13). Ačkoliv tyto názvy se samozřejmě používají pro životní cyklus IS jako celku nebo pro životní cyklus softwarových aplikací či jejich projektů, je možné tyto cykly použít i na

Bc. Lukáš Plachý (54219)
plachyl@std.fbm.vutbr.cz

jedinou část – ona sama je takovým projektem (který je sub-projektem projektu většího).

Pokud se týká agilnosti metodiky, pokud se podíváme na záměry specifikované v kapitole *2.3 Agilní versus rigidní* dostáváme se bezesporu do sporu. Ale jak již bylo uvedeno, je sice bezesporu hezké že jsou upřednostňovány sociální interakce, umožněny změny atp. ale to bohužel neznamená, že kroky, které zde byly uvedeny, nebudou potřeba. Bohužel budou, a je jen otázka, jak moc formálně budou provedeny. Zkušenější vývojář bezesporu na menším problému může zvládnout tyto kroky takříkajíc „zpaměti. Jenže jak jsme uvedli, klíčové pojmy jsou „systematičnost“ a „důslednost“. A těch nás žádná agilní metodika nezbaví!

Bc. Lukáš Plachý (54219)
plachyl@std.fbm.vutbr.cz

5 Závěr

V práci jsme navrhnuli metodiku, která umožňuje systematické a důsledné přenesení reálného stavu obchodních procesů do modelu architektury software tak, aby byla zachována především integrita návrhu a nebyly zaneseny zbytečné chyby a nepřesnosti.

Shrnuli jsme potřeby programátora, aby mohl kvalitně a jednoduše implementovat navržené funkcionality.

Specifikem metodiky je její přínos v jasné specifikaci a znázornění souvislostí mezi informacemi, které tvoří model softwarové aplikace a které má pomoci návrháři pracovat s modelem „komplexně“ (jako například s Rubikovou kostkou) tak, aby vždy mohl mít na paměti jasně specifikované a strukturované veškeré aspekty návrhu.

Díky tomu se dostává nejen návrháři software ale i jeho vedoucím do ruky nástroj, pomocí kterého je možné specifikovat zcela přesně strukturu informací a kroky k jejich zpracování, které se v rámci jakéhokoliv projektu týkají návrhu aplikace a které umožní nezanedbat nic důležité a při tom představují minimální sadu informací jejich souvislostí nutných ke správnému popisu aplikace.

Jak už bylo uvedeno, ačkoliv se jedná o metodiku z hlediska informací, které specifikuje, avšak bez splnění požadavku úplnosti. To by také mělo být dalším cílem práce autora na tuto diplomovou práci navazující – pomocí stejného principu specifikovat jasně dané kroky vedoucí od začátku projektu informačního systému až po jeho ukončení. První z navazujících prací by se měla ještě zabývat pouze výsekem, kterým je implementace softwarové aplikace jako takové (tj. počínaje bodem 0 kdy jedinou informací je fakt, že některé (zatím neznámé) procesy budou pokryty softwarovou aplikací), až po její úspěšné zavedení do provozu, tedy včetně implementace a testování. Další práce by pak měla pokrýt kompletní životní cyklus informačního systému. To vše s důrazem na výše uvedené vazby a tedy systematické a důsledné udržené integrity projektu.

Je rovněž předpokládáno přepracování tohoto díla do populární formy, které zde bylo zpracováváno zejména s ohledem na akademickou hodnotu a omezení rozsahu spíše s důrazem na abstraktní teoretickou stránku věci, avšak pro popularizaci mezi semi-odbornou veřejností by měla být změněna forma prezentace.

Bc. Lukáš Plachý (54219)
plachyl@std.fbm.vutbr.cz

6 Citovaná literatura

1. **NĚMEČEK, Petr, Prof. ing. DrSc. a ZICH, Robert, Ing. Ph.D.** *Podnikový management.* Brno : FINAL TISK s.r.o. Olomoučany, 2004. 80-214-2780-9.
2. **Furnham, Adrian.** The Incompetent Manager.
3. **Helliwell, Charlie.** Incompetence in Management. *jobsite.* [Online] 14. Srpen 2007. [Citace: 2. Únor 2008.] http://blog.jobsite.co.uk/charlie/archives/2007/08/incompetence_in.html.
4. **Řepa, Václav, a další.** *Metodika vývoje informačního systému s pomocí nástroje Power Designer.* [PDF] Praha : Katedra informačních technologií VŠE, 2006.
5. **KADLEC, Václav.** *Agilní programování.* BRno : Computer press, 2004. 80-251-0342-0.
6. **kol.** Manifesto for Agile Software Development. *Manifesto for Agile Software Development.* [Online] 2001. http://agilemanifesto.org/.
7. **VLK, Tomáš.** Manifest agilního vývoje softwaru z osobní perspektivy. [Online] http://www.tril.cz/manif.html.
8. **SMRŽ, Michal.** Proč mám alergii na agile development. *Scream.* [Online] 28.. Říjen 2007. http://www.scream.cz/2007/10/28/proc-mam-alergii-na-agile-development/.
9. **PLACHÝ, Lukáš.** Struktura SW aplikace pro podporu IS MP Brno. . Brno : Statutární město Brno, Městská policie Brno, 2004. 1.
10. **Consortium, EC.** HealthPlus - Documentations. Brno : autor neznámý, 2006-2008.
11. **Meriam-Webster.** Merriam-Webster Dictionary. [Online] http://www.merriam-webster.com/dictionary/model.
12. Model (abstrakce). *Wikipedia - otevřená encyklopedie.* [Online] 27. Listopad 2007. http://cs.wikipedia.org/wiki/Model_(abstrakce).
13. **KOCH, Miloš, Doc. Ing. CSc.** *Datové a funkční modelování.* Brno : AKADEMICKÉ NAKLADATELSTVÍ CERM s.r.o. Brno, 2004. 80-214-2724-8.
14. **KIRSTEN, Wolfgang, a další.** *Caché - Databáze postrelačního typu a tvorba aplikací.* Brno : CP Books, a.s., 2005. 80-251-0491-5.
15. **COCKBURN, Alistair.** *Use cases - Jak efektivně modelovat aplikace.* Brno : CP Books, a.s., 205. 80-251-0721-3.

16. Peter Chen. *Wikipedia - free encykolpedia.* [Online] http://en.wikipedia.org/wiki/Peter_Chen.
17. **Essential Strategies, Inc.** Data modeling. [Online] Essential Strategies, Inc., 1999. http://www.essentialstrategies.com/publications/modeling/.
18. **KOSEK, Jiří.** *XML pro každého podrobný průvodce.* Praha : Grada publishing, spol. s. r. o., 2000. 80-7169-860-1.
19. **BENEŠ, Michal.** *Přehled OO notací a metodik.* [editor] Pavel DRBAL. Praha : Vysoká škola ekonomická, 2008.
20. **IDS Scheer AG.** ARIS - manuály, informační materiály, interní firemní prezentace. Saarbrücken : autor neznámý, 2003-2008.
21. **ARLOW, Jim a NEUSTADT, Ila.** UML2 a unifikovaný proces vývoje aplikace. [překl.] Bogdan Kiszka. Brno : Computer Press, a.s., 2007. 2. 978-80-251-1503.
22. Business Process Modeling Notation. *Wikipedia, the free encyclopedia.* [Online] 2007. http://en.wikipedia.org/wiki/Business_Process_Modeling_Notation.
23. Adaptive Software Development. *Wikipedia, the free encyclopedia.* 2005.
24. **BACH, James.** Risk and Requirements-Based Testing. *Computer.* 1999, stránky 129-130.
25. **ALDORF, Filip.** *Metodika RUP.* Praha : VŠE, Katedra informačních technologií, 2007.

7 Slovník pojmů a zkratek

Pojem	Význam
Agilní metodika	viz. metodika, agilní
Analytik	Osoba provádějící analýzu, tj. rozklad systému na menší celky.
Atribut(y)	Vlastnosti, ev. strukturovaná data popisující nějakou entitu.
CVS, SVN	Concurrence Version System – systém pro správu verzí zdrojového kódu. Jedná se o systém, kdy v centrální repozitoři jsou uloženy veškeré zdrojové kódy projektu odkud si je jednotliví vývojáři „půjčují“ pro změny. Lepší CVS umějí sloučit neprotichůdné změny dvou a více vývojářů nad stejným kódem.
Data	Údaje (zde zpravidla v elektronické formě) zachycující nějakou informaci.
Entity	Jedná se o úkazy zpravidla reálného světa znázorněné jako třídy objektů v systému.
Informace	Je mírou redukce nejistoty, k níž dojde při přijetí zprávy
Informační systém	Informační systém je soubor lidí, technických prostředků a metod (programů), zabezpečující sběr, přenos, zpracování, uchování dat, za účelem prezentace informací pro potřeby uživatelů činných v systémech řízení
IT, informační technologie	Technologie zachycující, zpracovávající a nezřídka kdy i řídící tvorbu dat, obvykle pomocí prostředků číslicové elektrotechniky.
kodér	viz.: programátor
Instance	Viz.: třída
Metodika	Doporučený souhrn etap, přístupů, zásad, postupů, pravidel, dokumentů, řízení, metod, technik a nástrojů. Určuje **kdo, kdy, co a proč** má dělat během vývoje a provozu IS.
Metodika, agilní	Metodiky které – na rozdíl od tradičních metodik – umožňují (přímo očekávají) změnu vlastního zadání, tedy toho **co** má být dodáno
Model	Reprezentace určitého objektu nebo systému, pojatá z určitého úhlu

Pojem	Význam
	pohledu.
Patterns	Návrhové vzory – doporučené způsoby (algoritmy) řešení určitých problémů
Proces	Posloupnost jednotlivých kroků transformující určitý vstup na výstup.
Programátor	Osoba provádějící tvorbu posloupnosti instrukcí, pomocí kterých jsou IT technologie schopny vykonávat operace dle předem stanovených vzorů.
Rubikova kostka	Druh hlavolamu, kdy každá strana krychle je rozdělena na 3×3 polí a cílem je mít na každé straně krychle všech 9 polí stejné barvy. Pomocí otáčení částí kostky (vždy na předělu mezi poli) je možné tyto pole promíchat a cílem je právě jejich opětovné sestavení.
Software	Posloupnosti instrukcí, pomocí kterých jsou IT technologie schopny vykonávat operace dle předem stanovených vzorů.
SQL	Structured Query Language – strukturovany dotazovací jazyk, jazyk vytvořený pro získávání dat z databází relačního typu.
Trigger	Objekt tvořený zpravidla kódem SQL provádějící akci jazykem definovanou automaticky na základě nějaké události v systému.
Třída	Zde pojem z objektového programování – jedná se o reprezentaci nějakého typu objektu. Třídou je například „*jablko*“. Víme, že existuje jistý obecný popis jablka (tedy existuje definice „obecného“ jablka) avšak „*jablko, které držíme v ruce*“ (má konkrétní atributy – tj. například barvu, váhu, velikost, aktuální umístění („v ruce“)…) je již instancí této obecné třídy.
Transformace	Přeměna jednoho objektu (ev. informace) ve druhý (ev. ve druhou)
Uložená procedura	Objekt tvořený zpravidla kódem SQL provádějící akci jazykem definovanou (zpravidla celou sadu jednotlivých příkazů) na základě volání z jiného programu.

Bc. Lukáš Plachý (54219)
plachyl@std.fbm.vutbr.cz

8 Přílohy

Sekce umístěná v práci v souladu s článkem 5, odst. 1 směrnice rektora č.9/2007. K této práci nejsou dodávány žádné přílohy.