INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ

SOFTWAROVÉ INŽENÝRSTVÍ

URČENO PRO VZDĚLÁVÁNÍ V AKREDITOVANÝCH STUDIJNÍCH PROGRAMECH

JAROSLAV ŽÁČEK

ČÍSLO OPERAČNÍHO PROGRAMU: CZ.1.07
NÁZEV OPERAČNÍHO PROGRAMU:
VZDĚLÁVÁNÍ PRO KONKURENCESCHOPNOST
OPATŘENÍ: 7.2
ČÍSLO OBLASTI PODPORY: 7.2.2

INOVACE VÝUKY INFORMATICKÝCH PŘEDMĚTŮ VE STUDIJNÍCH PROGRAMECH OSTRAVSKÉ UNIVERZITY

REGISTRAČNÍ ČÍSLO PROJEKTU: CZ.1.07/2.2.00/28.0245

OSTRAVA 2014

Tento projekt je spolufinancován Evropským sociálním fondem a státním rozpočtem České republiky

Recenzent: doc. Ing. František Huňka, CSc.

Název: Softwarové inženýrství
Autor: Jaroslav Žáček
Vydání: první, 2017
Počet stran: 139

Jazyková korektura nebyla provedena, za jazykovou stránku odpovídá autor.

© Jaroslav Žáček
© Ostravská univerzita v Ostravě

OBSAH

1 ÚVOD **5**

1.1 CO V TEXTU NALEZNETE? 5
1.2 CO NENÍ NÁPLNÍ 6

2 HISTORIE SOFTWAROVÉHO INŽENÝRSTVÍ **7**

3 ZÁKLADNÍ POJMY A ARCHITEKTURY IS **11**

3.1 SOFTWAROVÉ INŽENÝRSTVÍ 12

4 GLOBÁLNÍ ARCHITEKTURA **15**

5 PROCESNÍ ŘÍZENÍ **24**

5.1.1 Podnikový proces *26*
5.1.2 Výhody procesního řízení *28*
5.1.3 Grafická reprezentace podnikového procesu *28*
5.2 IT STRATEGIE, STRATEGICKÉ ŘÍZENÍ 31
5.3 MĚŘENÍ EFEKTIVNOSTI IS 32
5.3.1 Ukazatele přínosů ICT *33*
5.3.2 Magický trojúhelník kvality *36*
5.3.3 CMMI a ISO 9000 ..... *36*

6 PROJEKTOVÁNÍ IS, ARCHITEKTURY IS **39**

6.1 PROJEKTOVÁNÍ, METODIKY, SOFTWAROVÉ INŽENÝRSTVÍ 39
6.2 VODOPÁDOVÝ VS. ITERATIVNÍ PŘÍSTUP 41
6.3 ISO/IEC 12207 A DALŠÍ STANDARDY 41
6.4 ARCHITEKTURY INFORMAČNÍCH SYSTÉMŮ 42
6.4.1 Globální architektura informačního systému *44*
6.4.2 Dílčí architektury informačního systému *44*

7 SPECIFIKACE POŽADAVKŮ **47**

7.1 TRADIČNÍ PŘÍSTUP 49
7.2 USE CASE 49
7.2.1 Chyby při tvorbě Use Case *51*
7.3 AGILNÍ PŘÍSTUPY 52
7.4 FURPS+ 53
7.5 STANDARD IEEE 830 54
7.6 NÁSTROJE 56

8 ŽIVOTNÍ CYKLUS VÝVOJE IS **59**

8.1 ITERACE 61
8.2 RUP FÁZE 62
8.3 INCEPTION PHASE 64
8.3.1 Milník LOM *66*
8.4 ELABORATION PHASE 67
8.4.1 Iterace *68*
8.4.2 Milník LCA *72*
8.5 CONSTRUCTION PHASE 73
8.5.1 Iterace *73*
8.5.2 Milník IOP *75*
8.6 TRANSITION PHASE 75

8.6.1 Cíle ... *76*
8.6.2 Testování ... *77*
8.6.3 Lessons learnt ... *77*
8.6.4 Milník PRM ... *77*
8.7 UML V PROCESU VÝVOJE ... 78

9 TESTOVÁNÍ SOFTWARE ... 83

9.1 TESTOVÁNÍ VE VÝVOJOVÉM PROCESU SOFTWARE ... 84
9.2 DRUHY TESTŮ ... 87

10 PROVOZ A ÚDRŽBA PODLE ITIL ... 91

10.1 ŠPATNÝ SCÉNÁŘ ... 91
10.2 LEPŠÍ SCÉNÁŘ ... 95
10.3 CO JE ITIL ... 98
10.4 STRUČNÝ PŘEHLED PROCESŮ SS A SD ... 102

11 APLIKACE IS, SYSTÉMOVÁ INTEGRACE ... 108

11.1 ERP ... 109
11.2 CRM ... 112
11.3 SCM ... 114
11.4 BUSINESS INTELIGENCE ... 115
11.5 SYSTÉMOVÁ INTEGRACE ... 116
11.6 OUTSOURCING ... 118

12 CASE SYSTÉMY ... 122

12.1 DRUHY CASE SYSTÉMŮ ... 122
12.2 KOMPONENTY CASE SYSTÉMŮ ... 124
12.3 VOLBA A HODNOCENÍ CASE NÁSTROJŮ ... 125
12.4 ZKUŠENOSTI S CASE A MYLNÉ PŘEDSTAVY ... 126

13 DOKUMENTACE SW ... 128

13.1 FORMA DOKUMENTACE ... 128
13.2 DOKUMENTACE ARCHITEKTURY ... 129
13.3 NÁSTROJE PRO DOKUMENTACI ... 131
13.3.1 XML ... *131*
13.3.2 JavaDoc ... *133*
13.3.3 IEEE 1063-2001 ... *137*

1 Úvod

V této kapitole se dozvíte:

- Co je náplní tohoto učebního textu.
- Co není náplní tohoto textu

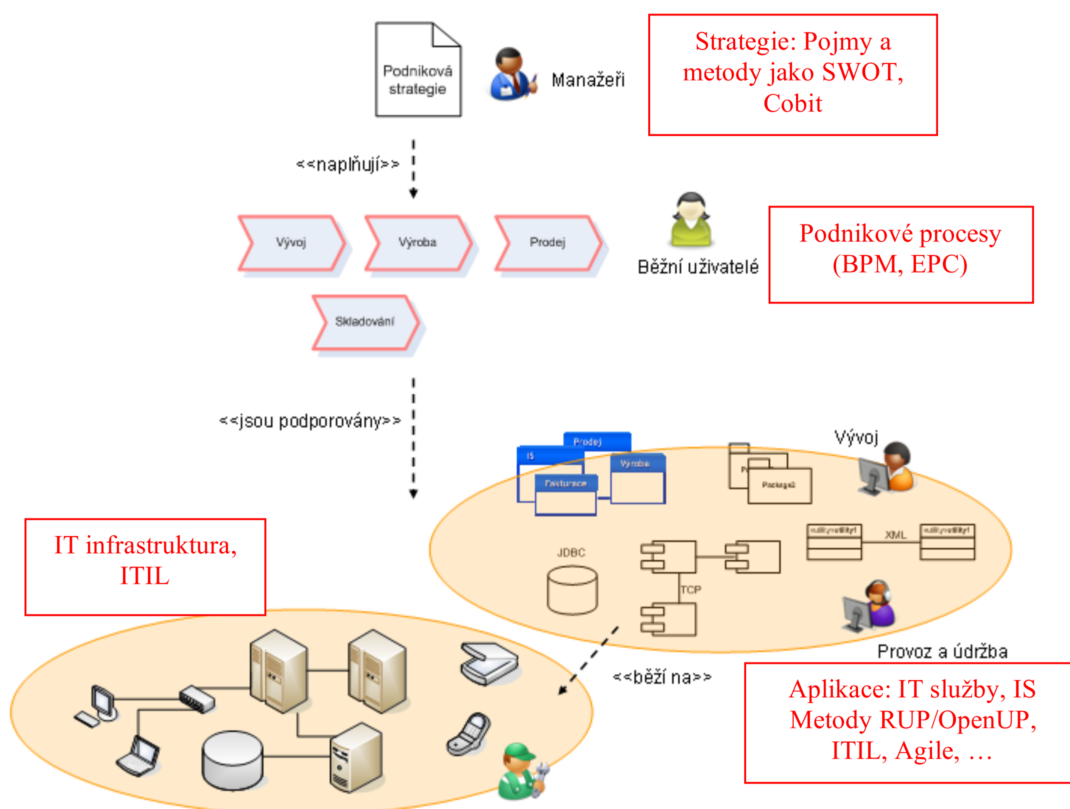
1.1 Co v textu naleznete?

Tento text shrnuje v kostce znalosti z mnoha oborů a disciplín, které by měl mít absolvent informatického oboru. Náplň předmětu tedy překračuje hranice samotného „softwarového inženýrství“, které je zaměřeno pouze na vývoj a provoz software a tedy na inženýrské disciplíny, které provádíme, abychom dostali fungující software naplňující potřeby uživatelů. V textu se postupně budeme věnovat základním pojmům v oblasti informatiky a informačních systémů, architekturám IS, vlastnímu softwarovému inženýrství a přidruženým disciplínám jako je projektové řízení, problematika CASE nástrojů či dokumentování SW projektů. Vše bude zasazeno také do rámce organizace, zmíníme pojmy jako podniková strategie a podnikové procesy. V neposlední řadě se budeme stručně zabývat i dnes tak populárním outsourcingem a velmi důležitou stránkou jako je použitelnost aplikací.

V textu nenaleznete žádnou CASE study, jelikož je každá organizace a každý projekt odlišný a tudíž by toto nemělo velký význam. Spíše se zaměříme na vysvětlení a demonstraci některých klíčových konceptů a principů, jejichž samotná adaptace v projektu přináší výsledky. Koncepty jsou demonstrovány na množství příkladů.

Odborné pojmy budeme uvádět často v jejich zavedených anglických verzích, jelikož české jsou často neustálené nebo zavádějící. Navíc je angličtina jazykem IT a její zvládnutí je jedním ze základních požadavků kladených na pracovníky v oboru IT.

Následující obrázek zachycuje problematiku předmětu. Od cílů organizace (definované manažery ve strategii firmy) přes podnikové procesy (naplňují tyto cíle, typicky jde o prodej a nákup, fakturaci, skladování, výrobu, vývoj, atd.), ICT na podporu a automatizaci podnikových procesů (informační systémy, portály, databáze, velká úložiště dat) až k sítím a hardware na kterých tyto systémy běží. Budeme se zabývat jak vlastním popisem strategie, podnikových procesů, informačních systémů, stejně jako typy těchto aplikací a také metodami a postupy pro jejich vývoj a správu.



Obr. 1 Kontext předmětu SWENG/XSWEN. Problematika sahá od strategie podniku až po aplikace a hardware, sítě na podporu běhu organizace, resp. jejich podnikových procesů.

1.2 Co není náplní

Náplní textu není detailní popis konceptů a disciplín zde představených. U všech témat se budeme snažit odkazovat na další kvalitní české, hlavně pak zahraniční zdroje, které mohou sloužit k dalšímu studiu.

Text se také nezabývá technologiemi nebo programovacími jazyky, ty jsou jen doplněním/naplněním, některých zde představených postupů či konceptů.

2 Historie softwarového inženýrství

V této kapitole se dozvíte:

- Jaký je podíl hardware, software a údržby na ceně software v dnešních dnech?
- Co je to výzkum Standish Group a co přináší?
- Co to je tzv. softwarové krize?

Po jejím prostudování byste měli být schopni:

- Pochopit proměnu ceny softwarových projektů.
- Popsat příčiny a důsledky softwarové krize.
- Vysvětlit, jak se komunita vypořádala se softwarovou krizí.

Klíčová slova této kapitoly:

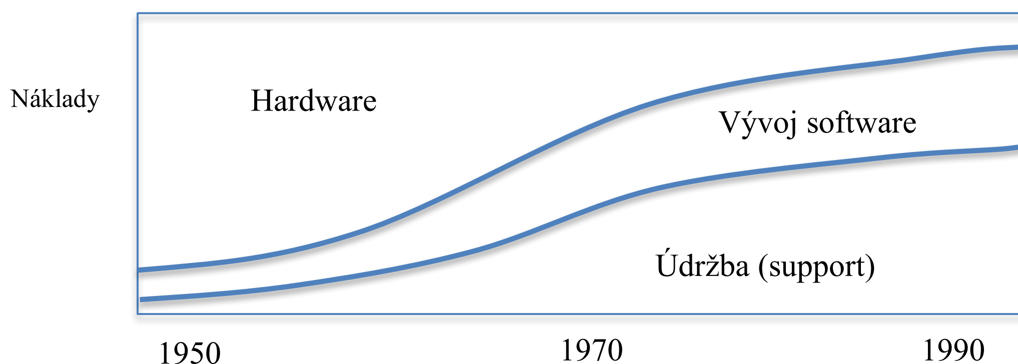
IT rozpočet, softwarová krize, CASE, Chaos Report

Doba potřebná ke studiu: 2 hodiny

Průvodce studiem

Kapitola popisuje postupný vývoj v oblasti softwarového inženýrství a dává ho do kontextu s první a druhou softwarovou krizí. Součástí textu jsou také stručně zmíněny techniky, které pomohly softwarové krize překonat.
Na studium této části si vyhraďte 2 hodiny.

Pro pochopení současných přístupů a obecných problémů v oblasti softwarového inženýrství je potřeba uvést některé historické souvislosti. Vývoj softwarového inženýrství byl výrazně ovlivněn změnou poměru mezi pořizovací cenou hardware a cenou softwarové výbavy. Specifickou části je pak samotná údržba. Ta se zpočátku orientovala převážně na hardware (pořízení nového záznamového média, výměna procesoru, sběrnice), později při vzrůstající komplexnosti softwarového vybavení se zvyšovala cena údržby za software a v současné době podle některých odhadů tvoří až 60% přijmu průměrné IT společnosti zabývající se vývojem software.



Obr. 2: Vývoj nákladů na hardware, software a údržbu v čase

Softwarové inženýrství

V 50. letech došlo z hlediska vývoje software k prvnímu významnému milníku, a to ke změně poměru nákladů mezi softwarovým produktem a hardwarem, na kterém je tento produkt provozován. V průběhu času do současnosti se cena pořizovacích nákladů na hardware neustále snižuje. Výpočetní výkon tehdejších počítačů byl mnohonásobně menší, než výkon dnešního průměrného mobilního telefonu a spolehlivost minimální. Na těchto počítačích běžely velmi jednoduché programy s dávkovým zpracováním. Programy byly uloženy na děrném štítku, který musel programátor ručně "vyštípat na papír". Děrný štítek se přinesl do výpočetního střediska a za nějaký čas (zpravidla do druhého dne) dostal výsledek výpočtu. Software byl tedy jednoduchý, nenáročný na paměť co do uložení kódu, bez interakce s uživatelem a pracoval s ním pouze jeho tvůrce. Postupem času s vývojem počítačů se hardware stával menší, rychlejší a levnější. Kvůli většímu dostupnému výkonu bylo možno budovat složitější a funkčně obsáhlejší programy. Na programu začíná pracovat více lidí a jakmile se do software zapojila interaktivita s uživatelem, začali programy využívat i běžní uživatelé. V dnešní době pořízení hardware je pouze minoritní částí IT rozpočtu firmy při realizaci IT zakázky. Naopak udržitelnost již nasazeného software je pro jeho použití klíčová a zároveň finančně nejnáročnější část jeho životního cyklu.

Významnou složkou IT rozpočtů firem tedy nyní tvoří z větší části vývoj nového software a z největší části údržba již dříve vyvinutého softwarového vybavení společnosti. Tato změna je zapříčiněna narůstající komplexností programového vybavení. Složitost hardware se sice také zvětšuje, ale stále se jí daří jistým způsobem automatizovat a to značně snižuje obecnou cenu hardwarového vybavení. Při kreativní duševní činnosti, jako je vývoj složitějších softwarových aplikací, nelze vždy použít unifikovaný automatizovaný proces a proto se vývoj aplikací pro konkrétní společnost prodražuje. Společným atributem většiny softwarových produktů se stávalo zpoždění zavedení software do provozu, neplánované zvyšování rozpočtu projektů a funkčnost, která neodpovídala původnímu zadání.

Pokud tedy chceme za použití software celkově ušetřit, je vhodné se věnovat z části optimalizace metody vývoje software a optimalizace procesu údržby, což spadá celkově do oblasti softwarového inženýrství. To nám nabízí množství praktik, jak dosáhnou celkové úspory při nasazení a udržování softwarových informačních systémů.

V 60 letech bylo již programové vybavení tak rozsáhlé, že v roce 1968 a 1969 byly zavedeny pojmy *Softwarové inženýrství* a problémy s vývojem označeny jako *Softwarová krize*. Softwarová krize se projevovala (a stále projevuje) neúnosným prodlužováním a prodražováním projektů, nízkou kvalitou výsledných produktů, problematickou údržbou a nízkou produktivitou práce programátorů. Hledání jednoduchého a účinného řešení na existující problémy vedlo k zavedení strukturovaného programování.

O trvání softwarová krize a o míře její závažnosti se můžeme přesvědčit například ze studie Standish Group Report, která byla zveřejněna v USA. Tato studie se nazývá Chaos report a od roku 1994 je každoročně vydávána jako obraz současného stavu projektů v malých, středních a větších firmách nasazujících softwarové systémy.

V 70 letech se výzkum v nově definované oblasti orientoval na identifikaci dobrých programovacích praktik (tzv. best practices). Tyto přístupy se již dříve v praxi osvědčily a je tedy vhodné je znovupoužít. Dále se začíná

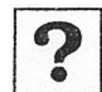
brát v úvahu lidský faktor a vývoj je unifikován definováním jednoduchého životního cyklu tvorby software. Při vývoji komplexních aplikacích se vytváření nové pracovní role (např. vedoucí týmu). Začíná se také používat přístup *shora-dolů*, který je podpořen modulárním programováním (vznikají jazyky C, Pascal). Jednoduché techniky strukturovaného programování nahrazuje jednoduchá metodika strukturovaná analýza - návrh.
Snahy o návrh různých formálních metod pro automatizovanou syntézu programů nebyly příliš úspěšné pro programy s vyšším počtem řádků kódu. Na druhou stranu se velmi osvědčily snahy o zavedení abstrakce a dekompozice rozsáhlejších systémů na moduly. Tyto snahy tak zavedly důležitý pojem abstraktní datový typ. Koncem 70. let se pak začínají formovat datově a procesně orientované metody a je kladen větší důraz na úvodní etapy životního cyklu software. Vývojáři se tak začali zaměřovat na zlepšení metod specifikace požadavků, analýzu požadavků a na návrh. Principy softwarového inženýrství se začínají využívat v počítačovém průmyslu.

V 80 letech se do vývoje zapojují Comupter Aided Software Engineering (CASE) nástroje pro podporu vývoje software. Tyto nástroje pomáhají v orientaci nad složitějším návrhem (grafické zobrazení) a podporují týmový vývoj. Původní zaměření CASE nástrojů bylo generovat kód z visuální předlohy a tím úplně nahradit ruční psaní kódu. Tato myšlenka se později ukázala jako velmi problematická a i dnes nejsou CASE nástroje pro generování funkčního kódu bez zásahu programátora spíše výjimkou (zpravidla doménově specifické modelování). CASE nástroje se ovšem velmi osvědčily jako pomocný nástroj pro tvorbu software pomocí modelů na všech úrovních abstrakce.
Objevují se také nové programovací paradigmata, nejvýznamnější z nich je objektově orientované programování.

Devadesátá léta byla zaměřena na znovupoužitelnost a komponentní architekturu. V oblasti metodik vývoje jsou významné změny zaznamenány v samotném sledování procesu vývoje a jeho tzv. vyzrálosti. Objektově orientované programování se stává hlavní programovací paradigma pro vývoj software. Samotný vývoj software se snaží implementovat i nové trendy v oblasti výzkumu softcomputingu - umělá inteligence, fuzzy přístup.

Ze současných výzkumů je patrné, že se softwarovou krizi do dneška nepodařilo úplně vyřešit. Stále existuje významná část softwarových projektů, které se vyznačují výše zmíněnými problémy. Do popředí se dostávají agilní metodiky vývoje a procesní frameworky (OpenUP, RUP) [2]. Na důležitosti nabývají také měkké aspekty softwarového vývoje - složení týmů programátorů dle jejich povah, různé osobností testy, aplikace základních principů z typologie osobnosti.

Kontrolní otázky:



1. Proč vznikla první a druhá softwarová krize?
2. Jakým způsobem byly softwarové krize vyřešeny?
3. Jak se změnila cena za IT služby v poměru hardware/software v průběhu času?

Úkoly k zamyšlení:

Zamyslete se nad cenou vydávanou společnostmi za IT služby. Co je v dnešních dnech pro softwarové společnosti důležité jako zásadní zdroj jejich příjmů?

Korespondenční úkol:

Vytvořte dokument, ve kterém bude srovnání starších programovací jazyku (pascal) a jazyků moderních (Java, C#). Jaký má dopad z hlediska softwarového inženýrství použití moderních programovacích jazyků na efektivitu práce?

Shrnutí obsahu kapitoly

V této kapitole jste se seznámili s historií softwarového inženýrství a jeho proměnami v čase. Postupně byly vysvětleny problémy, které při vývoji aplikaci vznikali a také techniky, které tyto problémy minimalizují.

3 Základní pojmy a architektury IS

V této kapitole se dozvíte:

- Jaké je základní rozdělení IT?
- Z čeho je složen informační systém?
- Jaká je definice softwarového inženýrství?

Po jejím prostudování byste měli být schopni:

- Načrtnout základní rozdělení IT.
- Určit všechny potřebné složky informačního systému.
- Definovat pojem softwarové inženýrství a říci, na jaké oblasti se dělí.

Klíčová slova této kapitoly:

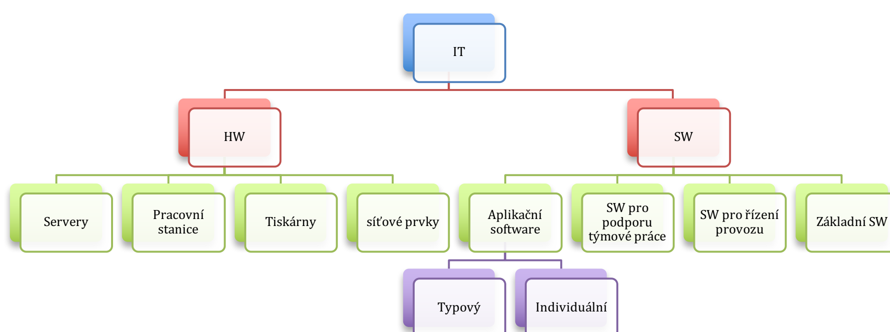
Informační technologie, informační systém, typový software, individuální software.

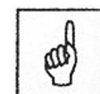
Doba potřebná ke studiu: 2 hodiny

Průvodce studiem

Kapitola nejprve uvádí rozdělení IT a později ho dává do kontextu s prvky informačního systému. Dále pak definuje pojem softwarové inženýrství dle IEEE Standard Computere Dictionary a také nabízí volnou formu definice.
Na studium této části si vyhraďte 2 hodiny.

V této kapitole se budeme zabývat základním rozdělením IT a také pojmem informační systém. Ukážeme si, že informační systém není pouze kus software, který můžete nainstalovat na infrastrukturu, ale také vše potřebné k tomu, aby tento software mohl fungovat. Již dříve jste se učili, že IT se dělí na složku hardware a software. My si toto rozdělení ještě upřesníme.





Obr. 3: Rozdělení IT

Na obrázku vidíme rozdělení informačních technologií (IT) v různých kategoriích, přičemž některé z nich si v textu zmíníme podrobněji. *Informační technologie* jsou dle definice hardwarové a softwarové prostředky pro sběr, přenos, uchování, zpracování a distribuci informací.
Informatická služba je relativně samostatná část IS viditelná koncovému uživateli a zaměřená na podporu jednoho nebo více procesů organizace. Zmiňujeme-li pojem aplikace, máme na mysli z čeho je IS tvořen, v případě

služby k cenu příslušná část IS v organizaci slouží, kdo je jejím provozovatelem (dodavatelem) a kdo jejím uživatelem (zákazníkem).
Informatický zdroj je komponenta (HW, SW, data-informace-znalost) nutná k tvorbě a provozu informatické aplikace nebo informatické služby.
Pojem *informační systém* je uživateli velmi často používán intuitivně a je pod ním často myšleno téměř veškeré softwarové vybavení, které má podnik k dispozici. Jaká je tedy definice, co přesně je informační systém a z čeho se skládá? Informační systém organizace je systém informačních technologií, dat a lidí, jehož cílem je efektivní podpora informačních a rozhodovacích procesů na všech úrovních řízení organizace (firmy). Vývoj a provoz IS jsou ovlivňovány radou aspektů. Informatická aplikace je relativně samostatná část IS (zahrnující HW, SW a data), vzniklá nebo zabudovaná do IS jedním projektem (např. e-mail, správa majetku, účetnictví).

Informační systém se tedy skládá z několika aspektů, které ovlivňují jeho existenci a další vývoj:

- Hardware – servery, stanice, tiskárny, skenery.
- Software – základní, typový a aplikační software.
- Sítě – propojení těchto komponent dohromady.
- Data a datové zdroje.
- Peopleware – lidé (uživatelé, správci, programátoři, konzultanti)
- Orgware – organizační struktura firmy a její pravidla
- Okolí – zákazníci, zákony, konkurence, trh, státní instituce.

IS by měl pomáhat plnit podnikové cíle. To znamená, že vychází ze strategie podniku a snaží se nějakým způsobem usnadnit fungování či úplně automatizovat podnikové procesy dané organizace. O pojmech jako strategie, podnikové procesy budeme mluvit v dalších kapitolách. Nyní si pouze řekneme, že informační systém by měl řešit nějaký problém v problémové doméně a přinášet nějakou hodnotu. Příkladem může být:

- Automatizované objednání nových dílů při poklesu jejich množství na skladě.
- Urychlení vyřizování a zpracování objednávek.
- Automatické zaslání upozornění (notifikace) studentovi, které informuje o končící platnosti výpůjčky knihy v knihovně.
- Evidence nákupního košíku prostřednictvím věrnostních karet zákazníků v supermarketu.

3.1 Softwarové inženýrství

Pro to, abychom se mohli bavit o oblasti softwarového inženýrství, měli bychom si tento pojem přesně definovat. Pomoci by nám mohla například definice IEEE, jakožto autority v oblasti IT:

Software engineering is the application of a systematic, disciplined, quantifiable approach to development, operation, and maintenance of

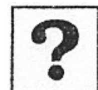
software; that is, the application of engineering to software. (IEEE Standard Computer Dictionary, 1990)

My se můžeme spokojit s volnou interpretací, že softwarové inženýrství je systematický, disciplinovaný a měřitelný přístup k vývoji, nasazení a údržbě software.

Při počátečním seznámení s vývojem aplikací může člověk nabýt dojmu, že softwarové inženýrství je programování. Při malých aplikacích tomu tak skutečně i může být. Pokud ovšem vytváříte rozsáhlý informační systém, pak zjistíte, že samotné programování (nazývejme raději implementace) je pouze jednou částí v celkovém procesu vývoje software.
V dnešních dnech naleznete softwarové vybavení prakticky všude (momentálně se složitější softwarové systémy snaží prosadit v náramkových hodinkách). Softwaru jsou svěřeny důležité informace (státní správa), peníze (banky) a mnohdy i lidské životy (kardiostimulátor). S ohlédnutím na oblasti, ve kterých je software využíván je potřeba pro ně zvolit specifický proces vývoje. Jeden bude zaměřen na spolehlivost, jiný na bezpečnost, jiný na správnost vložených informací. V každém softwarovém projektu je také podmínka, aby byl co nejlevnější. To je ovšem v přímém rozporu s požadavkem např. na bezpečnost - tu zajistíme neustálým penetračním testováním a audity a to určitě nebude levné. Zvýšení spolehlivosti software se také na ceně nepromítne pozitivně. Proto nám zde pomáhá softwarové inženýrství, abychom mohli určit "rozumnou" míru bezpečnosti daného software a přiměřených nákladů.
Softwarové inženýrství zahrnuje tři hlavní oblasti:

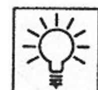
- Řízení projektu - Definování a řízení životního cyklu projektu (abychom věděli co máme v který čas dělat, např. abychom nejprve nedělali implementaci a poté teprve návrh).
- Techniky - Objasnění jakým způsobem můžeme provést analýzu, návrh, implementaci, podporu již běžícího software atd. (při analýze můžeme využít např. UML diagramy).
- Báze znalostí - Je přínosná pro samotného softwarového inženýra, umožňuje využívat již ověřené postupy a techniky, nabízí mu návody k řešení konkrétních situací (např. pomoc při budování distribuovaného systému na platformě Java).

Kontrolní otázky:



1. Jakým způsobem dělíme IT?
2. Jaké oblasti zahrnuje softwarové inženýrství?
3. Jaká je definice softwarového inženýrství?

Úkoly k zamyšlení:



Jenou z hlavních oblastí softwarového inženýrství je báze znalostí. Kde byste tuto bázi znalostí hledali?

Korespondenční úkol:

Vyberte si libovolný Vám známý "program" a dejte ho do kontextu s pojmem informační systém. Postupně vypište všechny složky, které jsou nutné, aby Váš "program" mohl pracovat (např. lidské zdroje, infrastruktura).

Shrnutí obsahu kapitoly

V této kapitole jste si rozšířili dělení IT. Poté jsme definovaly samotný pojem softwarové inženýrství a rozdělili samotný softwarový produkt dle způsobu jeho použití. Na závěr kapitoly je uvedeno rozdělení softwarového inženýrství do oblastí.

4 Globální architektura

V této kapitole se dozvíte:

- Co to je globální strategie?
- Jakým způsobem globální strategii určíme?
- Co to je podnikový proces?

Po jejím prostudování byste měli být schopni:

- Definovat globální strategii a určit její hlavní složky.
- Popsat podnikový proces a jeho vztah ke globální architektuře.
- Určit globální podnikovou strategii pomocí SWOT analýzy.

Klíčová slova této kapitoly:

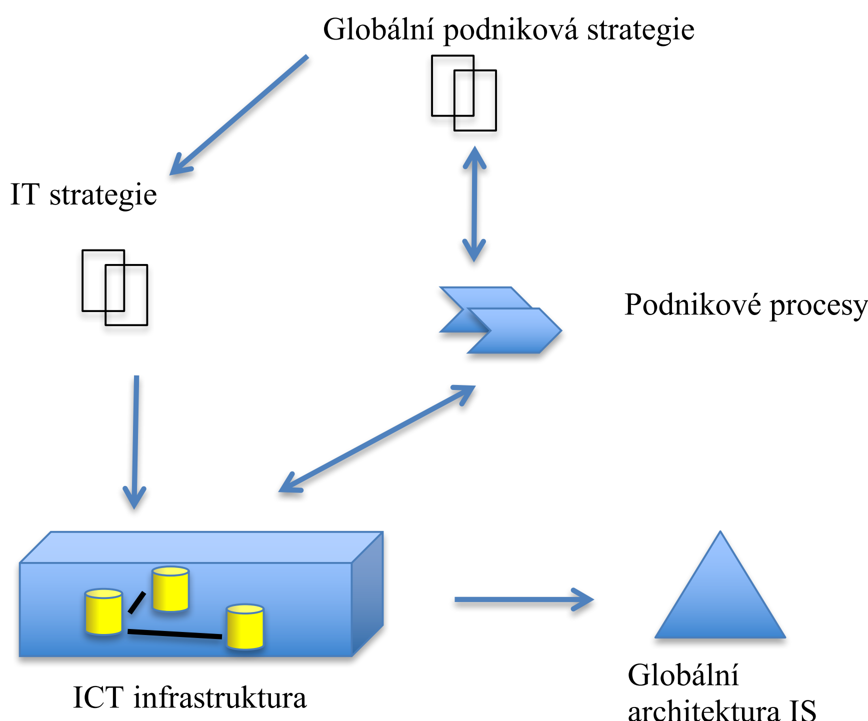
Globální strategie, SWOT analýza, podnikový proces, IT strategie.

Doba potřebná ke studiu: 5 hodin

Průvodce studiem

Kapitola se zabývá globální podnikovou architekturou, určení podnikové strategie a jejího vztahu k podnikovým procesům. Pro určení globální strategie je využito SWOT analýzy.
Na studium této části si vyhraďte 5 hodin.

V této kapitole se budeme zabývat důležitou oblastí, která nám dá kontext pro celý předmět. Vysvětlíme smysl existence každé organizace (vyjma neziskových), popíšeme, co jsou to podnikové procesy, jaké je jejich místo v organizaci a také ukážeme místo a smysl ICT technologií v organizaci.



Smyslem existence každého podniku je vytvářet zisk, bez ohledu na to, jestli vyrábím, nakupuji a prodávám, vyvíjím či poskytuji služby. Smysl existence podniku je definován v tzv. globální (podnikové) strategii. Tato strategie stručně říká, co děláme, kdo jsou naši zákazníci, na jaké trhy se orientujeme, kde na to vezmeme (peníze, lidi, technologie). Strategie pouze udává směr, o její realizaci se starají podnikové procesy. Podnikový (business process) proces je pak sled aktivit, které musíme vykonat, abychom vyprodukovali nějaký výstup. Příkladem podnikových procesů může být prodej a nákup, fakturace, naskladnění, výzkum, výroba, vývoj, personalistika atd. Proto, aby tyto procesy fungovali efektivně a pokud možno automatizovaly manuální, opakující se činnosti, slouží informační a komunikační technologie (ICT). ICT musí podporovat procesy vorganizaci a musí pomáhat naplnit cíle definované globální podnikovou strategií. Směr a cíle ICT jsou definovány v IT strategii, která určuje co a jak budeme pomocí ICT podporovat, automatizovat, jaké technologie a platformy použijeme a jaké jsou preferované modely pořízení SW (nákup, vývoj, systémová integrace, ...). Tato IT strategie musí být v souladu s podnikovou strategií. Obrázek ještě naznačuje malou část IT strategie – globální architekturu IS (nám již známou pyramidu).

Podnikové procesy jsou v dnešním řízení firmy klíčové a protože každý správný software je budován pro podporu podnikových procesů, je nutné si je zde uvést. Pro pochopení samotného zavedení procesního řízení je potřeba se podívat do minulosti. Samotný pojem podnikový proces definoval již v 18 století Adam Smith:

"One man draws out the wire, another straights it, a third cuts it, a fourth points it, a fifth grinds it at the top for receiving the head: to make the head requires two or three distinct operations: to put it on is a particular business, to whiten the pins is another ... and the important business of making a pin is, in this manner, divided into about eighteen distinct operations, which in some manufactories are all performed by distinct hands, though in others the same man will sometime perform two or three of them."

V té době pochopitelně neexistovala výpočetní technika, ovšem vznikala potřeba vytvořit jakýsi postup kdo co dělá při vytvoření výrobku, na jehož výrobě se podílí více lidí. Ve 20 století pak Henry Ford zavedl jeho slavnou pásovou výrobu - jeho předpoklad byl, že dělníci jsou líní a proto je potřeba přidat pás, který bude neustále udržovat stejné tempo výroby. Tím ovšem zavedl první automatizaci podnikového procesu (návod jak vyrobit auto - podnikový proces, běžící pás - jeho automatizace). V dnešní době jsme se již myšlením trochu posunuli a tak aby bylo možné určit definici podnikového procesu, je nutné nejprve nadefinovat jednotlivé dílčí pojmy. Obecných definic pojmu proces je více, zde bychom uvedli definici podle K. Luďka [Lud05]:

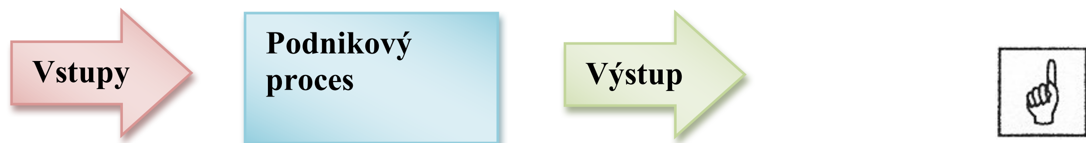
Proces je sled dílčích operací, které na svém konci mají jasně definovaný výstup.

Obdobně je možné nalézt definici pro pojem podnik podle G. Wöhe [Wöh95]:

Podnik je plánovitě organizovanou hospodářskou jednotkou, v níž se zhotovují a prodávají věcné statky a služby.

Skloubením těchto dvou pojmů je možné definovat samotný pojem podnikový proces. Takových definic je velmi mnoho, ale významově jsou téměř totožné. Například V. Řepa [Řep06] jej definoval následovně:

Podnikový proces je souhrn činností transformující souhrn vstupů na souhrn výstupů, které podnik provádí za účelem plnění podnikových cílů.



Obr. 5: Podnikový proces

Jinými slovy je podnikový proces sled kroků, které vytváří produkty a služby transformací vstupů na výstupy (**Obr. 5**). Dále je nutné zmínit, že každý proces musí mít svého vlastníka a pevně stanovené hranice. Podnikové procesy je možné obecně rozdělit do 3 kategorií podle toho, komu jsou určeny jejich výstupy [Šeb01]:

- Hlavní procesy – výstupy jsou orientovány na externího zákazníka. Příkladem může být dodávka produktu, platba, analýza trhu, servis a další.
- Podpůrné procesy – jedná se o interní procesy zaměřené na interního zákazníka (např. podnikové útvary, či jiné procesy), zde se jedná například o údržbu, výcvik, správa dokumentů, administrativa a podobně.
- Řídící procesy – ovlivňují hlavní i podpůrné procesy pro zajištění plnění podnikových cílů. Zde patří například plánování, kontroly, řízení lidských zdrojů a podobně.

Pro pochopení pojmu podnikový proces a jeho vztah ke kontextu podniku nám toto minimum postačuje, následující kapitola pak rozebírat tento pojem detailně včetně příkladů.

4.1 Podniková strategie

Globální podniková strategie určuje poslání firmy, celopodnikové cíle, priority, podmínky a zdroje pro dosažení těchto cílů. Zejména musí strategie určit následující:

- hlavní předmět podnikání,
- skupiny zákazníků, na které je podnik orientován,
- nabízené produkty či služby,
- hlavní obchodní partnery (zejména ve smyslu určení místa podniku v dodavatelsko-odběratelských sítích),
- zdroje (lidé, znalosti, finance, technologie,...) nutné pro dosažení stanovených cílů.

Při tvorbě globální strategie organizace slouží konceptuální model tvorby globální strategie. Tento model definuje hlavní zaměření podniku, dále podnikové cíle a jejich priority, definuje také zdroje pro realizaci cílů, způsob ověřování jejich naplňování a osoby odpovědné za jejich dosažení.

Konceptuální model se skládá z několika bodů, jsou jimi:

- SWOT analýza,
- formulace poslání podniku,
- definice globálních podnikových cílů,
- vymezení globálních podnikových funkcí,
- tvorba strategií pro globální podnikové funkce,
- vyhodnocení a změny.

4.1.1 SWOT analýza

SWOT analýza je způsob, jakým lze určit možnosti naší firmy. Pro pochopení této techniky si nejprve musíme rozebrat, co znamenají jednotlivé písmena:

- S - Strength - reprezentuje silné stránky - v čem je naše firma dobrá a jedinečná na trhu
- W - Weaknesses - reprezentuje slabé stránky - v čem firma zaostává za konkurencí
- O - Opportunities - reprezentuje příležitosti, kterých bychom mohli využít, abychom postavení naší firmy na trhu zlepšili
- T - Threats - popisuje hrozby, které nám můžou v podnikání zabránit

SWOT analýza má spoustu možných variant použití v různých oblastech, např. analýza produktu, osoby, používaného software, organizační jednotky, týmu, města atd. Slouží jako strukturovaný přístup k identifikaci výchozího stavu. Pro grafickou reprezentaci si většinou nakreslíme kříž, do kterého vepíšeme jednotlivé složky:

Obr. 6: SWOT analýza - základní koncept

Zde je potřeba zmínit, že ne všechny podněty vycházení ze společnosti samotné a proto je musíme rozdělovat na vnitřní původ a *vnější původ*. Nemůžete například ovlivnit, jestli konkurence na stejné ulici jako sídlí vaše firma neotevře také pobočku a neodvede tím vaše zákazníky. V tomto případě se jedná o hrozbu vnější. Vnější hrozby reprezentují složky O a T (příležitosti a hrozby).
Naopak dostatek lidských zdrojů (kmenových zaměstnanců) se může počítat jako silná stránka a v tomto případě se jedná o *vnitřní původ*. Ten reprezentují složky S a W (silné a slabé stránky).
Dále pak můžeme rozdělit kvadranty na pomocné a škodlivé. Pokud do SWOT analýzy přidáme nějaký bod, musíme pro něj zároveň definovat strategii. Ta nám říká, jakým způsobem budeme na identifikovaný bod reagovat. V případě silných stránek potřebujeme definovat strategii pro jejich rozvoj. Pokud se zajímáme o slabé stránky společnosti, musíme mít nějakou strategii na jejich odstranění. V příležitostech pak musíme vědět, jak jich využijeme a pokud se zaměříme na hrozby, tak ty je potřeba minimalizovat. Hrozby nikdy nemůžeme zcela odstranit - např. konkurenční podnik můžete sice koupit, ale to vám nezaručí, že konkurence zmizí úplně. Pro zápis strategií může posloužit následující obrázek:

SWOT analýza		Analýza vnitřního prostředí	
		Silné stránky (Strenghts)	Slabé stránky (Weaknesses)
Analýza vnějšího prostředí	Příložetitosti (Opportunities)	**Strategie** maximalizací silných stránek – maximalizovat příležitosti	**Strategie** minimalizací slabých stránek – maximalizovat příležitosti
	Hrozby (Threats)	**Strategie** maximalizací silných stránek – minimalizovat hrozby	**Strategie** minimalizací slabých stránek – minimalizovat hrozby

Obr. 7: SWOT analýza detailně

Dříve bylo zmíněno, že SWOT analýzu můžeme použít prakticky na cokoliv. Nejprve si tedy uvedeme obecný příklad:

Obr. 8: SWOT analýza - obecný příklad (zdroj: http://theanspruchsvoll.wordpress.com/category/uncategorized/)

Americký systém středního vzdělávání je kreditový. Aby mohl student získat středoškolský diplom, musí mít předepsaný počet kreditů. Střední škola se proto rozhodla vytvořit reklamní kampaň, která na tuto skutečnost studenty lépe upozorní a dá jim šanci poslední kredity získat. Aby reklamní kampaň měla úspěch, pomohli si SWOT analýzou, která jim sdělila následující:
Silná stránka pro cílovou skupinu je jasná - student může získat potřebné kredity. Slabá stránka ovšem je, že to není zadarmo, počet studentů je omezen a v konkrétní čas je nabízena vždy jen jedna hodina (výuka neprobíhá paralelně). Příležitost je rozšířit tyto kurzy i mezi jiné studenty než ty v posledním ročníku a také zvýšit počet míst. Hrozby pak byly identifikovány jako problém při organizaci doplňkové výuky a samotný fakt, že chybějící kredity si student může doplnit na jiné střední škole.
Zde je vidět, že byla SWOT analýza použita pro identifikaci vhodného zacílení reklamní kampaně a s IT nemá mnoho společného. Pojďme si tedy ukázat příklad z IT oblasti - zavedení nového informačního systému do společnost:

	Vnitřní faktory	Vnější faktory
Pozitivní faktory	*Silné stránky:* - informační systém splňující mezinárodní standardy - sofistikované řešení zaměřené speciálně na řízení a plánování výroby - realizační tým se zkušenostmi z jiných implementací a znalostmi problematiky řízení výrobních procesů - odborník (v projektovém týmu zákazníka) na většinu interních procesů firmy se zkušenostmi implementace předcházejícího IS	*Příležitosti:* - zajištění růstu konkurenceschopnosti na základě zvýšení dodavatelské spolehlivosti, snížení průběžné doby výroby, snížení nákladů a zvýšení efektivity podnikání - vyřešení on-line odvádění výroby s návazností na výpočet mezd
Negativní faktory	*Slabé stránky:* - časově náročná spolupráce projektového týmu (ČZUB) po celou dobu trvání implementace - nutné delegování vlastních pravomocí klíčových členů projektového týmu na další pracovníky	*Hrozby:* - změny v personálním obsazení projektových týmů - neochota přizpůsobení se změnám a tím novým procesům uvnitř firmy

Obr. 9: SWOT analýza - příklad zavedení IS do společnosti

Společnost Česká Zbrojovka Uherský Brod zaváděla nový informační systém do podniku a potřebovala odhadnout možné dopady zavedení systému ještě před jeho samotným vytvořením. V rámci SWOT analýzy identifikovali postupně silné stránky, slabé stránky, příležitosti a hrozby. Povšimněte si, že jednotlivé body SWOT analýzy nejsou pouze o software samotném, ale také o doprovodných jevech souvisejících se zavedením informačního systému - neochota přizpůsobit se změnám a tím novým procesům uvnitř firmy.

4.1.2 Formulace poslání podniku

Dalším bodem konceptuálního modelu tvorby globální podnikové strategie, který následuje po provedení SWOT analýzy je formulace poslání podniku. Je třeba stanovit kritické faktory rozvoje podniku – ty jsou vymezeny ve SWOT analýze. Následně je třeba na identifikované kritické faktory adekvátně reagovat. To znamená začít uskutečňovat navrhované akce ze SWOT analýzy z důvodu odstranění negativně působících faktorů či udržení a zlepšení pozitivních faktorů. Vedení firmy by si mělo položit tyto otázky:

- Jaké potřeby chce firma uspokojit?
- Pro jaké skupiny zákazníků?
- V jakém teritoriu?
- Jakou technologií?
- Je nezbytné na tyto otázky odpovědět.

4.1.3 Definice globálních podnikových cílů

Třetím bodem tvorby modelu je definice podnikových cílů. Globálních cílů podniku může být několik, záleží na tom, z jakého hlediska je bereme. Může to být například z hlediska:

- vlastníků podniku,
- vrcholového vedení,
- pracovníků podniku,
- společnosti.

Cílem vlastníků podniku může být zatraktivnit firmu za účelem jejího prodeje. Vrcholové vedení podniku se může snažit zlepšit chod procesů a cash flow. Ve spojení s lepším cash flow pak zlepšit finanční sílu a možnosti podniku. U pracovníků společnosti může být cílem dobré pracovní prostředí, adekvátní odměna za odvedenou práci a dobrý systém motivace.

4.1.4 Vymezení globálních podnikových funkcí

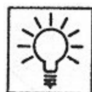
Má-li podnik stanovené cíle, lze přikročit k vymezení jeho funkcí, což je také dalším bodem návrhu konceptuálního modelu globální podnikové strategie. Stejně jako SWOT analýzu provádíme pro všechny faktory (interní i externí, organizační jednotky, apod.), tak i funkce podniku definujeme pro všechny jeho části – organizační jednotky. Počínaje vrcholovým řízením organizace a marketingem přes nákup, prodej, výrobu, skladování, služby až po výzkum a vývoj, ekonomiku či informační systém organizace.

4.1.5 Tvorba strategií pro globální podnikové funkce

Předposledním bodem modelu je tvorba strategií pro různé funkce podniku. Tak například v marketingové strategii budeme definovat jak sbírat, skladovat a využít data o našich zákaznících v souladu se zákonem, jakou mají mít strukturu a obsah, způsob a rozsah reklamy výrobků, prezentaci firmy veřejnosti, apod. Finanční strategie může obsahovat způsob využití volných finančních prostředků, nastavení úvěrové politiky, způsob výpočtu penále pohledávek nebo definici vzájemných zápočtů mezi firmami. Personální strategie se zabývá stavem zaměstnanců, jak početním, tak znalostním. Definujeme systém hierarchického postupu, odpovědností pracovníků, odměn a výhod. Navrhujeme školení a další kvalifikační kurzy, také rekvalifikační kurzy s možnostmi spolupráce s pracovními úřady, apod. Informační strategie definuje potřebu a možný rozvoj informačních zdrojů v organizaci.

Kontrolní otázky:

1. Co to je globální strategie?
2. Co to je SWOT analýza a k čemu slouží?
3. Jaký má vztah podnikový proces ke globální architektuře?



Úkoly k zamyšlení:

Jak se bude lišit globální architektura banky od globální architektury softwarové firmy?

Korespondenční úkol:

Vyberte si libovolný Vám známý informační systém a vytvořte na něj SWOT analýzu. Pro každý bod SWOT analýzy musí být definována i strategie pro odstranění/udržení/rozvinutí/minimalizaci.

Shrnutí obsahu kapitoly

Tato kapitola zavádí pojem globální architektura a popisuje její dílčí prvky. Pro určení poslání podniku je použita SWOT analýza. Technika je nejprve vysvětlena na obecném příkladu a poté upřesněna na konkrétním nasazení informačního systému do společnosti.

5 Procesní řízení

V této kapitole se dozvíte:

- Co je to procesní řízení?
- Z čeho se skládá podnikový proces?
- Jak měřit efektivnost podnikového procesu?

Po jejím prostudování byste měli být schopni:

- Vysvětlit rozdíl mezi funkčním a procesním řízením.
- Definovat svůj vlastní podnikový proces.
- Navrhnout vlastní metriku a ohodnotit podnikový proces.

Klíčová slova této kapitoly:

Událost, aktivita, grafická notace podnikového procesu, funkční řízení, procesní řízení, CMMI.

Doba potřebná ke studiu: 12 hodin

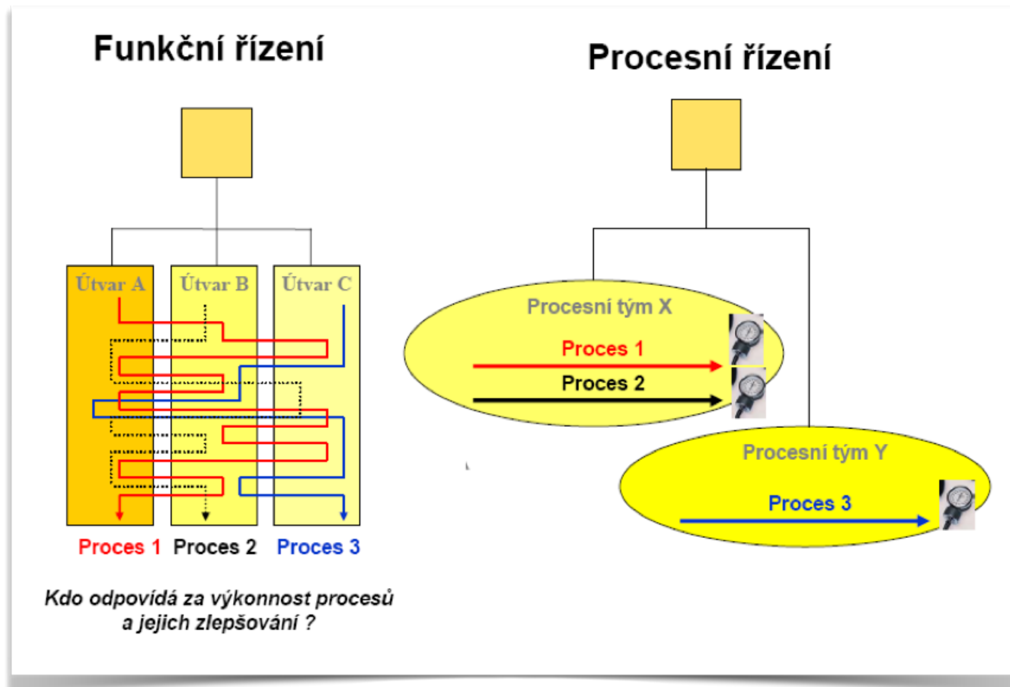
Průvodce studiem

Kapitola se v první části zabývá rozdílem mezi funkčním a procesním řízením a podrobně definuje podnikový proces. Dále jsou uvedeny elementy podnikového procesu a způsob jeho grafického vyjádření včetně příkladů. Poslední část kapitoly se věnuje měření efektivnosti informačních systémů, konkrétně pak metrikám. Na závěr je zmíněn úvod do CMMI a jeho vztah k efektivitě podnikových procesů.
Na studium této části si vyhraďte 12 hodin.

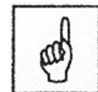
Ještě než zmíníme, co to je proces, a jakým způsobem identifikovat, popisovat a zlepšovat procesy v organizaci, zobecníme procesní řízení z úrovně projektů na řízení podniku. Do dnešní doby (po téměř 200 let) byl hojným manažerským způsobem využívaným k řízení organizace funkční přístup. Tento přístup vychází z myšlenek Adama Smithe, že výrobní procesy mají být rozloženy na nejjednodušší úkony. Funkční jednotky tedy slouží k rozdělení složitějších, sofistikovaných činností a k jejich dekompozici na jednoduché kroky, které může vykonávat i nekvalifikovaný dělník (viz A. Smith: Bohatství národů, napsaná v roce 1776). Tento přístup má však několik zásadních nevýhod, které si zde stručně popíšeme.

Procesu se účastní a pracuje na něm několik týmů, tyto týmy vykonávají pořád stejné činnosti a dokáží se v nich zlepšovat. Tato výhoda umožní zdokonalit pouze jeden část řetězce, který na projektu pracuje, ale cílem není zlepšovat jednotlivé kroky, ale celý výsledný postup. Důvodem je fakt, že pokud začneme optimalizovat jednu část systému bez ohledu na ostatní, můžeme sice docílit vylepšení efektivity této části systému, ale celkově může systém na efektivitě ztratit. Je to dáno tím, že vylepšení jedné části může znamenat zhoršení v jiné části. Například změna požadavků na vstupní informace do části systému může negativně ovlivnit ostatní části systému.

Další nevýhodou je komunikační bariéra mezi jednotlivými týmy, které si musejí předávat jednotlivá dílčí řešení, informace či produkty. Týmy mají jiné vedoucí, jiné zkušenosti, jiné znalosti a proto předávání je problém. Při předávání se tedy často nějaká část informací ztrácí, zůstává pouze v hlavách pracovníků jednoho týmu.



Obr. 10: Funkční a procesní řízení



Procesy musí produkovat nějaké výstupy, jinak by byly zbytečné. Výstupem bývá většinou nějaký produkt, služba nebo informace. Častou chybou ve funkčně řízených organizacích bývají procesy, které neprodukují žádné produkty, ale jen uspokojují vnitřní požadavky organizační struktury.

Poslední nevýhodou funkčního přístupu, kterou zmíníme, je nedefinovaná či nejasná zodpovědnost za daný proces a jednotlivé činnosti. Ve funkčním pojetí totiž zodpovědnost postupně přechází z jednoho manažera funkčního týmu na další manažery. V případě problému je pak velice složité domáhat se zodpovědnosti za chybu.

Výsledkem těchto problémů a nevýhod ve funkčně řízených společnostech bývá jejich špatně dokumentované chování a postupy. Každá znalost je držena v hlavách jednotlivých členů týmu, v případě odchodu zaměstnance neztratíme pouze jeho, ale rovněž významné části znalostí společnosti.

Procesní řízení organizace (tedy nejen jejich projektů) výše zmíněné nevýhody a problémy odstraňuje. Nyní se již konečně můžeme dostat k definici procesu a jeho vlastnostem.

5.1.1 Podnikový proces

V odborné literatuře můžeme najít několik definic procesu obecně, či v našem kontextu podnikového (čili byznys) procesu, přičemž všechny jsou konzistentní, v souladu.

Harrington říká [Har97], že proces je po částech uspořádaná množina aktivit, které přinášejí přidanou hodnotu. Proces musí mít svého vlastníka. Rovněž má vstupy a musí mít výstupy.

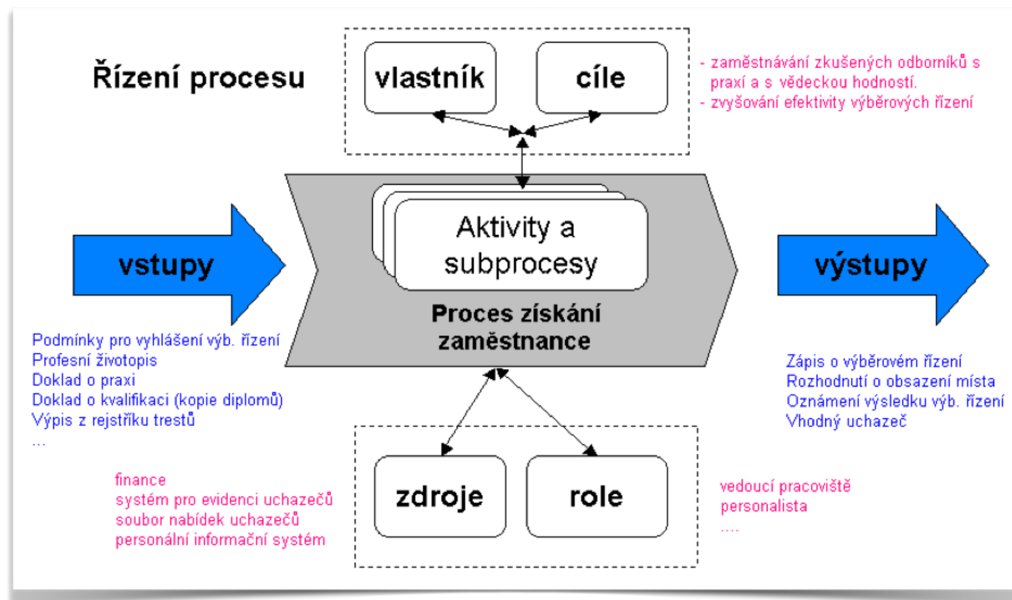
M. Robson a P. Ullah definují proces následovně [Ro96]: Proces je tok práce postupující od jednoho člověka k druhému a v případě větších procesů i z jednoho oddělení do druhého, přičemž procesy lze definovat na celé řadě úrovní. Vždy však mají jasně vymezený začátek, určitý počet kroků uprostřed a jasně vymezený konec.

Velký propagátor reengineeringu podnikových procesů M. Hammer definuje proces následovně [Ha03]: Proces je soubor činností, který vyžaduje jeden nebo více druhů vstupů a tvoří výstup, který má hodnotu pro zákazníka.

Výše uvedené definice, i když vypadají jako odlišné, říkají v podstatě stejné: proces představuje posloupnost aktivit, která je vykonávána, aby bylo dosaženo cíle. Cílem může být například uvaření oběda. Aktivita může být například připravení ingrediencí. Proces musí mít zodpovědnou osobu. Zodpovědná osoba nemusí nutně aktivity vykonávat, ale je zodpovědná za celkový výsledek procesu. Například šéfkuchař je zodpovědný za pokrm, ale obvykle jej sám nevaří. Proces rovněž má vstupy, jako například ingredience a musí mít výstupy, jinak by byl zbytečný. Tímto výstupem je v našem případě hotový připravený pokrm. Podnikové procesy musí naplňovat podnikové cíle, musí tedy ctít a vycházet z podnikové strategii, která tyto cíle definuje.

Jinými slovy se dá říci, že proces je jakási kuchařka, která popisuje jak postupovat. Kdo postup zná, nemusí jej používat denně, kdo jej nezná, bude ze začátku přesně postupovat podle něj. Postup říká, jak na sebe kroky navazují, co bude krokem následujícím, co bylo předchozím a s jakým výstupem, dále nám říká jaké zdroje, dokumenty, či směrnice budeme kjeho vykonání potřebovat apod.

Následující příklad ukazuje *proces získání zaměstnance*. Proces podporuje a naplňuje dva podnikové cíle (cíl č. 1 a 2), definujeme vstupy, které následně transformuje na výstupy procesu, jeho role, které vykonávají činnosti a řídí a spotřebovávají zdroje.



Obr. 11: Podnikový proces - nábor zaměstnanců

Podnikový cíl 1: zaměstnávání zkušených odborníků spraktickými zkušenostmi a s vědeckou hodností (vzděláním).

Podnikový cíl 2: zvyšování efektivity výběrových řízení.

Vlastníkem procesu výběrového řízení může být vedoucí personálního oddělení nebo v menších organizacích samotný vedoucí.

Vstupy procesu získání zaměstnance jsou dokumenty podmínky pro vyhlášení výběrového řízení, profesní životopis, doklad o praxi, doklad o kvalifikaci či výpis z rejstříku trestů, dalším vstupem jsou například nabídky jednotlivých uchazečů či uchazeči samotní.

Výstupy procesu jsou vhodný vybraný uchazeč a dále dokumenty: zápis o výběrovém řízení, rozhodnutí o obsazení místa, oznámení o výsledku výběrového řízení.

Role, které se tohoto procesu účastní jsou zřejmé, jedná se o personalistu(ku), dále vedoucího pracoviště a uchazeče. Další role mohou být zapisovatel, popř. psycholog.

Tyto role spotřebovávají určité **zdroje**, například technologickým zdrojem, který je nutný pro potřeby personalisty je informační systém s evidencí detailů o uchazečích a také personální informační systém, do kterého je zaevidován nový zaměstnanec (vybraný uchazeč). Dalšími zdroji je čas nutný k vykonání výběrového řízení a jeho uspořádání a přípravě a také finance, které jsou k tomuto potřeba (tisk inzerátů, mzdy zaměstnanců podílejících se na výb. řízení).
Příklad neobsahuje dekompozici procesu na jeho podprocesy a činnosti.

Cílem procesního řízení není pouze definovat procesy a tímto skončit. Důležité je procesy implementovat a skutečně se podle nich řídit. Organizace definují

procesy také scílem zpřehlednit jejich chování a rovněž umožnit její vylepšování. Procesní řízení je tedy základem pro neustálé zlepšování. Důvod je ten, že procesy umožní lépe pochopit společnost, její chování, strukturu, potřeby a slabé stránky a také je možné u procesů definovat konkrétní atributy, které pak měříme a vyhodnocujeme.

5.1.2 Výhody procesního řízení

Jaké jsou tedy výhody procesního řízení? Oproti funkčnímu řízení procesní řízení definuje striktně zodpovědnost za proces. Tato zodpovědnost je dána na všech úrovních. Procesní mapa umožňuje definovat hierarchii procesů a zodpovědnost je v procesním řízení definována na všech úrovních. Jelikož proces definuje aktivity, které nejsou předávány dále pryč z procesního týmu, je zodpovědnost striktně dodržována a zpětně vysledovatelná.

Procesní řízení poskytuje vysokou možnost optimalizace. Je to dáno množstvím informací, které popisy procesů poskytují. Optimalizace může být manuální, či automatická s podporou softwaru.

Procesní řízení umožňuje zprůhlednit fungování a chování společnosti a to navenek i zevnitř. V dnešní době společnosti velice často spolupracují s jinými. Společnost má své dodavatele, zákazníky a partnery. Aby tyto vztahy byly efektivní je třeba pracovat na chápání potřeb druhých stran. Namodelované procesy ve vztahu k ostatním organizacím umožňují lépe definovat tyto vztahy.

Každá společnost definuje své pracovní postupy a chování. Procesy jsou jednou z možností. Výhodou procesů je fakt, že tento popis je unifikovaný a lehce čitelný. Běžný způsob popisu chování společnosti je neunifikovaný a pro každou část společnosti se liší.

5.1.3 Grafická reprezentace podnikového procesu

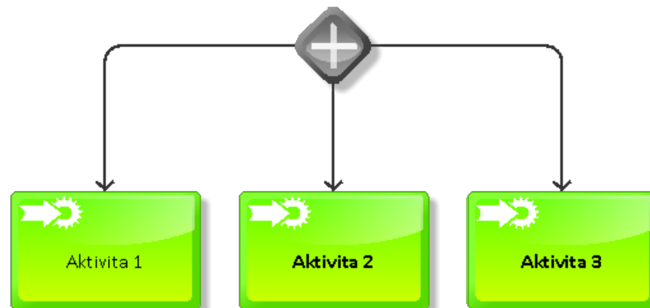
Pro definici podnikového procesu je dobré použít grafickou podobu. Jedna z možností je použít software Aris Express, který pro modelování podnikových procesů využívá následující grafickou notaci:

Obr. 12: Elementy pro modelování podnikových procesů

Pro modelování tedy můžeme použít následující prvky:

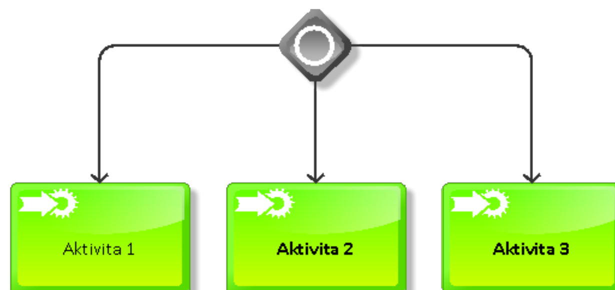
- Událost - vyjadřuje stav či ukončený děj, např. Faktura je odeslána zákazníkovi, Balík byl odeslán.
- Aktivita - reprezentuje činnost, kterou je potřeba provést, např. Registrace na předmět, Přihlášení na zkoušku aj.
- Logické spojky - AND, OR a XOR.

Abychom mohly v podnikovém procesu zavést rozhodování, musíme při jeho budování využít právě logické spojky. I když máte za sebou již teoretický základ logiky, neuškodí si tyto spojky připomenout. Logická spojka AND nám definuje, že je potřeba paralelně vykonat všechny následující aktivity, které větví:



Obr. 13: Logická spojka AND

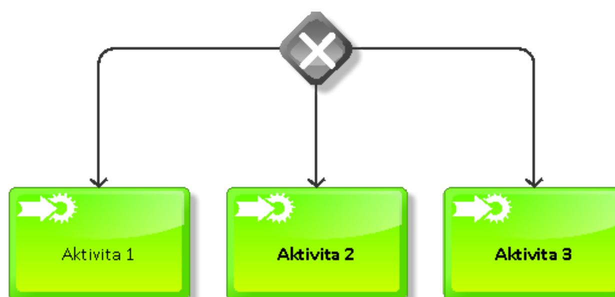
Pokud bychom chtěli modelovat situaci, kdy pro složení notebooku budeme potřebovat všechny hardwarové komponenty (pevný disk, základní deska, baterie, šasi), použijeme spojku AND a musíme tedy vykonat dle Obr. 13 realizovat aktivitu 1, 2 i 3 zároveň.
Logická spojka AND ovšem nestačí pro modelování všech situací. Pokud si sestavujete rozvrh, můžete si vybrat jen některé předměty. Zde bychom mohli použít spojku OR, která nám dovoluje vytvořit kombinaci mezi nadcházejícími aktivitami:



Obr. 14: Logická spojka OR

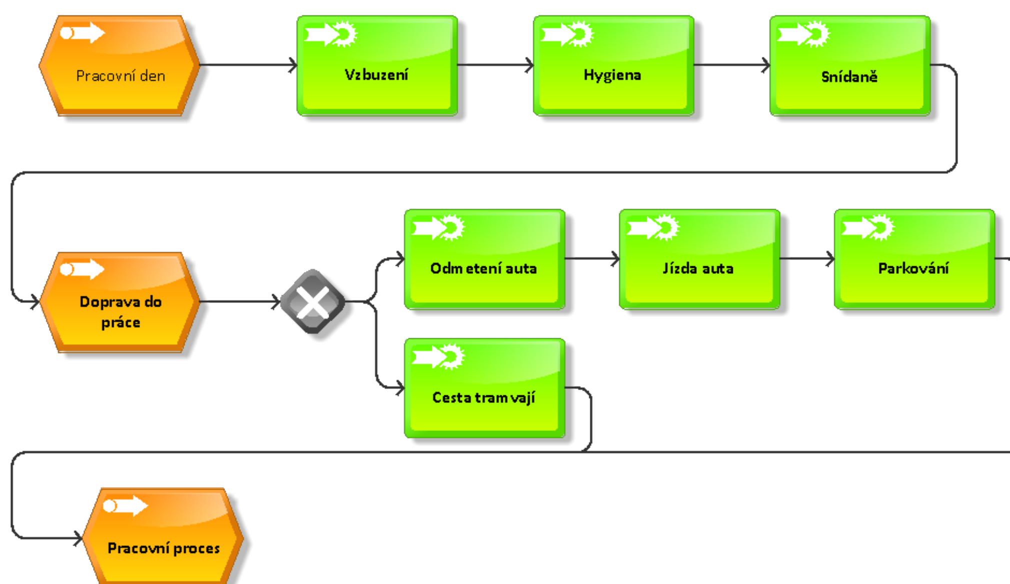
Zde bychom si mohli vybrat aktivitu 1 v kombinaci s aktivitou 3, nebo ve speciálním případě dokonce všechny tři navazující aktivity (tento speciální případ je ekvivalentní se spojkou AND).

Poslední spojka, kterou můžeme použít je vhodná pro modelování situací, kdy si potřebujeme striktně vybrat pouze jeden směr. Pokud cestujete do školy, máte zpravidla na výběr mezi autem, MHD, případně pěší chůzí. Musíte si ovšem vybrat pouze jednu variantu, není možné jet MHD a zároveň i autem. Zde tedy využijeme spojku XOR:



Obr. 15: Logická spojka XOR

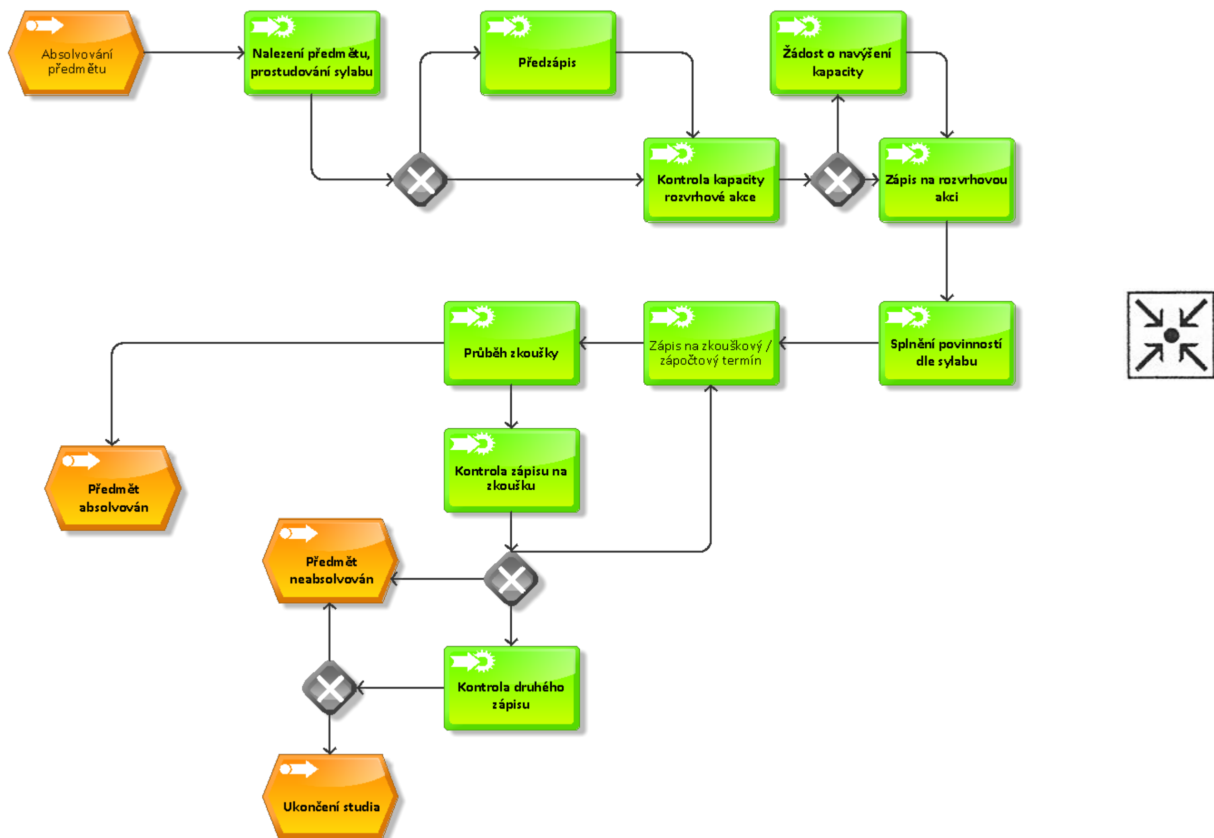
Kombinací výše uvedených prvků tak můžeme vytvořit proces počátku pracovního dne. Nejprve se vzbudíme, poté vykonáme ranní hygienu a snídani. Dále pak vybereme dopravní prostředek a pokud zvolíme v případě lednových dnů jízdu autem, přináší to s sebou další činnosti. Tento proces bychom mohli graficky vyjádřit následujícím způsobem:



Obr. 16: Proces - pracovní den

Událost "Doprava do práce" je zde oddělena z toho důvodu, že by se mohla stát vlastním procesem (tzv. sub-procesem). Událost "Pracovní proces" pak z důvodu zjednodušení není rozepsána na aktivity, ale pro lepší pochopení si ji můžete sami určit a konzultovat s vyučujícím.

Pro dobré procesů je dobré vztáhnout příklady na studentům dobře známou problémovou doménu. Proto zde uvádíme příklad, jak by mohl vypadat proces absolvování předmětu na Ostravské univerzitě:



Obr. 17: Proces - absolvování předmětu na OSU

Náš podnikový proces má jasně definovaný vstup - Absolvování předmětu a také jasně definované výstupy - Předmět absolvován, Předmět neabsolvován a Ukončení studia.

V návrhu jsme počítali s následujícími alternativami:

- Některé předměty mají předzápis, jiné nemají.
- Pokud máte ve studijním plánu předmět označen jako A a má vyčerpanou kapacitu, můžete požádat o její navýšení.
- Pokud zkoušku nesložíte napoprvé, máte možnost jít na další dva termíny.
- Pokud předmět označen jako A neabsolvujete dvakrát po sobě, následuje ukončení studia.

Logické spojky nám zde zaručí opakovatelnost celého procesu.

5.2 IT strategie, strategické řízení

Další stavební kostkou z našeho Obr. 4 je IT strategie a ICT infrastruktura. ICT musíme, stejně jako podnikové procesy, řídit. Strategické řízení IS/IT je kontinuálním procesem. Cílem je efektivně využít informační systémy, technologie (IT služby) k vytvoření přidané hodnoty produktů a služeb, které nabízí organizace svým zákazníkům. Není to tedy pouze vytvoření dokumentu IT strategie (stejně jako tomu není v případě tvorby a řízení strategie firmy, projektového plánu vývoje SW apod.). Jako každá strategie je i IT strategie zaměřena dlouhodobě, tzn. Na období 3-5 let dopředu. V případě vytváření podnikové strategie provádíme následující tři kroky:

- Analýza a zhodnocení současného stavu IS/IT.
- Definice cílového stavu IS/IT (podle cílů byznysu).
- Návrh postupu, jak dosáhnout cílového stavu ze stavu aktuálního.

Jelikož jsme řekli, že strategie a strategické řízení není jen o tvorbě dokumentu, je nutné vpravidlených intervalech také kontrolovat postup a doplňovat důležité změny, fakta. IT strategie je důležitá v tom, že:
- Určuje rozvoj společnosti v oblasti IS/IT.
- Slouží jako zdroj informací pro poptávkový dokument pro případné dodavatele.
- Definuje vazby mezi jednotlivými projekty IT a ostatními projekty v organizaci (zavádění ISO, nová výrobní linka, ...).
- Obsahuje podklady pro tvorby rozpočtů a plánování investic.

Jaké jsou cíle procesu definování informační strategie? Obsahem informační strategie je komplexní pohled na celou problematiku IS/IT v podniku. Výsledkem procesu definování strategie je tedy nalezení odpovědí (rozhodnutí) na následující otázky:

- Jak může IT přidat hodnotu našim produktům?
- Jaký IS zvýší nejvíce naši konkurenceschopnost?
- Kdo a jak má řídit rozvoj a provoz IS/IT?
- Jak má být rozvoj a provoz IS/IT organizován?
- Kolik prostředků máme vydávat na rozvoj a provoz IS/IT?
- Kde a jak máme získávat tyto zdroje a jak hodnotit jejich efektivnost?
- Jak vychovávat a motivovat pracovníky ve využívání IS/IT?

Ke strategickému řízení IT přispívají i různé frameworky, jako například COBIT či ITIL, které poskytují systematický pohled na rozpočet, využití kapacit a IT infrastruktury či IT služeb podle vývoje byznysu apod. Více o těchto frameworcích viz samostatná kapitola dále v textu.

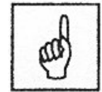
Více o propojení strategie podniku s IT strategií a podnikovými procesy viz [Pro06] či podrobněji v [So06].

5.3 Měření efektivnosti IS

Pokud se rozhodneme nějaký IS zavést a podpořit či zautomatizovat naše interní podnikové procesy, propojit je s dodavateli či zákazníky, je vhodné tuto snahu měřit. Cílem je zjistit, zda se nám implementace daného IS vyplatila, zda nám v něčem pomohla, či jsme jen utratili peníze. Měřit bychom neměli jen počáteční přínos, ale náklady a přínosy průběžné. Jelikož dnes IS dávno nejsou konkurenční výhodou ale spíše nutností, kvůli neustále rostoucí náročnosti zákazníků, globální konkurenci, vzniku nových trhů, výkyvům trhů. Manažeři potřebují aktuální informace vysoké kvality. Informační systém bychom neměli zavádět protože to řekl náš nový zahraniční vlastník nebo proto, že jej má konkurence. Je třeba najít si čas a stanovit si *důvody, účel a cíl zavádění IS*. Ten by měl korespondovat se strategií firmy. Motto, které ctí profesor Molnár

říká: „Pravou hodnotu informačního systému si uvědomíme teprve až v okamžiku, kdy o něj přijdeme.“
Problematika hodnocení efektivnosti IS je do značné míry otázkou nejen potřeb a jejich efektivního uspokojování, ale také otázkou očekávání, kterou mají lidé (koneční hodnotitelé a příjemci užitku). V podnikové sféře můžeme identifikovat 4 kategorie subjektů a jejich očekávání:

- Majitelé – IS/IT by mělo přinášet trvalé zhodnocování jejich majetku vloženého do podniku.
- Manažeři – IS/IT má dávat možnost úspěšně řídit podnik tak, aby bylo dosahováno žádoucích výsledků s minimem potřeby zdrojů jim svěřených do správy.



- Zaměstnanci – IS/IT by měl nabídnout lepší pracovní podmínky, vyšší společenský status a větší pocit sounáležitosti s podnikem.
- Zákazníci – díky výše zmíněnému by měl dostávat produkt či služby s vyšší přidanou hodnotou za přijatelnou cenu.

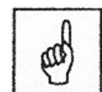
5.3.1 Ukazatele přínosů ICT

Podstatné pro celkovou efektivnost IS/IT je, aby na konci každého projektu (implementace IS/IT) byl spokojený uživatel, a to na všech stupních řízení a ve všech oblastech užití IS/IT. Očekávání mohou být samozřejmě různá a lišit se mohou podle postavení a role uživatele v podniku a také podle toho jak je počítačově gramotný. Systematizace přínosů je nutná proto, abychom mohli tyto ukazatele hned na počátku životního cyklu definovat pro konkrétní aplikaci a podnik. Systematizace je nutná nejen z důvodu definice, ale i způsobu vyhodnocení a stanovení konkrétní odpovědnosti za dosažení určité hodnoty definovaných ukazatelů.

Pro hodnocení přínosů a efektivnosti IS používáme **měření** (formální popis vlastností zvoleného výseku světa). V IS/IT se setkáváme spíše s pojmem **metrika**, což však zpohledu statistiky pojem nesprávný. Funkce, která zkoumaným entitám (objektům či jevům) v reálném světě přiřazuje čísla se nazývá *míra* a číslo přiřazené dané entitě pak *hodnota míry*. Měření převede zkoumaný výsek empirického světa na nějakou formální strukturu. Z ní pak pomocí různých metod můžeme odvodit nové poznatky o zkoumaném výseku světa (tzv. interpretace) [Van07].

Metriku můžeme definovat následovně: „Metrika je přesně vymezený finanční či nefinanční ukazatel nebo hodnotící kritérium, které je používáno k hodnocení úrovně efektivnosti konkrétní oblasti řízení podnikového výkonu a jeho efektivní podpory prostředky IS/ICT. Skupinu metrik sdružených za určitým cílem (tzn. vztahujících se ke konkrétní oblasti, procesu či projektu) nazýváme „portfolio metrik“, viz (Učeň, 2002).“

Základní kategorie metrik jsou:

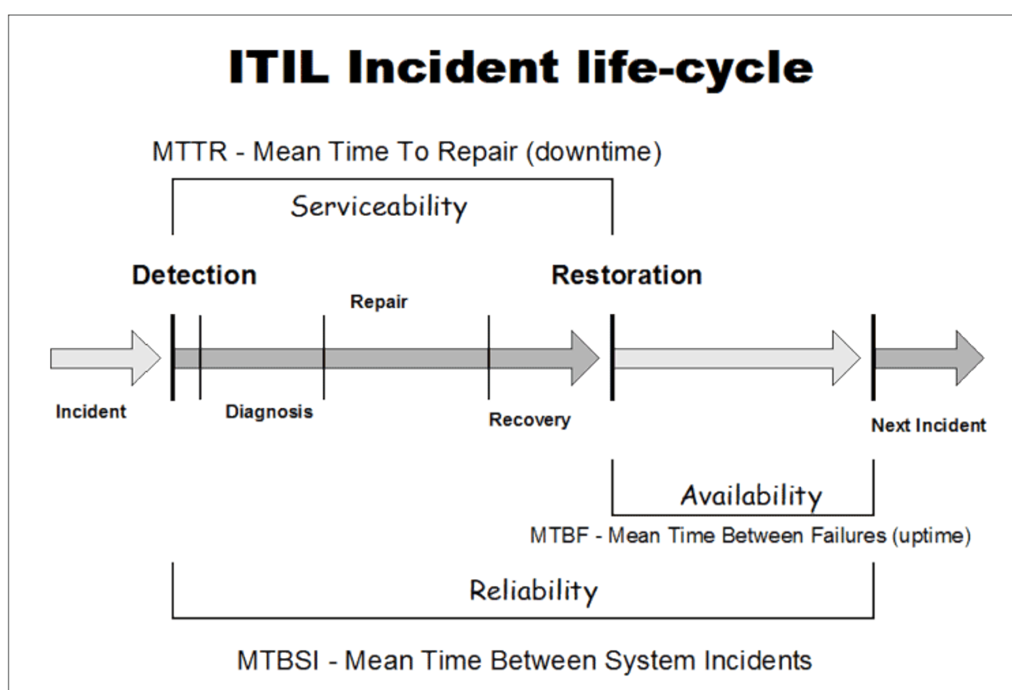


- **Tvrdé metriky** – objektivně měřitelné, snadno měřitelné, často převoditelné na finanční ukazatele.
- **Měkké metriky** – nejsou objektivně měřitelné, mohou využívat například stupnic 0...10 či 0...100, týkají se hodnocení cílů, příkladem

může být hodnocení podpory IT (spíše subjektivní uživatelské hodnocení).

Příklady metrik IS:

- Průměrná doba mezi výpadky (Mean time between failures - MTBF) – průměrná doba mezi jednotlivými výpadky systému, říká nám, jak často dané problémy nastávají.
- Dostupnost (za období) – udávaná většinou vprocentech nebo hodinách, říká, kolik procent či hodin celkového času musí aplikace běžet. Pokud vezmeme v potaz aplikaci běžící 24hodin a jako období pro vyhodnocení 1 den, pak 90% dostupnost aplikace znamená, že aplikace musí být dostupná po dobu 21h a 36min (90% z celkové doby 24h). Tato míra může být brána v celku nebo jako součet za den.
- Response time – doba odezvy funkčnosti aplikace, často uváděna v ms.



Obr. 18: Průměrný čas mezi hlášenými incidenty (zdroj [CMG])

Ukazatele přínosů můžeme klasifikovat z několika hledisek:

- finanční (měřené v peněžních jednotkách) a nefinanční (počet, čas, apod.),
- kvantitativní (kardinální stupnicí) a kvalitativní (ordinální pořadovou či logickou stupnicí splněno – nesplněno),
- přímé (jednoznačný příčinný vztah k dosažení přínosu) a nepřímé (zástupné ukazatele vyjadřující změnu),
- krátkodobé (projeví se do půl roku po implementaci) a dlouhodobé (za více let),
- absolutní (vyjádřené měřitelnou hodnotou) a relativní (bezrozměrným poměrovým číslem).

Finanční ukazatele:

Určují se většinou v etapě plánování, kdy zdůvodňujeme ekonomickou výhodnost investice. Potom se aplikuje některý ze standardních ukazatelů efektivnosti investic jako jsou analýza nákladů a přínosů, diskontovaný cash flow, doba návratnosti investice, apod. Velmi jednoduchým a silným ukazatelem je také tzv. ROI – Return on Investment, česky návratnost investice. Tento ukazatel by měl být jedním ze základních finančních ukazatelů a počítá se například takto:

$$ROI = \frac{úspory}{náklady} x100\%$$

Hodnota vyšší než 100% znamená přínos, hodnota menší než 100% znamená větší investici, než byly přínosy této investice. ROI se počítá většinou z ročních čísel, může být však také měsíční nebo za jiné období. Logicky ale musí být oba údaje za stejné období.

Další finanční ukazete mohou být například:

- finanční úspora na obsluhu a správu systému,
- cena zpracování jedné transakce,
- hodnota celkového vlastněného ICT, infrastruktury a systémů, tzv. Total Cost of Ownership (TCO),
- cena změny/chyby,
- aj.

Nefinanční ukazatele:

Důležitým měřitelným nefinančním ukazatelem přínosů IS/IT je produktivita. Produktivita poskytuje informaci o vztahu mezi vstupními náklady a výstupním užitkem. Je to poměr mezi množstvím vstupů a množstvím výstupů za určitou časovou jednotku. Můžeme vyhodnocovat například produktivitu výroby zboží v korunách na pracovníka za rok, počet obsloužených zákazníků jedním pracovníkem za den, apod. Na růst produktivity může působit celá řada sezónních faktorů, proto musíme srovnávat srovnatelné časové úseky. Ostatní měřitelné nefinanční ukazatele, např. podle podnikových procesů:

- zkrácení průběžné doby vývoje/výroby,
- snížení počtu reklamací,
- zvýšení počtu zákazníků,
- zvýšení podílu na trhu,
- rozšíření výrobního sortimentu.

Nekvantifikované ukazatele:

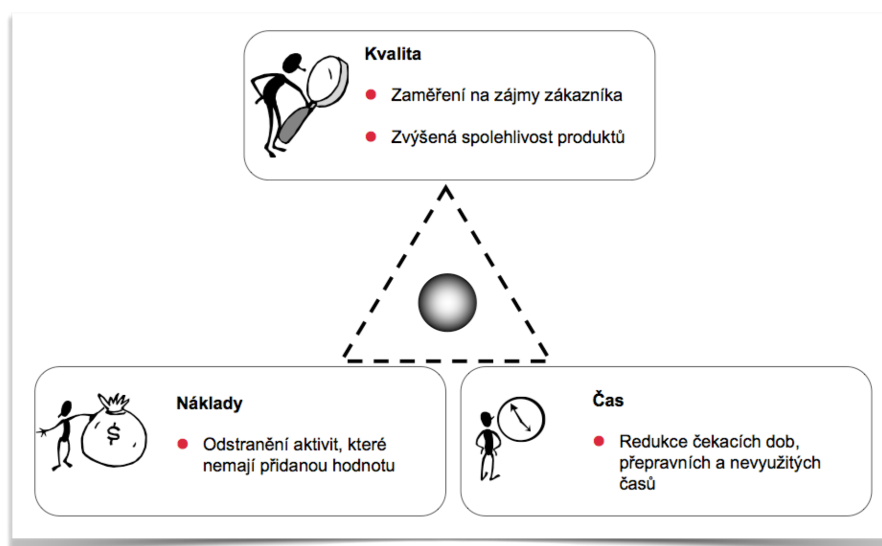
Někdy také nazývány měkké ukazatele. K hodnocení změn měkkých ukazatelů si musíme většinou najít nějaký tvrdý (kvantifikovaný) ukazatel, jehož změna reflektuje co možná nejlépe žádoucí změnu měkkého ukazatele – tzv. zástupný ukazatel. K měkkým ukazatelům patří (kromě nefinančním měřitelných) zejména kvalitativní ukazatele jako:

- zlepšení dobrého jména podniku (hodnoceno průzkumy),

- spokojenost zákazníků (průzkumy, růst zákazníků),
- zvýšení zákaznické věrnosti (opakované objednávky),
- zlepšení pracovního prostředí,
- zvýšení kvalifikace pracovníků podniku.

5.3.2 Magický trojúhelník kvality

Uživatelé vnímají kvalitu podle charakteristik uvedených v předchozích dvou odstavcích, často by ale chtěli všechno ihned a pokud možno zadarmo, což z důvodu fyzických omezení není možné.



Obr. 19: Magický trojúhelník kvality

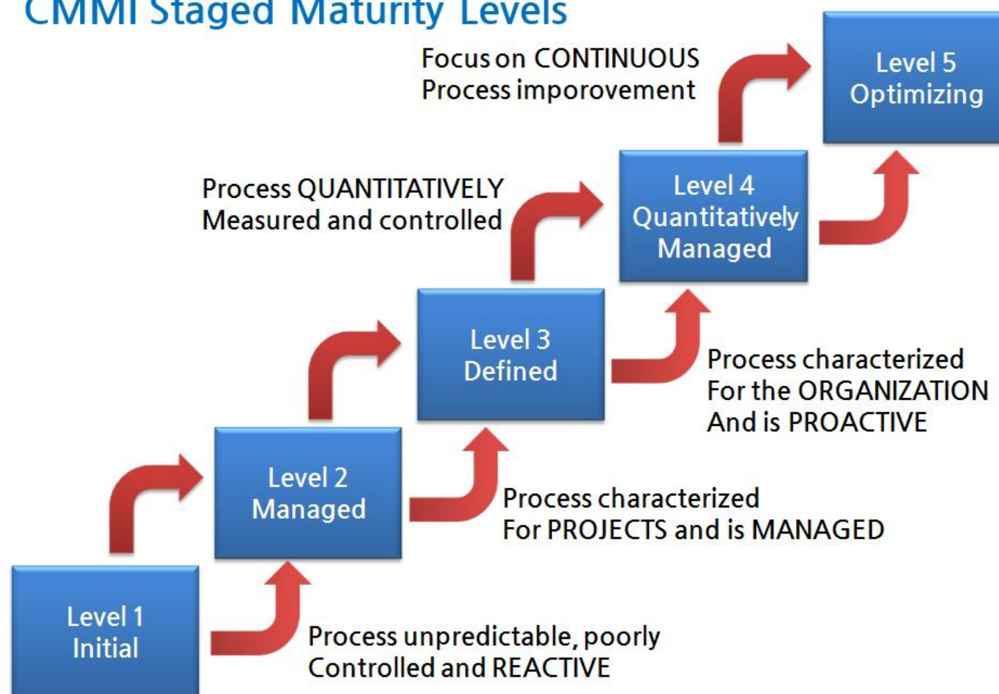
To nás zavedlo ke klasickému *trojúhelníku kvality* (viz obrázek). Při řízení projektů musí být brán v potaz (viz Obr. 19), který popisuje pevný vztah 3 hlavních faktorů: času, nákladů a výkonu/kvality. Z pohledu zákazníka bude vždy požadavek na co nejkvalitnější produkt v krátkém termínu a s nízkými náklady. Při plánování projektu musí vedoucí projektu brát tyto faktory na zřetel a počítat s tím, že pokud bude chtít zkrátit termín musí automaticky počítat s většími náklady nebo snížením kvality, při snaze zvýšit kvalitu je třeba navýšit náklady a/nebo prodloužit termín apod. Nalezení optimálního poměru mezi kvalitou, cenou a časem závisí nejen na zkušenostech uživatelů a dodavatelů, ale také na jejich vzájemné spolupráci při jeho hledání.

5.3.3 CMMI a ISO 9000

Kvalitu můžeme samozřejmě hodnotit také z podhelud celé organizace a to podle standardů či existujících nástrojů. Toto nám pomáhá odlišit organizace, které mají vyspělejší procesy či způsob řízení od ostatních. Některé společnosti spolupracující se státní sférou dokonce potřebují tuto certifikaci jako nutnou podmínku pro získání státních zakázek. Nutnou podmínkou pro certifikaci kvality ISO 9001 je mít zmapované procesy společnosti, což zahrnuje vytvoření procesní mapy a také vytvoření detailní dokumentace jednotlivých procesů. Určitou nevýhodou ISO certifikace je, že auditoři vyžadují pouze

dokumentaci procesů, ale již nezkoumají, zda daná organizace opravdu tyto procesy realizuje v praxi. Můžeme se tedy potom setkat se situací, že organizace má popsané a dokumentované procesy, je vlastníkem ISO normy 9001, ale ve skutečnosti provádí činnosti odlišně, než jsou popsány v dokumentaci, která jim certifikaci přinesla.

Druhou možností, jak ohodnotit kvalitu organizace, respektive jejích procesů je CMMI – Capability Maturity Model Integration. Autorem této metody je SEI (Software Engineering Institute). Jedná se o hodnocení vyspělosti (maturity) klíčových procesních oblastí. Celkovou vyspělost má podnik potom pouze pokud má tuto vyspělost ve všech oblastech pro danou úroveň. Pokud ne, splňuje pouze tzv. capability vdané oblasti. Pokud podnik nedělá nic a řekněme, že může fungovat i chaoticky, má CMMI Level 1. Dostat se na vyšší úrovně, už znamená dělat věci efektivněji, správně, neprodukovat zbytečné vedlejší výstupy, automatizovat činnosti, apod. Jednotlivé úrovně CMMI modelu viz následující obrázek.



Obr. 20: CMMI model vyspělosti procesů

V případě vývoje software je pro dosažení úrovně dva (Maturity Level 2) nutné mít proces pro správu požadavků (requirements management), řídit a monitorovat projekt, mít správu konfigurací a verzí, provádět měření a vyhodnocování metrik. Je nutné dodat, že stejně jako ISO, i CMMI levely se v praxi často „prodávaly“ či se potřebné dokumenty falšovaly, existovali tedy společnosti s maturity level 5, aniž by takto efektivně opravdu fungovaly. Toto bylo běžné hlavně v Asii. Proto SEI, vytvořilo novou verzi CMMI v 1.2, kde se kontrolují i tzv. druhotné artefakty (záznamy z mítinků, e-mailová komunikace, oznámení v kalendářích apod.). Skutečnost je nyní taková, že se spíše vyplatí opravdu věci dělat správně a efektivně, než všechny tyto artefakty falšovat. Více o CMM viz [SEI] či [SEI02].

Kontrolní otázky:

1. Jaký je rozdíl mezi funkčním a procesním řízením?
2. Jak je definován podnikový proces?
3. Co je to CMMI?

Úkoly k zamyšlení:

Jak se bude lišit globální architektura banky od globální architektury softwarové firmy?

Korespondenční úkol:

Vytvořte vlastní proces, který zahraničním studentům vysvětlí, jak si v menze objednat jídlo. Nezapomeňte na to, že nejprve je potřeba se před první objednávkou registrovat. Také nezapomeňte na fakt, že pokud rušíte objednávku v den výdeje jídla, přesouvá se objednávka do tzv. burzy. Pro grafické znázornění procesu využijte software Aris Express.

Shrnutí obsahu kapitoly

Tato kapitola se zabývá procesním řízením, které v úvodu porovnává s dříve provozovaným funkčním řízením. Poté definuje detailně podnikový proces včetně jeho vlastností a nabízí grafickou notaci, kterou je možno proces znázornit. Pro posouzení kvality procesu je vysvětlen pojem softwarová metrika a uveden příklad výpočtu ROI. Závěrem je pak vysvětlen model vyzrálosti procesů CMMI.

6 Projektování IS, architektury IS

V této kapitole se dozvíte:

- Co je to vlastně softwarové inženýrství?
- Jaké existují základní modely vývoje IS/Software?
- Co jsou architektury IS?
- Jaké jsou rozdíly mezi iterativním a vodopádovým přístupem?

Po jejím prostudování byste měli být schopni:

- Pochopit a vysvětlit základní modely vývoje IS/SW.
- Porovnat iterativní a vodopádový přístup k vývoji SW.
- Popsat rozdíl mezi globální a dílčí architekturou IS.

Klíčová slova této kapitoly:

Softwarové inženýrství, metodiky, metody, techniky, iterativní vývoj, architektury informačních systémů, ISO 12207.

Doba potřebná ke studiu: 4 hodiny

Průvodce studiem

Kapitola se věnuje představení pojmů a modelů z oblasti vývoje software, které jsou nezbytné pro pochopení následující problematiky. Dále jsou představeny také architektury informačních systémů.
Na studium této části si vyhraďte 4 hodiny.

6.1 Projektování, metodiky, softwarové inženýrství

Projektování software je proces tvorby nového SW a jeho uvedení do provozu. Tento proces je řízen a má určitá pravidla a doporučení, kterými se při vývoji řídíme. Takový proces se nazývá *metodikou*. Metodika mám říká kdo, kdy, co a proč má dělat během vývoje a provozu SW. Příkladem metodiky jsou klasické OMT, MDIS či agilní Extrémní programování. *Metoda* nám říká, co je třeba dělat v určité fázi nebo činnosti vývoje a provozu. Metody používají různé techniky, každá technika nám říká, jak se dobrat požadovaného výsledku. Metodou je například SWOT analýza používaná ve spoustě oblastí. Nakonec se zmíníme ještě o *nástrojích*, které jsou prostředkem k uskutečnění určité činnosti v procesu vývoje a provozu SW. Nástrojem jsou RAD systémy, CASE systémy, automatické testovací a buildovací nástroje apod.

Definice říká, že metodika je doporučený souhrn principů, konceptů, dokumentů, metod, technik a nástrojů pro tvůrce softwarových (informačních) systémů, který pokrývá celý životní cyklus informačních systémů. Metodika určuje kdo, kdy, co, jak a proč má dělat během vývoje a provozu SW. Metodika napomáhá k tomu, aby byl systém přínosem pro uživatele a celou organizaci. Napomáhá k tomu, aby byly provedeny všechny potřebné činnosti tvorby SW, a to ve správné časové posloupnosti. Metodika také napomáhá k dobré organizaci práce na projektu a jeho dobré a srozumitelné dokumentaci a v neposlední řadě k optimalizaci spotřeby zdrojů při tvorbě a provozu SW.

Dalším pojmem, který budeme dále v textu zmiňovat je *framework*. Framework je konceptuální struktura, která slouží pro řešení komplexního problému určité problémové domény. Jako příklad technologického frameworku můžeme uvést Struts, což je implementační framework pro jazyk Java, řešící vrstvení, ukládaní a komunikaci pro webové aplikace. Jedná se vlastně o znovupoužitelný softwarový systém (většinou jádro, architektura systému). Procesní framework tedy bude s využitím výše zmíněného popisu struktura, která definuje role, činnosti a artefakty (tj. základní elementy procesu), které jsou třeba k vykonání určitého procesu, např. procesu vývoje software. Pro každý konkrétní projekt si pak vývojový tým pomocí procesního frameworku vydefinuje svůj vlastní proces vývoje SW na základě předchozích zkušeností, rozsahu projektu, zkušeností týmu a různých doporučení.

Dalším druhem frameworku je tzv. procesní framework. Tento pojem, resp. koncept má blízko k metodice, je však obecnější, rozsáhlejší a není předepisující. Pokud řeším nějaký problém a postupuji podle metodiky, najdu si na konkrétní straně v knize, jak tento problém vyřešit, metodika toto přesně říká. Procesní framework je oproti tomu spíše sada best practices (praxí ověřených řešení) pro různé problémy. Procesní framework tedy slouží jako zdrojová knihovna, ze které si (kromě nezbytného základu, jádra ve formě principů) vybíráme jen to, co nám přinese nějakou hodnotu. Nemusíme tedy vykonávat zbytečné či málo přínosné aktivity, produkovat nevhodné výstupy, modely, dokumenty.

Posledním pojmem, který je důležité zmínit, je *softwarové inženýrství*. Co to tedy je, čím vším se máme v tomto předmětu zabývat? Definice říká [Von02], že softwarové inženýrství je inženýrská disciplína zabývající se praktickými problémy vývoje rozsáhlých softwarových systémů. Z pohledu dané definice vyplývá, že vývoj softwarového systému zahrnuje celou řadu faktorů nutných k úspěšnému vytvoření požadovaného produktu, jedná se především o:

- technické aspekty zahrnující počítačovou infrastrukturu a softwarové vybavení;
- netechnické aspekty dané organizační strukturou organizace vyvíjející daný produkt a jejími ekonomickými možnostmi;
- znalostmi z oblasti specifikace požadavků na softwarový produkt, jeho analýzy, návrhu, implementace, testování a na konec také instalace u zákazníka;
- lidské zdroje schopné aplikovat výše uvedené znalosti a uplatnit je tak při realizaci softwarového systému;
- řízení spjaté s vývojem samotného produktu umožňující efektivní využití všech výše uvedených faktorů s cílem vytvořit produkt požadované kvality.

Více o problematice tvorby software, či konkrétně tvorby informačních systémů, lze nalézt například v učebních textech [Pro07] nebo [Von02], dále pak v knihách [Ře99], [Ka04], [Bu05].

6.2 Vodopádový vs. iterativní přístup

Spousta problémů při vývoji software je způsobena nevhodným přístupem k vývoji software, respektive jeho vodopádovou formou. Hlavními rysy tohoto typu přístupu je tvorba detailních plánů hned na úvod, kdy ještě nevíme spoustu věcí, nepočítá se v plánech s riziky a nečekanými událostmi (problémy s technologií, složitost problémové domény, ...); požadavky uživatelů jsou zmraženy, nemohou se měnit; integrace komponent a testování probíhá až na konci projektu, kdy je díky nepřesným plánům málo času na řešení odhalených chyb a také je na toto odhalování již příliš pozdě. Problémy jsou také v tom, že jednotlivé týmy rolí (např. analytik, návrhář, programátor, tester) pracují v průběhu projektu odděleně a vytváří velké množství specifikací, které musí další role opět nastudovat. Toto trvá nějaký čas, ztrácí se tak spousta informací zachycených „mezi řádky“ a navíc mohou být tyto specifikace subjektivně interpretovány.

Iterativní způsob vývoje, který je řízen riziky a Use Casy, o němž se budeme bavit v jedné z kapitol dále v textu, se snaží tyto problémy odstranit nejen identifikací a snahou o odstranění rizik, těsnou spoluprácí jednotlivých vývojářů v průběhu celého projektu nebo neustálou integrací a testováním. Nejedná se však pouze o změnu procesu, který by byl pořád stejný pro všechny projekty. Každý produkt vyžaduje jinou kvalitu (např. SW do kritických produktů ve zdravotnictví či letectví musí být stabilnější a kvalitnější než produkty pro domácí PC), jinou úroveň dokumentace a také jinou výkonnost, dostupnost, stejně jako existují různé možnosti znovupoužitelnosti. Iterativní způsob tedy není jen jiným druhem pevně daného procesu, ale spíše způsobem myšlení a souborem principů, přičemž v každém projektu můžeme klást důraz na jiný. Hlavní rozdíly vodopádového (resp. vylepšeného spirálového modelu) a iterativního jsou shrnuty v následující tabulce:

Vodopádové principy	Iterativní (agilní principy)
Zaměřen na procesy, předpokládá jejich opakovatelnost.	Zaměřen na lidi – motivace, komunikace prvořadá.
Pevné, podrobné plány definovány na úvod, kdy je spousta nejasností.	Pro celý projekt pouze road map. Podrobné plány jen pro iterace (kratší časové úseky, max. 2 měsíce).
Rizika jsou často překvapení, přináší problémy.	Řízen riziky – nejrizikovější věci řešíme nejdříve.
Integrace a testování až na konci.	Průběžná integrace a testování.
Změny nejsou vítány.	Počítá se změnami, přijímá je.
Často zaměřen na tvorbu dokumentů bez přidané hodnoty a jejich revize.	Zaměřen na fungující SW (hodnota pro zákazníka).
Buildy a testy až na konci, často přeskočeno nefunkční testování.	Automatizované buildy a testy.
Za kvalitu odpovědní pouze testeři, QA manažeři nebo často nikdo.	Všichni (celý tým) odpovědní za kvalitu produktu.

Tabulka 6-1: Vodopádový vs. Iterativní přístup

6.3 ISO/IEC 12207 a další standardy

V oboru informačních technologií také samozřejmě existují standardy, jedním z nich je standard ISO/IEC 12207 (viz [ISO]), který definuje proces životního

cyklu software. ISO/IEC 12207 popisuje důležité komponenty procesu vývoje SW a obecné souvislosti, které určují vzájemné působení těchto komponent. Popisuje celý proces od konceptualizace, až po ukončení provozu (tzv. retirement). Norma definuje tři kategorie procesů:

- Primární procesy – akvizice, dodávka, vývoj, provoz, údržba.
- Podpůrné procesy – dokumentace, řízení konfigurací, zajištění kvality, ověřování, potvrzení, společné revize, audit, řešení problémů.
- Organizační procesy – řízení, infrastruktura, zdokonalení, trénink.

ISO 12207 definuje proces, který je chápán jako modulární a adaptovatelný pro různé typy projektů. Je založen na dvou principech: modulárnost a odpovědnost. Dalším příkladem podobně založeného procesu je např. DoD, standard amerického ministerstva obrany pro tvorbu softwarových projektů založený na vodopádu.

6.4 Architektury informačních systémů

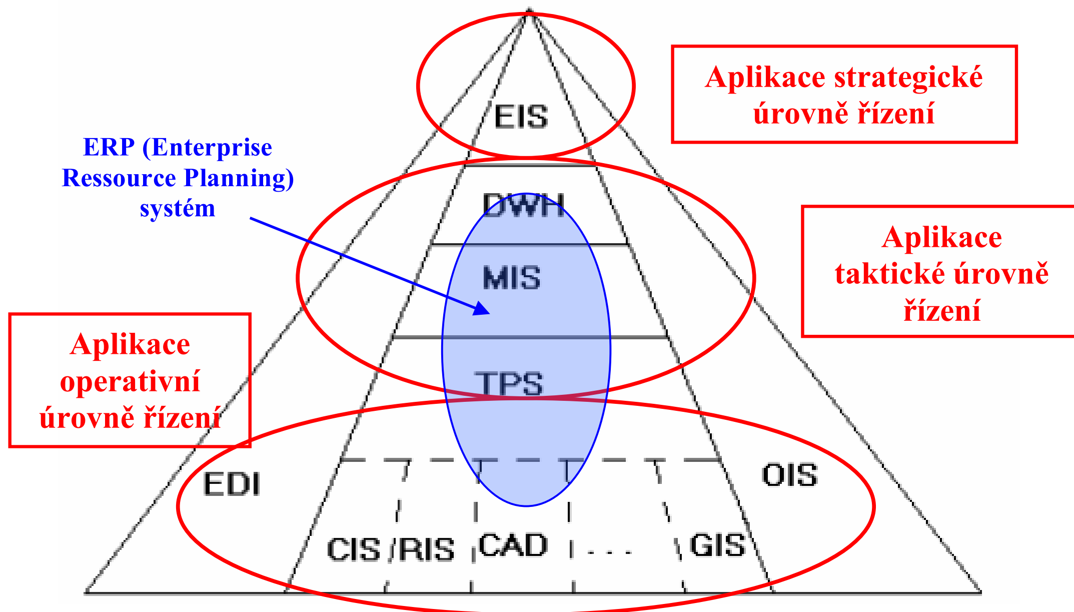
Architektura tvoří klíčový prvek řízení IS, z něhož pak vycházejí detailní analytické i plánovací charakteristiky celého IS. Architektura musí respektovat strategii podniku a IT strategii, podnikové cíle a cíle IS. Do architektury se musí promítat stav a rozvoj produkčních a řídících aktivit a odpovídajících zdrojů. Podstatou a účelem architektury informačního systému je podpora následujících vlastností: strategická orientace, pokrytí uživatelských požadavků, integrovatelnost, otevřenost, jednoduchost, flexibilita, udržovatelnost, efektivní provozuschopnost, viz [Do97]. Architektura IS vyjadřuje celkovou vizi. Je oproštěna od veškerých detailů, vychází ale z pochopení ekonomických, výrobních a obchodních cílů, které organizace sleduje. Musí být jednoduchá a srozumitelná, je to jakýsi skelet, na který se navěšují další funkce systému.
Neexistuje-li architektura IS, můžeme se setkat s těmito problémy:

- Nepokryté požadavky na funkce IS, jiné funkce jsou naopak zbytečné (jaksi navíc).
- Schází potřebné nástroje – tvorba v málo výkonném prostředí, bez CASE, bez specialistů na tvorbu IS, na počítačové sítě, na databáze, …
- Časté přestavby z důvodů množících se požadavků uživatelů.
- Draze nakupený SW nepoužitelný z důvodů kompatibility SW nebo HW a nepoužití standardů (rozdílné uživatelské rozhraní aplikací, různé databázové systémy, …) → poruchy, údržba bez řádné dokumentace.

Architektura musí být postavena tak, aby respektovala dynamiku změn v procesech a zdrojích a promítat je do navazujících aktivit řízení informačního systému.

Následující obrázek zobrazuje možný příklad vizualizace globální architektury podnikového IS. Z modelu můžeme vidět jednotlivé typy aplikací využívané na různých hierarchických úrovních organizace. V podstatě to znamená, že vysocí manažeři firmy budou využívat jiné aplikace (trendy trhů a produktů, trendy poptávky, počty a dopad chyb našeho produktu pomocí tzv. BI aplikací) než dělnické profese (systémy pro plánování výroby, zákaznické či objednávkové

systémy apod.). Tento typ modelu také vizuálně říká, kolik dat existuje a je produkováno kterou vrstvou a kolik lidí daný typ aplikací využívá.



Obr. 21 Příklad architektury podnikového systému

EIS (Executive IS) – podporuje vrcholové řízení organizace (strategie podniku, finanční řízení, trendy na trzích a v poptávce po našich produktech).
DWH (Data warehouse) – datový sklad, podpora řízení na základě analýz rozsáhlých dat.
MIS (Management IS) – podpora taktické a operativní úrovně řízení (účetnictví, nákup, prodej, sklad, …).
TPS (Transaction processing system) – bezprostředně spojený s typem provozu v rámci dané organizace (systémy bezprostředně podporující dílenské, skladové, transportní operace výrobních podniků, rezervační systémy dopravních společností, zákaznické systémy energetických společností).
CIS (Customer IS) – zajišťuje bezprostřední styk se zákazníkem (odečty spotřeby energie, fakturaci na zákazníka, …).
RIS (Reservation IS) – rezervační systémy v dopravních organizacích, cestovních kancelářích.
GIS (Geographic IS) – podpora kreslení a vyhodnocování map, tvorba územních modelů.
CAD (Computer aided design) – konstrukční a návrhářské práce v průmyslu, počítačová podpora návrhu výrobku.
CAM (Computer aided manufacturing) – automatizovaná podpora řízení výrobních provozů.
OIS (Office IS) – podpora rutinních kancelářských prací (elektronická pošta, správa a zpracování dokumentů).
EDI (Electronic data interchange) – podporuje elektronickou výměnu dat mezi obchodními partnery, bankami, ústavy, apod.

Příklad z obrázku ukazuje ***architekturu informačního systému pro výrobní podnik*** (resp. aplikace používané na jednotlivých vrstvách). Pokud budeme uvažovat architekturu informačního systému například ***banky***, pak bude

strategická a taktická úroveň prakticky stejná (manažerské reporty či účetnictví potřebuje banka stejně jako výrobní firma a manažery na této úrovni zajímají stejné makroekonomické údaje), pouze aplikace na operativní a částečně na taktické úrovni se budou lišit. Místo řízení a plánování výroby či CAD systémů bude mít banka zákaznické systémy, systémy na podporu investování, elektronické bankovnictví apod.

Pokud se bavíme o architektuře informačního systému, musíme zmínit další pohledy a vrstvy. Obr. 21 znázorňuje jednotlivé typy aplikací na jednotlivých vrstvách řízení organizace. Dále však existuje rozdělení z jiného pohledu. Jedná se o vrstvu prostředí, aplikační a technologickou:

- Vrstva prostředí reprezentuje ekonomické prostředí, legislativu, organizační strukturu, personální kapacity a jejich kvalifikace, zkušenosti v IT a motivaci pro IT.
- Vrstva aplikační pokrývá provozované a řešené projekty, jejich dokumentace, funkční a datové specifikace, organizační pravidla jejich řešení a provozu, aplikační SW.
- Vrstva technologická pokrývá návrh a provoz počítačových sítí, vymezení jednotlivých komponent IT, což představuje základní software, technické prostředky včetně jejich vazeb a vnitřní struktury.

6.4.1 Globální architektura informačního systému

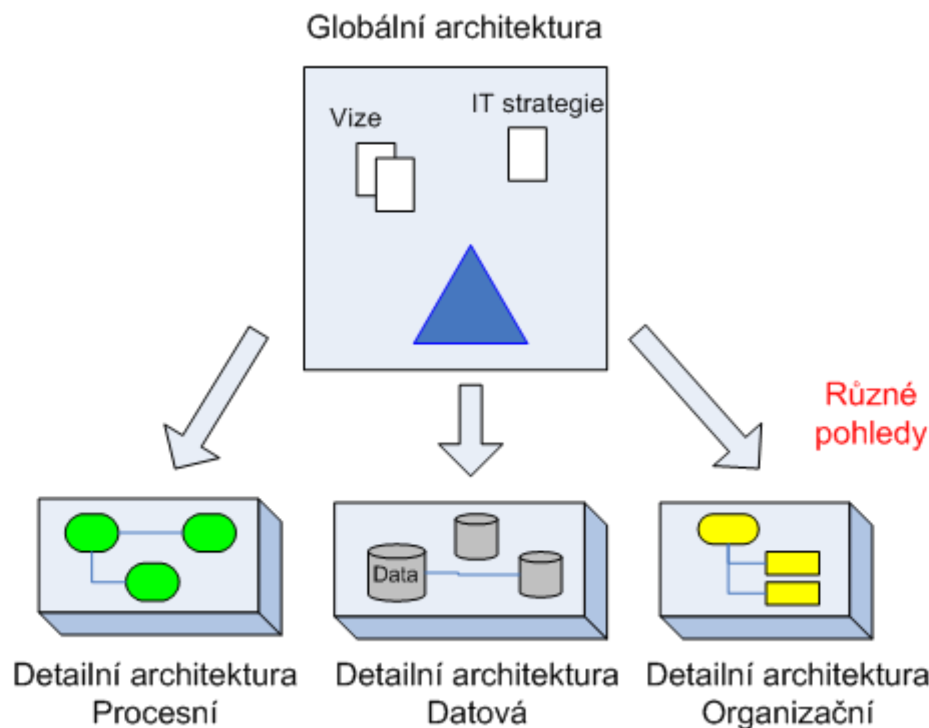
Globální architektura je hrubý návrh celého IS/IT. Je to vize budoucího stavu. Zachycuje nejen jednotlivé komponenty IS/IT a jejich vzájemné vazby. Globální architektura je složena z tzv. bloků. Blok je množina informačních služeb, funkcí, které slouží k podpoře podnikových procesů (jednoho nebo více). Jsou to vlastně hlavní úlohy odpovídající optimalizovanému uspořádání procesů a zdrojů. Můžeme také říci, že jsou to množiny pro různé uživatelské skupiny – partneři, zákazníci, zaměstnanci, veřejnost, apod. Příkladem těchto bloků jsou EIS, DWH, MIS, TPS, … (viz Obr. 21 výše).

6.4.2 Dílčí architektury informačního systému

Jedná se o detailní návrh IS z hlediska různých dimenzí. Určení obsahu těchto dimenzí IS/IT:

- Funkční – funkční struktura, náplň jednotlivých funkcí.
- Procesní – vymezení klíčových procesů a vazeb v IS/IT, (kontextový diagram, EPC diagramy, diagramy toků dat – DFD, síťové diagramy).
- Datová – určení datových objektů a zdrojů v rozlišení na interní a externí zdroje, návrh datových entit, databázových souborů a jejich uložení.
- Softwarová – rozlišení na ASW, ZSW nebo systémový SW.
- Technická – postihuje celý komplex prostředků počítačové a komunikační techniky.
- Organizační – zahrnuje organizační strukturu a vymezení organizačních jednotek.
- Personální – zahrnuje profesní a kvalifikační struktury.

Každá z těchto dimenzí je popsána svými atributy (identifikace, název, klíčové problémy, …). Součástí modelu řízení IS/IT (v návaznosti na architekturu) je i analýza a plánování všech podstatných vazeb mezi dimenzemi.



Obr. 22 Globální a detailní architektury a jejich vztah

Na závěr jen upozorníme, že pojmy architektura informačního systému a architektura software se dost liší. Rozdílné jsou už proto že informační systémy a software mají jiný záběr, rozměr. Jak jsme si již řekli výše, pokud mluvíme o IS, máme na mysli i veškeré organizační aspekty, okolí, hardware, lidskou složku atd., nejen operační systém a vlastní software. Kdežto pokud mluvíme o architektuře SW, jedná se pouze o podmnožinu (vrstvy software, použité technologie na těchto vrstvách, způsob komunikace mezi nimi).

Více o architekturách informačních systémů lze nalézt například v Dohnal, J., Pour, J.: Architektury informačních systémů [Do97].

Kontrolní otázky:

1. Co je to metodika?
2. Co je to architektura IS?
3. Co je to informační systém?
4. Co je to softwarové inženýrství?
5. Jaké jsou problémy projektů tvorby SW?
6. Vyjmenujte základní rozdíly mezi vodopádem a iterativním vývojem.

Úkoly k zamyšlení:

V kapitole byl uveden základní rozdíl mezi iterativním a vodopádovým přístupem. Zamyslete se nad jedním z bodů, jímž je automatizované buildování

(sestavení aplikace ze zdrojových kódů) a testování. Jaké přínosy toto může mít pro roli vývojáře/programátora?

Korespondenční úkol:

Zmínili jsme, že v iterativním způsobu vývoje netvoříme na začátku detailní plán celého projektu, ale pouze jakousi road map. Pokuste se zamyslet nad obsahem takového dokumentu celkového plánu projektu, jelikož nějaký celkový plán je určitě nutné vytvořit. Co (časy, zdroje, milníky, ...) by podle Vás měl takový plán obsahovat a proč?

Shrnutí obsahu kapitoly

V této kapitole jsme stručně zopakovali pojmy z oblasti vývoje software. Zmínili jsme také problematiku architektur informačních systémů. Na závěr kapitoly jsme představili standardy pro vývoj software a také hlavní rozdíly mezi iterativním a vodopádovým způsobem vývoje.

7 Specifikace požadavků

V této kapitole se dozvíte:

- Co je to proces specifikace požadavků.
- Různé přístupy ke specifikaci požadavků.
- Co jsou to use case?
- Jaké máme standardy?

Po jejím prostudování byste měli být schopni:

- Porozumět základním principům specifikace požadavků software.

Klíčová slova této kapitoly:

Specifikace požadavků, SRS, use case, user stories, story boards, FURPS.

Doba potřebná ke studiu: 4 hodiny

Průvodce studiem

Kapitola zmiňuje principy tvorby specifikace požadavků, a to jak proces samotný (jak bychom měli postupovat), tak různé formy zápisu požadavků. Jako jeden z důležitých podpůrných faktorů jsou zmíněny také nástroje na podporu zachycení požadavků na SW.
Na studium této části si vyhraďte 4 hodiny.

Prvním krokem při vývoji SW je ujasnění si toho, co vlastně máme vytvořit. Zásadní chybou je často tvorba něčeho, aniž bychom věděli čeho. Prvním krokem je tedy definice potřeb zákazníka, jeho problémů a následná identifikace účastníků (tzv. stakeholders) s jejich požadavků. Proč se zabýváme touto disciplínou podrobněji, než disciplínami jinými? Hlavním důvody jsou následující:

- Podle Chaos reportu organizace Standish Group jsou požadavky jedním z *přispivatelů k problémům* softwarových projektů. Průzkum z roku 1994 a 1997 na 1. místě. V roce 2001 a 2006 na 2. místě!
- Výzkumy a zkušenosti hodnotí *měnící se požadavky* jako jednu z příčin eúspěšných projektů.
- Pouze 20% implementovaných požadavků je opravdu potřeba, 45% nikdy nepoužito! (Zdroj: Chaos report z roku 2001 [Sta02]).

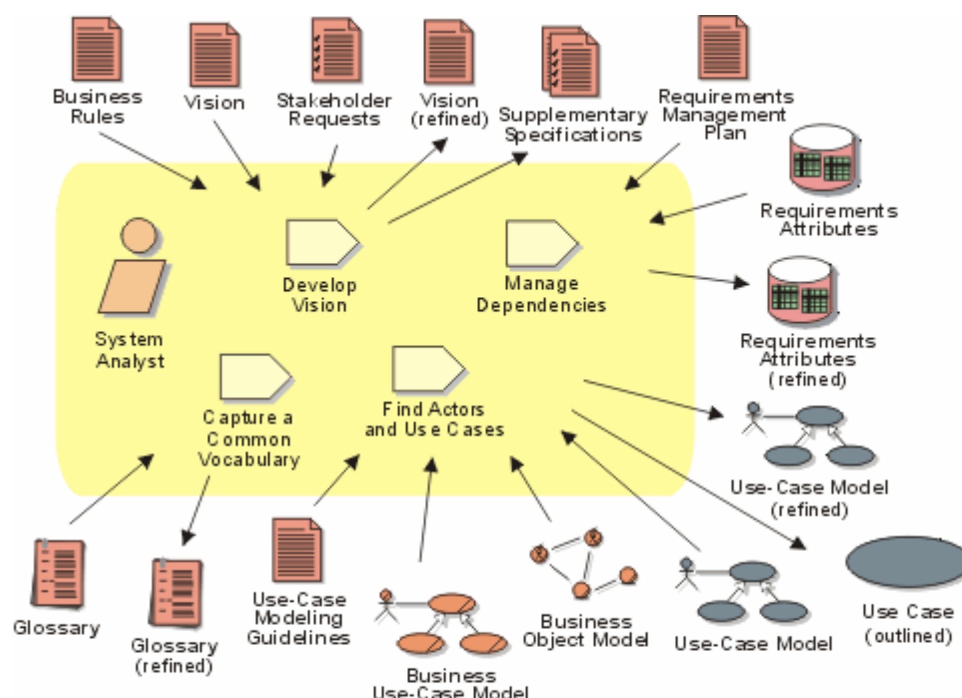
Obecně můžeme říci, že existují dva základní druhy požadavků na systém:

- Funkční požadavky – popisují, co má systém dělat, jeho budoucí chování (use case systému).
- Nefunkční požadavky – popisují další vlastnosti systému, které však nejsou funkčnostmi, laicky řečeno je nenajdete v menu aplikace (jedná se např. o možnosti přístupu k systému z více míst, maximální přístupovou dobu, počet operací za čas, implementované standardy, ..).

Poslední pojem na úvod, který osvětlíme je SRS. Tato zkratka znamená Software Requirements Specification, česky: specifikace softwarových

požadavků. Jedná se o dokument, či elektronický artefakt, který zachycuje požadavky. Forma a způsob uchování tedy není tolik důležitá.
Obecný postup při specifikaci požadavků na budoucí software je následující:

1. Pochopení problému a jeho analýza – provádíme v problémové doméně, cílem je porozumět problému a nalézt k němu řešení.
2. Identifikace všech se zájmem na projektu (tzv. stakeholders) – každý může mít jiný zájem, příliš mnoho stakeholderů s rozdílnými zájmy může způsobit neúspěch projektu.
3. Definice systému (scope) a jeho hranic (boundaries) – společně se stakeholdery, co ještě bude systém za problém řešit a co už ne, jaké budou jeho přibližné rysy (features).
4. Identifikace omezení, který musí systém mít – použití technologie, vazby na systémy, bezpečnostní požadavky, apod.



Obr. 23 Příklad definice systému podle RUP – role, aktivity, artefakty

Způsobů získávání požadavků je několik. Můžeme využít IT strategii firmy, stejně jako provést analýzu existujícího systému. Pokud vytváříme systém nový, je třeba tyto představy získat od budoucích uživatelů. Toto je zdrojem problémů, jelikož lidský mozek je dobrý v popisu hmatalených věcí, tj. Při hře s existujícím produktem, předmětem. Naopak špatní jsme v momentě, kdy si máme něco pouze představit a tuto představu pak kompletně popsat. Jedním z řešení je tedy aplikace principy iterativního vývoje, kdy v častých intervalech ukazujeme zákazníkovi, co jsme vytvořili. Mezi způsoby, jak identifikovat můžeme zařadit:

- Rozhovory s vybranými uživateli – může být ve formě připravených otázek, které přesně dodržujeme nebo ve formě volného rozhovoru.
- Requirements workshop – jedná se o časově omezenou schůzku, kdy například formou brainstormingu generujeme možné požadavky. Tento

workshop je řízen byznys analytikem, který vede směr úvah tam, kam potřebuje.

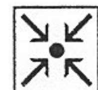
- Prototyp – hrubý nástřel rozhraní (HTML, GUI)
- User stories + post-it lístečky – kreslený vzhled GUI a jeho přetváření pomocí nalepovacích štítků (tzv. Post-it).

7.1 Tradiční přístup

Tradiční přístupy identifikovaly veškeré požadavky na budoucí systém na úvod projektu, posléze je „zmrazily" a postupně analyzovaly, vytvořily architekturu, implementaci, nakonec bylo řešení integrováno a pokud zbyl čas, tak i testováno. Tento přístup způsoboval řadu problémů:

1. Požadavky byly ve formě detailních popisných vět (viz ukázka níže) bez návazností, hierarchií, celkového pohledu, což způsobovalo:
    a. Rozdrobení požadavků na vymezené funkčnosti a ztráta celkového přehledu (tzv. big picture).
    b. Možná kontradikce mezi požadavky (jak víme a ověříme, že požadavek FREQ-0311 na straně 22 není v kontradikci s požadavkem FRE-Q2057 na straně 117?)
    c. Nejasné či nečitelné vazby, hierarchie, vztahy mezi požadavky.
2. Změna, která přináší zákazníkovi přidanou hodnotu, šla jen velmi težko implementovat v průběhu již započatého vývoje.
3. Popis funkčností byl tvořen z pohledu systému, ne z pohledu uživatele, který ho bude používat.

Následující tabulka ukazuje velmi stručný výřez takovéto specifikace.



Kód požadavku	Popis	Priorita
FREQ-001	Systém ověří platnost uživatelského jména a hesla.	Musí mít
FREQ-002	Systém umožní vložit položku do košíku.	Musí mít
FREQ-003	Systém umožní editovat obsah košíku.	...
...	...	...
FREQ-713		Může mít

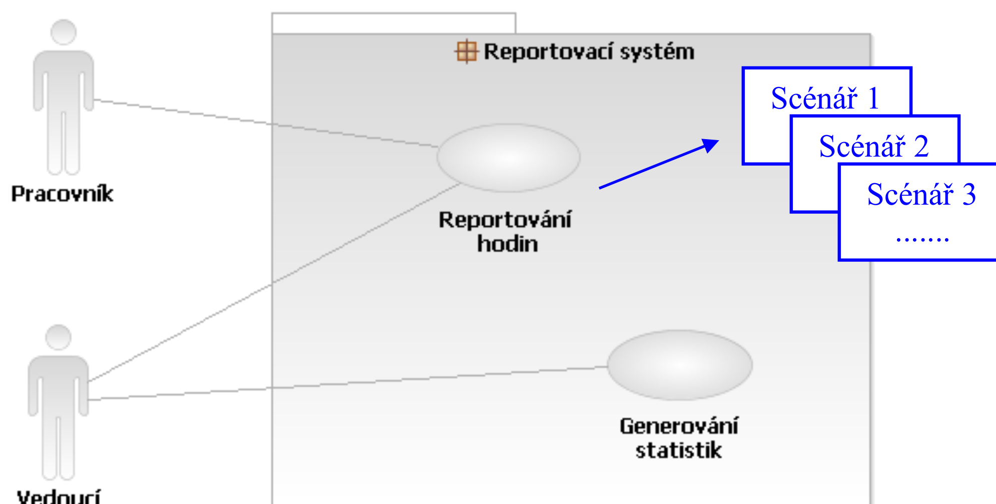
Tabulka 7-1: Funkční specifikace požadavků

V následujícím textu stručně představíme modernější a efektivnější přístupy k definici požadavků, konkrétně to jsou use case (česky někdy zvené případy užití či modely jednání), FUPRS+ přístup a také standard pro specifikaci funkčních požadavků IEEE 830.

7.2 Use Case

Začněme technikou zvanou Use Case, česky někdy nazývány případy užití. Jak jste mohli vidět v kapitole věnující se životnímu cyklu, je RUP „Use Case driven" – řízen Use Casy. To znamená, že vývoj, analýza, plánování, testování je prováděno podle use case, nebo jsou tyto v průběhu znovupoužity. Jak již víme, tato grafická technika slouží k identifikaci požadavků na systém. Na rozdíl od klasické specifikace reprezentované seznamem požadavků, je tento přístup uživatelsky orientován. Ukazuje, co který uživatel (role) od budoucího systému očekává a jakým způsobem ho používá.

Tvorba Use Casů začíná identifikací aktorů – budoucích přímých uživatelů systému. S jejich pomocí poté identifikujeme očekávané vlastnosti a chování systému na abstraktní úrovni.



Obr. 24 Use Case reportovacího systému a scénáře jednoho UC na dokreslení kontextu

Use Casy popisují chování, které očekává uživatel od vytvářeného systému. Toto chování – Use Case – je pak popisováno scénáři, jedná se o mírně formalizovaný (strukturovaný) příběh, který popisuje, jakým způsobem uživatel využívá konkrétní funkčnost systému.

Šablona pro detailní popis Use Case [Kr03a]:

Introduction:	Short introduction to the specification
Use Case Description:	Brief description conveys the purpose of the Use Case
Pre-condition:	The state that must be present when a use case may start
Flow of Event:	Description of the dialog between actor and system
Basic Flow:	The "Happy day scenario"
Alternative Flows:	Exception and alternatives
Special Requirements:	Nonfunctional requirements specific to the Use Case
Post-condition:	Possible states after at use case has finished
Extension Points:	Definitions of locations of extension points
References:	List of all references

Příklad jednoduchého use case pro námi zachycený reportovací systém může být následující:

Název: Reportování hodin
Aktor: Zaměstnanec
Počáteční podmínka: Zaměstnanec odpracoval určitou dobu na projektu
Tok událostí (basic flow):

1. Zaměstnanec vybere správu projektů.
2. Systém zobrazí správu projektů.
3. Zaměstnanec vybere projekt na kterém pracoval.

4. Zaměstnanec zadá počet hodin a datum.
5. Systém ověří zadaná maximální počet hodin a datum.
6. Zaměstnanec odešle zapsaná a ověřená data.
7. Systém uloží data.

Alternativní toky:

4a. …

Můžete si všimnout, že jsme uvažovali pouze scénář, kdy je vše v pořádku, nebrali jsme v potaz potenciální problémy při špatně zadaném datu (novější než dnes), při více zadaných hodinách atd. Těmito stavy se zabývají alternativní a chybové toky.

7.2.1 Chyby při tvorbě Use Case

Běžné chyby, se kterými se lze setkat při tvorbě Use Case, jsou následující:

- *1) Příliš mnoho UC a jejich funkční dekompozice* – způsobeno nepochopením použití UC, UC jsou funkčně dekomponovány, což je chybný přístup. Jako UC vystupují jednotlivé scénáře a ne jejich obálka. Tím ztrácíme celkový pohled na daný příběh a hlavně vazby mezi jednotlivými scénáři, což způsobuje špatnou interpretaci požadavku a možné redundance kódu (stejná věc psána 2x pro oba scénáře).

- *2) CRUDL use casy a scénáře* – nevhodně formulované use casy či jejich scénáře, které nenásledují kroky, které vykonává uživatel ale spíše kopírují databázové operace (Create, Read, Update, Delete, List). Takové UC a scénáře jsou často produktem programátorů.

  Příklad chybného použití bodu 1 a 2: místo našeho use case „*Reportování hodin*" na Obr. 24 je funkční dekompozice chování systému na více use casů následující:
  - vložení hodin,
  - zobrazení vložených hodin,
  - aktualizace vložených hodin,
  - výmaz vložených hodin

  Use Case je příběh, který popisuje celek používaný uživatelem formou příběhu s několika alternativními toky (místo vkládání, editace apod.)

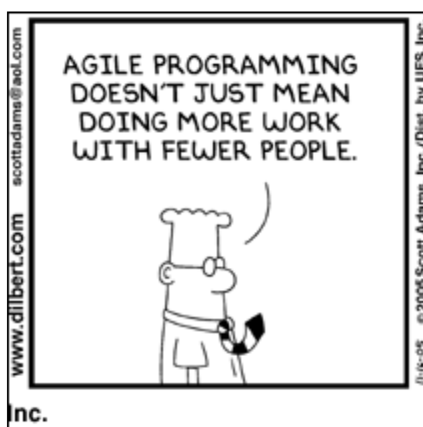
- *3) Zahrnutí návrhových rozhodnutí (technologických pojmů)* – tato chyba nás nutí dělat návrhová rozhodnutí unáhleně, předčasně. Návrhová rozhodnutí odkládáme do nejzazšího možného termínu, jelikož mají vliv na architekturu a mohou ovlivnit spoustu věcí, resp. způsobit spoustu problémů.

  Příklad chybného použití, bod 3: Popis toku scénáře obsahuje pojmy jako: Klikneme na tlačítko, vybereme z databáze, vyhledávací klíč, seznam s rolovací lištou zobrazený červeně apod.

Jak use case řídí vývoj, jak podle nich plánujeme, jak je realizujeme či jak je můžeme znovupoužít pro dokumentaci či testy bylo naznačeno v předchozí kapitole. Blíže k detailnímu popisu a tvorbě Use Case a také ke specifikaci požadavků viz [Cockburn05].

7.3 Agilní přístupy

Jelikož je dnes také pro vývoj software používáno agilních přístupů a většinou úspěšněji než vodopádových přístupů, řekneme si pár slov i o způsobu specifikace požadavků pomocí tzv. user stories. Tento přístup je využíván například v XP (extrémní programování Kenta Becka) či ve strumu (Schwaber, Sutherland). Více o samotných metodách např. viz skripta „Informační systémy 2“, nebo česky psaná kniha [Ka04], či anglická literatura [Be04], [Sch04].



Obr. 25 Dilbert zobrazující klasické nepochopení Agile (Zdroj: www.dilbert.com)

User stories jsou velmi názorné a mocné formy psaní požadavků software. Stejně jako use case popisují budoucí funkčnost z pohledu uživatele, což je velmi důležité. Lidé mají rádi příběhy, proto je tato forma požadavku srozumitelnější, navíc popisuje systém přesně tak, jak ho daný uživatel používá. Pro user stories platí několik zásad:

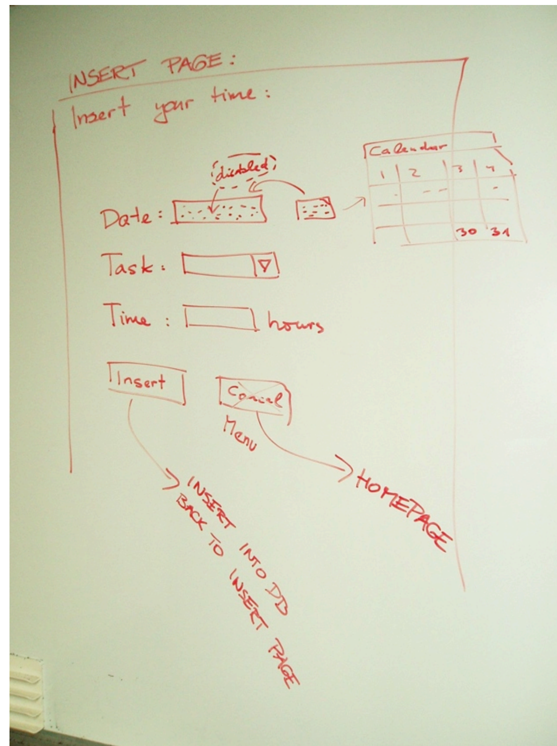
1. Jsou psány textovou formou.
2. Je používáno názvů a pojmů běžných pro řeč uživatele (byznys názvosloví).
3. Jsou psány na malé kartičky či papírku, aby nenarostly do velkých rozměrů.
4. Jsou psány zákazníkem.

Příklady user stories:

Student si může zakoupit měsíční parkovací pass online.

Parkovací pasy lze platit pomocí kreditní karty.

Parkovací pasy lze platit pomocí systému PayPal ™.



Obr. 26 Story board

User stories mohou a často v Agile bývají doplněny tzv. story boards (viz předchozí obrázek). Tyto náčrtky zobrazují možné budoucí GUI a jeho funkce. Pomocí nich jsme schopni si s uživateli vyjasnit spoustu nejasností týkajících se fungování aplikace a jejího vzhledu.

Bohužel user stories mají několik problémů, resp. Je mohou způsobovat:

- User stories oproti use cas postrádají celkový pohled (popisují jen 1 story, ne celý UC a propojení jednotlivých scénářů jako UC) a mohou způsobit špatnou interpretaci developerem.
- UC realizace chybí – formalizovaný mezikrok mezi vágním požadavkem a kódem.

7.4 FURPS+

Tento přístup popisuje jaké požadavky bychom neměli opomenout, spíše, než formu jejich zápisu. Use case jsou zaměřeny hlavně na zachycení funkčních požadavků. Pro vytvoření stabilní architektury je, jak již víme, potřeba uvažovat také nefunkční požadavky. Tento systém klasifikace požadavků z pohledu architektury navrhovaného systému byl vytvořen Robertem Grady ve společnosti Hewlett-Packard. Oblasti, které daný systém klasifikace požadavků uvažuje jsou následující (název je odvozen od počátečních písmen anglických slov):

- **F**unctionality (funkcionalita)
- **U**sability (použitelnost)
- **R**eliability (spolehlivost)
- **P**erformance (výkonnost)

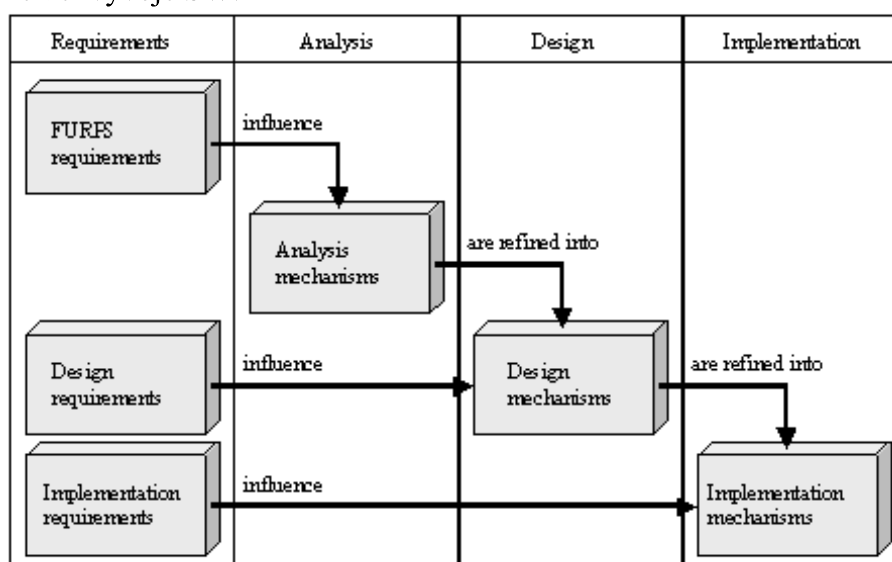
- Supportability (podporovatelnost)
- + znamená, že bychom neměli zapomenout ani na návrhové, implementační a fyzické požadavky.

Následující tabulka ukazuje architektonicky významné funkční požadavky:

Function	Description
Auditing	Provide audit trails of system execution.
Licensing	Provide services for tracking, acquiring, installing, and monitoring license usage.
Localization	Provide facilities for supporting multiple human languages.
Mail	Provide services that allow applications to send and receive mail.
Online help	Provide online help capability.
Printing	Provide facilities for printing.
Reporting	Provide reporting facilities.
Security	Provide services to protect access to certain resources or information.
System management	Provide services that facilitate management of applications in a distributed environment.
Workflow	Provide support for moving documents and other work items, including review and approval cycles.

Tabulka 7-2: Architektonicky významné funkční požadavky (zdroj: IBM Rational Edge)

Následující postup ukazuje způsob transformace FURPS požadavků do výsledného produktu. Upozorníme jen, že daný obrázek nezobrazuje vodopádový model, nýbrž jednotlivé disciplíny podle RUP, tj. vlastně jednu iteraci vývoje SW.



Obr. 27 Transformace požadavků FURPS do výsledného produktu, swimlanes reprezentují disciplínu, ne fázi vodopádu (zdroj: Rational Edge)

7.5 Standard IEEE 830

Jelikož jsme v oblasti vývoje software, existuje samozřejmě spousta doporučení a standardů. Pro oblast specifikace softwarových systémů si představíme standard, respektive doporučení IEEE 830. Tento přístup, stejně jako FURPS+, popisuje jaké požadavky bychom neměli opomenout, spíše, než jejich formu zápisu. Plný název normy IEEE 830 zní „IEEE Recommended Practice for Software Requirements Specification“.

Dokument IEEE 830 obsahuje:

- Zaměření (scope) tohoto doporučení, jímž je popis praktik SRS za účelem vývoje SW, částečně také za účelem výběru SW.
- Odkazy na relevantní standardy IEEE (např. IEEE 610.12 – Standard Glossary of Software Engineering Terminology; IEEE 1042 – Guide to SW Configurations Management).
- Definice používaných pojmů jako je kontrakt, zákazník, dodavatel, uživatel.
- Čím se zabývat v SRS, tj. funkcionalita, externí rozhraní, výkon, atributy, návrhová omezení.

Součástí dokumentu je také doporučení týkající se formy a obsahu vlastní specifikace (SRS). Konkrétně je zmíněno a vysvětleno, že dokumentace musí být correct (přesná), unambiguous (jednoznačná), complete (kompletní), consistent (konzistentní), ranked for importance (ohodnocená podle důležitosti), verifiable (ověřitelná), modifiable (přizpůsobitelná), traceable (sledovatelná).

Následující struktura zobrazuje příklad náplně dokumentu specifikace požadavků (SRS) podle IEEE 830:

Table of Contents

1. Introduction
   - 1.1 Purpose
   - 1.2 Scope
   - 1.3 Definitions, acronyms, and abbreviations
   - 1.4 References
   - 1.5 Overview
2. Overall description
   - 2.1 Product perspective
   - 2.2 Product functions
   - 2.3 User characteristics
   - 2.4 Constraints
   - 2.5 Assumptions and dependencies
3. Specific requirements (See 5.3.1–5.3.8 for explanations of possible specific requirements. See also annex A for several different organizations of this section of the SRS.)

Appendixes

Index

Šablona pro specifikaci požadavků, různé druhy pohledů na popis systému:

A.1 Template of SRS section 3 organized by mode: Version 1

3 Specific requirements
- 3.1 External interface requirements
  - 3.1.1 User interfaces
  - 3.1.2 Hardware interfaces
  - 3.1.3 Software interfaces
  - 3.1.4 Communications interfaces
- 3.2 Functional requirements
  - 3.2.1 Mode 1
    - 3.2.1.1 Functional requirement 1.1
    - .
    - .
    - .
    - 3.2.1.*n* Functional requirement 1.*n*
  - 3.2.2 Mode 2
  - .
  - .
  - .
  - 3.2.*m* Mode *m*
    - 3.2.*m*.1 Functional requirement *m*.1
    - .
    - .
    - .
    - 3.2.*m*.*n* Functional requirement *m*.*n*
- 3.3 Performance requirements
- 3.4 Design constraints
- 3.5 Software system attributes
- 3.6 Other requirements

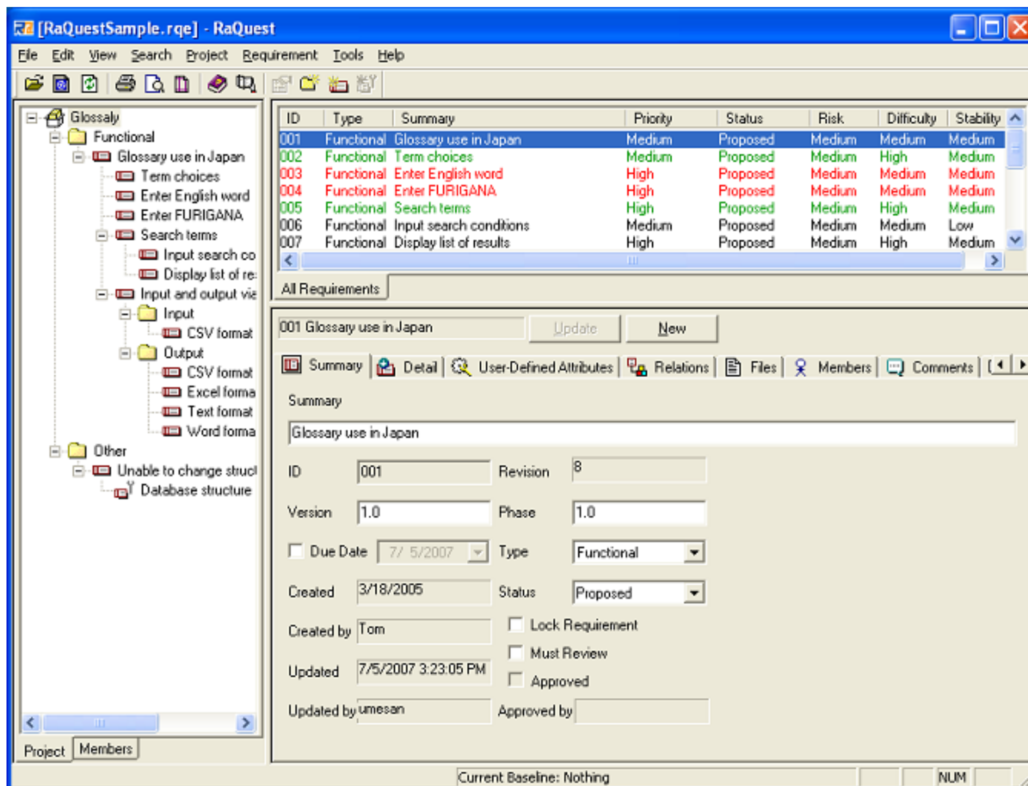
7.6 Nástroje

Velmi důležitým bodem specifikace požadavků je podpora nástroji. Mezi základní nástroje, které by měly podporovat tuto disciplínu patří hlavně CASE nástroje, částečně pak také IDE nástroje. Jaké funkčnosti bychom měli od těchto systémů očekávat? Nejdůležitější jsou následující:

- Automatizace a usnadnění činností.
- Kontrola konzistence modelů, požadavků.
- Tracebility (a to až na úroveň kódu).
- Znovupoužití (reuse) pro testy, dokumentaci, plánování iterací (v případě UC přístupu).
- Možnost generování GUI (není nejpodstatnější).

Tuto podporu by měly poskytovat hlavně CASE nástroje z kategorie upper CASE (např. Rational RequisitePro). Více o CASE nástrojích, jejich rozdělení, funkcích viz jedna z následujících kapitol.

Upozorníme na jednu důležitou věc, nástroje musí podporovat proces vývoje software, který jsme ve společnosti implementovali. Jinak není možné jeho efektivní použití, budeme nuceni vytvářet zbytečné artefakty nebo nebudeme naopak moci vytvořit potřebné. Navíc výkonné nástroje neodstraní špatný či neexistující proces identifikace a definice požadavků (tzv. Requirements Management)!

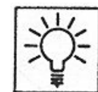


Obr. 28 Příklad systému pro správu požadavků

Kontrolní otázky:

1. Co je to proces identifikace a správy požadavků?
2. Jaké znáte metody či přístupy pro jejich zachycení?
3. Jaké nevýhody mohou provázet použití user stories?
4. Jak by měly podporovat nástroje proces správy požadavků?

Úkoly k zamyšlení:



Pokuste se zamyslet nad tím, jaké problémy při vývoji software může způsobit neexistence CASE nástroje, které dané činnosti automatizuje a umožňuje automatickou kontrolu. Je jeho použití nutnost? Nestačí mnohdy tabule či tužka a papír?

Korespondenční úkol:

V korespondenčním úkolu jste se zamysleli nad tím, jaké problémy při vývoji software může způsobit neexistence CASE nástroje. Nyní se zamyslete nad celým procesem identifikace a zaznamenání požadavků. Co za problémy (obecně v celém vývoji) můžeme očekávat, pokud tento proces neexistuje, není následován nebo je nevalně definovaný?

Shrnutí obsahu kapitoly

V této kapitole jsme stručně zmínili principy tvorby specifikace požadavků, a to jak proces samotný (jak bychom měli postupovat), tak různé formy zápisu požadavků: tradiční přístup, use case přístup, FURPS+, IEEE 830 či agilní formu jejich dokumentace a identifikace. Jako jeden z důležitých podpůrných faktorů jsou zmíněny také nástroje na podporu zachycení a dokumentace požadavků na SW.

8 Životní cyklus vývoje IS

V této kapitole se dozvíte:

- Co jsou to principy RUPu?
- Jaké jsou základní principy RUPu?
- Co tyto principy zdůrazňují?
- Před čím varují?

Po jejím prostudování byste měli být schopni:

- Porozumět základním principům iterativního vývoje řízeného riziky.

Klíčová slova této kapitoly:

Principy RUP, iterativní vývoj, iterace, fáze, řízení rizik, kvalita, UML.

Doba potřebná ke studiu: 6 hodin

Průvodce studiem

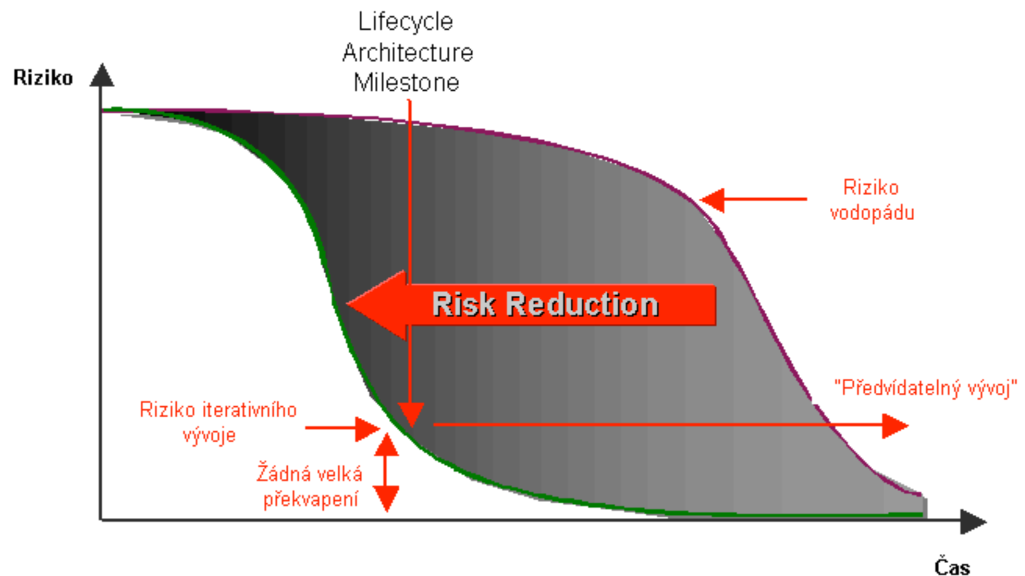
Kapitola popisuje iterativní, riziky řízený vývoj podle RUP, podrobně se věnuje popisu jednotlivých fází RUP. V neposlední řadě je stručně představeno UML a jeho místo ve vývoji podle RUP.
Na studium této části si vyhraďte 6 hodin.

Iterativní vývoj je vystavěn na několika základních principech, které si nyní představíme. Některé z nich již byly nastíněny také v předchozí kapitole, konkrétně se tedy jedná o:

- Snahu atakovat rizika projektu co nejdříve a neustále.
- Ujištění se, že dodáváme zákazníkovi přidanou hodnotu.
- Zaměření na spustitelný software.
- Zapracovat změny v časných fázích projektu.
- Brzké nastínění spustitelné architektury.
- Znovupoužití existujících komponent.
- Úzká spolupráce, všichni jsou jeden tým.
- Kvalita je způsob provádění celého projektu, nejen část (testování).

Tyto body jsou obsaženy také v klíčových principech (key principles) RUPu stejně jako v OpenUP, jedná se o:

a) Přizpůsobte proces potřebám projektu (Adapt the process),
b) Vyvažujte vzájemně si konkurující požadavky všech zúčastněných na projektu (Balance competing stakeholders priorities),
c) Spolupracujte napříč týmy (Collaborate across teams),
d) Demonstrujte hodnotu v několika iteracích (Demonstrate value iteratively),
e) Pracujte s úrovní abstrakce (Elevate the level of abstraction),
f) Zaměřte se na kvalitu (Focus on quality).



Obr. 29 Úroveň rizik pro různé přístupy vývoje (zdroj: RUP)

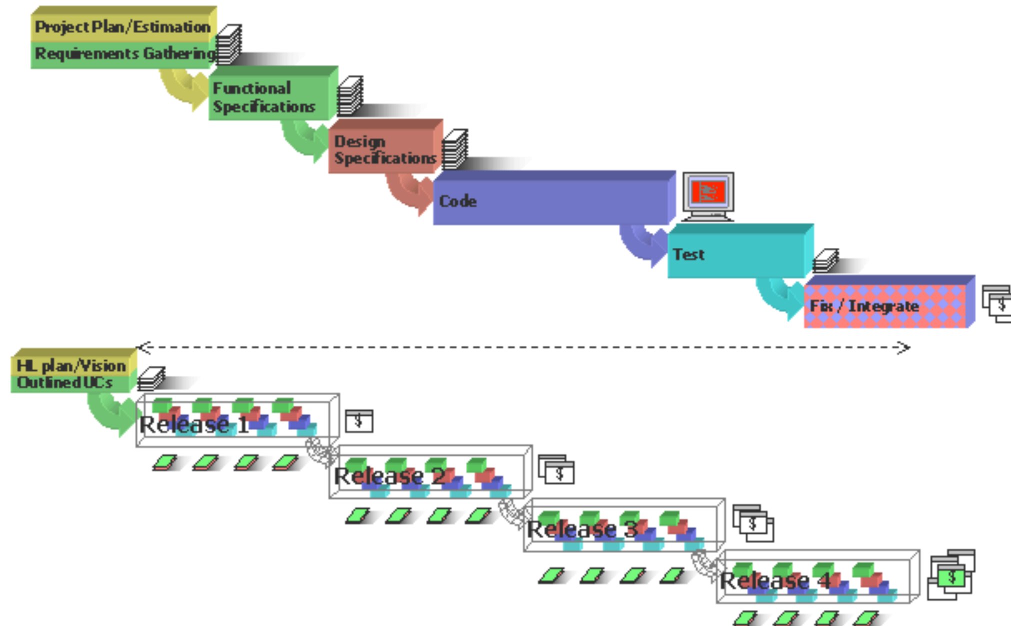
V této kapitole se budeme věnovat náplni jednotlivých fází projektu podle RUP. Na úvod ukážeme rozdíl mezi klasickými fázemi vodopádu a fázemi v iterativním vývoji. V tradičním modelu jsou fáze definovány/rozděleny podle rolí, ve fázi specifikace požadavků prostě definujeme veškeré požadavky na systém, v analýze toto všechno zanalyzujeme dopodrobna, abychom dále mohli navrhnout řešení atd. Tento model však přináší několik problémů (vodopád viz Obr. 30). Předně díky chybějící dennodenní spolupráci mezi členy týmu, jsou často požadavky chápány vývojáři jinak, než to ve skutečnosti zamýšlel analytik a přál si uživatel. Je to způsobeno tím, že analytik definuje požadavky a tyto sepíše do nějakého rozsáhlého dokumentu následujícím způsobem:

- Systém by měl mít toto ...
- Systém by měl mít tamto ...
- Systém by měl dělat toto ...
- Systém by měl splňovat standard ....
- Systém by měl umožnit ...

Takový dokument poté předá návrháři a je většinou převelen na jiný projekt. Bohužel si s sebou pak odnese i znalost zákazníka a jeho prostředí, pochopení jeho problémové domény a další důležité věci či vazby, které jsou psány mezi řádky. Tudíž o nich nemůže mít návrhář či programátor ani tušení, jelikož se neúčastnili sezení se zákazníkem a analytik již není k dispozici, aby toto vysvětlil. Navíc takový dokument může být těžce použitelný, představme si 20 stran takových požadavků, jak zajistíme, že na straně 12 není požadavek protichůdný proti požadavku na straně 19? Jak zajistíme či naznačíme návaznost a závislosti mezi jednoduchými požadavky, když použijeme pouhý text?

Dalším problémem je tvorba detailních plánů hned na úvod, kdy ještě nevíme spoustu věcí, nepočítá se v plánech s riziky a nečekanými událostmi (problémy s technologií, složitost problémové domény, ...), proto je tak jednoduché plány přestřelit nebo naopak (což je o moc více obvyklé) podhodnotil.

Díky způsobu vývoje, kdy je třeba mít na začátku definované všechny požadavky, se setkáváme s dalším problémem. Uživatel nám nadiktuje opravdu vše, co by kdy mohl potřebovat. Tím se dostaneme do stavu na (Standish Group výzkum), kdy máme v aplikaci 80% rysů, které uživatel nikdy nevyužije nebo je využije velice zřídka. Jako vývojáři však všechny tyto zbytečné rysy musíme vytvořit. Potom pravděpodobně není problém s produktivitou vývoje.



Obr. 30 Tradiční vs. iterativní model vývoje

Posledním významným problémem, který ve spojitosti s tradičním modelem zmíníme je integrace komponent a testování. To probíhá až na konci projektu, kdy projdeme všemi fázemi. V této části projektu je však nejen díky nepřesným plánům již málo času na řešení odhalených chyb a také je na toto odhalování již příliš pozdě, stojí více, než kdyby např. sami vývojáři spouštěli Unit testy hned po napsaní kódu, kdyby byla architektura integrována a ověřena dříve apod.

8.1 Iterace

Jak je patrné z Obr. 30, mezi fázemi RUPu, resp. iterativního vývoje a vodopádu je zásadní rozdíl. Iterativní vývoj probíhá v několika tzv. iteracích (opakováních). Takových zásadních rozdílů je hned několik:

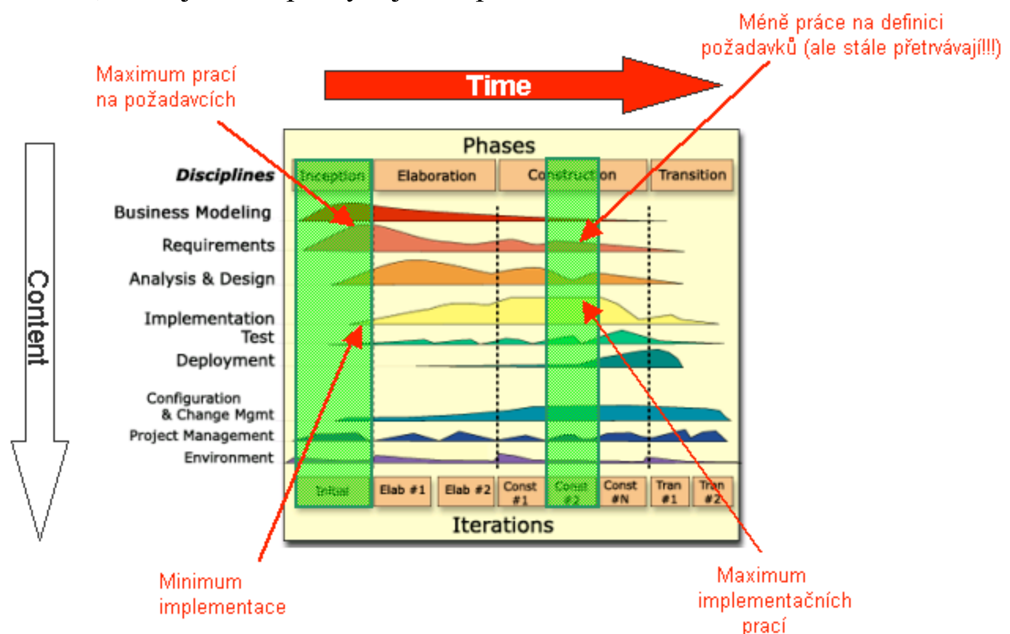
- Každá iterace produkuje spustitelný a otestovaný build obsahující nově implementované funkčnosti (scénáře) – proto abychom ho mohli dát k dispozici uživateli a dostali od něj zpětnou vazbu, jdeme správným směrem, pochopili jsme jeho potřeby dobře?
- Každá iterace má definovaný přesný cíl, který se snažíme naplnit (paralelní analýza, návrh, implementace a testování vybrané nové funkčnosti – jejich scénářů).

- Iterace je miniprojekt, což znamená, že má svůj začátek, konec a pevně definovaný časový rozsah (většinou 2-6 týdnů) – což se již předvídá a detailně plánuje lépe, než například 2 roky.
- V průběhu jedné iterace provádíme všechny disciplíny! Tj. definice požadavků, analýza a návrh, implementace, integrace a testování!!!
- Zřetězením iterací nabalujeme jednotlivé funkčnosti až do výsledného produktu.

Hlavní výhodou je, že již po relativně krátké době má zákazník k dispozici nějakou verzi výsledného produktu, i když jde třeba o nestabilní produkt, tak nám již může říct, co se mu líbí, nelíbí (poskytne velmi cenou zpětnou vazbu) a může s ním dokonce začít pracovat, čili aplikace již může vydělávat, i když neobsahuje zdaleka tolik rysů jako výsledný produkt. V průběhu každé fáze je možno mít 1 až n iterací v závislosti na typu projektu, blíže viz následující kapitoly. Na Obr. 31 jsou 2 z iterací naznačeny zeleně.

8.2 RUP fáze

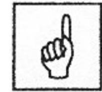
Jak již bylo naznačeno výše, RUP fáze jsou zcela odlišné od fází vodopádu, nejde tedy jen o jejich „přejmenování“. Fáze v RUP jsou spíše jednotlivé statusy projektu, jeho evoluce v čase. Obecně můžeme říci, že výstupem každé fáze iterativního projektu je spustitelný kód. Výsledkem fází vodopádového projektu jsou dokumenty, modely a další artefakty týkající se podobných činností, které je třeba při vývoji SW provést.



Obr. 31 Dvojrozměrný model RUP - fáze a disciplíny

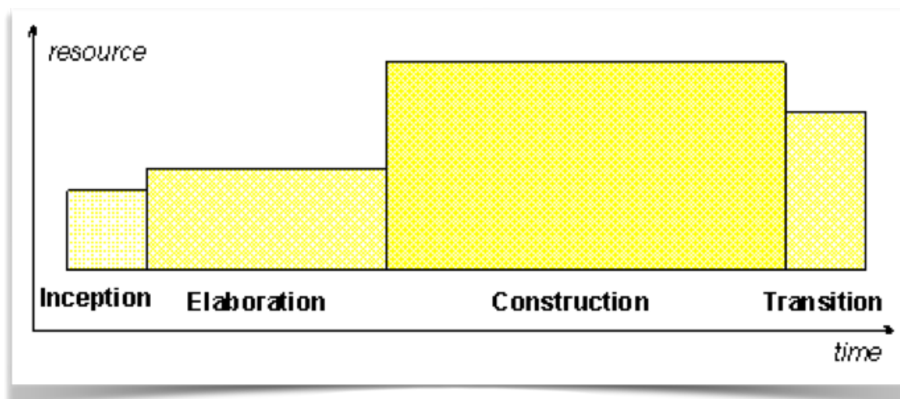
Výsledkem Inception je pochopení problematiky, vize projektu, identifikovaná rizika. Výstupem Elaboration je spustitelná, otestovaná architektura (= fungující část aplikace). Výstupem Construction je beta-release aplikace, relativně stabilní, opět spustitelná, téměř kompletní aplikace. Výstupem Transition je pak již produkt připravený k finálnímu nasazení včetně veškeré dokumentace a hardware. Zásadní rozdíl je také v tom, že každá fáze může

obsahovat několik iterací, v nichž se však vždy provádí všechny disciplíny s různým objemem prací – což je reprezentováno plochou pod danou křivkou (viz Obr. 31)



Je nutné zdůraznit, že každé fáze se většinou účastní různý počet vývojářů. V úvodu, kdy je třeba identifikovat požadavky, často komunikovat se zákazníkem, navrhnout architekturu, ověřit komunikaci s jinými systémy apod. jsou zainteresováni často jen projektový manažer (Project Manager), systémový analytik (System Analyst), zákazník, architekt, návrhář testů (Test Designer). V pozdějších fázích, kdy je stabilní architektura přibude větší množství vývojářů i testerů. Tito lidé (různé role), ale stále pracují spolu v jednom či více týmech a jsou k dispozici pro vysvětlení nejasností!!!

Poslední zásadní věcí týkající se fází, je rozložení prací na projektu v jednotlivých fázích. Následující obrázek a tabulka toto ukazují. Je vidět, že cílem Inception je opravdu rychle definovat vizi a rizika projektu a základní projektové věci a co nejrychleji přejít do Elaboration, abychom mohli zákazníkovi co nejdříve poskytnout spustitelný SW. Účast zdrojů na této fázi je omezená, většinou se jedná o projektového manažera, systémového analytika, architekta, test manažera a test designera + zástupci zákazníka a uživatelů. Celkově by Inception měla zabírat 10% celkového času, spíše méně.



Obr. 32 Objem prací a časové trvání jednotlivých fází RUP (zdroj: RUP)

	Inception	Elaboration	Construction	Transition
Pracnost	~5 %	20 %	65 %	10%
Plán	10 %	30 %	50 %	10%

Tabulka 8-1: Průměrné časy trvání jednotlivých fází RUP (zdroj: RUP)

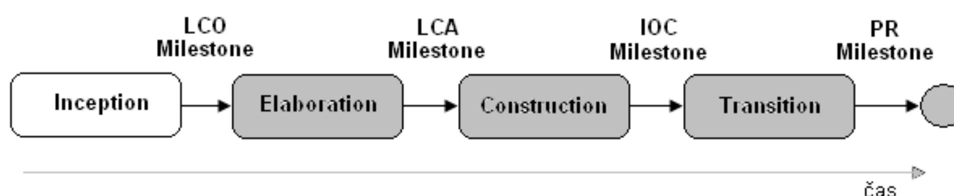
Elaboration obsahuje o málo více zdrojů a časově je její trvání zhruba 30% celkového času. Z obrázku i tabulky je patrné, že nejvíce času a nejvíce zdrojů vyčerpá Construction, kdy je v několika iteracích implementován výsledný produkt. Časově to znamená přibližně 50% celkového času projektu. Transition již pak zabírá 10% času, ale zdrojů je využíváno pořád hodně, jelikož doděláváme zbývající jednodušší rysy, řešíme finální vyladění produktu, jeho výkonnost, kompletujeme dokumentaci a provádíme závěrečné testy.

Následující text se věnuje popisu jednotlivých fází, jejich náplně a milníků detailněji. Více se také zabývá počtem potřebných iterací v jednotlivých fázích.

8.3 Inception phase

Cílem úvodní fáze je pochopení cílů projektu, požadovaných rysů aplikace, výběr vhodné technologie, definice vývojového procesu a výběr a nastavení nástrojů. Přesněji má Inception fáze těchto 5 cílů:

1. Porozumění tomu, co vytvořit – vytvoření vize, definice rozsahu systému, jeho hranic; definice toho, kdo chce vytvářený systém a co mu to přinese.
2. Identifikace klíčových funkcionalit systému – identifikace nejkritičtějších Use Casů.
3. Návrh alespoň jednoho možného řešení (architektury).
4. Srozumění s náklady, plánem projektu, riziky.
5. Definice/úprava procesu, výběr a nastavení nástrojů.



Obr. 33 Fáze Inception

V průběhu první fáze proběhne ve většině projektů pouze jedna jediná iterace. Proto, abychom dosáhli cílů této fáze, je však možné provést více iterací. Mezi důvody, které k tomuto přispívají můžeme zařadit následující:

- Rozsáhlý projekt, kde je těžké pochopit zaměření a rozsah systému.
- Jedná se o nový systém v neznámé problémové doméně, je obtížné definovat, co by měl systém dělat.
- Vyskytují se velká technická rizika, která je třeba snížit implementací prototypu nebo konceptem architektury před vlastním představením / schválením projektu.

Výsledkem této iterace (celé Inception fáze) je vize projektu. Vize nám říká, kterým směrem se bude projekt ubírat, je však definována z pohledu uživatelů aplikace (definuje jeho klíčové potřeby a rysy aplikace), ne technickou řečí! Obsahem vize jsou nastíněné klíčové požadavky na systém, vize tedy poskytuje jakýsi základ pro detailnější technické požadavky. Na vizi by se měli všichni účastníci projektu shodnout, pokud shoda nenastane, není možné jít do další fáze, jíž je Elaboration.

Příklad jednoduché vize pro jednoduchý projekt časovače, který pracuje na stanice vývojáře a reportuje určitý čas strávený prací na daném projektu:

Osobní časovač: Vize
Problém Gary není schopný sbírat konsistentní časové údaje od vývojářů reprezentující čas strávený na

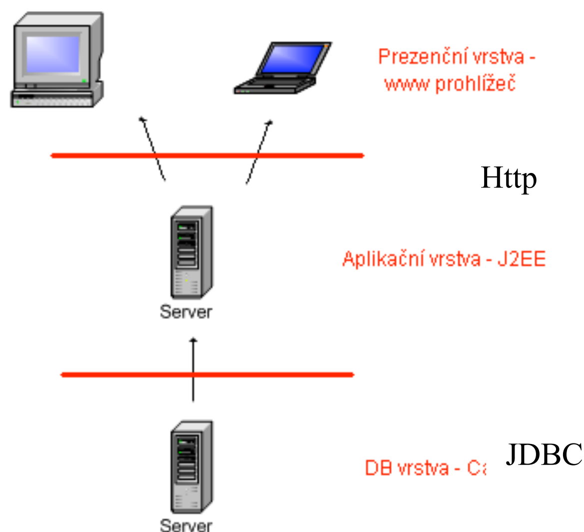
různých projektech. Není tedy možné monitorovat a porovnat postup oproti plánům, fakturovat řádné časy, platit externí spolupracovníky a samozřejmě také na základě těchto dat dělat věrné odhady dalších iterací.
Řešení Osobní časovač (OČ) měří čas strávený na projektech, shromažďuje a ukládá tato data pro pozdější zobrazení (stylem Post-it poznámek), aby mohl Gary systematicky organizovat a hodnotit projekty, sledovat aktuální postup prací a ty porovnávat s plánovanými odhady pro jednotlivé projekty
Zainteresované strany (Stakeholders) - jednotlivý vývojáři - pracovníci administrativy - projektový manažer
Use Case (nyní identifikované) - Změř čas aktivity - Sesbírej týdenní data - Sluč / konsoliduj data pro každý projekt - Nastav nástroj a databázi pro projekt

Tabulka 8-2: Příklad vize

Cílem Inception fáze je také určit, zda má smysl pokračovat dále v projektu, proto si musíme být jisti, že existuje alespoň jedna architektura, která umožní implementovat systém s rozumným podílem rizik a s rozumnými náklady.

Příkladem může být architektura klient-server (C/S), kterou můžeme realizovat pomocí několika vrstev a také s využitím několika technologií:

- První varianta může obsahovat 2 vrstvy – databázovou a tlustého klienta (prezenční + aplikační).
- Druhá varianta může být složena ze tří vrstev (viz následující obrázek) – prezenční reprezentována www prohlížečem, aplikační realizována webovým serverem s J2EE technologií a datová reprezentována post-relační databází Caché.



Obr. 34 Varianta Klient-server (C/S) architektury

Nezbytné komponenty architektury jsou také implementovány, abychom snížili na přijatelnou úroveň či odstranili možná technologická rizika související s použitou technologií, kompatibilitu, komunikaci s ostatními systémy či nákladovou stránku údržby nebo nutnost potřeby využití nějaké komponenty.

Tým se může pro naplnění cíle 3 ptát také na architekturu a technologie použité v předchozích projektech podobného rozsahu a zaměření, na jejich údržbu a cenu. Dále by se měl zabývat otázkami potřeby softwarových komponent a možnosti znovupoužití či nákupu již existujících. S tím také souvisí náklady na jejich pořízení a spojená rizika.

Pro celý projekt je kritické pochopení toho, co chceme vytvořit, stejně tak kritické je ale také vědět, jak toho dosáhnout a s jakými náklady. Hodně nákladů je například spojeno se zdroji a také se snižováním/odstraňováním rizik. Podle zdrojů, které máme k dispozici, jsme schopni odhadnout nejen dobu, ale také přibližnou cenu projektu. Výsledkem tohoto bodu je dokument zvaný Business Case. Dokument popisuje ekonomické přínosy produktu z pohledu kvantifikovatelných veličin. Dobrým příkladem může být použití návratnosti investic, tzv. ROI (Return of Investments), jenž vypočítáme ze vzorce (*100 abychom dostali procenta):

$$ROI = \frac{zisk - náklady}{náklady} \quad \text{nebo} \quad ROI = \frac{úspory}{náklady}$$

Příklad:

Pokud nás tedy softwarový projekt stojí 1.000.000 Kč a díky němu (automatizace práce zaměstnanců) ušetříme za rok při 25 zaměstnancích s náklady za jednoho 500 Kč za hodinu a při 200 pracovních dnech práci 5ti zaměstnanců, bude zisk následující:

8 hodin * 500 Kč/h * 5 zaměstnanců * 200 pracovních dnů = 4.000.000 Kč

pak výsledná návratnost investic bude (úspory/zisk * 100):

$$ROI = \frac{4.000.000}{1.000.000} = 400\%$$

Je zřejmé, že v tomto případě se projekt vyplatí, jelikož návratnost investic je 400%, což znamená, že vložená investice se nám vrátí 4krát.

V malém projektu může mít Business Case formu e-mailu nebo poznámky, v rozsáhlejším pak bude mít klidně několik stránek.

8.3.1 Milník LOM

Na konci Inception fáze následuje první milník projektu nazýván Lifecycle Objective Milestone (LOM). Milník je určen ke zhodnocení cílů celého projektu. Pokud nejsme schopni tohoto milníku dosáhnout, měl by být projekt

zrušen nebo přerušen. Důvodem neshody může být neshoda na rozsahu funkčností produktu, přílišné náklady, neexistující technologie schopná dostát našim požadavkům, nevýhodnost/nevratnost investice apod.

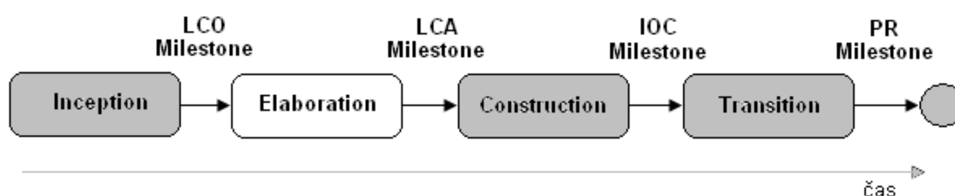
Pokud je produkt odsouzen k záhubě, je lepší ho ukončit dříve, než později, jelikož nás tím pádem připraví o míň peněz. Iterativní přístup společně s definovanými milníky nám umožňuje identifikovat tuto skutečnost dříve, než například ve vodopádovém přístupu.

LOM milník definuje následující evaluační kritéria:

- Shoda účastníků projektu (samozřejmě včetně zákazníka) na rozsahu projektu, počátečních nákladech a odhadu plánu, který bude dále upřesňován.
- Shoda na identifikaci správných požadavků a porozumění jim.
- Shoda na věrnosti odhadů nákladů a plánu, prioritách, rizicích a vývojovém procesu.
- Shoda na tom, že počáteční rizika byla identifikována a existují strategie na jejich snížení.

8.4 Elaboration phase

Po úspěšném projítí první fází máme tedy představu o tom, co budeme vyvíjet, kdo budou uživatelé a co jim má nový systém přinést. Na této představě jsme se shodli se všemi účastníky projektu. Dále máme identifikovány klíčové funkčnosti a tyto stručně popsány. Navrhli jsme alespoň 1 možné architektonické a technologické řešení, vytvořili jsme hrubý plán projektu a identifikovali náklady. V neposlední řadě máme definovaný proces vývoje a nastavené prostředí a nástroje.



Obr. 35 Fáze Elaboration

Pokud jsme dosáhli prvního milníku (Lifecycle Objective Milestone – LOM), přechází projekt do druhé fáze zvané Elaboration. Cílem této fáze je definovat a nastínit architekturu systému, abychom na jejím základě mohli v následující fázi navrhnout a implementovat zbývajících 70-80% nekritických Use Casů (funkčností). Jak již bylo řečeno, architektura se vyvine z nejdůležitějších požadavků (z těch, které na ni mají dopad) a také z ohodnocení rizik. Cílem této fáze je hlavně snížení či odstranění rizik, a to ve čtyřech hlavních oblastech:

- Rizika spojená s požadavky na systém (Vyvíjíme správnou aplikaci?).
- Rizika architektonická (Poskytujeme správné řešení?).
- Rizika spojená s náklady a plány.

- Rizika procesní a spojená s prostředím a nástroji (Máme správný proces a nástroje?)

V následujícím textu se budeme podrobněji opět věnovat každému z cílů této fáze. Nejdříve však vysvětlíme, kolik iterací je vhodné naplánovat pro tuto fázi v případě různých typů projektů.

8.4.1 Iterace

Většinu rizik snížíme tím, že v této fázi vyprodukujeme spustitelnou architekturu našeho výsledného řešení. Tímto konkrétně demonstrujeme jeho proveditelnost a nespoléháme na možné mylné předpoklady. Pokud navrhujeme systém s použitím stejné technologie jako v předchozích projektech, jsme schopni cíle fáze naplnit většinou v jediné iteraci, jelikož existuje menší množství rizik, které potřebujeme odstranit. Navíc můžeme znovupoužít řešení či komponenty z předchozích projektů, což urychluje náš postup.

Naopak, pokud nemáme zkušenosti s danou problémovou doménou, pokud je systém velmi komplexní nebo pokud používáme novou technologii, bude zapotřebí dvou či tří iterací k vytvoření stabilní architektury a zmírnění největších rizik. Dalšími přispívateli k většímu počtu iterací jsou distribuovaný vývoj, velký počet uživatelů systému, bezpečnostní požadavky a další.

První iterace v Elaboration by měla zahrnovat:

- Návrh, implementace a testování malého počtu kritických scénářů, pomocí kterých identifikujeme typ architektury a potřebné mechanismy, rozhraní. Toto se snažíme provést co nejdříve z důvodu snížení největších rizik.
- Identifikace, implementace a testování malé množiny základních mechanismů v architektuře.
- Počáteční hrubý návrh logické struktury databáze.
- Detailní vypracování událostí hrubé poloviny Use Casů, které zamýšlíme detailně popsat v Elaboration fázi (podle priorit).
- Důkladnější testování pro ověření architektury a ujištění se, že největší architektonická rizika byla snížena na únosnou mez.

Druhá iterace v Elaboration by měla zahrnovat:

- Oprava všeho, co nebylo správné v první iteraci.
- Návrh, implementace a testování zbývajících architektonicky významných scénářů.
- Nastínění a implementace paralelismů, procesů, threadů na takové úrovni, která je potřeba k identifikaci potenciálních technických problémů. Zaměření této iterace je také na testování výkonnosti, zátěže a rozhraní mezi subsystémy, stejně jako na externí rozhraní.
- Identifikace, implementace a testování zbývajících mechanismů v architektuře.
- Návrh a implementace první verze databáze.
- Detailní popis zbývající poloviny Use Casů, které chceme blíže specifikovat v Elaboration.

- Testování, ověření a úpravy architektury do její stabilní verze, na ni pak můžeme dále navěsit další funkčnosti systému.

Pokud v těchto iteracích přijdou zásadnější zásahy do architektury např. vinou změnových požadavků, je vhodné přidat další iteraci, abychom měli jistotu (výsledky testů), že je architektura opravdu správná a stabilní. Toto pravděpodobně způsobí zpoždění projektu, ale je to o mnoho levnější, než celé řešení stavět na tekutých píscích (rozuměj na nestabilní architektuře).

V úvodní fázi (Inception) jsme definovali vizi produktu a detailně popsali přibližně 20% těch nejdůležitějších Use Casů (z hlediska architektury). Tento cíl říká, že v průběhu Elaboration budeme podrobněji popisovat další Use Casy. Některé z nich mohou být natolik jednoduché nebo pouze používající jiná data, že je posuneme až do Construction fáze nebo dokonce nemusí být formálně vůbec popsány. Jejich detailní popis totiž nepřinese žádný přínos ke snížení nějakého rizika. Předmětem Elaboration může být také konstrukce prototypu uživatelského rozhraní pro významné Use Casy, nad kterým si budeme následně s uživateli vyjasňovat funkcionalitu, kterou mají Use Casy poskytovat.

Pokud se jedná o složitější problémovou doménu, můžeme vytvořit doménový model pro popis vztahů jednotlivých entit a samozřejmě doplnit či opravit doménový slovník, který byl vytvořen v Inception fázi.

Pro detailní popis jednotlivých Use Casů je vhodné si vytvořit časově omezený úsek (stejně jako v Inception), abychom nezabředávali do přílišných detailů. Pokud se jedná o menší projekt a implementaci bude provádět stejný člověk, který specifikuje Use Casy, je možné dokumentací jejich popisu strávit méně času a vrátit se k němu až po implementaci a otestování daných UC. Na konci Elaboration by mělo být popsáno přibližně 80% Use Casů, některé nové UC je možné nalézt i v Construction, nemělo by to však být pravidlem.

Dalším cílem Elaboration je návrh a implementace, ověření základu celého řešení, tzv. architektury, řekneme si nejdříve stručně, co to architektura je. Architektura je část systému, část řešení, řeší komunikaci, ukládání, uživatelské rozhraní, technologie a podobné věci, ale pouze v míře nezbytně nutné na základě kritických (nejdůležitějších) Use Casů. Při tvorbě architektury uvažujeme následující:

- subsystémy řešení (stavební bloky) a jejich rozhraní,
- jejich interakce za běhu programu pro naplnění identifikovaných scénářů,
- implementace a testování prototypu (rizika snížena, ověřeno řešení, výkonnost, škálovatelnost, náklady).

Proto abychom byli schopni ověřit správnost a proveditelnost navržené architektury, potřebujeme více než jen revidované modely či stránky papíru. Potřebujeme spustitelnou architekturu, kterou můžeme testovat, tj. spustitelný kód. Nyní představíme několik kroků, které je vhodné provést (samozřejmě opět podle typu projektu) pro dosažení spustitelné, stabilní a testovatelné architektury.

Prvním krokem je definice subsystémů, klíčových komponent a jejich rozhraní, pomocí kterých budou komunikovat. V tomto kroku je vhodné uvažovat použití existujících frameworků (z předchozích projektů, či existujících na trhu) předtím, než budeme na zelené louce navrhovat architekturu vlastní. Potenciálními zdroji pro identifikaci komponent jsou doménové objekty z doménového modelu.

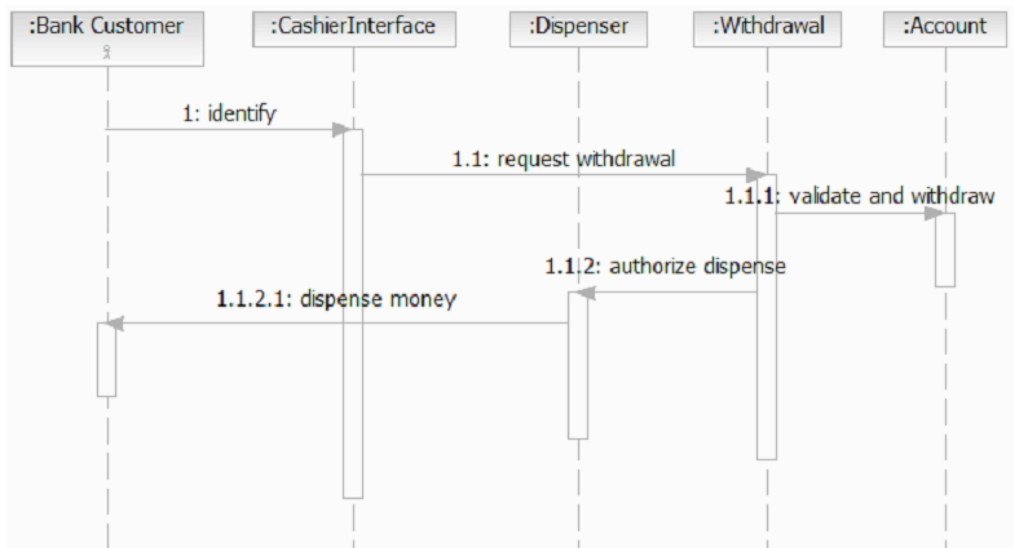
Dalším krokem je použití architektonicky významných Use Casů pro definici architektury. Jak již bylo řečeno, je to typicky 20-30% Use Casů, které jsou kritické. Je třeba brát v úvahu taktéž nefunkční požadavky a nalézt a implementovat Use Casy, které odhalí potencionální problémy a rizika a ověří jejich proveditelnost. Pokud mluvíme o implementaci Use Casů máme v těchto fázích na mysli pouze 1 nebo 2 scénáře. To znamená ideální („happy day") scénář + například jeden chybový pro ověření způsobu zachycení a zpracování výjimek. Na paměti je třeba mít účel těchto všech kroků ve Elaboration fázi, tím je snížení rizik našeho řešení na akceptovatelnou úroveň či jejich úplné odstranění.

Dalším krokem je návrh (design) kritických Use Casů, jinými slovy také realizace Use Case. Ten popisuje, jak jsou konkrétní objekty realizovány v návrhovém modelu z pohledu jejich spolupráce. Toto může být provedeno formálně či opět neformálně, pouze náčrtky na tabuli nebo ve vizuálním modelovacím nástroji, **Obr. 36** ukazuje zápis pomocí UML sekvenčního diagramu. Návrh probíhá v několika krocích:

1. Nástin analytických objektů.
2. Definice chování jednotlivých analytických tříd (jejich odpovědnost).
3. Detailní popis těchto tříd (přesnější pochopení odpovědností).
4. Návrh Use Casů – komunikace (pořadí, způsob, ...).
5. Rozpad (upřesnění) analytických tříd na návrhové.

Konsolidace a seskupení identifikovaných tříd do balíčků je dalším krokem. Tyto balíčky nebo subsystémy vytváříme podle několika aspektů:

- Předměty budoucích častých změn do 1 balíčku (např. rozhraní aktora).
- Pravidla viditelnosti (vícevrstvá aplikace nebude obsahovat v 1 balíčku třídy z více vrstev).
- Budoucí konfigurace produktu (výsledný produkt může být skládán z různých částí aplikace).



Obr. 36 UML sequence diagram

Dalším důležitým krokem je návrh databáze, jelikož většina dnešních systémů využívá pro persistentní objekty některý ze systémů řízení báze dat (SŘBD). Cílem je pochopení, jak budou persistentní data ukládána a opět vyvolávána zpět (mechanismus a technologie).

Jak již bylo naznačeno v předchozím kroku, jsou důležitým aspektem architektury také různé mechanismy v architektuře. Jedná se o běžné situace a problémy, které jsou řešeny například pomocí návrhových vzorů (persistence, autorizace, komunikace, ...).

Integrace komponent je předposledním bodem, který zmíníme. Integrace určuje v jakém pořadí a jaké komponenty budou integrovány. Toto definujeme paralelně s identifikací analytických tříd. Integrace komponent je důležitá, jelikož sestavené a kompilované komponenty jsou předmětem testování, abychom viděli, zda splňují požadované chování a výkonnostní či bezpečnostní požadavky. Integrace je neustálou aktivitou, kterou provádíme v průběhu všech iterací a to většinou denně (např. night builds) minimálně však alespoň 2x týdně. Kritickými faktory této aktivity je fungující konfigurační management stejně jako automatizované buildy.

Testování kritických scénářů je posledním krokem. Testování je totiž kritickým aspektem Elaboration. Je to ukazatel postupu projektu v čase (co je již hotovo, otestováno, kde zbývají problémy apod.). Nejlepší cestou k přesvědčení se, že máme snížena všechna důležitá rizika, je otestovat spustitelnou architekturu, což znamená:

- Kritické scénáře byly správně implementovány a poskytují předpokládanou funkčnost.
- Architektura poskytuje dostatečnou výkonnost.
- Architektura podporuje a je schopna pojmout nezbytnou zátěž (např. 1000 současných uživatelů).
- Rozhraní s externími systémy fungují tak, jak bylo předpokládáno.

- Byly testovány také ostatní nefunkční požadavky, které nebyly zachyceny a popsány výše.

Je třeba si uvědomit, že pokud jsme prošli až sem, tak máme některé části systému vyvinuté v docela pokročilém stupni, stále ale máme implementováno pouze 10-20% celého řešení. Většinou se jedná o úspěšné scénáře 20-30% Use Casů. Provedli jsme tedy něco ode všech disciplín, ale stále nám zbývá nějakých 80% systému navrhnout a implementovat. Dobrou správou je, že tato část byla tou nejnáročnější z celého systému, díky tomu jsme snížili nejvýznamnější rizika projektu.

Na konci Elaboration máme přesnější informace, které nám dovolí aktualizovat a upřesnit projektový plán a odhad nákladů. Máme totiž již:

- vytvořen detailní popis požadavků – rozumíme přesně jaký systém budeme vytvářet,
- implementovánu kostru řešení (architekturu),
- sníženu většinu rizik (což redukuje nadhodnocení či podhodnocení odhadů plánů a nákladů),
- přesnější porozumění, jak efektivně pracuje náš tým pomocí definovaného procesu s danými nástroji a s danou technologií (jelikož jsme prošli celý životní cyklus již minimálně jednou).

Nyní tedy zpřesníme odhady nákladů a plánů projektu (podle rozsahu projektu to mohou být následující dokumenty: Vision, Business Case, Software Development Plan, Project Plan), aktualizujeme seznam rizik a akcí na jejich odstranění/snížení.

8.4.2 Milník LCA

Milníkem Elaboration fáze je tzv. Lifecycle Architecture milestone (LCA). Nyní máme detailně prozkoumány cíle a rozsah systému, vybranou architekturu a identifikována a snížena největší rizika. Opět platí, že pokud nejsme schopni dosáhnout tohoto milníku, je vhodné projekt ukončit.

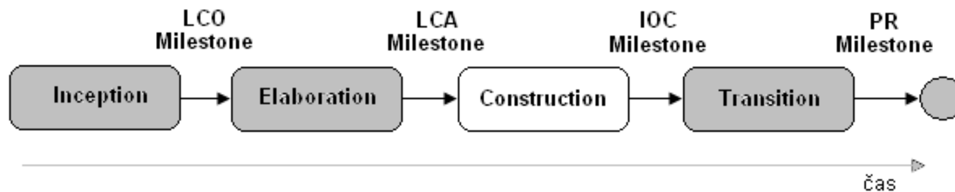
Zda jsme dosáhli tohoto milníku nám pomůže zjistit následující kontrolní seznam:

- Je vize produktu stabilní, jsou stabilní požadavky?
- Máme stabilní architekturu?
- Jsou klíčové postupy a přístupy, které budeme používat, otestovány a je dokázána jejich použitelnost?
- Ukázalo testování spustitelného prototypu, že jsou klíčová rizika identifikována a vyřešena?
- Máme definovány plány iterací pro následující Construction fázi v náležitých podrobnostech, abychom byli schopni podle nich postupovat?
- Jsou tyto plány podpořeny důvěryhodnými odhady?
- Naplněním plánu s použitím definované architektury dosáhneme cílů shrnutých ve vizi?
- Jsou aktuální náklady akceptovatelné vůči plánovaným?

Tato revize může trvat pro rozsáhlejší projekty den i více. Menší projekty mohou provést ohodnocení během hodinového sezení.

8.5 Construction phase

Fáze Elaboration byla ukončena interním releasem základní, spustitelné architektury systému, která umožnila identifikovat a implementací přímo ověřit největší technická rizika (soupeření o zdroje, výkonnostní rizika, zabezpečení dat, …). Následující fáze zvaná Construction, která je předmětem této kapitoly, je zaměřena na detailní návrh, implementaci a testování, aby bylo zajištěno zhmotnění kompletního systému.



Obr. 37 Fáze Construction

Předmětem prací v této fázi je návrh a implementace zbývajících přibližně 80% Use Case a finální implementace původních 20%, které představují kritické (hlavní) požadavky zákazníka. Dosud byla implementována pouze malá podmnožina z celkového kódu aplikace. Tato fáze je proto také časově nejnáročnější a účastní se jí největší počet lidí, hlavně programátorů a testerů. V průběhu Construction budou identifikována další rizika, na která se musíme zaměřit. Neměla by však být kritického rázu a tudíž by měla mít pouze malý vliv na architekturu systému. Pokud by tomu bylo naopak, značí to nekvalitní práci v předchozí fázi. Kritickými faktory úspěchu pro tuto fázi jsou zajištění celistvosti architektury, paralelní vývoj, správa konfigurací a změnové řízení (Configuration & Change Management) a v neposlední řadě automatizované testování. Zajímá nás také správná rovnováha mezi kvalitou, záběrem systému (jeho rozsáhlostí) časem a detailností či dokonalostí implementovaných požadavků.

Cíle Construction fáze lze definovat následovně:

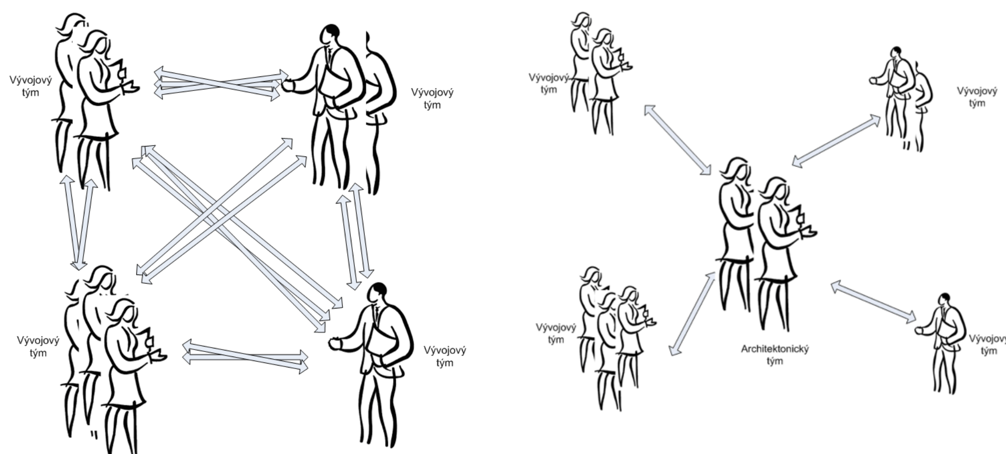
- Minimalizace nákladů na vývoj, dosažení určitého stupně paralelního vývoje (pro efektivnější využití zdrojů).
- Iterativní vývoj kompletního produktu, který bude připravený k doručení uživatelské komunitě. (beta release – první funkční verze aplikace)

8.5.1 Iterace

Počet iterací této fáze se bude opět lišit projekt od projektu, v zásadě lze říci, že jich bude více než v jiných fázích, typicky 2-4. Plánování iterací bude opět řízeno Use Casy, kdy nejdříve budeme implementovat ty nejdůležitější pro zákazníka, či technicky nejrizikovější, což může znamenat implementaci pouze některých scénářů (hlavně u technických rizik). Řečená zásada se týká hlavně první iterace v této fázi.

Cílem správně provedené Elaboration fáze je základní architektura systému, která je otestovaná a spustitelná. Cílem bylo vytvořit kostru komunikačních mechanismů, ukládání a správu dat a další. Pokud byla architektura navržena správně a je robustní, je nyní jednodušší pokračovat ve vývoji, jelikož tyto mechanismy můžeme využívat a znovupoužít, další kód je na tuto architekturu „navěšen“.

Jednou z výhod existence architektury systému je jasná definice odpovědností částí systému rozdělených do dobře definovaných subsystémů. To umožní jednotlivým vývojovým týmům při paralelním vývoji nezasahovat si do svých subsystémů. Samozřejmě, vývojáři musí rozumět celému systému, ale měli by mít přidělenou určitou část, podsystém, na kterém pracují.



Obr. 38 Organizace kolem architektury minimalizuje přílišné komunikační zatížení

Tento způsob je nazýván organizace kolem architektury a snaží se efektivně nahradit komunikaci tváří v tvář, která je důležitá, ale v případě velkého vývojového týmu by neúměrně narostla (geometricky!) a snížila efektivitu vývojového týmu. Toto můžeme omezit existencí jednoho týmu, který je odpovědný za architekturu a několika podtýmů odpovědných za jeden nebo několik podsystémů. Komunikace mezi těmito týmy je pak zprostředkována týmem odpovědným za architekturu, jelikož může řešit problémy a těžkosti spojené s celkovým řešením, stejně jako s jednotlivými rozhraními a například mít poslední slovo (rozhodovat o jejich struktuře).

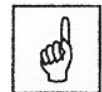
Velmi důležitým aspektem této fáze je také správa konfigurací (CM – Configuration Management). CM je definován a vybudován ve fázi Inception a vyladěn ve fázi Elaboration. V průběhu vývoje vzniká spousta různých souborů budoucí aplikace. Sledovat všechny jejich verze a změny je velmi složité, zvláště v případě iterativního vývoje, kdy neustále vytváříme nové verze, provádíme jejich integraci a testování. CM nám umožní jít zpět k posledním fungujícím verzím, umožní sestavovat build ze správných verzí či umožní přístup k některým souborům pouze vybrané skupině vývojářů. Existuje-li funkční správa konfigurací, mohou se vývojáři věnovat jen a pouze vlastnímu vývoji a tím zvýšit jeho efektivitu.

V průběhu Elaboration jsme detailně popsali pouze kritické Use Case nebo ty, které mají vliv na architekturu. Méně významné UC s malým dopadem na architekturu byly přesunuty do fáze Construction. Jedná se hlavně o UC, jejichž funkcionalita je podobná jako již implementovaných UC, ale využívají se při tom jiné entity, datové objekty, aktoři či rozdílné uživatelské rozhraní (UI).

Na konci Construction fáze provádíme také tzv. beta-relase, jehož předmětem je testování do něhož jsou zahrnuti vybraní uživatelé. V rámci Construction je třeba připravit úspěšně otestovaný beta-release. Je třeba, aby byly implementovány všechny rysy aplikace, mohou být však ještě nevyřešeny některé kvalitativní problémy, jako je menší dostupnost či odezva aplikace, nesmí se však ztrácet data apod. Stejně tak musí být připravena nápověda v aplikaci, instalační instrukce, uživatelské manuály a tutoriály, jinak nemůžeme dostat od uživatelů (beta-testerů) zpětnou vazbu. Pro některé projekty je také třeba připravit se v Construction na finální nasazení produktu, což zahrnuje:

- Tvorbu materiálů pro trénink uživatelů a správců aplikace.
- Přípravu prostředí pro nasazení (nákup nového HW, konvertování dat apod.) a přípravu dat.
- Příprava dalších aktivit zahrnujících marketing, distribuci, prodej.

8.5.2 Milník IOP



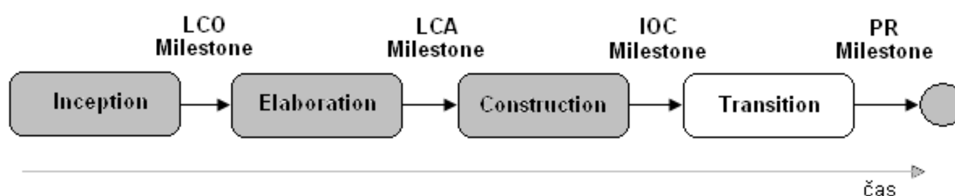
Tento milník je velmi důležitý, jelikož nám říká, zda je produkt připraven pro nasazení a beta-testování.

Zda jsme dosáhli tohoto milníku nám pomůže zjistit následující kontrolní seznam:

- Je produkt dostatečně stabilní a vyzrálý, aby mohl být rozeslán mezi komunitu uživatelů?
- Jsou aktuální výdaje na zdroje oproti plánovaným stále akceptovatelné?

8.6 Transition phase

Poslední fáze představovaného iterativního způsobu vývoje se nazývá Transition. Jejím cílem je především finální vyladění produktu a to nejen z pohledu funkcionality, ale také z pohledu výkonnosti, uživatelské použitelnosti a vůbec celkové kvality. Je také důležité si opět uvědomit, že artefakty, o kterých budeme opět mluvit, nemusí být vůbec formální (dokument či model v nějakém nástroji), je možné je mít ve formě fotek whiteboardu, či na něm přímo ponechané, dále ve formě náčrtků nebo je mít jen v hlavě.



Obr. 39 Fáze Transition

Beta-release, který byl nasazen mezi vybrané uživatele v rámci Construction fáze není finální produkt, je třeba ho stále ještě vyladit. Proto i zpětná vazba od uživatelů by měla zahrnovat jen body týkající se výkonnosti, instalace, použitelnosti. Žádné velké změny by neměly být v této fázi prováděny, již se s nimi nepočítá, např. nutnost změn v architektuře v této fázi jednoznačně indikuje špatně provedenou Elaboration a také částečně Construction a evokuje spíše vodopádový přístup, než správně pochopené a provedené iterace řízené riziky. Cílem Transition může být také kompletování některých scénářů, které byly z důvodu podobnosti s ostatními nebo kvůli jejich jednoduchosti přesunuty do fáze Transition (některé případně na konec Construction).

Jednoznačně se vymezíme od tradičních metodik. Cílem Transition není pozdní testování a integrace, které ve vodopádu teprve odhalují vzniklé problémy. Naopak, do této fáze již vstupuje relativně hotová integrovaná, spustitelná, stabilní a testovaná aplikace obsahující téměř všechny funkčnosti.

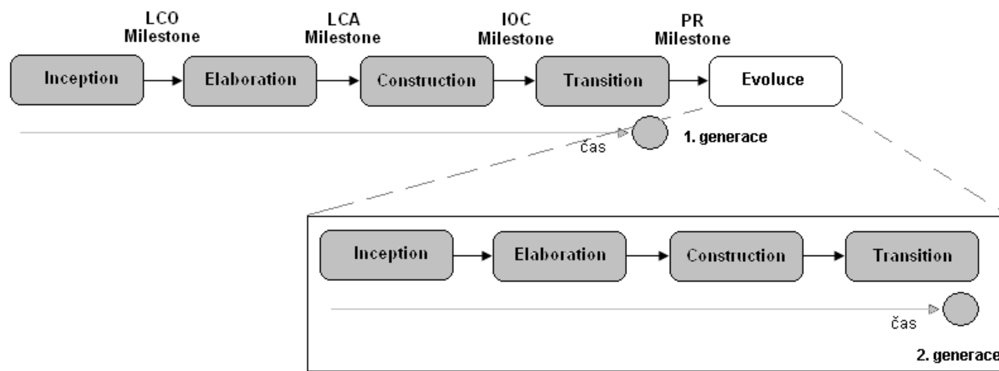
8.6.1 Cíle

Cíle této fáze jsou následující:

1. Beta-testování – zjištění, zda jsme naplnili očekávání uživatelů.
2. Školení uživatelů a správců aplikace.
3. Příprava prostředí a dat.
4. Příprava dalších aktivit zahrnujících marketing, distribuci, prodej – tvorba letáků s popisem produktu, white papers, technical papers, case study, demo nahrávek, zpráv pro tisk.
5. Dosažení souhlasu uživatelů, že aplikace splňuje jejich představy zachycené v dokumentu Vize (Vision).
6. Zlepšení průběhu budoucích projektů díky ponaučením z tohoto projektu tzv. lessons learnt.

Transition fáze může být velmi jednoduchá, stejně jako velmi komplexní v závislosti na druhu projektu. Může zahrnovat provoz starého systému paralelně s novým, migraci a transformaci dat, školení uživatelů či přizpůsobení podnikových procesů. Typické projekty obsahují v této fázi pouze jednu iteraci, která je zaměřena na opravu chyb a vyladění aplikace.

Kromě dodání výsledného produktu je třeba také dodat zákazníkovi či třetím stranám, které budou provozovat, spravovat nebo dále rozvíjet stávající aplikaci, další artefakty jako je dokumentace uživatelská i technická popisující například architekturu systému. Aplikace na konci této fáze však neumře, pouze ukončíme vývojový cyklus, za kterým však mohou následovat další, viz následující obrázek.



Obr. 40 Vývojové cykly více verzí produktu

V případě evoluce, čímž je rozuměn vývoj další verze, jsou většinou překryty fáze Transition právě dokončované verze a Inception životního cyklu verze nové.

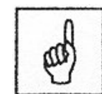
8.6.2 Testování

Součástí Transition fáze je také testování a to jak regresní, protože, jak již bylo řečeno i v Transition můžeme provádět návrh a implementaci některých rysů, tak akceptační. Pokud je vývoj ukončen, chyby opraveny a vytvořen build, je tento v Transition opět ještě testován podle standardního testovacího cyklu. Pořád však mějme na paměti, že se nejedná o vodopádový model, že testování a integrace neprobíhá až ve fázi Transition! Testování probíhá neustále v rámci minimálně Elaboration, Construction a Transition fází. Na konce každé iterace v těchto fázích je produkován spustitelný build, nové funkčnosti jsou integrovány a je provedeno unitové testování (provádí ještě samotný programátor), integrační, systémové, funkční testy a také regresní, abychom si byli jisti, že jsme nenarušili dříve vytvořené funkčnosti.

8.6.3 Lessons learnt

V této fázi je také vhodné shromáždit data o projektu a strávit chvíli jejich analýzou, abychom zjistili, co fungovalo a co ne. Výsledkem mohou být doporučení pro příští projekty, abychom se vyvarovali opětovným chybám. Můžeme znovupoužít nastavení prostředí (jako struktura repository, nastavení nástrojů), některé komponenty apod.

8.6.4 Milník PRM



Posledním milníkem je tzv. Product Release Milestone, který ukončuje čtvrtou a poslední fázi životního cyklu RUP. Cílem milníku je zjistit, zda byly naplněny cíle, které jsme si předsevzali a zda můžeme/chceme začít další vývojový cyklus. V případě pokračování je tato fáze prováděna zároveň jako Inception dalšího cyklu.

Primárními evaluačními kritérii Transition fáze jsou následující otázky:

- Jsou uživatelé spokojeni?
- Jsou aktuální výdaje versus plánované akceptovatelné; pokud ne, jaké akce mohou být v příštích projektech provedeny, abychom tomuto problému předešli?

Osobní časovač (OČ): Jednoduchý projektový plán

Pondělí	**Úterý**	**Středa**	**Čtvrtek**	**Pátek**
Inception Vize Plán Business Case Seznam rizik **LCO:** Souhlas od Garyho **Elaboration** Prototyp	Prototyp Zmírnění rizik **LCA:** Souhlas od Garyho Use Casy Testy	**Construction** **C1** Návrh Programování Testování **C2** Návrh Programování Testování	**C3** Návrh Programování Testování **IOC:** ukázat první beta verzi **Transition** Zlepšování Doručení	Časový buffer

Tabulka 8-3: Příklad projektového plánu

Zmínili jsme, že v projektovém plánu definujeme počty iterací v jednotlivých fázích, jejich stručnou náplň a jejich cíle, pokud jsou již známé. Příkladem cílů jednotlivých iterací může být pro systém Telefonního přepínače následující:

- Iterace 1: Hovor mezi lokálními stanicemi.
- Iterace 2: Přidání externích hovorů a správa účastníků.
- Iterace 3: Přidání telefonního záznamníku a konferenčních hovorů.
- Iterace 4: ...

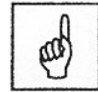
8.7 UML v procesu vývoje

UML je doporučeným nástrojem v Unified Processu, stejně jako v RUP, proto si něco povíme i o něm. Nejdříve tedy co UML je a co není, jakou má historii. UML není metodikou ani programovacím jazykem, je to pouze vizuální modelovací nastroj určený pro modelování převážně objektově orientovaných systémů. S žádnou konkrétní metodikou také není svázán (i když je například hojně využíván již zmíněným UP nebo RUP).

UML nabízí vizuální syntaxi pro modelování během celého vývojového cyklu (analýza až nasazení). UML slouží pro modelování čehokoliv, podporuje různé aplikační domény (od real-time systémů až po expertní systémy). UML je nezávislý na programovacím jazyku, i když nejlepší použití je samozřejmě s objektově orientovanými jazyky jako je Smalltalk, Java nebo C#, vhodný je však i pro hybridní jazyky (C++ nebo Visual Basic).

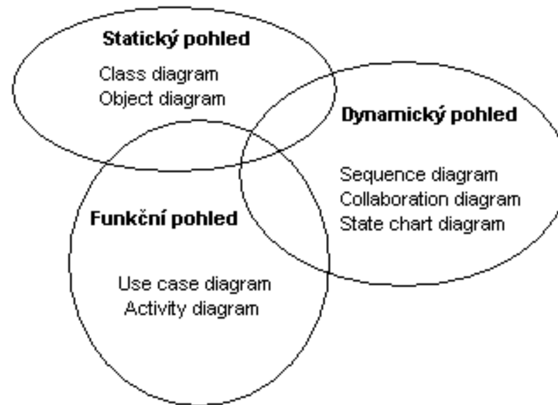
Povíme si také stručně něco z historie. Do roku 1994 panoval ve světě objektově orientovaných metod chaos, existovalo několik různých jazyků a metod pro vizuální modelování. Jedním z prvních pokusů o sjednocení byla metodka Fusion (1994), do její přípravy ale nebyli zapojeni tvůrci metod s největším podílem na trhu (Booch, Jacobson, Rumbaugh), proto se neujala. Booch a Rumbaugh se později spojili ve firmě Rational Corporation a začali pracovat na tvorbě jazyka UML. V roce 1996 navrhlo sdružení OMG UML

jako standard objektově orientovaného jazyka pro vizuální modelování, v roce 1997 byl OMG přijat. Nyní (rok 2015) existuje UML ve verzi 2.5.



UML definuje základní pohledy (viz obrázek):

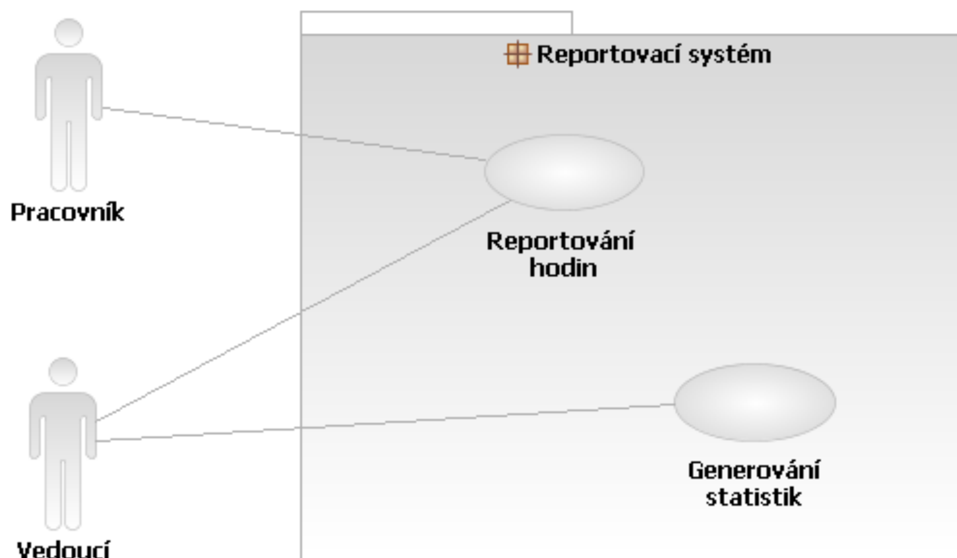
- Statický pohled
- Dynamický
- Funkční



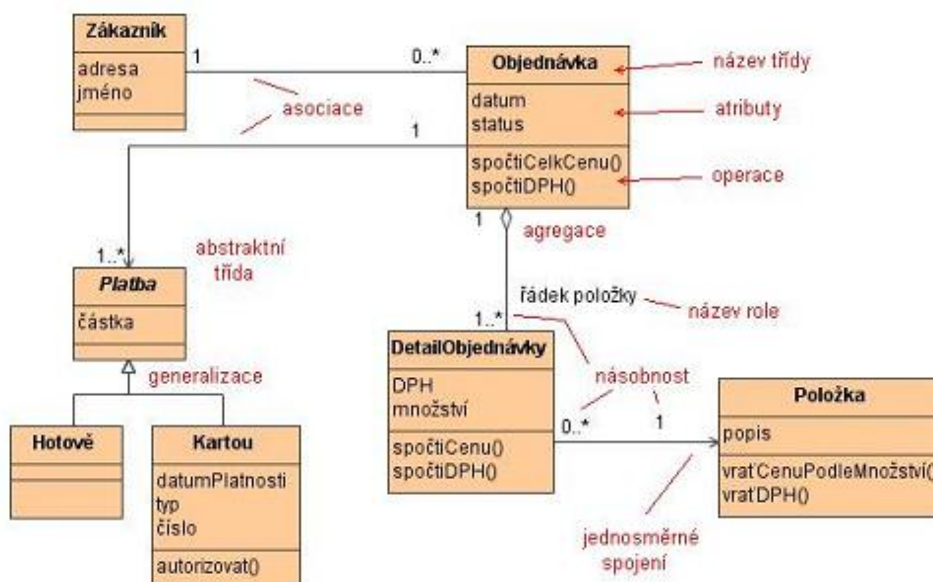
Obr. 41 Pohledy UML

Základními modely používanými v moderním iterativním vývoji software jsou:

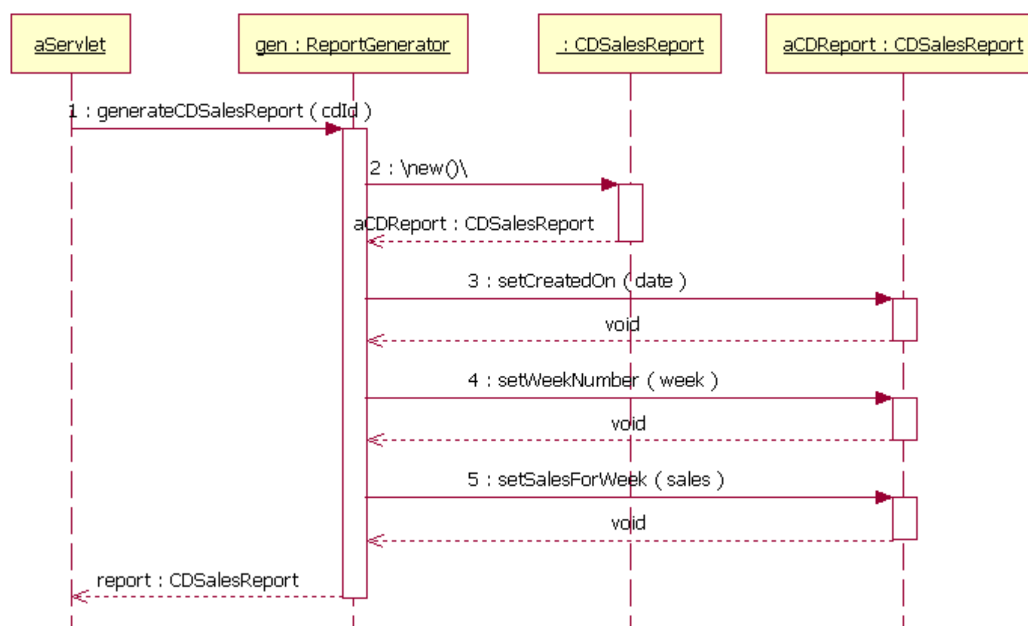
- Use case,
- Diagram tříd (objektů) + balíčky,
- Sekvenční diagram.



Obr. 8-42: Use case model



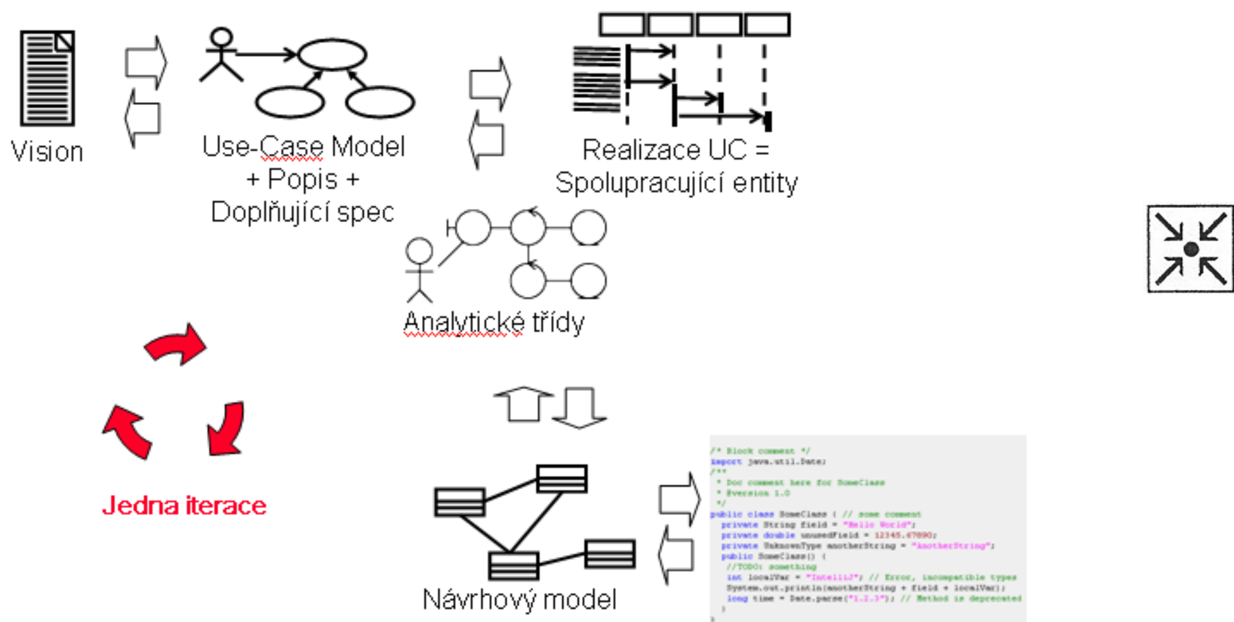
Obr. 43 Diagram tříd



Obr. 44 Sekvenční diagram

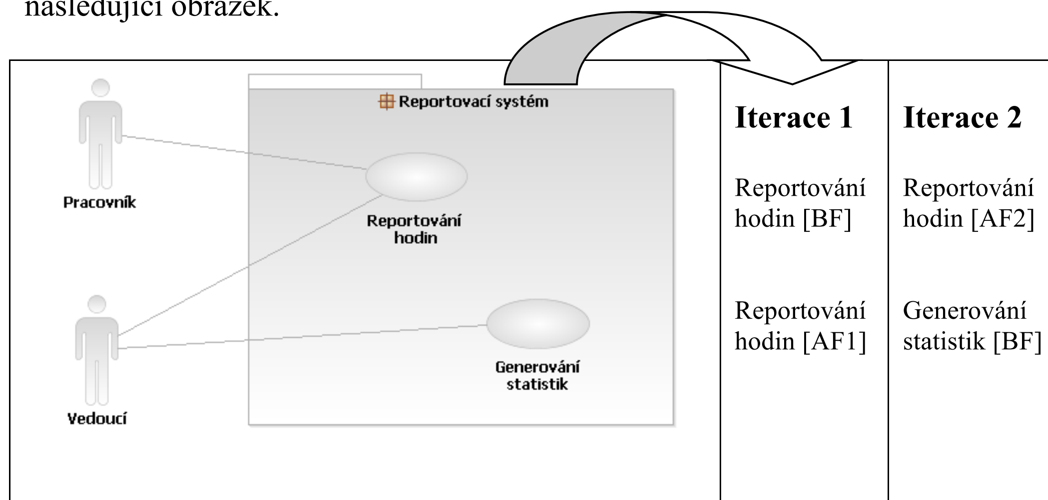
Use Case umožňují modelovat chování aplikace z pohledu uživatele, tj. jak uživatel používá (bude používat) danou aplikaci. Toto chování je slovně popsáno pomocí tzv. scénářů, které obsahují hlavní a alternativní toky událostí. Proto, abychom mohli tento vágní, jazykový popis implementovat, existují tzv. use case realizace. Jedná se o formalizovaný přepis bodů scénáře do diagramu sekvencí. Tento postup nám umožní identifikovat byznys objekty, které se potom určitým způsobem promítnou do návrhových objektů aplikace. Přepis či transformace návrhového modelu do kódu je již relativně snadným krokem. Následující obrázek ukazuje tento postup. Dané kroky opakujeme vždy v každé iteraci, kdy kompletně od požadavků po spustitelný kód implementujeme 1

nebo několik scénářů 1 či několika use case podle důležitosti z pohledu zákazníka.



Obr. 45 Vývoj řízený UC – v rámci jedné iterace kompletně analyzujeme, navrhneme, implementujeme a testujeme jednu část aplikace (scénář či celý use case) přes všechny technologické vrstvy (uživatelské rozhraní, aplikační logika, databáze)

Jelikož je RUP řízen use casy, slouží tyto také k plánování vývoje. Projektový plán (vysoko-úrovňová road map) obsahuje jako cíle jednotlivých iterací dané scénáře use case. Promítnutí use case modelu do projektového plánu zachycuje následující obrázek.



Obr. 46 Plánování projektu pomocí use case

Posledním příkladem znovupoužití use case, který si ukážeme, je případ testů.

Číslo TC	Tok	Hodnota	Výstup	Průběh
TC#001	UC1 hlavní tok	3h	3h	OK
TC#002	UC1 alternativní tok 1	-1h	0h	OK
…	…	…	…	…

Tabulka 8-4: Test case

Pro testování funkčností systému musíme projít ty kroky, které bude procházet uživatel, tj. jak bude on aplikaci používat = opět lze odkázat na scénáře konkrétního use case. Proto můžeme udržovat test case v konsistentním stavu s use case při jakékoliv změně (novém požadavku uživatele), jelikož je na ně pouze odkazováno a nedržíme tedy stejnou informaci na několika místech.

V neposlední řadě lze use case použít jako uživatelskou dokumentaci systému. Kdy konkrétní text scénáře okopírujeme a pouze doplníme o implementační obrazovky (ať ve formě testu či jako HTML stránku).

Kontrolní otázky:

1. Co je cílem fáze Inception?
2. Co je cílem fáze Elaboration?
3. Co je cílem fáze Construction?
4. Co je cílem fáze Transition?
5. Jaký je vztah mezi vodopádovým modelem a iterativním přístupem?
6. Kdy se snažíme odstranit technická rizika projektu?
7. Co vše by měla obsahovat Vize (Vision) vytvořená v Inception?

Úkoly k zamyšlení:

Pokuste se zamyslet nad finančními přínosy iterativního a vodopádového přístupu. Kdy aplikace vyvinutá tím kterým způsobem může začít vydělávat a jaké jsou možnosti variabilního, rozloženého financování v čase?

Korespondenční úkol:

Vypracujte seznam rizik včetně priorit a akcí na jejich odstranění či zmírnění pro projekt výstupu na Mount Everest. Pro každé riziko napište, ve které fázi a v jaké iteraci této fáze budou odstraněna/snížena.

Shrnutí obsahu kapitoly

V této kapitole jste se seznámili se čtyřmi fázemi iterativního způsobu vývoje podle metodiky RUP a také s milníky, které tyto fáze uzavírají. Zásadním rozdílem oproti vodopádu je neustálá spolupráce všech zúčastněných na projektu, fáze jsou spíše evolučními fázemi projektu, ne funkčně oddělenými bloky, v neposlední řadě je pak velmi brzy produkován hmatatelný výstup (spustitelná aplikace, ne jen vývojářské dokumenty), který je dán k dispozici zákazníkovi, čímž je umožněna zpětná vazba a jsou snížena určitá rizika plynoucí z nepochopení potřeb zákazníka. Jako poslední bod byl stručně zmíněn use casy řízený (use case driven) vývoj a místo UML ve vývoji software podle RUP.

9 Testování software

V této kapitole se dozvíte:

- Proč je nutné software testovat?
- Kdy mám při vývoji software testovat?
- Jaké jsou druhy testů a k čemu se používají?

Po jejím prostudování byste měli být schopni:

- Realizovat testování v současných metodikách vývoje software.
- Vybrat, které druhy testů jsou vhodné při vývoji software.

Klíčová slova této kapitoly:

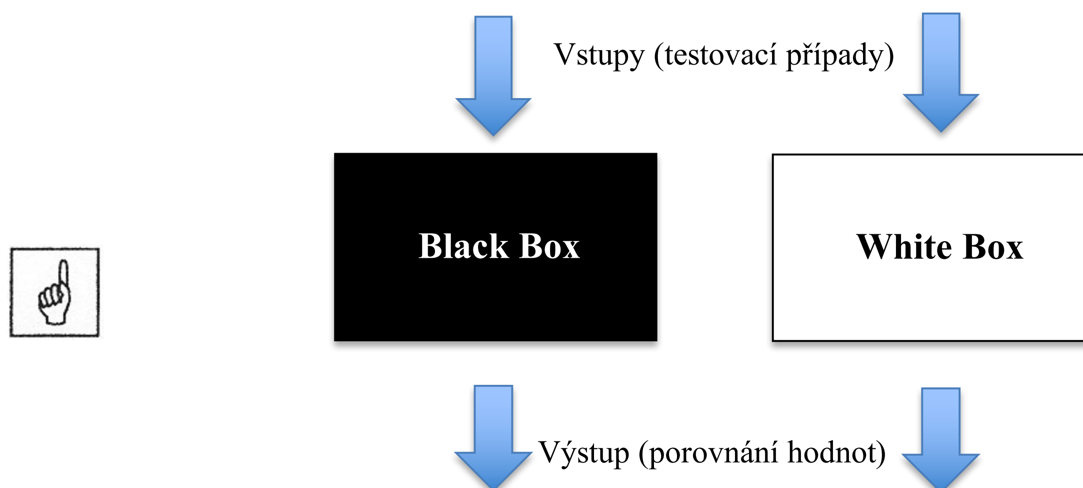
Testování, xUnit, funkční testy, nefunkční testy.

Doba potřebná ke studiu: 5 hodin

Průvodce studiem

Kapitola se věnuje zajištění kvality softwarového produktu. Nejprve je definován pojem testování, poté je vysvětleno, jakým způsobem se testování projevuje v různých metodikách softwarového vývoje. Dále je uvedeno základní rozdělení testů a popsány nejčastější testy používané v praxi.

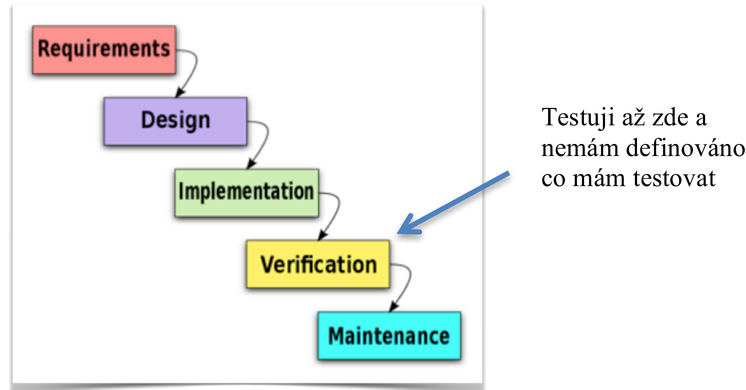
Málokdo z nás dokáže vytvořit složití software bez jediné chyby. Není to tím, že bychom byli špatní programátoři, ovšem běžný člověk udrží pozornost přibližně 45 minut a programátor je nucen bez přestávek pracovat mnohem delší dobu. Problém nejsou ani tak chyby samotné, ale nastavení vhodné míry kvality výsledného produktu. To, že je software kvalitní, nemusí znamenat nulovou chybovost. Problém tedy je nastavení vhodné míry kvality. Obecně kvalitu můžeme definovat jako schopnost objektu být použit. Chyba je pak vlastnost objektu, která snižuje jeho kvalitu. Testováním software se tedy snažíme co nejvíce chyb odstranit, abychom zvýšili výslednou kvalitu produktu. R. Patton o testování prohlásil, že cílem softwarového testera je vyhledávat chyby co nejdříve a zajistit jejich nápravu. Brzká identifikace chyby je pro nás velmi důležitá, protože čím dříve chybu odhalíme, tím jednodušeji (a hlavně levněji) ji opravíme. Pro tuto činnost budeme tedy potřebovat disciplínu nazvanou testování, která bude ověřovat kvalitu objektu tak, že hledá jeho chyby.
Obecně existují základní dva přístupy k testování: Black Box a White Box.



Přístup Black Box naznačuje, že jako testeři neznáme přesnou implementaci programu - vnitřek krabice je nám neviditelný. Díky tomu jako testeři nejsme zatížení implementačními detaily. Pokud jako programátoři vytvoříte formulář se vstupním polem a napíšete k němu popisek "Vložte kladné číslo", tak by vás zřejmě nikdy nenapadlo zadat vstupní hodnotu "A", protože to není číselná hodnota a program by v lepším případě vyhodil uživateli pro něj nesrozumitelnou výjimku, v horším se úplně ukončil. Běžný uživatel ovšem tento vstup vygenerovat může.
Přístup White Box je pak způsob testování, kdy máme k dispozici dodatečné informace o tom, jak je konkrétní problém programem řešen. Nemusí to byt pouze zdrojové kódy, ale také vývojový diagram nebo vstupní omezující podmínky. Tento druh testu používají zpravidla sami programátoři.

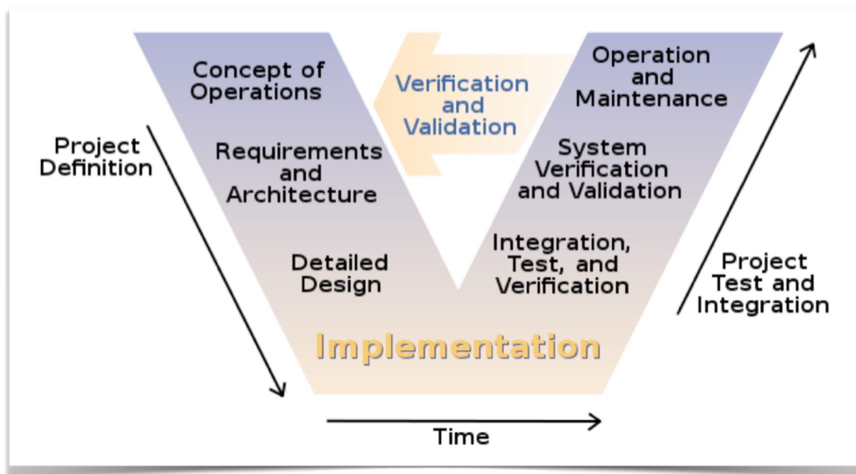
9.1 Testování ve vývojovém procesu software

Z předchozích kapitol již víte, že existuje více postupů (vývojových procesů) jak software vytvořit. Z pohledu aplikovatelnosti testování si tedy musíme položit následující otázku: *Musíme testovat ve všech fázích vývojového procesu?* Odpověď není lehká, protože to vždy záleží na konkrétní metodice vývoje. První pokud o udržení kvality software byl zakomponován již ve vodopádovém modelu - fáze testování (někdy uváděná jako verifikační). Testovalo se tedy pouze v této fázi a kvůli problémům s vodopádovým modelem vývoje u velkých projektů se často netestovalo vůbec. Navíc vodopádový model tuto fázi nerozděloval.



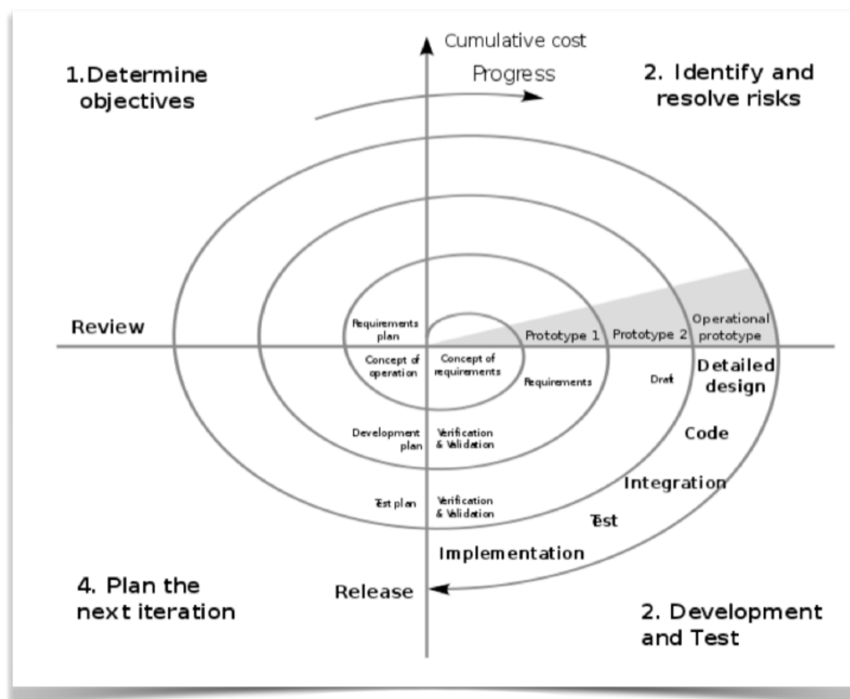
Obr. 47 Zjednodušený vodopádový model

Vy ovšem již nyní tušíte, že jiným způsobem se testuje uživatelské prostředí a jiným způsobem se testuje výpočet faktoriálu z příkazové řádky. Tento problém se snažil vyřešit V-Model, který byl rozšířením původního vodopádového modelu vývoje. Ten přinesl podstatné rozšíření tím, že rozdělil fázi testování na integrační testy, verifikační testy a údržbu a podporu. Toto rozdělení navíc namapoval na jednotlivé disciplíny konkrétní způsob testování. Disciplína sběr požadavků tak zároveň slouží pro definici akceptačních testů, disciplína návrhu a definici architektury se připravují integrační testy a disciplína moduly vyprodukuje (kromě zdrojového kódu) unitové testy.



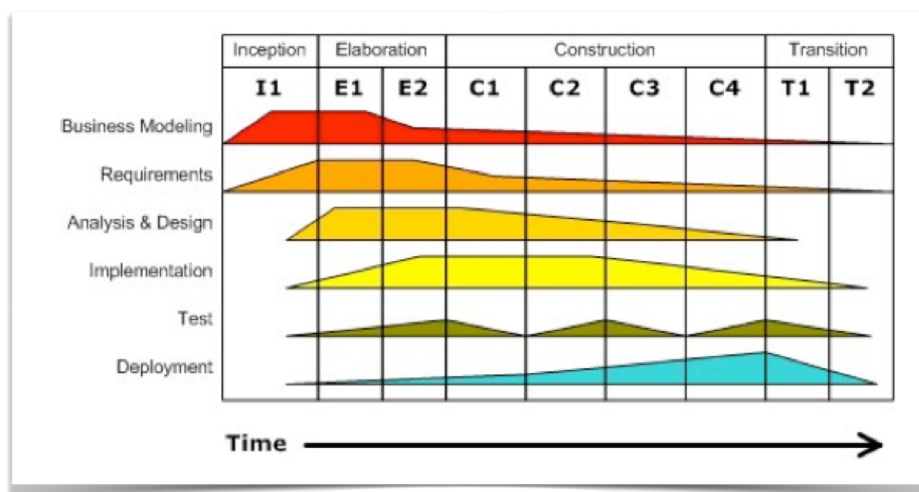
Obr. 48 V-Model

V-Model sice přinesl určité zlepšení kvality výsledného produktu, ovšem byl založen na vodopádu, a proto na velkých projektech zpravidla selhává. Cestou k iterativně-inkrementálnímu způsobu vývoje je dobré připomenout ještě spirálový model. Ten se snažil zajistit kvalitu software tím, že rozdělil dobu vývoje na jednotlivé prototypy, které se vždy před prezentací testovali. Výhodou je, že máme oproti vodopádu více než jednu možnost zvýšit výslednému produktu kvalitu. Nevýhodou pak, že doba vypuštění prototypu byla plánována zpravidla na půl roku a testovat dvakrát do roka je nedostatečné.



Obr. 49 Spirálový model

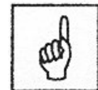
V současné době se jeví jako nejvhodnější iterativně-inkrementální způsob vývoje, který nám předepisuje testování neustálé. V praxi to funguje tak, že programátor vytvoří část kódu a poté si na něj sám napíše test. Ti lepší vývojáři píší test dříve, než vytvoří samotný kód (zájemce odkazuji na způsob vývoje známý jako Test-Driven Development). Tím, že jedna iterace trvá maximálně 6 týdnů, zkracuje se nám velmi výrazně perioda testování. Na následujícím obrázku vidíte objem práce, které zabírá testování v celkovém čase vývoje. Všimněte si, že testování neprobíhá po každé iteraci, ale kontinuálně přes celé trvání iterace.

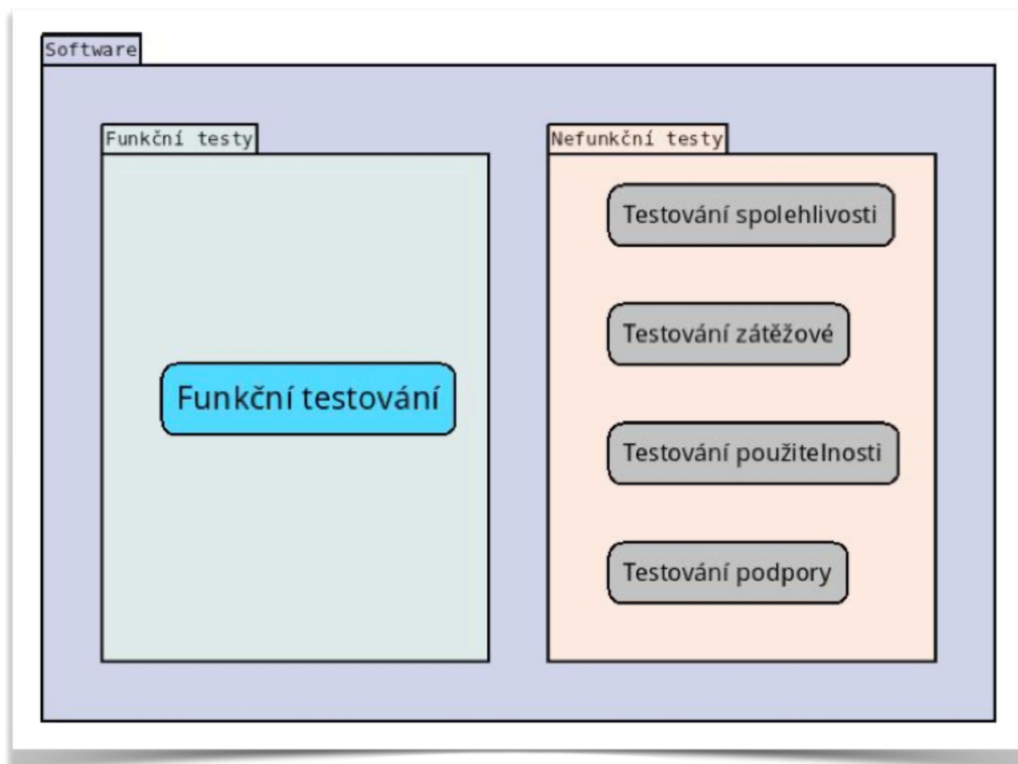


Obr. 50 Objem testování v iterativně-inkrementálním vývoji

9.2 Druhy testů

Stejně jako vývoj software dle UML má několik stupňů abstrakce, můžeme podobně rozdělit i testování. Budeme tedy používat základní rozdělení na testy funkční a nefunkční. Možná si říkáte, proč bychom dělali nefunkční testy, když potřebujeme, aby software fungoval. Samotné funkční testy jsou totiž orientovány na přímé požadavky uživatele. Nepřímo ale všichni uživatelé chtějí, aby byl software spolehlivý, přijatelně rychlý i při větší zátěži a aby byl jeho výrobcem dlouho podporován.





Obr. 51 Základní rozdělení testů

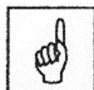
Do funkčních testů pak spadá vše, co ověří kvalitu funkcí vyžadovaných zákazníkem. Testování funkčních požadavků je dobré podpořit nějakou normou či doporučením. Z kapitoly o životním cyklu software bychom mohli využít třeba normu ISO/IEC 12207 a rozdělit testování na:

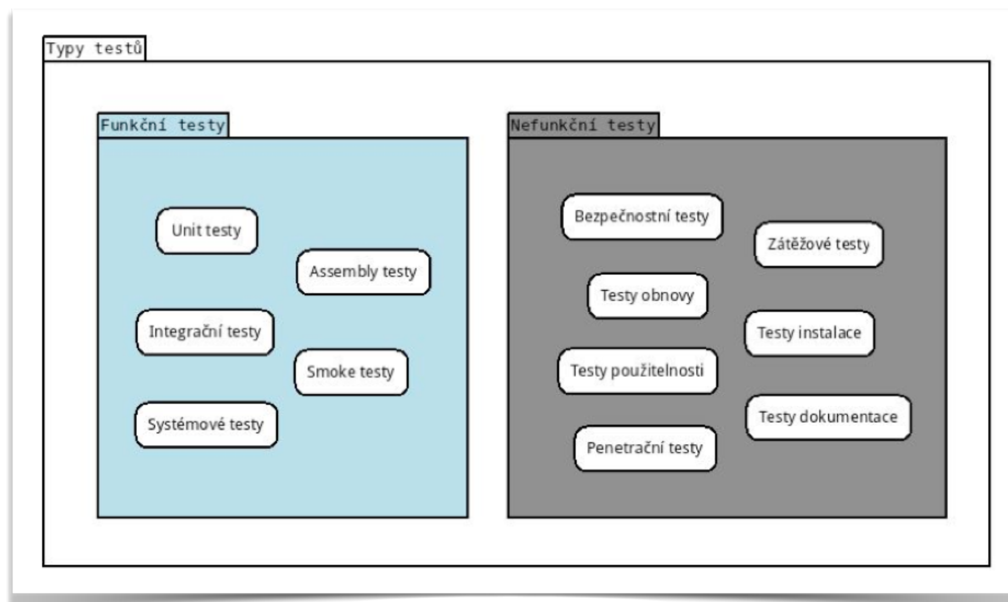
- Hardwarové požadavky,
- softwarové požadavky,
- ostatní požadavky.

Pokud bychom testování vzali z pohledu specifikace požadavků za pomocí FURPS+, pak by testování mohlo být rozděleno následovně:

- Funkcionalita - systém je schopen exportovat výstup do PDF.
- Použitelnost - prvky GUI jsou dostatečně velké pro ovládání na tabletu.
- Spolehlivost - doba mezi výpadky je průměrně 3000 hodin.
- Výkonnost - aplikace funguje rozumnou rychlostí i když s ní pracuje naráz 3000 zaměstnanců.
- Podporovatelnost - aplikace je postavena na moderní technologii .NET, a proto je dobrý předpoklad budoucího rozšíření.

Některé z výše uvedených příkladů lze zajistit pouze smluvně. U ostatních si ukážeme, že k nim existuje konkrétní test. Že je systém schopen exportovat PDF můžeme ověřit unitovým testováním, nebo testováním uživatelským. Prvky GUI lze ověřit vždy uživatelským testováním. Pokud bychom chtěli testovat průměrnou dobu mezi výpadky, zaplatíme si cloudovou službu a vytvoříme makro, které bude simulovat běžný provoz aplikace. Ověření výkonnosti pak provedou zátěžové testy. Zde vidíte, jak jsou nefunkční testy důležité a nelze je tedy při vývoji software opomenout. Všechny běžně používané testy jsou zařazeny do kontextu našeho rozdělení na následujícím obrázku:





Obr. 52 Druhy testů

Zkusme si je nyní trochu podrobněji popsat:

- **Unit testy** - tyto testy se používají na nejnižší úrovni - testujeme metody tříd. Největší testovaná jednotka je třída, nejmenší pak metoda. Pro tyto testy je typické použití frameworku xUnit (nUnit, jUnit, phpUnit). Jako testeři napíšeme vlastní metodu, ve které porovnáváme očekávané vstupy s těmi reálnými pomocí metod assert.
- **Assembly testy** - provádí se zpravidla po unitovém testování a mají za úkol zjistit, zda spolu jednotlivé části kódu již dříve otestovány unit testy fungují.
- **Integrační testy** - zde testujeme, jestli jednotlivé třídy pracují v pořádku spolu v hotovém systému. Většinou se testují hraniční rozhraní a větší seskupení tříd. Tento způsob testování je výhodný, pokud pracujeme v týmu a každý programátor má určen svůj vlastní subsystém.
- **Smoke testy** - pokud je vytvářený software většího rozsahu, může trvat testování delší dobu (klidně i v řádu hodin). Programátor proto vytypuje klíčové hraniční rozhraní a provede se test jen na těchto pár místech. Pokud je zde nalezena chyba, je zpravidla rychle opravena a nemusí se čekat na výsledky kompletního testování. U jednodušších systémů se může stát, že by se smoke testy kryly s testy integračními.

- **Systémové testy** - tyto druhy testů mají za úkol zkontrolovat, zda aplikace jako celek funguje správně a dodává zákazníkovi přidanou hodnotu. Proto se provádějí v několika kolech.
- **Akceptační testy** - tyto testy provádí zákazník a většinou se provádějí před podpisem předávacího protokolu a doručením finální uživatelské dokumentace. Jedná se o tzv. proklikání software a zkontrolování funkčnosti objednaných rysů software. Zákazník akceptačním testováním samozřejmě není schopen pokrýt všechny případy a stavy, proto pouze akceptační testování pro zajištění kvality nestačí.

Mezi nejčastější nefunkční testy používané v praxi patří:

- **Bezpečnostní testy** - jedná se o ošetření technické odolnosti proti chybným vstupům, ověření autorizace a autentizace. Mezi běžně odhalené chyby pak patří např. SQL injection, nebo cross-site scripting.
- **Zátěžové testy** - tyto testy mají za úkol potvrdit nám, že systém je schopen mít rozumnou odezvu (rozumějte v řádu ms) i při zvýšené zátěži. Testovací software pro zátěžové testování vytvoří několik simultánních požadavků na testovaný systém a měří, za jak dlouho mů přijde na požadavek odpověď.
- **Testy obnovy** - testy obnovy mají za úkol zjistit, jak se aplikace vypořádá s obnovou do chodu. Typický test obnovy je během práce s programem vypnout počítač hlavním vypínačem, poté nastartovat znovu aplikaci a kontrolovat konzistenci dat. V případě distribuovaných aplikací se test obnovy provádí odpojením ethernetového kabelu a následně kontrolou došlých dat.
- **Testy instalace** - testujeme, jestli je software schopný instalace na konkrétní platformě. Tento problém se zdá být jednoduchý, představte si ale aplikaci, která se instaluje způsobem, že nejprve stáhne ke klientovi část zdrojového kódu a ten teprve poté zkompiluje a integruje di výsledné aplikace. Požadavky na automatizovanou kompilaci u klienta již není triviální. Do kategorie testů instalace spadá i možná aktualizace programu.
- **Testy použitelnosti** - tyto testy pokrývají rozsáhlou skupinu, použitelnost můžeme u webové aplikace například definovat tak, že bude použitelná i na slabším internetovém připojení. Z hlediska desktopové aplikace by mohlo být hledisko použitelnosti bráno tak, že aplikaci lze provozovat i na starších operačních systémech.
- **Testy dokumentace** - pokud předáváme software, je jeho nezbytnou součástí uživatelská dokumentace. Z pohledu uživatelské dokumentace testujeme srozumitelnost a konzistentnost. Kromě dokumentace pro uživatele se často může předávat i programátorská dokumentace - obvykle ve formě JavaDoc. Na testování programátorské dokumentace lze nastavit jednoduchou metriku - procento pokrytí kódu komentáři. Nejedná se o kontrolu každého řádku kódu, ale o číslo, které vyjadřuje poměr celkového počtu metod a metod opatřených komentáři.
- **Penetrační testy** - testy průniku mají za úkol vyzkoušet co nejvíce známých způsobů průniku do konkrétního systému. Pokud například použijete pro tvorbu redakčního systému Wordpress, musíte jej správně nastavit. V opačném případě otevíráte útočníkovi zadní vrátka do systému. Pro často používané systémy existuje sada penetračních testů

(skriptů), které spustíte a po dokončení vám testy vypíší možné bezpečnostní rizika. Kromě špatných nastavení frameworků může v systému být i obecná bezpečnostní chyba. Proto jsou penetračními skripty většinou testovány i nejčastější bezpečnostní chyby v software před jejich záplatováním.

Kontrolní otázky:

1. Co znamená testovací přístup Black Box a White Box?
2. Kdy testujeme v iterativně-inkrementálním způsobu vývoje?
3. Co je to smoke test?

Úkoly k zamyšlení:

Pokuste se zamyslet nad tím, kdy je pro nás testování přínosné a kdy nás naopak časově zbytečně zatěžuje. Pro které typy softwarových projektů je vhodné použít zátěžové testy, pro které typy projektů pak penetrační testy.

Korespondenční úkol:

Vyhledejte na internetu nějakou instalaci redakčního systému (Wordpress, Joomla, Drupal) a pomocí on-line nástroje pro penetrační testování ověřte, zda neobsahuje nějaké zranitelnosti. Výstup z nástroje pak pošlete na kontrolu.

Shrnutí obsahu kapitoly

V této kapitole jste se dozvěděli hlavní důvody proč testovat software a základní členění testování. Dále je zde uvedeno, jakým způsobem je testování software integrováno do stávajících metodik vývoje software. Závěrem kapitoly je uvedeno několik druhů testů, které se v praxi používají včetně příkladů jejich použití.

10 Provoz a údržba podle ITIL

V této kapitole se dozvíte:

- Proč je nutné definovat procesy pro údržbu a podporu provozované aplikace?
- Co je to ITIL?
- Jaké procesy definuje ITIL?

Po jejím prostudování byste měli být schopni:

- Pochopit potřebu existence standardních procesů pro provoz a údržbu.
- Vyjmenovat procesy ITIL a jejich účel.

Klíčová slova této kapitoly:

Procesní řízení, ITIL, provoz a údržba, ISO.

Doba potřebná ke studiu: 6 hodin

Průvodce studiem

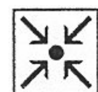
Kapitola představuje oblast provozu a údržby vyvinutého software pomocí standardních postupů, konkrétně pomocí procesů definovaných frameworkem pro provoz a údržbu ITIL. Text vysvětluje, co je to ITIL a na příkladu ukazuje jeho aplikaci. Stručně jsou pak popsány jednotlivé procesy ITILu.
Na studium této části si vyhraďte 6 hodin.

Dosud jsme se v tomto textu mimo jiné zabývali iterativním vývojem software, který je zákazníkovi doručován v pravidelných releasech. Na základě zpětné vazby je pak možné aplikaci dále upravovat, doplňovat a vylepšovat, aby zákazník dostal řešení, které skutečně řeší jeho problémy v business doméně, byl spokojený a toto řešení mu přinášelo zisk a konkurenční výhodu. Stejně důležitou částí jako vývoj softwarového produktu je však také jeho provoz. Právě na tuto oblast se zaměříme v následujícím textu.

Kritickým faktorem pro úspěšný vývoj, doručení a provoz softwarového produktu je kooperace tří skupin, jedná se o zákazníky a uživatele, dále o vývojáře a v neposlední řadě o provozovatele (údržbu) aplikace.

Následující příklad se snaží ukázat problémy, které nás mohou potkat při provozu existujícího software. Dozvíme se, proč je důležité mít definované standardní procesy a jaké v této oblasti existují best practices.

10.1 Špatný scénář



Provoz aplikace doprovází několik nutných zásahů. Pokud provozujeme ekonomickou aplikaci, je třeba dbát na to, abychom implementovali nové zákony a směrnice. Pokud provozujeme výrobní aplikaci, je třeba reflektovat změny technologie, výrobních linek apod. Evidence, ohodnocení, implementace, testování a integrace změny (či nové funkčnosti) je však pouze jedním zásahem do provozované aplikace. Daleko závažnějším případem, který

je třeba co nejdříve řešit je výpadek aplikace nebo neočekávané chování, které znemožňuje uživatelům systému práci. Čím déle a čím více uživatelů nemůže pracovat, tím více peněz toto organizaci stojí.

Jednoduchý příklad:

Náklady na jednoho zaměstnance ...... 500 Kč / h

Počet zaměstnanců .... 200 celkem

Výpadek aplikace 3h

Nemožnost práce 30 zaměstnanců po dobu 3 hodin z důvodu výpadku aplikace stojí organizaci: 30 x 3 x 500 = 45.000 Kč!!!

Z tohoto důvodu potřebujeme mít také mechanismy, které nás o výpadku informují a nástroje, které nám pomohou výpadek vyřešit. S tím souvisí nejen hledání příčiny, ale také hledání informací o hardware a software daného uživatele, o jeho konfiguraci a jednotlivých verzích software, který používá. Dobrým pomocníkem je v tomto případě také znalostní báze obsahující řešené problémy s popisem příznaků a s řešením.

Následující příklad ukáže, jaké může neexistence těchto mechanismů (procesů) způsobit problémy. Budeme se zabývat řešením incidentů, problémů (skrytá příčina způsobující incidenty) a změn od počátku (jejich nahlášení či zjištění) až po jejich vyřešení (implementace do provozního prostředí). V příkladu si ukážeme špatný i dobrý scénář, aby byl patrný rozdíl.

Účastníci:

Mary … uživatel mající problém,
Pete … aplikační programátor,
John … systémový administrátor,
A další osoby vystupující podle potřeby.

Pokuste se sami najít problémy a jejich příčiny v následujícím „katastrofálním“ případu, některé kroky jsou naznačeny cíleně příliš extrémně, aby byly možné problémy viditelné na první pohled.

Pondělí ráno

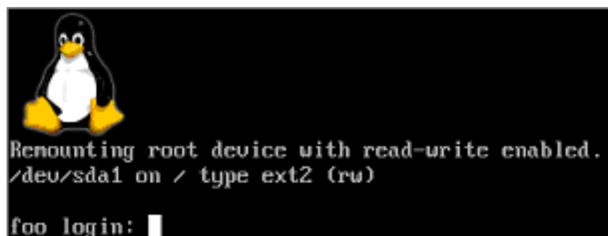
- Mary používá ke své práci program SkladAObjednávky v1.1, po hodině práce program přestane odpovídat a zhroutí se a již nejde znovu spustit.
- Mary zavolá aplikačního programátora Peta, jelikož jsou kamarádi a znají se, aby jí pomohl s daným problémem.
- Okamžitě po hovoru na to Pete zapomene, jelikož má spoustu práce s chystaným buildem. Mary se však v průběhu společného oběda připomene.
- Pete se jí při obědě zeptá na nějaké symptomy (chybové hlášky, apod.), Mary si už na moc od rána nevzpomíná, ale něco Petovi přece jen poví.
- Po obědě se však Pete opět vrací ke své práci a na Mary zapomíná.
- Mary pořád nemůže pracovat s aplikací na zpracování objednávek a dělá nedůležitou práci.

Pondělí odpoledne

- Mary opět volá Petovi, aby se informovala o pokroku.
- Pete se naštve, protože ho Mary vyrušuje od práce a on potřebuje dokončit build pro zítřejší nasazení.
- Pete tedy přestane programovat a začne se zabývat identifikací problému, proč aplikace nefunguje jak má.
- Mary pořád vykonává nepodstatnou práci, důležité objednávky stále nejsou zpracovány.
- Odpoledne v 16h jde Mary domů.
- Pete zůstává v práci až do 20h a hledá kde je program provozován, na jakých serverech běží (HW) a jaký middleware používá – aplikační server a databázi (SW).

Úterý ráno

- Pete potřebuje dokončit build určený k nasazení, proto přichází dřív do práce i přesto že včera pracoval do večera.
- Mary přišla do práce raději později než obvykle, protože si nebyla jista, zda už bude její aplikace funkční.
- Poté, co Pete dokončil build, pokračuje v hledání serverů z předchozího dne – nachází je, jedná se o Linux servery.
- Pete je naštěstí linuxový nadšenec a tudíž zná Linux prostředí (příkazovou řádku), ale bohužel nemá uživatelský účet.
- Pete se ptá Johna (sysadmin) na nějaké přihlášení.
- John jako Petův dobrý kamarád mu dá root heslo.



Úterý odpoledne

- Pete se přihlásí jako root a najde program SkladAObjednávky v1.1.
- Náhodou si při startu MC všimne, že je disk serveru plný.
- Smaže tedy nějaké dočasné soubory, to vše jako root.
- Poprosí Johna aby restartoval Oracle DB a Apache, jelikož zjistil, že je program využívá a že oba nejedou.
- John je restartuje, bez toho aniž by někdo uvědomil další uživatele serveru. Nyní program SkladAObjednávky v1.1 opět běží.
- Pete volá Mary, Mary začíná opět pracovat s aplikací.
- Pete se vrátí k práci na buildu pro zítřejší nasazení – spouští a analyzuje testy.
- Zůstává v práci opět až do 20h do večera.

Středa ráno

- Mary zpracovává poslední objednávky, ale po 2 hodinách práce se opět objeví stejný problém.
- Volá Petovi, ale nikdo telefon v jeho kanceláři nebere, jelikož je Pete u zákazníka a instaluje build, který poslední dva dny dolaďoval a testoval.
- Mary mu tedy zavolá na soukromý mobilní telefon.
- Pete volá zpátky Johnovi, aby se na problém podíval, jelikož je s ním od včerejška alespoň trochu obeznámen – ví o které servery se jedná.
- John opět vidí plný disk, smaže opět nějaké dočasné soubory a některé nepotřebné logy.
- Poté John restartuje opět všechny služby operačního systému a vidí, že opět všechno funguje správně.

Středa odpoledne

- John píše skript, který bude pravidelně zálohovat na jiný server a mazat dočasné soubory a některé nepodstatné logy.
- John si také zkopíroval všechny logy programů běžících na tomto serveru, aby mohl zjistit, kde je chyba, proč se disk stále plní.
- Mezi tím, než se mu nahrají všechny soubory začne analyzovat již stažené, asi po hodině práci si všimne, že se velmi dlouho stahuje Oracle DB log – důvodem je jeho velikost (960 MB!!!).
- Prozkoumá tedy tento log přímo na serveru a zjistí, že obsahuje programátorské výpisy.
- Přepíše skript tak, aby mazal i Oracle log a původní zálohuje.
- Mary může opět pracovat s aplikací, problém se již neopakuje.
- John prohledává Internet a hledá ve fórech, zda se s tímto problémem již někdo setkal a zda je problém řešen. Nenajde žádnou odpověď, reportuje tedy chybu společnosti Oracle.

Závěry:

I když je daný příklad velmi ostrý a chyby viditelné, spousta organizací opravdu pracuje na podobných principech a nepřipadá jim to divné, jsou-li na toto upozorněny. To je bohužel moje osobní zkušenost. Z tohoto příkladu můžeme udělat několik závěrů:

- Pete je přetížený a naštvaný.
- Build může obsahovat chyby, kterých si díky únavě a napětí nemusel všimnout.
- Pete přistupuje na servery, kam nemá právo přístupu a to dokonce jako root a navíc zde maže jako root soubory!
- Mary nemohla zpracovávat téměř dva dny objednávky!
- Znovu se opakující incidenty.
- Kořenová příčina incidentů stále nevyřešena!
- John jako administrátor začíná zkoumat problém (dělat svou práci) až třetí den od incidentu.

10.2 Lepší scénář

Nyní si povíme, jak by stejný případ mohl vypadat v případě, kdyby daná organizace měla implementovány procesy podle ITIL. Opět se zde vyskytují stejní účastníci, jen průběh řešení incidentu bude odlišný.

Účastníci:
Mary … uživatel mající problém,
Adam … pracovník podpory,
A další osoby vystupující podle potřeby.

Pondělí (Incident Management):

- Mary používá ke své práci program SkladAObjednávky v1.1, po hodině práce program přestane odpovídat a zhroutí se a již nejde znovu spustit.
- Mary vytvoří záznam o incidentu[1] v Service Desk nástroji, který je jediným kontaktním místem (SPOC – Single Point Of Contact). Záznam je reportovaný na IT službu SkladAObjednávky.
- Incident manažer přiřadí kategorii (aplikace) a prioritu (vysoká) incidentu a přiřadí incident Adamovi.
- Mary je o přiřazení incidentu řešiteli notifikována automaticky e-mailem.
- Adam obdrží o svém přiřazení taktéž notifikaci, přeruší aktuální práci z důvodu vysoké priority nového incidentu a začne jej řešit.
- Adam začne prozkoumávat incident s použitím znalostní báze (KB – Knowledge Base), konfigurační databáze CMDB a automatických nástrojů.
- Adam ví, na kterém serveru program běží a jaké jiné programy využívá díky katalogu IT služeb (viz Tabulka 10-1).

[1] Pojem ITIL. Incident je událost, která znemožní pracovat nebo způsobí omezení určité IT služby.

Service Name	Service Users	Configuration Items
StockControl	Mary ...	StockControl Program v1.1 Tomcat v5.5 Oracle 9i Red Hat Enterprise Linux 5 Server Prague Switch1 Switch3 Intranet
Internet	...	Internet Service Provider Firewall Zone F v3.2

Tabulka 10-1: Příklad jednoduchého katalogu IT služeb

- Adam díky automatickému nástroji pro správu konfigurace zjistí, že disk serveru, na kterém aplikace běží, je plný.
- Smaže tedy dočasné soubory a začne prozkoumávat pouze log pro Oracle DB a Apache, jelikož ví z katalogu IT služeb, že služba tyto využívá.
- Okamžitě si všimne velikého logu Oracle databáze.
- Zálohuje a vymaže tento log, restartuje pouze danou službu a vyzkouší její funkčnost.
- Okamžitě také připraví skript, který bude pravidelně zálohovat na jiném stroji a poté mazat původní Oracle log (tzv. workaround) a instaluje ho. Služba stále funguje dobře.
- Poté vytvoří v Service Desk aplikaci záznam o problému (přiřadí Oracle skupině, která řeší problémy s Oracle databázemi a připojí odkaz na zálohovaný log) – proto, aby se zkoumala a vyřešila kořenová příčina tohoto incidentu, jelikož ji sám neodhalil.
- Nakonec Adam aktualizuje záznam o incidentu a uzavře jej.
- Mary je o uzavření / vyřešení opět automaticky notifikována e-mailem, takže nyní ví, že může danou IT službu opět využívat.
- Po uzavření incidentu vytvoří Adam záznam ve znalostní bázi (KB) s popisem incidentu, jeho symptomů a přiloží workaround, který daný incident řeší. Tento záznam má pomoci rychle vyřešit incident tohoto typu, pokud se náhodou někdy v budoucnu opět objeví.

Obr. 53 Příklad záznamu o incidentu v systému OmniTracker

Využívali jsme funkci Service Desk a proces zvaný Incident Management. V této fázi je však vyřešen pouze incident. Viděli jsme, že během několika hodin/možná desítek minut, kdy se okamžitě incidentem začaly zabývat osoby k tomu určené, mohla Mary opět pracovat. Důležitá služba sloužící ke zpracování objednávek tedy není 3 dny nedostupná, zákazníci nemuseli vůbec nic poznat. Nyní je však třeba vyřešit ještě kořenovou příčinu (pojmy ITIL tzv. problém) tohoto incidentu, aby se v budoucnu neopakoval. Podívejme se, jak budeme toto řešit s pomocí procesu Problem Management:

- Je zformován tým Problem Managementu (PrM), jelikož v Service Desku vznikl požadavek na řešení problému.
- Rachel, specialistka na Oracle, je přiřazena k danému problému a začne zkoumat záznam o problému, přidružený incident a také připojený log soubor Oracle databáze.
- Vidí, že v logu se objevují programátorské výpisy.
- Připojí se k Oracle webu, k jejich nástroji na reportování chyb, ale nenajde zde žádnou podobnou chybu.
- Proto zde, v Oracle reportovacím nástroji ihned vytvoří záznam o chybě.
- Po několika dnech je notifikována e-mailem, že Oracle vydal opravnou záplatu, která řeší tento problém.
- Rachel tedy vytvoří požadavek na změnu (RfC – Request for Change), který požaduje a vysvětluje nutnost implementace této záplaty.

Nyní existuje řešení kořenové příčiny a je požadavek na implementaci tohoto řešení do provozního prostředí. Za schválení požadavku, otestování a nasazení záplaty jsou zodpovědné procesy Change a Release Managementu. Postup by mohl vypadat následovně:

- Změna je schválena Change Managerem, jelikož její implementace téměř nic nestojí, řeší kořenovou příčinu, tudíž odstraňuje dočasné řešení, tzv. workaround.
- Oracle záplata je otestována v testovacím prostředí, všechny testy proběhnou v pořádku, je tedy možné záplatu nainstalovat i do provozního prostředí.
- Záplata je instalována do provozního prostředí ve večerním okně určeném pro údržbu (od 2.00 do 3.00 v noci) tzv. maintenance window.
- Po úspěšné instalaci záplaty je odstraněno dočasné řešení – skript.
- Poté Rachel doplní řešení problému a uzavře záznam.
- Nakonec aktualizuje záznam ve znalostní bázi vytvořený Adamem a přidá k němu odkaz na záplatu řešící tento problém.

Použití formálních, popsaných a automatizovaných procesů definovaných podle ITIL umožnilo provést všechny nutné aktivity mnohem efektivněji než ad hoc přístup v prvním případě.

Na závěr můžeme říct následujících několik bodů. Incident byl vyřešen mnohem dříve, než v prvním případě. Řešili ho lidé k tomu určení a jeho řešení neovlivnilo práci jiných lidí v IT oddělení (např. programátorů). Tito lidé věděli, co a jak mají dělat. Navíc hned ten samý den proběhlo zkoumání a řešení kořenové příčiny, z důvodu zamezení opakujících se incidentů a z důvodu strukturálního řešení, ne jen dočasného workaroundu.

Automatizované nástroje usnadnili spoustu práce s diagnostikou a hledáním. O všem existují záznamy v nástroji Service Desk, je snadné vysledovat změny a kroky provedené pracovníky podpory. Znalostní báze (KB) může pomoci při příštím řešení podobných incidentů/problémů.

10.3 Co je ITIL

Zkratka ITIL znamená IT Infrastructure Library, anglicky mluvícím je tedy zřejmé, že se jedná o nějakou knihovnu, která nám dává doporučení co a kdy dělat při provozu a údržbě IT služeb. Hned na začátku zmíníme, že ITIL neříká jak toto dělat, pouze říká co a kdy. Jelikož mluvíme o termínu správa IT služeb, měli bychom si nejdříve říci, co to je IT služba.

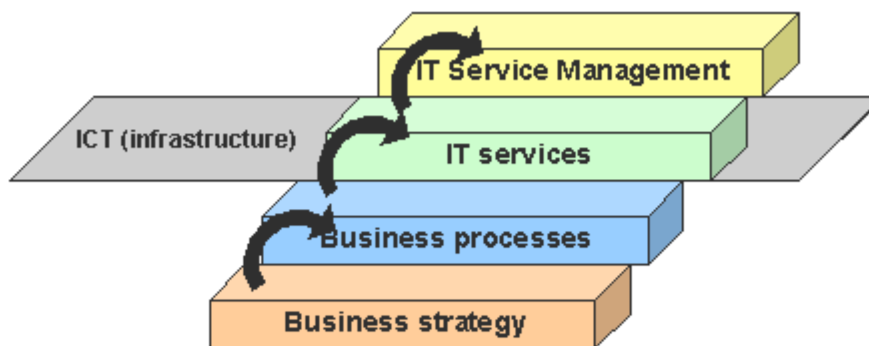
IT služba je skupina příbuzných funkcí jednoho či více IT systémů. Z pohledu uživatele se jeví jako celek, jako dále nedělitelná entita. Cílem IT služby je podpora podnikových procesů. IT služba se skládá ze software, hardware, komunikačních linek. IT potřebuje mít tyto údaje o každé službě k dispozici z důvodu monitoringu, implementace změn, reportování.

Příkladem IT služeb může být:

- E-mail
- Tisk dokumentů (na tiskárnu, do pdf, ...)
- Skladování
- Repozitory pro verzování software

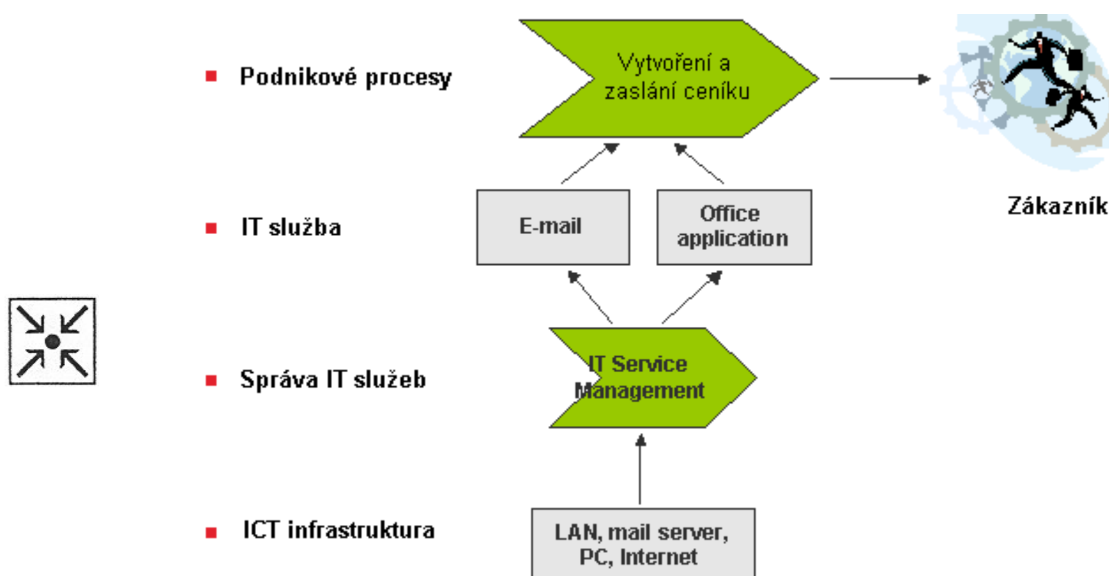
Druhým pojmem, který je pro ITIL klíčový je **proaktivní přístup**. Klasický reaktivní přístup pouze reaguje na události, které nastaly. Oproti tomu proaktivní přístup se zabývá aktivní detekcí a řešením možných problémů, potřebných změn v ICT infrastruktuře, které by v budoucnu mohly vyvolat incidenty či přinést problémy.

Nyní se můžeme vrátit k vysvětlení pojmu **správa IT služeb**. Následující obrázek znázorňuje místo IT služeb a jejich správy, podnikových procesů a podnikové strategie. Základem podniku je jeho podniková strategie, která definuje směr, kterým se chce podnik ubírat, co prodává či jaké nabízí služby, na jaké zákazníky se zaměřuje, v jakých lokalitách chce působit, zda bude mít kamenné pobočky či pouze elektronický obchod a spoustu dalšího. Pro realizaci těchto cílů slouží podnikové procesy (více o podnikových procesech a procesním řízení viz [Luk04] a [Pro06]). Proto, aby mohly být podnikové procesy efektivně vykonávány, používáme IT služby, jež dané podnikové procesy podporují. K provozu těchto služeb využíváme ICT infrastrukturu. Samotné služby pak potřebují být také nějakým způsobem spravovány, tj. definovány, nasazovány, provozovány, k tomu slouží správa IT služeb



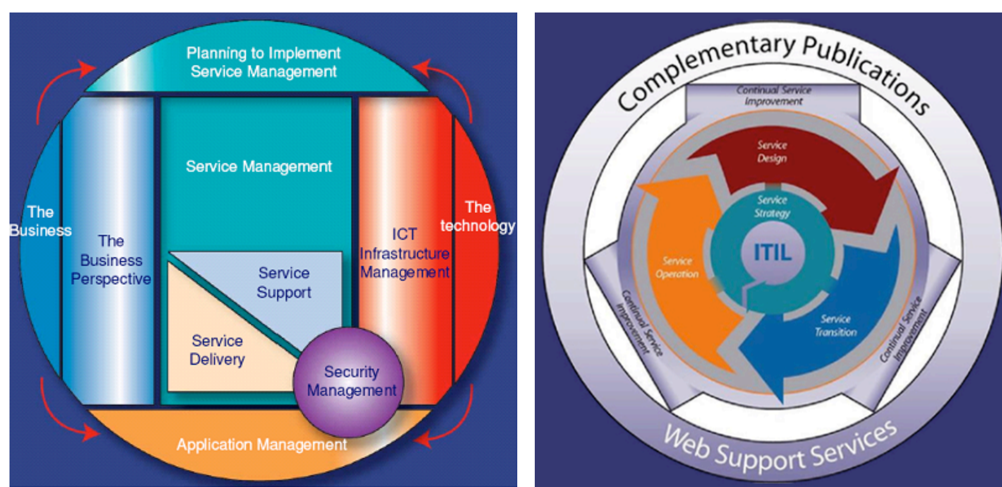
Obr. 54 Vztah podnikových procesů, IT služeb a jejich správy

Následující příklad ukazuje situaci na příkladu, aby čtenář lépe pochopil, co že je to ten podnikový proces a co že je to IT služba. Cílem podnikového procesu je doručit zákazníkovy aktuální ceník produktů, k tomu slouží podnikový proces nazvaný „Vytvoření a zaslání ceníku“. Proto, abychom mohli výstup tohoto procesu uskutečnit, potřebujeme 2 IT služby, konkrétně nějakou Office aplikaci (např. Star Office StarWriter, StarCalc či MS Word, MS Excel, ...) a e-mail. Tyto IT služby spravuje správa IT služeb a provozovány jsou na ICT infrastruktuře organizace, což je připojení k síti, e-mailový server, PC stanice, na kterých je nainstalována nebo provozována Office aplikace a v neposlední řadě připojení k Internetu, aby mohl být e-mail odeslán. Pro jednoduchost vynecháváme další nezbytné prvky infrastruktury jako jsou switche, e-mailový server apod.



Obr. 55 Vztah podnikových procesů, IT služeb a jejich správy

ITIL je tedy frameworkem sloužícím ke správě IT služeb. Je to pouze framework, není metodikou, říká pouze co a kdy dělat, ale již neříká, jak toto dělat. ITIL vznikl jako framework postavený na nejlepších zkušenostech z praxe (best practices), na základě úspěšných projektů, stejně jako RUP v případě vývoje software. ITIL definuje terminologii, jeden jazyk, aby lidé mluvili stejnými slovy o stejných věcech. Dále definuje procesy, jejich aktivity a role. Konkrétně je ITIL knihovna publikací (knihy, CD) obsahující best practices pro správu IT služeb. Procesy zahrnuté v ITIL byly původně vyvinuty pro britskou vládu, nyní jsou de facto i de jure standardem používaným celosvětově.



Obr. 56 ITIL framework, vlevo v2, vpravo v3 (zdroj: ITIL)

První verze ITIL z konce 80. let měla 46 knih, přepracovaná v2 která spatřila světlo světa kolem roku 2000 má již pouze 8 knih a připravovaná verze 3 má 5+1 přehledovou knihu a také nový model životního cyklu. V době psaní tohoto učebního textu je aktuálně připravena k vydání verze 3, ale ještě není oficiální, proto se budeme věnovat popisu verze 2 i verze 3 (obě na Obr. 56).

Jak již bylo zmíněno, ITIL není standardem, vůči kterému je možné organizaci posuzovat a certifikovat. Tímto standardem je buď původní britská norma BS 15.000 přepracovaná v roce 2003 nebo nově vydaná mezinárodní ISO 20.000. Obě normy jsou na ITIL založené, vycházejí z něj. Pokud mluvíme o certifikacích, měli bychom se zmínit o třech možných úrovních certifikování, a to lidí, podniků a nástrojů.

Certifikace lidí (3 úrovně):

- Základní porozumění ITIL (ITIL foundation certificate in ITSM) – teorie a pojmy ITIL, základní pochopení.
- Praktik daného ITIL procesu (ITIL practitioner certificate in ITSM) – pro každý proces zvlášť, pro toho, kdo provádí dennodenní aktivity spojené s daným procesem.
- Manažerský pro dodávku (Service Delivery) a provoz služeb (Service Support) (Manager's certificate in ITSM) – člověk má hluboké porozumění v celé oblasti, může definovat procesy, pracovat jako konzultant.
- Zodpovědné 2 nezávislé certifikační autority EXIN a ISEB.

Certifikace podniků (procesů):

- ISO 20.000 – certifikace probíhá pro každý proces zvlášť.
- BS 15.000 – původní britská norma.

Certifikace nástrojů (podporujících ITIL procesy):

- Zodpovědná organizace PinkElefant
- Existují verifikační šablony[2] – žádný z nástrojů není verifikován na všech 10 procesů, nejlepší a nejrozsáhlejší (BMC Remedy, CA Unicenter ServiceDesk, HP Service Center a další) vyhovují 6-7 procesům SS a SD.

Struktura ITIL verze 2 (Obr. 56 vlevo) se skládá ze 7 knih:

- Business Perspective – poskytuje IT rady a návody pro lepší porozumění a přispění k cílům byznysu a jak služby lépe přizpůsobit a využít pro maximalizaci přínosů.
- Application Management – popisuje správu aplikací v průběhu celého životního cyklu, vhodnější využívat metody k tomu přímo určené jako je RUP, OpenUP, MDIS apod.
- Service Delivery – pokrývá procesy potřebné pro plánování a dodávku kvalitních IT služeb. Zaměřuje se na procesy s delším časovým dopadem, spojené se zlepšováním kvality dodávaných IT služeb.
- Service Support – popisuje procesy spojené s každodenními aktivitami podpory a údržby poskytovaných IT služeb.
- Security Management – popisuje proces plánování a správy definované úrovně bezpečnosti informací a služeb IT. Zahrnuje také analýzu a správu rizik a zranitelností a implementaci nákladově zdůvodněných protiopatření.

---

[2] Šablony viz https://www.pinkelephant.com/en-PH/ResourceCenter/PinkVerify/

- ICT Infrastructure Management – pokrývá všechny aspekty správy ICT infrastruktury od identifikace požadavků byznysu, přes nabídku až k testování, instalaci a následným operacím a optimalizaci komponent IT služeb.
- Planning to Implement Service Management – zaměřuje se na problémy a úkoly související s plánováním, zaváděním a zlepšováním procesů správy IT služeb. Zahrnuje také problematiku kulturních a organizačních změn, rozvoj vize a strategie.

Jádrem ITIL verze 2 jsou knihy Service Support a Service Delivery. Jimi se budeme zabývat podrobněji v další kapitole. Mezi většinou těchto knih existuje vzájemný přesah.

Struktura ITIL verze 3 (Obr. 56 vpravo) se skládá z 5ti knih + 1 úvodní:

- Service Strategy – zabývá se sladěním byznysu a IT, strategií správy IT služeb, plánováním.
- Service Design – řešení IT služeb, návrh procesů (tvorba a údržba IT architektury, postupů).
- Service Transition – předání IT služby do byznys prostředí.
- Service Operation – doručení a řídící aktivity procesu, správa aplikací, změn, provozu, metrik.
- Continual Service Improvement – hnací body zlepšení IT služeb, oprávněnost vylepšení, metody, praktiky, metriky.

- Official Introduction of the ITIL Service Lifecycle – základní koncept ITSM, místo ITIL v něm, nový model životního cyklu.

Verze 3 definuje nový životní cyklus IT služby od jejího plánování, přes doručení a zavedení až po provoz a nestálé zlepšování. Touto problematikou se verze 2 nezabývala, resp. nebyla řešena jako životní cyklus, ale pouze v rámci některých procesů Service Delivery.

Na závěr této kapitoly si ještě shrneme, co ITIL je a co není, co řeší a co neřeší. ITIL je procesním frameworkem sloužícím pro definici procesů v oblasti správy IT služeb. Vznikl jako soubor best practices z úspěšných projektů v praxi. Definuje činnosti a procesy, které je třeba vykonat (a říká kdy) a k nim zodpovědné role. ITIL však neřeší jak konkrétně tyto činnosti provádět, jakou mít organizační strukturu. Tyto aspekty ITIL konkrétně nedefinuje, protože je každá organizace jiný, každá má jinou kulturu a zvyky. Řekli jsme, že ITIL také není kompletní metodikou, ale procesním frameworkem. Není standardem, standardem jsou normy britská BS 15.000 a mezinárodní ISO 20.000. V poslední řadě ITIL neřeší/neobsahuje metodiku projektového řízení pro nasazení IT služeb, pouze doporučuje metodiku PRINCE2 (něco málo o této metodice naleznete v [Pro06]).

10.4 Stručný přehled procesů SS a SD

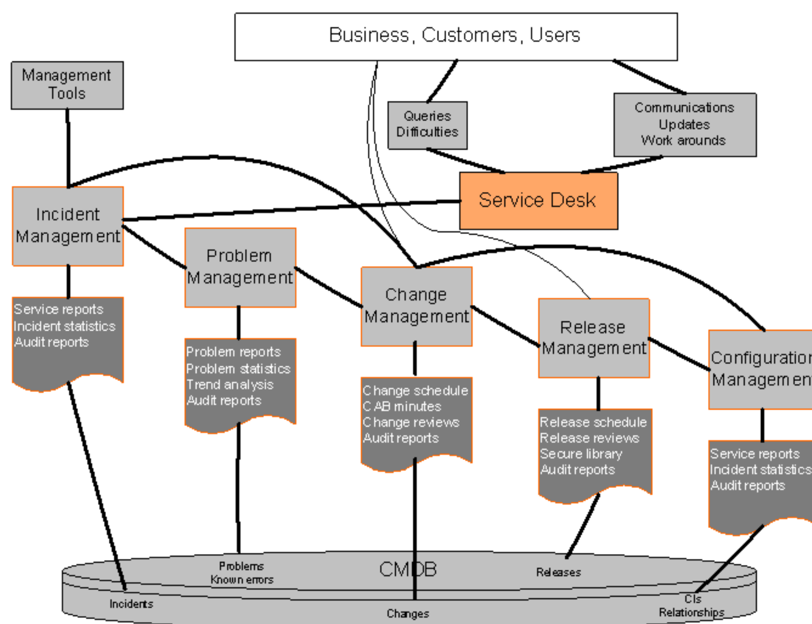
V příkladu z kapitoly 10.2 jsme se již setkali s některými procesy ITIL. V předchozí kapitole jsme vysvětlili základy týkající se ITIL, vazbu na standardy, co ITIL definuje a co nedefinuje. Nyní se budeme věnovat popisu

procesů vlastního jádra ITIL, tj. pouze dvou základním knihám verze 2: provozu služeb (Service Support) a doručení služeb (Service Delivery).

Service Support definuje následující procesy (viz Obr. 56):

- Incident Management (IM) – správa incidentů,
- Problem Management (PrM) – správa problémů,
- Change Management (CxM) – správa změn,
- Configuration Management (CoM) – správa konfigurací,
- Release Management (ReM) – správa releasů,
- Funkce Service Desk.

Ústředním bodem Service Support (dále SS) procesů je funkce Service Desku, o které jsme mluvili již v příkladu. **Service Desk** je jediným kontaktním místem pro zákazníky a uživatele a má vazbu na všechny, resp. většinu dalších procesů. Service Desk je vlastně aplikace obsahující evidence procesů, tyto evidence mohou být navzájem propojeny a záznamy na sebe odkazovat, např. problémy na incidenty, jak bylo ukázáno v našem příkladu. Cílem Service Desku je zajišťovat dennodenní kontakt se zákazníky, uživateli, pracovníky IT a externí podpory. Zajišťuje obnovu výpadku IT služeb a plní roli podpory 1. úrovně a koordinuje 2. a 3. úroveň podpory[3].



Obr. 57 ITIL procesy pro Service Support

Proces **Incident Management** je zodpovědný za obnovení normálního provozu IT služby. Snahou je nejen co nejrychlejší obnova, ale také minimalizace důsledků výpadku na provoz (na byznys – zákazníka a uživatele). Cílem IM je také zajistit, aby byly služby dodány zákazníkům v kvalitě, která byla dohodnutá v tzv. SLA dokumentu (o SLA více viz proces

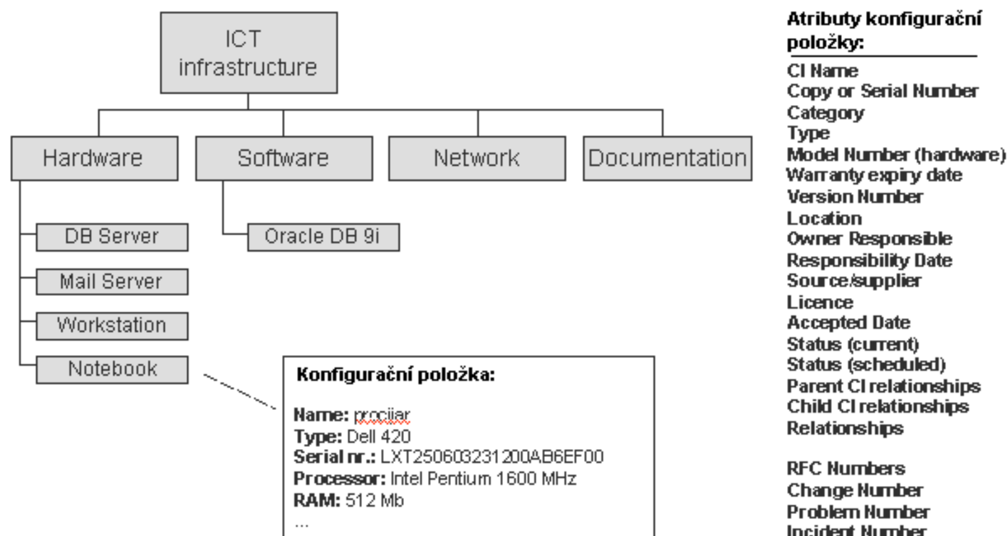
[3] 1. úroveň podpory – aplikační specialisté, řeší také pomoc v případě dokumentace, instalace.
2. úroveň podpory – techničtí specialisté, řeší technické problémy konkrétních aplikací.
3. úroveň podpory – specialisté na daný produkt, většinou třetí strany (výrobci).

SLM). Odpovídá tedy za správu všech incidentů od zjištění a vytvoření záznamu, až po vyřešení a uzavření.

Cílem procesu **Problem Management** je minimalizovat nepříznivé dopady incidentů a problému na byznys. PrM asistuje IM při řešení závažných incidentů. PrM je zodpovědný za evidenci náhradních řešení (workarounds), rychlých náprav jako známých chyb, a tam kde je to vhodné/finančně efektivní iniciuje změny trvale implementované do infrastruktury – strukturální změny. PrM rovněž analyzuje incidenty a problémy a zkoumá jejich trendy, aby proaktivně zabránil jejich dalšímu výskytu.

Pro efektivní zpracování a implementaci změn slouží proces **Change Management**. Změny jsou pomocí procesu řízeny od zaznamenání záznamu, přes filtraci, posouzení, kategorizaci až po plánování, testování a nasazení. Jedním z klíčových výstupů procesu je plán změn (Forward Schedule of Change), který obsahuje program změn dohodnutý se všemi odděleními/zákazníky, vytvořený podle dopadu na byznys.

Proces **Release Management** řeší změny IT služeb z pohledu globálních souvislostí. Zabývá se aspekty nasazení, akceptace a distribuce software a hardware. ReM je zodpovědný za sestavení, testování a distribuci releasů a také za jejich bezpečné uložení, k tomu slouží Definitive Software Library (DSL) pro software a Definitive Hardware Store (DHS) pro hardware.



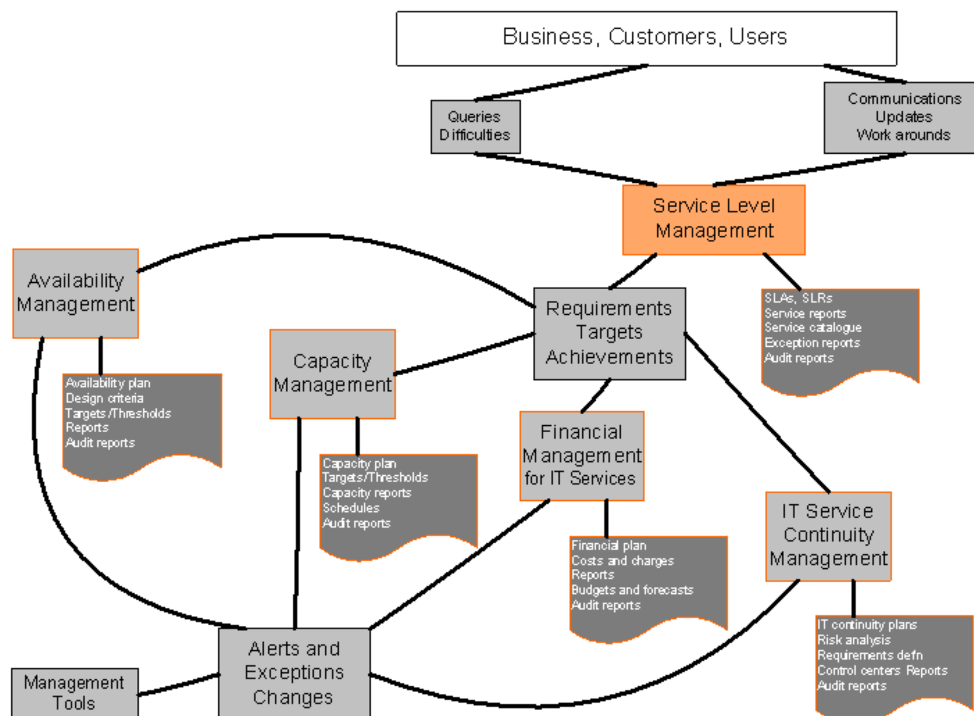
Obr. 58 Struktura konfiguračních položek, příklad atributů

Základnou, na které jsou postaveny ostatní procesy ITIL je proces **Configuration Management**. Základním prvkem tohoto procesu je tzv. konfigurační databáze (CMDB – Configuration Management Database), která se skládá z 1 nebo více fyzických databází či pohledů. CMDB obsahuje detailní záznamy o jednotlivých komponentách ICT infrastruktury, tyto prvky jsou označované jako konfigurační položky (CI – Configuration Items). Hlavním přínosem této databáze je zachycení vzájemných vztahů konfiguračních položek, což umožňuje nejen modelovat scénáře IF-THEN (co

se stane když ...), ale také v případě výpadku konkrétního prvku dohledat možné komplikace pro další prvky.

Service Delivery definuje následující procesy (viz Obr. 59):

- Service Level Management (SLM) – správa úrovně služeb,
- Capacity Management (CaM) – správa kapacit,
- Availability Management (AvM) – správa dostupnosti,
- IT Service Continuity Management (ITSCM) – obnova IT služby po výpadku,
- Financial Management for IT Services (FiM) – správa financí IT služeb.



Obr. 59 ITIL procesy pro Service Delivery

Procesem zodpovědným za dojednání kvality IT služeb je **Service Level Management**. SLM sbírá a hodnotí požadavky byznysu na IT služby, na jejich základě vytvoří a provozuje IT službu údaje o její úrovni/kvalitě sepisuje v dohodě o úrovni služby (SLA – Service Level Agreement). Pro podporu vlastního SLA je možné ještě dohodnout podpůrné smlouvy OLA – Operational Level Agreement a nebo externí UC – Underpinning Contract. Jedniné UC má status právního dokumentu, ostatní jsou pouze dohody, musí (měly by být) být tedy přílohou nějaké smlouvy o poskytování IT služeb. SLM proces je také zodpovědný za definici katalogu a údržbu služeb (viz **Obr. 60**), který popisuje jednotlivé IT služby, definuje zákazníky, kteří si je mohou objednat a využívat a pro potřeby IT definuje, které prvky ICT infrastruktury jsou využívány pro kterou IT službu. Díky tomuto seznamu (který je publikován prostřednictvím Service Desku a dostupný všem jeho uživatelům) je pak při výpadku některé konfigurační položky jasně vidět, které služby jsou

ohroženy, uživatelé o tom mohou být okamžitě obeznámeni a může být sjednána náprava. Zde je jasná vazba na Service Desk a jeho funkce.

Service Name	Description	Service Users		Configuration Items
StockControl	Service provides functions for order processing,	Trade dept. Warehousing dept. Marketing dept.		StockControl Program v1.1 Tomcat v5.5 Oracle 9i Red Hat Enterprise Linux 5 Server Prague Switch1 Switch3 Intranet
Internet	Service provides connection to Internet	All users		Internet Service Provider Firewall Zone F v3.2
Document Search	Service allows searching for documents	All users		Google Desktop Search

Obr. 60 Příklad katalogu IT služeb

Proces **Financial Management for IT Services** umožňuje provozovat IT jako byznys, definuje pravidla pro organizaci, která si je vědoma svých nákladů a je tedy nákladově efektivní. Základní aktivity zahrnují porozumění nákladům a jejich účtování, , prognózy budoucích finančních plánů a další. Proces definuje jednu volitelnou položku: účtování za IT služby, jež se snaží o navrácení nákladů na provoz IT z byznysu spravedlivým a poctivým způsobem.

Proces **Capacity Management** se zabývá, jak již název vypovídá, dostatečnou, resp. akorát potřebnou kapacitou k uspokojení požadavků a cílů byznysu. Pro tento účel vytváříme a postupujeme podle kapacitního plánu (CP – Capacity Plan), který je úzce spjatý s plány a cíly byznysu. CP pokrývá tři oblasti – správa kapacit z pohledu byznysu, IT služeb a zdrojů. Pro dosažení cílů používá proces obecné aktivity jako je řízení výkonnosti služeb (Performance Management), správa požadavků uživatelů (Demand Management) nebo škálování a modelování aplikací (Application Sizing and Modelling). Z poslední aktivity je zřejmé, že tento tým by měl spolupracovat s vývojovým týmem, minimálně v otázkách rozšiřitelné architektury a výkonnosti vyvíjené aplikace.

Proces **IT Service Continuity Management** produkuje plány obnovy, které jsou navrženy tak, aby po každém velkém výpadku / incidentu byly služby poskytovány na dohodnuté úrovni v dohodnutém čase (jak psáno v SLA). Tento proces je součástí podnikového plánu zachování byznysu (BCP – Business Continuity Plan). Cílem procesu ITSCM je napomoci byznysu minimalizovat narušení základních podnikových procesů během závažného incidentu a po něm. V rámci procesu jsou také pravidelně prováděny aktivity jako analýza dopadu na byznys, analýza rizik, testování a údržba plánů obnovy.

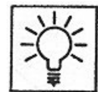
Klíčovým aspektem IT služeb je jejich dostupnost. Proces **Availability Management** je odpovědný za splnění požadované dostupnosti IT služeb či za

jejich překročení (vyšší hodnoty), stejně jako jejich proaktivní zlepšování. Pro dosažení těchto cílů proces monitoruje, měří, reportuje a vyhodnocuje klíčové metriky každé služby a jejich součástí. Mezi ně řadíme dostupnost (availability), spolehlivost (reliability), udržovatelnost (maintainability), praktičnost (serviceability) a bezpečnost (security).

Kontrolní otázky:

4. Co je to ITSM (správa IT služeb)?
5. Vyjmenujte procesy Service Supportu.
6. Vyjmenujte procesy Service Delivery.
7. Jaké procesy/funkce se zabývají co nejrychlejším vyřešením incidentu?
8. Jaký je rozdíl mezi incidentem a problémem?
9. Co je to IT služba?

Úkoly k zamyšlení:



Zamyslete se nad možnostmi zálohování existujících softwarových služeb, jaké všechny Vás napadnou (zálohy dat, záložní servery, prostory pro okamžitou instalaci, pojištění, záložní centra, ...). Jaký si myslíte, že bude rozdíl mezi vybranými a použitými možnostmi v bance, průmyslovém podniku a střední IT firmě?

Korespondenční úkol:

Zmínili jsme stručně problematiku Service Desku, což není jen nástroj, ale je to jediné kontaktní místo pro zákazníky (tzv. SPOC – Single Point of Contact). Jelikož toto místo slouží k zaznamenání incidentů, problémů, požadavků na změny, k nahlášení a dohodnutí implementace záplat, aktualizací s uživatelem, je zřejmé, že spokojenost uživatelů bude hodně záviset na lidech. Představte si že píšete inzerát, pokuste se vypsat, jaké schopnosti a dovednosti (včetně tzv. soft-skills) by měl mít takový pracovník a také jaké bude mít odpovědnosti.

Shrnutí obsahu kapitoly

V této kapitole jste se seznámili s oblastí provozu a údržby vyvinutého software pomocí standardních postupů (ITSM – IT Service Management), konkrétně pomocí procesů definovaných frameworkem pro provoz a údržbu ITIL. Text vysvětlil, co je to ITIL a na příkladu ukázal jeho aplikaci. V poslední kapitole jsme se stručně věnovali popisu jednotlivých procesů ITILu.

11 Aplikace IS, systémová integrace

V této kapitole se dozvíte:

- Jaké jsou aplikace podnikových informačních systémů?
- Co znamenají zkratky jako ERP, BI, SCM?
- Co je to systémová integrace?

Po jejím prostudování byste měli být schopni:

- Porozumět základní klasifikaci aplikací podnikových IS.
- Porozumět principům systémové integrace.

Klíčová slova této kapitoly:

Podnikový informační systém, ERP, CRM, SCM, BI, systémová integrace.

Doba potřebná ke studiu: 4 hodiny

Průvodce studiem

Kapitola zmiňuje kategorie a aplikace podnikových IS včetně příkladů a ukazuje jejich místo v podniku. Nedílnou součáatí zavedení, provozu a údržby těchto aplikací je pak systémová integrace.
Na studium této části si vyhraďte 4 hodiny.

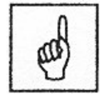
V jedné z úvodních kapitol o architekturách IS jsme se zmínili o možných variantách a aplikacích informačních systémů. Tyto aplikace, respektive části podnikového informačního systému musí podporovat všechny tři úrovně řízení a rozhodování v organizaci (strategické, taktické, operativní), resp. musí k tomuto rozhodování poskytovat údaje, data a měly by také podporovat podnikové procesy na těchto úrovních. Následující obrázek ukazuje kontext, kdy jádrem podnikového IS je tzv. ERP systém, pro podporu komunikace se zákazníky slouží tzv. CRM systém, pro řízení dodavatelského řetezce pak tzv. SCM a v neposlední řadě na stategické a částečně i taktické úrovni tzv. BI systémy pro analýzy, přehledy a podporu rozhodování.

Obr. 61 Kontextový pohled na úrovně řízení a aplikace IS v podniku

Podnikové procesy, které existují v organizaci by měly být vhodně podpořeny informačními technologiemi, proto lze podle podpory různých podnikových

procesů rozdělit aplikace informačního systému do několika základních kategorií, jedná se o:

- **ERP** (Enterprise Resource Planning) – jádro podnikového IS, zaměřené na řízení interních (převážně hlavních) podnikových procesů.
- **CRM** (Customer Relationship Management) – slouží pro podporu a automatizaci procesů směřovaných k zákazníkovi.
- **SCM** (Supply Chain Management) – slouží pro podporu a automatizaci dodavatelského řetězce.
- **BI**, často součást manažerských IS (tzv. MIS), sbírá data z předchozích a vytváří agregace, analýzy, trendy, které slouží pro podporu rozhodování manažerů.



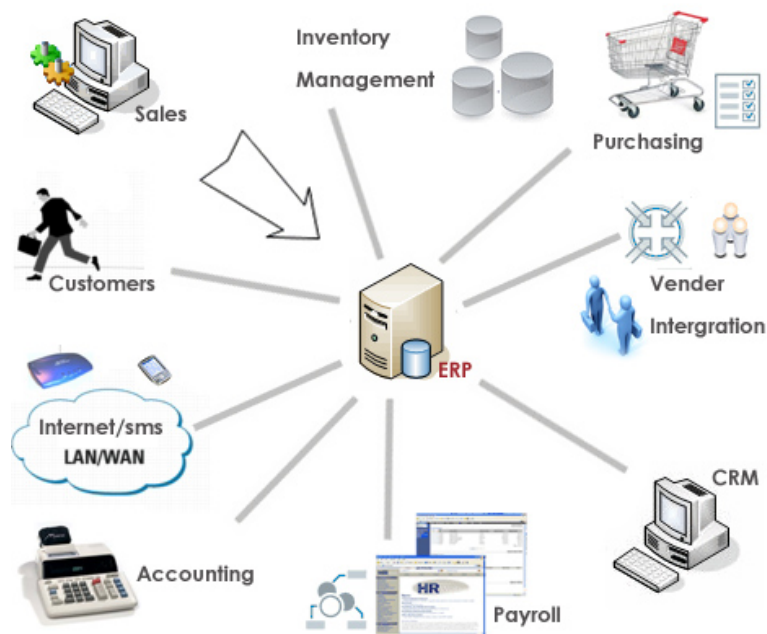
Dále v textu se budeme podrobněji zabývat jednotlivými částmi kompletního podnikového informačního systému. Dozvíte se tedy podrobněji, co znamenají všechny ty zkratky ERP, CRM, SCM, BI apod.

11.1 ERP

ERP neboli Enterprise Resource Planning system (plánování podnikových zdrojů) je softwarový nástroj, který je určen pro plánování a řízení podnikových procesů na všech úrovních (operativní až strategická). Integruje veškeré klíčové a řídící procesy, aby mohl poskytnout nadhled nad vším, co se děje ve firmě. ERP je tedy zaměřeno na interní procesy, tj. procesy, nad nimiž má organizace plnou kontrolu, jedná se o následující 4 kategorie procesů:

- výroba (pokud organizace vyrábí),
- logistika,
- personalistika (lidské zdroje),
- ekonomika.

ERP se snaží, aby celý systém, i když bude poskládán z menších systémů, komunikoval se všemi částmi a všechny funkce byly spolu provázány.



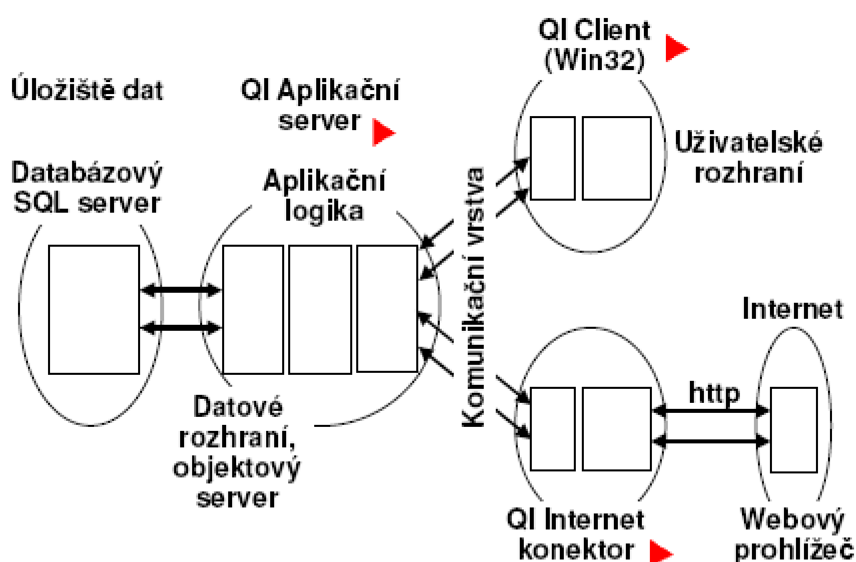
Obr. 62 Zdroje ERP

ERP se dělí do tří základních kategorií podle schopnosti pokrýt či integrovat zmíněné 4 procesy, resp. kategorie procesů:

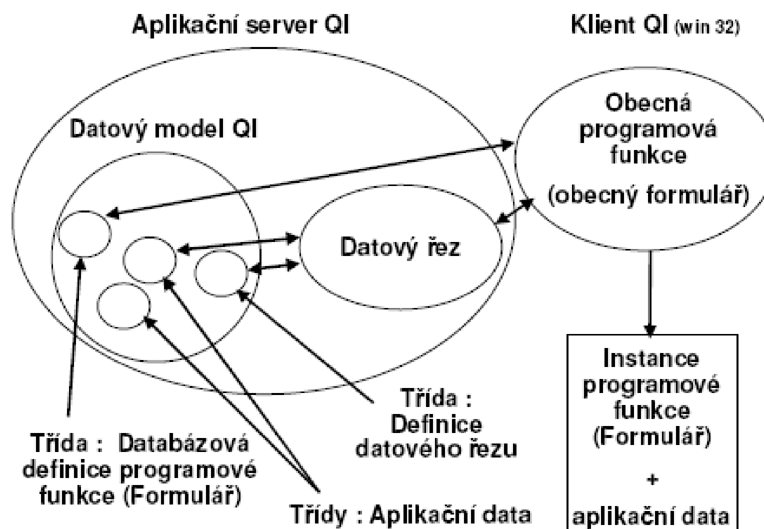
- **All-in-one** – pokrývají a integrují všechny tyto oblasti hlavních interních procesů. Výhodou je tedy pokrytí všech oblastí, ale za cenu nižší detailní podpory funkčnostmi, navíc customizace obecného řešení může být velmi nákladná.
- **Best-of-Breed** – nepokrývají všechny 4 kategorie interních procesů podniku, ale jsou zaměřeny spíše na jen na některou oblast pro kterou poskytují detailní funkčnosti nebo pokrývají širší oblast, ale jsou zaměřeny na konkrétní obor podnikání (prodej/nákup, vodní hospodářství, cestovní kanceláře, ...). V praxi mohou být nasazeny samostatně nebo jako součást celé koncepce ERP. Může způsobovat obtížnější koordinaci procesů, nekonzistence či redundance informací.
- **Lite ERP** – jedná se o odlehčené verze klasických ERP, převážně pro využití malými a středními podniky (tzv. SME). Výhodou tohoto řešení je nižší cena a snadná, resp. rychlá implementace řešení. Nevýhodou jsou pak určitá omezení, ať ve funkcionalitě nebo počtech současně připojených uživatelů, uložených datech apod.

Jelikož pouze zaměření na výše zmíněné 4 kategorie interních podnikových procesů bylo nedostatečné, dnešní ERP, označované již jako ERP II slouží také pro podporu externích procesů (řízení vztahů se zákazníky a řízení dodavatelského řetězce) a manažerského rozhodování.

V oblasti ERP lze ještě narazit na historickou zkratku MRP (Material Requirements Planning). Jedná se o automatizované plánování spotřeby materiálu



Obr. 63 Technologie IS QI

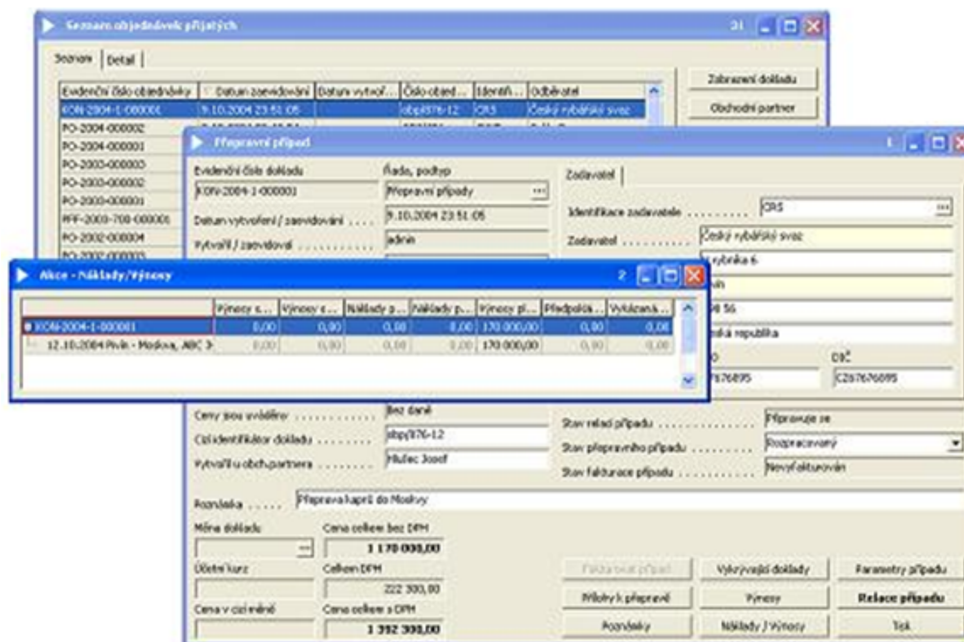


Obr. 64 Konstrukce aplikace v IS QI (QI Builder)

Příkladem automatizace procesů pomocí ERP může být automatické vystavení faktury a její odeslání na e-mail zákazníka v případě expedování výrobků a potvrzení výdeje v systému (vyskladnění).

Mezi konrétními ERP můžeme zmínit například:

- mySAP Business Suite (pro malé podniky pak SAP Business One),
- Oracle E-Business Suite (či převzaté JD Edwards EnterpriseOne, PeopleSoft Enterprise),
- Microsoft Dynamics AX,
- IFS aplikace,
- či dlouholetý český výrobce ERP DC Concept a jejich produkt QI.
- dále také další zdatní čeští konkurenti LCS Noris nebo IS Karat.



Obr. 65 Modul prodej a nákup v IS QI

11.2 CRM

Další oblastí či kategorií podnikové informatiky je CRM neboli Customer Relationship Management. Jedná se o systém komunikace a řízení vztahů mezi jednotlivými subjekty, nejde tedy pouze o podpůrný software ale i o externí procesy (marketing, prodej/nákup, servis, apod.), strategie či organizační strukturu. S rozmachem nových metod (cíleného) marketingu a také technologií, zvláště Internetu a mobilních technologií je růstový trend využití CRM zvláště znatelný.

V podstatě se jedná o systém práce s informacemi, které jsou strukturovaně zpracovávány a využívány k jednotlivým obchodním vztahům a obchodním případům. Potom jsme schopni:

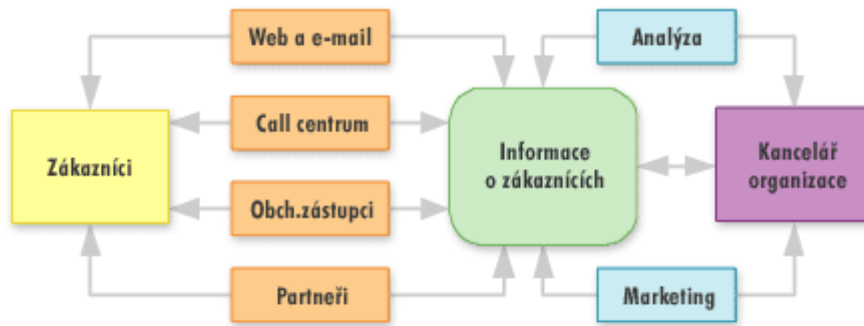
- porozumět potřebám zákazníka,
- kategorizovat zákazníky do různých skupin (podle odběrů, zaměření, důležitosti, ...),
- upravit nabídku služeb a produktů potřebám zákazníků.

Cílem CRM systémů je tedy naplnění a uspokojení potřeb zákazníků, stejně jako snaha o zvýšení ziskovosti. Kromě správy veškerých událostí ve firmě ve vztahu ke klientům a dodavatelům, je CRM velkým přínosem firmám z hlediska dlouhodobého využívání. CRM aplikace si vytváří statistické údaje o vývoji jednotlivých obchodních vztahů, zaznamenává historii komunikace mezi dodavatelem a odběratelem (a to z několika různých komunikačních kanálů, od různých zaměstnanců). Proto k informacím může rychle přistoupit nový zaměstnanec, aniž by bylo nutné pro takového zaměstnance vytvářet speciální soubor informací o přidělených obchodních případech.

Hlavní funkčností CRM není tedy jen podpora běžného provozu, kontaktu se zákazníky a podpora spolupráce, ale také analytické funkce využívající analýzy, datamining, OLAP systémy. Z těchto funkčností vyplývá i základní rozdělení CRM systémů:

- operativní (provozní) – podporují tzv. front-office procesy, jedná se o správu kontaktů, komunikace, zaznamenání historie.
- analytické – slouží pro analýzu zákaznických dat za různým účelem, např. cílené kampaně, analýza chování s cílem poskytnout lepší služby, předpovědi trendů, finanční analýzy.
- kolaborativní (pro podporu spolupráce) – koordinace několika komunikačních a informačních kanálů, poskytuje podporu (zahrnuje také infrastrukturu) pro efektivní řešení reklamací, dotazů, stížností.

Komunikace se zákazníky může být realizovaná mnoha informačními kanály, např.: e-mail, dopis, telefon, elektronický formulář nebo osobní kontakt.



Obr. 66 Kontextový pohled CRM

V souhrnu lze však konstatovat, že CRM je dnes softwarový produkt poskytující uživateli celkový přehled nad svými kontakty, obchodními případy a marketingovými akcemi pro podporu prodeje včetně správy času, úkolů a veškeré komunikace.

S dalším rozvojem technologií se také více dostupnými či běžnými stávají technologie jako VoIP (Voice over IP) – hlasová komunikace v prostředí Internetu či CTI (Computer Telephony Integration) – integrace telefonu se systémem, kdy systém podle identifikace volajícího zobrazí jeho detailní údaje apod.

Obr. 67 Příklad pokročilejších funkcí CRM aplikace (zdroj: salesboom.com)

Typickými zástupci typových CRM systémů jsou:

- Siebel (Oracle),
- mySAP CRM,

- Microsoft Dynamics CRM 3.0,
- Epiphany (SSA Global),
- či Oracle CRM.

Je však třeba zmínit, že velké procento CRM systémů je šito přímo na míru (zakázkové aplikace) a to hlavně v prostředí velkých korporací.

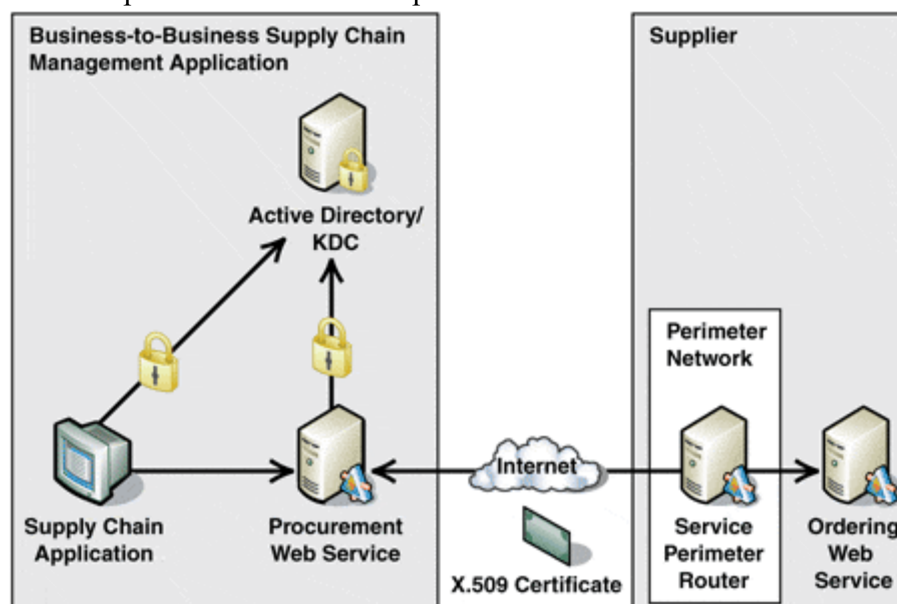
11.3 SCM

Jak již bylo zmíněno výše, jedná se o externí proces sloužící pro podporu a automatizaci dodavatelského řetězce. SCM (Supply Chain Management) tedy slouží pro správu poptávky a nabídky mezi různými organizacemi, zahrnující koordinaci a spolupráci s partnery, kteří mohou být dodavatelé stejně jako poskytovatelé služeb. Toky, které probíhají pomocí SCM jsou především hmotné toky (produkty), dále pak finanční (hlavně informace týkající se plateb a jejich typů) a informační (informace o stavu zboží na skladě, o zpracování objednávek) toky.

Existuje několik modelů popisujících aktivity spojené se správou a přesunem informací a materiálů přes hranice podniků, jedná se například o SCOR (Supply Chain Management Council) či GSCF (Global Supply Chain Forum). Obecně pak můžeme nutné aktivity rozdělit do následujících úrovní:

- Strategické – dlouhodobé zaměření (produkty, IT systémy, architektura, systémy, lokace či způsob propojení organizací); zahrnuje také strategické partnerství; koordinaci vývoje produktů apod.
- Taktické – kontrakty; tvorba a správa jednotlivých procesů; řízení zásob, jejich umístění, množství, frekvence dodávek.
- Operativní – denní plánování a distribuce, plánování a předpovědi zdrojů, poptávky; správa příchozích a ochozích operací, dodávek apod.

Příklad zabezpečeného řešení SCM podle Microsoft:



Obr. 68 SCM systém

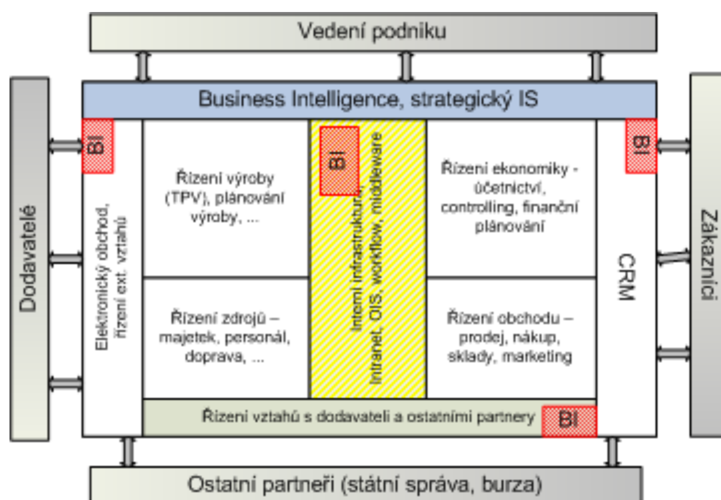
Příkladem provázání dvou podniků pomocí SCM může být následující. Při poklesu zboží pod určitou limitní úroveň je automaticky u dodavatele vystavena objednávka na předem dohodnutý objem daného a popř. jiného zboží a tato dopravena k nám do organizace.

11.4 Business Inteligence

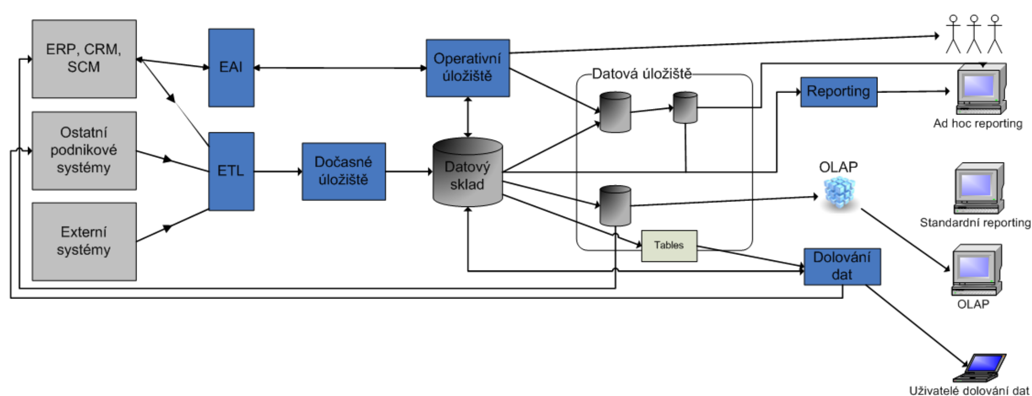
Jedná se o aplikace a metody nadřazené všem interním podnikovým procesům. Pomocí BI jsou sledovány, shromažďovány, analyzovány a zpracovávány údaje o podniku jako celku, nejen o zákaznících, trhu nebo konkurentech. Stejný termín se používá v souvislosti se správou, analýzou a vyhodnocováním velkých objemů dat, většinou v souvislosti s ukládáním surových dat, jejich správou a data miningem.

Cílem BI aplikací není pouze shromažďovat data, ale hlavně podporovat manažerská rozhodnutí (na základě poskytnutých dat). BI systémy poskytují historická a současná data stejně jako předpovědi, jak se může vyvíjet situace na trhu či jaká jsou rizika a to hned v několika scénářích v návaznosti na manažerská rozhodnutí.

Data jsou většinou skladována v datových skladech (data warehouse, data mart), proto pozor na možnou záměnu pojmů Business Intelligence a datové sklady. Datové sklady jsou spíše technologickým řešením využívajícím databáze a mohou sloužit pro uložení dat pro BI, která jsou sesbírána z různých částí informačního systému (z ERP, CRM či SCM). Důležitým pojmem v oblasti BI, stejně jako v oblasti podnikových procesů je KPI (Key Performance Indicators). Jedná se o klíčové ukazatele výkonnosti, které nám slouží k ohodnocení současného stavu podniku, resp. jeho procesů. Příkladem KPI může být průměrný počet zpracovaných objednávek za čas. KPI jsou či mohou tedy být také zpracovány BI systémy.



Obr. 69 Postavení BI v architektuře IS



Obr. 70 Komponenty BI

Mezi známé platformy BI patří například:

- SAS Enterprise Intelligence Platform,
- Cognos 8 BI,
- SAP NetWeaver Business Intelligence,
- Hyperion Solutions,
- Business Objects.

11.5 Systémová integrace

Systémová interace (SI) je disciplínou, která poskytuje prostředky pro vytvoření a pro permanentní údržbu podnikového informačního systému a to na všech úrovních (technologie, řízení, strategické plánování). Často se jedná o implementaci All-in-one ERP produktů, kdy firma poskytuje jak daný produkt, tak také služby systémového integrátora. Základní služby poskytované systémovým integrátorem jsou:

- projekce – analýza, návrh, implementace a instalace jednotlivých HW a SW komponent,
- výběr vhodných produktů pro realizaci – od aplikačního SW až po dílčí HW,
- instalační služby – HW, SW, sítě, kabeláž,
- školící služby,
- podíl na řízení projektů – poskytnutí odborníků či přímo řízení,
- zajišťování permanentního rozvoje (outsourcing).

Daný výčet poskytovaných služeb vyžaduje velké množství znalostí a zkušeností, systémový integrátor by měl tedy mít znalosti nejméně v následujících oblastech:

- procesní řízení a modelování podnikových procesů,
- architektury a konstrukce počítačů,
- struktura operačních systémů,
- aplikační programy a jejich tvorba (podle standarních postupů, např RUP),
- komunikační systémy,
- právní a ekonomické aspekty uplatnění VT

Výhodou využití služeb systémového integrátora je celkový pohled na danou problematiku a strategické (rozuměj dlouhodobé) využití vložených investic. Výchozím bodem návrhu IS je IT strategie, požadované funkce IS jsou odvozeny od podnikových cílů procesů. IS je řešen a realizován jako komplexní integrovaný systém, ne samostatné, nespolupracující jednotky – počítače a přídavná zařízení, sítě LAN a WAN, základní SW, technologicky orientovaný SW, aplikační SW, interní a externí datové zdroje. IS je také realizován jako integrovaný komplex služeb (zahrnující studie, projekty, implementaci, instalaci, školení, konzultace, vývoj, možný další provoz). IS je realizován jako otevřený systém splňující mezinárodní a podnikové standardy, nezávislost na výrobci HW a SW. Bývá rozvýjen podle jednotné metodiky s promyšlenou a srozumitelnou architekturou. V neposlední řadě je IS provozován na základě jednotné soustavy pravidel, které musí dodržovat všichni uživatelé systému.

Integrace může probíhat na několika úrovních:

- Integrace vizí – zabýváme se tím, jak podpořit konkurenceschopnost, jaké procesy budeme podporovat, jakými nástroji.
- Integrace podniku s okolím – přizpůsobení se měnícím požadavkům trhu, navázání informačního vztahu s externími partnery, poskytování vhodných informací o podniku.
- Integrace procesní – zkrácení doby jednotlivých procesů, jejich zefektivnění (potřeba méně podnikových zdrojů), optimalizace procesů z pohledu kvality výrobků.
- Integrace technologická – na úrovni dat (společná DB nebo datový sklad), hardware, software (propojení programů, automatizování činností), uživatelské rozhraní (stejné ovládání aplikací, stejné GUI - portály).

SI lze využít pouze na některou z aktivit, i když hlavní přínos je právě v jeho celistvosti. Následující tabulka ukazuje varianty vývoje IS včetně výhod a nevýhod každé z nich.

Varianty vývoje	Výhody	Nevýhody
1) Vlastní vývoj, nákup ostatních komponent, integrace a provoz vlastními silami	IS šitý na míru potřebám firmy, možnost růstu IS dle potřeb firmy, detailní znalost provozovaného IS/IT, konkurence nezná silné a slabé stránky IS firmy, dodavatel neodhalí strategii firmy, snadná reakce na potřeby uživatelů	Vysoké náklady, časová náročnost, obvykle nižší kvalita IS způsobená ne vždy vysokou kvalitou interních řešitelů, riziko nekonzistence systému při fluktuaci řešitelů, kooperativní náročnost
2) Vývoj externí softwarovou firmou, nákup, integrace, provoz vlastními silami	IS na míru, možnost růstu dle potřeb podniku, konkurence nezná silné a slabé stránky IS/IT, optimální využití znalostí interních a externích specialistů	Vysoké náklady, obvykle vyšší než u 1), dlouhá doba řešení (kratší než u 1), potíže s integrací celého IS/IT,nízká parametričnost IASW, rizika úniku informací
3) Nákup všech komponent formou TASW od různých výrobců, integrace a provoz je vlastní	Rychlá realizace, nízké náklady, lze vybrat osvědčená řešení pro každou část IS, TASW je parametrický – nové požadavky jsou řešeny jiným nastavením parametrů	Procesy v podniku se musí přizpůsobit možnostem TASW, různé části TASW od různých výrobců ⇒ obtížná integrace do IS, obtíže s údržbou vazeb a z toho plynoucí nestabilita IS
4) Nákup celého IS od generálního dodavatele – systémového integrátora (SI), provoz vlastními silami	Rychlá realizace, nízké náklady, profesionální řešení komponent i celého IS, lze vybrat osvědčená řešení, SW je parametrický – nové požadavky – přenastavení parametrů, integrace a stabilita garantována integrátorem, rozložení rizik mezi podnik a SI	Procesy v podniku se musí přizpůsobit možnostem TASW, závislost na generálním dodavateli, jeho schopnostech, serióznosti, rizika úniku informací mimo podnik
5) Tvorba IS generálním dodavatelem, externí provoz celého IS/IT	Viz bod 4) a navíc: rychlejší dosažení požadovaných služeb, snížení nároků na provozní personál – možnost soustředit se jen na svůj hlavní předmět činnosti, snadnější přizpůsobení kapacit IT podle potřeby, možnost využití nejprogresivnějších technologií	Viz bod 4) a navíc: růst závislosti na SI, vyšší náklady (ale vyšší kvalita služeb a spolehlivosti), při stejné úrovni služeb jsou náklady nižší
6) Nákup informatických služeb od různých poskytovatelů, externí provoz (ASP)	Podobně jako 5), navíc: vysoká flexibilita služeb (aktivace, deaktivace služby dle potřeby podniku), každá služba provozována optimálním dodavatelem, snížení nároků na provozní personál	Problémy s integrací služeb, problémy s HW, SW i datovou integrací, vyšší bezpečnostní rizika než u 5)

Tabulka 11-1: Varianty vývoje IS

Více a detailněji k problematice aplikací IS viz [So06], více k systémové integraci viz [Tv00] nebo [Vo99].

11.6 Outsourcing

V případě vývoje a provozu ICT vůbec se často zmiňuje pojem outsourcing, Global Development and Delivery (GDD) a také offshore. Co tyto a další pojmy znamenají nám objasní tato kapitola. Outsourcing obecně je proces delegace činností, které nejsou jádrem činností (core byznys procesy) dané organizace, na externího dodavatele. Tento dodavatel se většinou na danou oblast specializuje. Běžně je outsourcován úklid nebo stravování. Cílem je

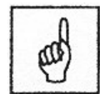
snížení nákladů a zefektivnění těchto procesů, organizace se pak může věnovat pouze své hlavní činnosti a být v ní více konkurenceschopná.

Pokud budeme mluvit o obecném outsourcingu lze opravdu převést na třetí stranu velké množství procesů či činností, například:

- již zmíněný úklid a stravování,
- marketing,
- účetnictví,
- ostrahu objektů,
- správu a provoz IT.

Outsourcing IT může mít několik variant, ICT infrastruktura zůstává majetkem organizace, ale je spravována třetí firmou nebo je naopak i s lidmi prodána (či převedena) na outsourcingovou firmu. Vlastnímu outsourcingu by měla předcházet analýza a strategické rozhodnutí managementu, jelikož organizace může na této formě spolupráce vydělat i prodělat. Proto je nutné stanovení strategických cílů a z nich vycházet. Z obou modelů samozřejmě plynou určité výhody a nevýhody.

Výhodou outsourcingu v případě ICT je předání práce odborníkům, kteří budou mít lepší znalosti, větší zkušenosti (a také lepší možnosti), než naši interní zaměstnanci. Další výhodou je rozložení plateb do pravidelných měsíčních splátek. Nevýhodou je pak předání dat a informací externí firmě, měla by být tedy důvěryhodná, aby nedošlo k jejich zneužití.



ASP – Application Service Provider je poskytovatel ICT ve formě služby, jedná se o nejexternější variantu outsourcingu (viz bod 6, Tabulka 11-1). Další varianta či obdoba tohoto pojmu je **SaaS** (Software as a Service), kdy je určitá aplikace poskytována a účtována jako služba.

Global Development and Delivery (GDD) je posledním pojmem z této oblasti. Jedná se o vývoj SW řešení několika týmy, které mohou být rozprostřeny po celém světě. Je zřejmé, že takto můžeme využívat nižších nákladů na pracovní sílu například ve východní Evropě či v Indii. Samozřejmě to ale vyžaduje mnohem větší nároky na řízení, spolupráci a synchronizaci jednotlivých týmů. Je důležité rozlišovat mezi pojmy GDD ve všech jeho variantách a outsourcingem. V prvním případě jde o fyzický přesun části většinou hlavních aktivit (ať už na externího dodavatele či na další jednotku v jiné lokaci). V případě druhého jde o vyvedení činnosti na třetí stranu, která následně zajišťuje její dodávku.

GDD existuje a využívá se z několika důvodů, není to jen nižší cena práce v různých částech světa. Další důvody jsou tedy následující:

- Kvalifikovaná pracovní síla (možnost využít více lidských zdrojů).
- Strategická výhoda (blíže byznysu, blíže zákazníkovi, mluvíme jeho řečí apod.).
- V neposlední řadě úspora nákladů.

V případě outsourcingu, resp. GDD existuje několik modelů, které mají označení podle vzdálenosti od sídla původního organizace. Jedná se o:

- Onsite (externí zaměstnanci jsou součástí týmu dané organizace, sedící přímo v jejích prostorách, často využívaný model také v ČR).
- Nearshore (využití sousedních zemí ve stejném regionu – střední Evropa, Skandinávie, apod.) – toto řešení umožňuje redukovat náklady na cestování, komunikace je oproti offshore snadnější díky stejnému časovému pásmu, podobné jsou také kultury a zvyklosti.
- Offshore (vlastní prostory, často projektově orientováno, někdy úkolově orientováno) – tento způsob přináší v případě vývoje SW velké problémy, pokud jsou offsete lidé bráni pouze jako další zdroje (stejně jako finance či materiál).

Tento model samozřejmě generuje určitá rizika, výše jsme již zmínili problémy s řízením a koordinací týmů. Není možné řídit týmy vzdáleně, ty musí fungovat jako samostatné jednotky, sami rozhodovat o určitých krocích, znát cíle a celkový obraz (big picture), ne chtít odpovědnost za práci bez možnosti o něčem rozhodovat. Dalším rizikem je nedostatečná spolupráce, dezinterpretace plynoucí z nepochopení, sdílení znalostí (neexistuje neformální komunikace v kuchyňce) a v neposlední řadě také kulturní odlišnosti (kdy například Číňan nikdy neřekne ne, i když se jedná o nereálnou věc).

Kontrolní otázky:

1. Jaké znáte aplikace podnikových informačních systémů?
2. Co je to ERP?
3. Jaké 3 druhy CRM znáte?
4. Co je to Business Intelligence?
5. Co je to SCM, co podporuje, čím se zabývá?
6. Čím se zabývá systémová integrace?
7. Co je to outsourcing a jaké znáte jeho formy?

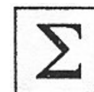
Úkoly k zamyšlení:

Pokuste se zamyslet nad využitím SCM ve vaší organizaci, určitě nějakou oblast najdete. Pokuste se popsat, co vše a jakým způsobem by mohlo být propojeno na úrovni IT aplikací, aby procesy ve vašem podniku byly efektivnější.

Korespondenční úkol:

Zmínili jsme rozdíl mezi datovými sklady a business inteligence. Pokuste se zamyslet nad nutnými technologickými aspekty BI aplikací. Pokuste se (s přihlédnutím k vašim znalostem konstrukce IT systémů, SŘDB, architektur IS) navrhnout technologickou architekturu BI systému.

Shrnutí obsahu kapitoly



V této kapitole jsme představili základní kategorie aplikací informačních systémů. Jednalo se o ERP systémy na podporu chodu interních podnikových procesů, dále CRM a SCM na podporu externích procesů (komunikace se zákazníky a dodavateli) a v neposlední řadě BI pro podporu rozhodování manažerů. Dodavatelem (a zároveň službou), který provádí návrh, implementaci a provoz takových složitých systémů je systémový integrátor. V souvislosti s provozem a vývojem ICT se mluví o outsourcingu, proto jsme zmínili i tento pojem, jeho formy, rizika.

12 CASE systémy

V této kapitole se dozvíte:

- Co jsou to CASE systémy?
- Jaké jsou jejich kategorie?
- Jak je to s podporou metodik?

Po jejím prostudování byste měli být schopni:

- Porozumět základním komponentách a kategoriím CASE.

Klíčová slova této kapitoly:

CASE, upper, middle, lower CASE.

Doba potřebná ke studiu: 4 hodiny

Průvodce studiem

Kapitola představuje CASE nástroje, jejich místo ve vývoji SW. Dále zmiňuje jednotlivé kategorie CASE podporující určité činnosti a také způsob výběru CASE pro vlastní potřeby.
Na studium této části si vyhraďte 4 hodiny.

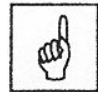
Vznik metodik vývoje software a diagramů pro znázornění systému z různých úhlů pohledu vedl k požadavku automatizace vývoje SW pomocí počítačů. Odpovědí se staly CASE – Computer Aided Software Engineering. CASE nástrojů existuje celá řada, a to jak pro strukturované, tak i pro objektově orientované metody vývoje. Některé CASE nástroje jsou integrovány do moderních prostředí pro vývoj software (např. firma Borland, IBM nebo Oracle, či možnosti rozšíření platformy Eclipse). Přestože vypadá, že tvorba diagramů v těchto nástrojích je jednoduchá, vyžaduje vysokou znalost a profesionálnost tvůrce a těch, kdo je používají. Druhým předpokladem úspěchu je vhodnost použitých metod, na kterých je CASE založen. Podle obratu na trhu CASE je jasné, že o CASE nástroje je velký zájem. Použití CASE neznamená jen kreslení různých modelů, jde o silné nástroje pro zajištění souvislostí, které člověk mentálně neumí pojmout.

12.1 Druhy CASE systémů

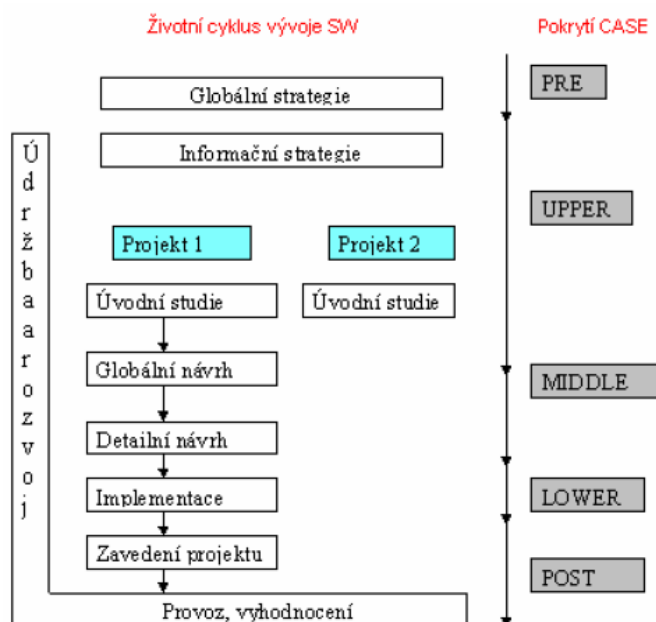
CASE systémy se využívají ve fázích specifikace požadavků, návrhu, kódování a údržbě. Nástroje použité v různých etapách se liší a je obvyklé, že pokrývají jen určité činnosti. Hranice mezi CASE a integrovanými vývojovými nástroji se postupně stírají (viz například CASE a také vývojový nástroj QI Builder informačního systému QI). Podle životního cyklu vývoje software lze CASE nástroje rozdělit do skupin:

- *Pre CASE* – podporuje tvorbu globální strategie.
- *Upper CASE* – podporuje plánování, specifikace požadavků, modelování organizace podniku a globální analýzu IS. Hlavním úkolem nástroje je analýza organizace, zobrazení procesů v organizaci, definice klíčových informačních toků a dokumentace zjištěných požadavků. Z těchto údajů je jasné použití při specifikaci cílů, počáteční specifikaci

požadavků a řízení projektů. Cílem je pochopit danou oblast a specifikovat systém jako celek. Hlavní nástroje jsou DFD a jejich varianty, ERD bez podrobných atributů, prostředky pro řízení projektů a sledování ekonomických skutečností, popis základních vlastností systému prostředky OO modelování.



- *Middle CASE* – podporuje podrobnou specifikaci požadavků a vlastní návrh systému. Tato třída CASE nástrojů je nejúspěšnější. Používají se pro podrobnou specifikaci požadavků, návrh systému, dokumentaci a vizualizaci systému. Obsahem středního CASE jsou metody a nástroje popisované v předchozích kapitolách: DFD včetně podrobného popisu procesů, datových úložišť, podrobné ERD, pro OOAN – diagramy tříd, instancí, přechodové diagramy, apod. Dále middle CASE obsahují systém správy dokumentů a konfigurace, systém pro vyhodnocování metrik, vývoj prototypů, návrh rozhraní, generátory obrazovek a sestav, generátory (kostry) definic dat. Hlavním cílem je formalizace specifikace a návrhu s možností snadných změn a komunikace se zákazníkem a také vytvoření modelů umožňujících generování návrhu. Tento druh CASE je jádrem komerčně dodávaných CASE systémů.
- *Lower CASE* – obsahují nástroje pro podporu kódování, testování a údržby, reverzního inženýrství. Integrovány jsou nástroje jako generátory kódu (generují jen kostru nebo až ¾ výsledného kódu, programátor doplňuje většinou jen detaily), prostředky pro reverse engineering (rekonstrukce dokumentace a modelů z existujícího SW), prostředky pro sledování a vyhodnocení metrik, prostředky plánování a zjištění kvality SW (sběr informací o průběhu testování, vyhodnocení výsledků testů, řízení testování), správa konfigurace, prostředky sledování a vyhodnocování práce systému. Funkce dolních CASE se často překrývají s funkcemi obecných vývojových prostředí.
- *Post CASE* – podporuje organizační činnosti (zavedení, údržbu a rozvoj IS).



Obr. 71 Pokrytí disciplín vývoje IS druhy CASE

12.2 Komponenty CASE systémů

Z toho, jaké jsou obecné funkce a vlastnosti CASE systémů vyplývá také z jakých komponent se tyto systémy skládají. Mezi důležité funkce a vlastnosti CASE systémů patří:

- konzistentní grafické ovládací prostředí (podle zásad tvorby GUI) – jednotný vzhled obrazovek, popisků, tlačítek, jednotné ovládání, použití symbolických ikon, apod.
- centrální databáze pro uchování informací o všech objektech IS (tímto způsobem se zaručí, že informace je použitelná v libovolném dalším kroku projektování),
- prostředky verifikace konzistentnosti dat a podpora normalizace dat,
- textový editor pro popis jednotlivých objektů – pro účely technické a uživatelské dokumentace systému, možnost jejího přímého generování ze systému,
- možnost rychlého návrhu uživatelských obrazovek včetně simulace vstupů a výstupů (je vyžadováno pro prototyping),
- generátor zdrojových programů (pro případy častého znovupoužití daného kódu až ¾ výsledného kódu),
- export / import dat – pro práci s modely a dokumentací, které byly vytvořeny v jiných programech nebo jsou v jiných programech dále využívány a zpracovávány.

Po vyjmenování základních vlastností a funkcí CASE systémů se zmíníme o jejich základních komponentách.

Grafický interface se skládá z obrazových primitiv, která jsou předdefinována (kružnice, čtverce, přímky, křivky, šipky), existuje možnost jejich definování s minimální námahou a s možností uchování. Jedná se v podstatě o elektronickou tabuli, na kterou analytik konstruuje grafy a diagramy. Další požadavky na tyto primitiva jsou podpora kontrolních procesů, kontrola toků.
Grafický interface také umožňuje editování vytvořených grafů (schopnost mazat, přepisovat, modifikovat grafické objekty). Úsilí jaké je třeba k vytvoření diagramu může být měřeno počtem stisků klávesy nebo kliknutí myši. Z dalších vlastností vyjmenujme přejmenování grafických objektů, obnovu objektů do jejich předchozího stavu (po výmazu, i více kroků zpět) – funkce „UNDO“ nebo změnu měřítka objektů.

Pokud jsme zmínili grafický interface, je jasné, že musí existovat nějaké vstupy, kterými bude možno s danými grafickými primitivy či s grafy pracovat, manipulovat a také pomocí nich celý systém ovládat. Mezi vstupní zařízení řadíme jak standardní vstupní periferie počítače (klávesnice, myš), tak také další technická zařízení jako scanner, světelné pero a speciální hardware. Vstupní interface zajišťuje přenos vstupní informace do systému a grafickou interpretaci vstupních operací. Jakýkoliv pohyb myši, stisk klávesy, pohyb světelným perem musí být promítnut do systému a následně na obrazovku počítače (ve spolupráci s výstupním interface). Existuje-li nějaké vstupní rozhraní (interface), mělo by existovat také výstupní. Výstupní interface se stará o provedení výstupů ze systému. Mezi výstupní technická zařízení může patřit monitor, tiskárna nebo plotter. Výstupní interface zahrnuje také definici

výstupního formátu tisků – formát papíru, kvalita tisku, fonty a velikost písma, okraje, tituly, apod.

Důležitou komponentou CASE systémů je slovník, jeho přítomnost v podstatě klasifikuje systém do rodiny CASE nástrojů. V některých nástrojích je slovník automatický (jakmile vytvoří uživatel objekt diagramu, je ve slovníku automaticky o tomto objektu vytvořen záznam). Velikost slovníku definuje maximální počet procesů, toků, datových objektů. Slovník funguje většinou také jako textový procesor pro účely dokumentace. Ve slovníku je obsažena definice datových struktur (použitých datových entit), definice vztahů v hierarchii procesů (rodič-potomek) a v neposlední řadě také relace mezi daty a procesy.

Modelování systémů je doprovázeno možností vytvářet vstupně-výstupní obrazovky. Vývojová prostředí čtvrté generace tyto prostředky obsahují, u CASE nástrojů to však obvyklé není. Každý CASE, který generování obrazovek podporuje může mít jinou úroveň použití grafiky při definici obrazovek, jiný stupeň sjednocení mezi použitou grafikou a textem na obrazovce, jinou úroveň integrace slovníku dat s výstupem.

Důležitou vlastností CASE systému je možnost kontroly kvality modelu. Systém kontroluje tvořené modely a diagramy podle pravidel tvorby a podle definovaných logických souvislostí. Dále kontroluje izolované a nedefinované jednotky dat, procesy a moduly bez specifikace, uložení dat jako externí zdroje nebo self vazby entit (vazby na sebe samu). K těmto kontrolám je použit slovník, kde jsou uloženy vazby a významy jednotlivých entit. Na základě těchto kontrol jsou generovány zprávy, které uživateli oznamují možnost nebo nemožnost provedení daného kroku. Samozřejmostí by měla být podpora generování seznamů datových položek a jejich atributů.

Jelikož jsou CASE systémy ve velkých firmách používány projekčními týmy v síťovém prostředí, měly by podporovat automatickou kompletaci komplexního modelu z jednotlivých částí (které dělali zvlášť jednotlivý řešitelé týmu), výměnu dokumentů, různé formy komunikace a také použití centrálního slovníku. Generování kódu nebo programu je také jednou z vlastností, podle které můžeme systém vybrat či hodnotit. Bereme v potaz dostupnost a složitost automatického generování kódu dle specifikace a typu uvedeného jazyka (jazyků), ve kterém je generování umožněno.

12.3 Volba a hodnocení CASE nástrojů

Při volbě moderních vývojových prostředí je třeba brát v úvahu následující kritéria:

- musí být vytvořeno vědomí formalizace metod a řízení projektu,
- předpoklad použití a znalosti blízkých metodik, na kterých je založen CASE,
- CASE by měl zahrnovat prvky procesního pohledu, měl by zahrnovat všechny nástroje střední úrovně a hlavní nástroje horní úrovně,
- výstupy CASE by měli umožňovat spolupráci s vývojovými prostředími a různými DB systémy, určité CASE mají zaměření na

různá prostředí, která podporují více (většinou dáno jedním výrobcem vývojového prostředí i CASE),
- CASE by měl podporovat moderní architektury jako C/S, třívrstvé, komponentové a distribuované aplikace, také aplikace založená na WS – webových službách,
- měl by splňovat obecné podmínky otevřenosti, nezávislosti na HW, měl by mít kvalitní podporu ze strany dodavatele.

Součástí hodnocení CASE je také kvalita jeho podpory určité metodiky. Některé nástroje podporují strukturované i objektově orientované metodiky, většinou však buď strukturované (CASE 4/0 – Yourdan) nebo objektově orientované (Rational Rose - RUP). Pokud je již v organizaci zavedená nějaká metodika, pak námi vybíraný CASE systém musí tuto metodiku podporovat. Vybraný CASE systém musí umožňovat podporu požadavků na modelování v reálném čase (modelování pomocí grafických nástrojů), správu konfigurací a verzí (CM), správu projektu a pokud možno i správu dokumentů. Jelikož CASE nástroje sledují ještě stále rychlý vývoj metodik, rychle zastarávají. Investice do CASE nástroje se vyplatí jen tehdy, jsou-li tyto nástroje systematicky a intenzivně využívány. Nástroje starší než 4 roky mohou být v dnešní době již morálně a metodicky zastaralé.

12.4 Zkušenosti s CASE a mylné představy

Použití CASE systému přinese pozitivní výsledky pouze v případě, že metody, na kterých je CASE založen, již tým používá a jsou mu blízké. Pouze samo zavedení CASE nevyřeší problémy projektu, ani nezlepší řízení projektu, pokud do té doby žádné neexistuje. Dalším úskalím může být odpor k novotám, ke změně nebo přehnané očekávání (opět, že CASE vše vyřeší za nás). CASE by neměl být napoprvé použit v plné šíři u rozsáhlého softwarového projektu. Měli bychom s jeho nasazením začít u zkušených vývojářů, na menším projektu se známou technologií a ve známé doméně (tedy stejně jako s nasazením vlastní metodiky). Je dobré začít od grafických prostředků (Use case, diagramy tříd, ERD) a postupně zvládat další prostředky. Osvědčují se i návrhy obrazovek (i když rychlejší a přínosnější může být její načrtnutí na papír či tabuli). Diagramy ulehčují komunikaci v týmu a pomáhají zpřesnit analýzu systému. Za těmito výhodami stojí úspěch středních CASE. Dolní CASE již tolik úspěšné nejsou, zřejmě z již zmíněného důvodu pokrytí jeho funkcí běžnými vývojovými prostředími. Co CASE systémy ještě dostatečně nepodporují je zavedení, sběr a analýza softwarových metrik a nedostatečné vazby na normy sady ISO 9000.

Jaké jsou některé mylné představy o těchto systémech, které jsme mohli slyšet, si zmíníme nyní. CASE není samo o sobě metodikou, ale používá již zavedené metodiky (např. CASE orientován na vodopádový proces vývoje). CASE není náhradou programovacích jazyků (může generovat části kódu, ale programovací jazyky nenahrazuje, minimálně detaily chování musíme napsat sami). Všechny CASE systémy pracují podobně v tom smyslu, že poskytují stejné výstupy. Užívání CASE zlepší práci vedoucích pracovníků podniku nebo specialistů pro IT – CASE je nástroj, který může zlepšit produktivitu práce, efektivita práce vždy závisí na osobních kvalitách jednotlivých pracovníků.

CASE odstraňuje potřebu disciplíny a přísného vývoje aplikací IT. Praxe ukázala, že systémy CASE často selhávají právě díky nedisciplinovanosti uživatelů (jako vždy, na lidech záleží v 1. řadě). Automatizací chaosu vznikne automatizovaný chaos. Od CASE se často očekává jako výstup tvorba aplikačního programového vybavení. Hlavní přínos přitom je v úplném poznání fungování podniku a vytváření úplných podkladů právě pro programování aplikací. Produktivita dosažená pomocí CASE není okamžitě zřejmá – na počátku práce je nutné vykonat velmi mnoho práce, která není dlouho vidět. Užívání CASE nezaručí konzistenci výstupů. Když se dá stejný CASE dvěma systémovým analytikům, mohou dospět k docela rozdílným závěrům.

Výčet několika známých CASE nástrojů:

1. Power Designer od společnosti Sybase (definování DB struktur, tvorba kostry DB aplikací včetně menu, …),
2. Skupina nástrojů Rational (Modeler, Architekt, RequisitePro, …) od společnosti IBM,
3. ORACLE Designer od společnosti Oracle (výborný a robustní v kombinaci s databází Oracle),
4. System Architect od společnosti Popkin Software and Systems Inc.,
5. a spousta dalších.

Kontrolní otázky:

1. Co je to CASE?
2. Jaké druhy CASE existují?
3. Jak byste postupovali při výběru CASE pro vaše potřeby?

Úkoly k zamyšlení:

V průběhu předchozích kapitol jsme probrali proces vývoje, specifikace požadavků, nástroje pro podporu vývoje a různé souvislosti. Zamyslete se nad tím, jak to všechno dát dohromady (repository pro ukládání kódu, IDE nástroje, CASE nástroje, build proces, proces vývoje), kdo by se toho měl účastnit, kdo je za to zodpovědný atd.

Korespondenční úkol:

V této kapitole jste dostali informace týkající se CASE a jejich implementace. Zamyslete se nad tím, jakým způsobem byste vybírali CASE pro vaše potřeby a jaké rizika případně problémy vás mohou ohrozit a co mohou způsobit.

Shrnutí obsahu kapitoly

V této kapitole jsme se zabývali CASE nástroji, řekli jsme si jaké známe kategorie, z jakých částí se skládají a nastínili jsme jejich místo ve vývoji IS. Zmínili jsme také zkušenosti a správný způsob výběru CASE.

13 Dokumentace SW

V této kapitole se dozvíte:

- Proč se zabývat dokumentací SW systémů?
- Proč je dokumentace důležitá?
- Jaké nástroje můžeme využít?
- V jaké formě má dokumentace existovat?

Po jejím prostudování byste měli být schopni:

- Porozumět účelu a formě dokumentace software.

Klíčová slova této kapitoly:

Dokumentace architektury, XML, JavaDoc, IEEE1063.

Doba potřebná ke studiu: 3 hodiny

Průvodce studiem

Kapitola zmiňuje principy, formu a účel softwarové dokumentace. Proč a jak dokumentujeme SW systémy, v jaké formě a jaké nástroje k tomu můžeme využít.
Na studium této části si vyhraďte 3 hodiny.

Jelikož dokumentace systémů, stejně jako specifikace požadavků (podrobněji probrána v jedné z předchozích kapitol) bývá v IT průmyslu a hlavně v softwarovém inženýrství problematická či nedostatečná, věnujeme jí taktéž samostatnou kapitolu. Většina vývojářů má ráda psaní kódu, vymýšlení algoritmů, tvoření efektivních konstrukcí, ale jen nerada píše dokumentaci vlastní práce. Stejně jako neprogramujeme vše od začátku a můžeme využívat existující knihovny, moduly, frameworky, existují i možnosti efektivní dokumentace softwarových systémů (znovupoužití use case scénářů, zachycení detailů v kódu a testech apod.). Dokumentace je totiž důležitá a potřebná pro spoustu další práce, pro potřeby dalších rolí. Netvoříme ji jen proto, aby existovala a mohli jsme ji štosovat do sloupců a měřit v metrech, například:

- Specifikace požadavků je důležitá pro testery, kteří verifikují systém, zda dělá správně to, co je požadováno.
- Tým údržby potřebuje dokumentaci architektury, aby věděl, jak je systém navržen a napsán, jaké má vrstvy, které části a jak (pomocí jakých pravidel/protokolů) komunikují.
- Zákazníci potřebují uživatelskou dokumentaci pro podporu práce s naprogramovaným softwarem.

13.1 Forma dokumentace

Jak již bylo zmíněno několikrát v předešlých kapitolách, není v softwarovém inženýrství často rozhodující forma zachycení, „zhmotnění“ artefaktu (model, dokument, test skript, ...), ale zda vůbec danou informaci máme, sdílíme ji či

zda danou aktivitu vůbec provádíme. Toto zdůrazníme zvlášť v případě dokumentace. V závislosti na rozsahu projektu, složitosti technologie, distribuovanosti týmu a zkušenostech členů, množství zainteresovaných stran a dalších faktorech, použijeme různou formu a množství dokumentace. V případě malého, zkušeného týmu, který sedí pohromadě, technologie a problémová doména je známá, můžeme použít pouze ofocený náčrtek architektury z tabule jako architektonický dokument. V opačném případě (velký a distribuovaný či nezkušený tým, složitá či nová technologie, neznámá doména) může být potřeba např. slovník dané domény, model byznys procesů, detailnější dynamický model komunikace jednotlivých SW komponent, dokumentace a nutnost prototypu atd. Proto by pro každý projekt měla platit jiná pravidla dokumentace softwarového systému a důležitých rozhodnutí.

Pokud píšeme zdrojový kód, je vhodné okomentovat kroky daného řešení, i když se nám to může zdát zbytečné. Mnoho lidí si myslí, že je to ztráta času a že to nepotřebují, protože každý programátor zná svůj zdrojový kód. To však platí pouze do chvíle, než se vrhne na jiný projekt, či stačí jenom dovolená a už mu bude trvat pochopit, proč napsal to či ono. Navíc, dnešní vývoj SW je týmová práce, tudíž je samozřejmostí okomentovat svůj výtvor pro ostatní členy v týmu, aby mohli navázat na naši myšlenku, či navrhnout zlepšení. Nejdůležitější výhodou psaní komentářů je možnost uložení naší práce jako knihovny. V tomto případě ostatní nevidí, jak program uvnitř pracuje, vidí jen to, co jim nabízí. Bez kvalitní dokumentace by byl takový program nepoužitelný, minimálně těžko použitelný.

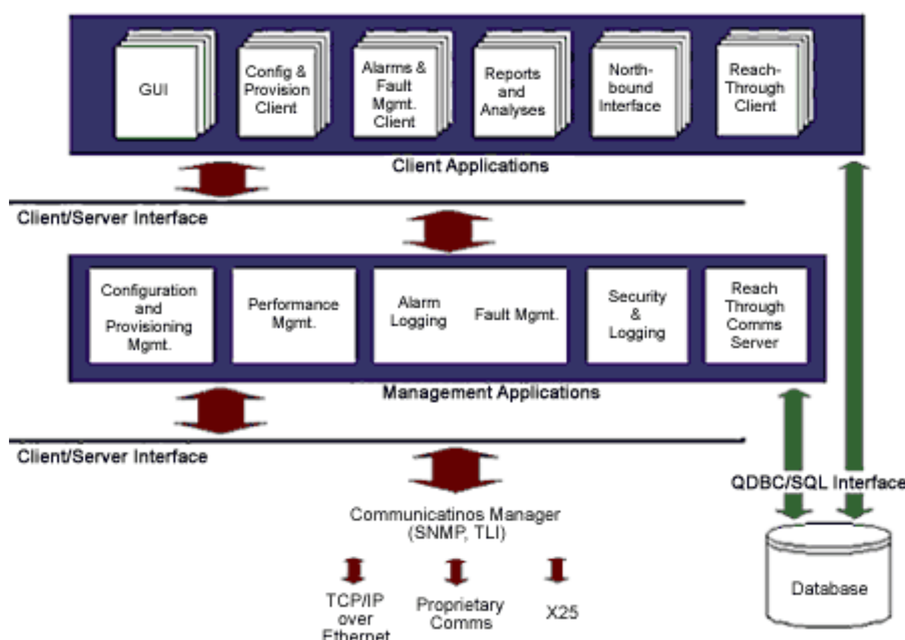
13.2 Dokumentace architektury

V zásadě můžeme říct, že pro dokumentaci architektury je vhodné využít i modelovacích prostředků a CASE nástrojů. Detaily jsou zachyceny a okomentovány ve vlastním kódu, unit testech či test case, ale celkový přehled je nutné někde zachytit. Pro tento účel můžeme použít modelovací nástroje, kreslítka či pouze náčrty s využitím tabule (vidíte, že forma zachycení informace není opět důležitá a bude různá v různých projektech) a samozřejmě doprovodný text. Důležité je aby tento dokument či náčrtek obsahoval základní architektonické informace, tj.

- jaké vrstvy bude aplikace mít,
- jak budou tyto vrstvy komunikovat,
- jaké jsou základní odpovědnosti těchto vrstev,
- vazby na další systémy, rozhraní,
- jakou technologii použijeme pro kterou vrstvu.

V případě architektury je třeba dokumentovat různé pohledy na systém, tzv. view. Statické uspořádání částí aplikace, dynamické chování (komunikace komponent) a popis chování systému z pohledu uživatele jsou základními pohledy, které by měly být zachyceny. Detailněji o této problematice viz např. [SEI2]. Důležité je mít u každého modelu vysvětlující legendu či popsané, co znamenají všechny ty čáry (např. JDBC, TCP, RMI komunikace; volání, instanciace apod.), bloky, čtverečky, kolečka (třídy, use case) apod. Pokud používáme například UML, stačí samozřejmě uvést o jaký diagram se jedná.

Z zásadě však lze pro dokumentaci architektury doporučit následující zásady:
- základní model architektury je zachycen formou modelů (class diagram a rozdělení do balíčků, vrstev, diagram sekvencí)
- jednotlivé detaily jsou obsaženy a dokumentovány (komentáře + JavaDoc komentáře) v kódu,
- další detaily v unit testech a test case,



Obr. 72 Příklad základního modelu architektury EMS systému (zdroj: IEC)

Z pohledu UC driven přístupu pak využíváme pro dokumentaci celého systému minimálně následující:
- Use case – funkční pohled na systém z pohledu uživatele.
- UML Diagram tříd/objektů (spolu s balíčky) – statický pohled.
- UML Sekvenční diagram pro popis dynamického chování.

Poznámka na konec k úplnosti popisu: opět zmiňujeme pouze (podle našeho názoru) nejefektivnější a minimální způsob dokumentace software (stejně jako v případě tvorby softwarové specifikace). Samozřejmě existuje spousta dalších pohledů, doporučení či standardů, technologicky specifických doporučení, názorů. Nezmiňujeme například vůbec dokumentaci databáze, jelikož většina (všechna) logiky by měla být implementována v aplikační logice systému a samotné databázové schéma by mělo být samo-dokumentované, tzn. Dostatečně srozumitelně strukturované a pojmenované.

K dokumentaci softwarového systému se můžeme samozřejmě postavit holisticky, úplně, lze říci až vědecky, např. SEI (Software Engineering Institute při Carnegie Mellon University, např. autor CMMI) má dokonce speciální metody a workshopy zaměřené na dokumentaci architektury, jedná se o ATAM (Architecture Trade-off Analysis Method) a QAW (Quality Attribute Workshop), viz [SEI2]. My raději upřednostňujeme minimalistický, efektivní přístup.

V neposlední řadě je třeba zmínit, že čím více dokumentace máme a na čím více místech (= opakované informace v různých dokumentech), tím více jí musíme aktualizovat a vystavujeme se nebezpečí nekonzistence daných informací. O velmi chybovém prostředí ani nemluvě.

13.3 Nástroje pro dokumentaci

Dokumentaci je možné tvořit ručně, ale také lze použít různé nástroje a techniky, které nám tuto nepříjemnou práci usnadní. Jednou z pomůcek je XML ve všech jeho možnostech a variantách, další pak například Java nástroj pro generování dokumentace ze zdrojových kódu JavaDoc. O těchto si nyní něco řekneme blíže, stejně jako o standardu IEEE 1063 pro tvorbu dokumentace softwarových systémů.

13.3.1 XML

Zkratka XML znamená eXtensible Markup Language. Jde o značkovací jazyk vyvinutý konsorciem W3C. Díky možnosti definovat strukturu dokumentu a díky oddělené definici vzhledu, je možné tento typ dokumentů zpracovávat strojově. Proto je tak vhodný k použití pro dokumentační účely. V XML lze používat vlastní tagy, které dokáží mnohem přesněji označit význam prezentovaných informací. To má velký význam především pro vyhledávání informací, kdy lze pomocí XML doplnit do dokumentu množství meta-informací.

Nyní si XML stručně představme. Mějme dokumentaci, která má nějaký název, autora a verzi. XML dokument pak může vypadat následovně:

```
<dokument>
  <název>Dokumentace SW systému</název>
  <autor>Jarek</autor>
  <verze>1.0</verze>
</dokument>
```

Díky struktuře dokumentu je zřejmé o co se jedná, že jsem jejím autorem a je ve verzi 1.0. Struktura XML dokumentu se definuje pomocí jiného souboru (DTD či XML schématu), o čemž si řekneme později.

XML dokument musí obsahovat vždy jen jeden kořenový element, jména elementu nesmí obsahovat mezeru, atributy jsou uzavřeny v uvozovkách, všechny tagy musí být párové, tzn. používáme otevírací a ukončující tag. XML je case sensitive, tj. rozlišuje velikost písmen, proto musíme dávat pozor přo pojmenovávání entit. Jelikož XML používá jako standardní znakovou sadu Unicode (ISO 10646), lze používat národní znaky i v názvech elementů a atributů. Tolik stručně k syntaxi XML dokumentů. Proto, abychom mohli ověřit i sémantiku dokumentu, existuje

Strukturu XML dokumentu (resp. jeho sémantiku) můžeme definovat pomocí DTD (Document Type Definition) nebo XML schématu. Nejběžněji používané DTD jsou například:

- Hypertext markup Language (HTML) – známé HTML pro tvorbu webových stránek, existují DTD pro jeho jednotlivé verze 2.0, 3.2 a 4.0.
- Synchronized Multimedia Integration Language (SMIL) – jazyk protvorbu multimediálních prezentací.
- DocBook – DTD pro psaní technické dokumentace, je velmi rozšířené a přímo podporováno mnoha aplikacemi.
- Scalable Vector Graphics (SVG) – vektorový formát pro 2D grafiku určený zejména pro webové stránky, na vývoji pracuje W3C.

DTD k našemu příkladu:

```
<!DOCTYPE docka [
<!ELEMENT docka (dokument+)>
<!ELEMENT dokument (název+, autor, verze, popis?)>
<!ATTLIST dokument
      jazyk CDATA #REQUIRED
>
<!ELEMENT název (#PCDATA)>
<!ELEMENT autor (#PCDATA)>
<!ELEMENT verze (#PCDATA)>
<!ELEMENT popis (#PCDATA)>
]>
```

DTD popisuje strukturu dokumentu následovně. Nejdříve vytvoříme soubor *docka.dtd*, který obsahuje definici struktury. Jako první definujeme kořenový element (root) docka, který se může vyskytnout pouze jednou. Deklarací + u tagu dokument říkáme, že daný dokument musí existovat v XML dokumentu alespoň jednou. Dále obsahuje dokument alespoň 1 název, atribut jazyk a povinné údaje autor a verze. Volitelným údajem je popis, což je označeno znakem ?.

Při práci s XML dokumenty používáme automatizované nástroje, které usnadňují práci. Jedním z nich jsou XML parsery a validátory. Tyto programy za nás ověří validitu dokumentu a odhalí možné chyby. Jelikož XML dokument validuje oproti DTD definici, musí být tato správná. DTD samo nelze validovat, jelikož se nejedná o XML dokument. Dále nelze také kontrolovat datové typy (čísla, datum a čas, …). Proto existuje XML Schéma, které tyto problémy odstraňuje. XML Schéma, samotné je XML dokumentem, můžeme je tudíž validovat pomocí standardních nástrojů. Navíc umožňuje u jednotlivých elementů definovat datové typy či přesněji vymezené výskyty.

Příklad XML Schématu, které se nevztahuje k našemu příkladu:

```
Příklad XML Schématu
<schema xmlns="http://www.w3.org/2001/XMLSchema"
          xmlns:b="http://businesscard.org"
          targetNamespace="http://businesscard.org">

  <element name="card" type="b:card_type"/>
  <element name="name" type="string"/>
  <element name="logo" type="b:logo_type"/>

  <complexType name="card_type">
     <sequence>
          <element ref="b:name"/>
          <element ref="b:logo" minOccurs="0"/>
     </sequence>
  </complexType>
</schema>
```

Na začátku kapitoly bylo zmíněno, že XML dokumenty neřeší vzhled. Pokud tedy chceme data reprezentovat, musíme využít jiných možností. Jednou z nich je využít XSL (eXtensible Stylesheet Language), které definují podobně jako kaskádové styly v HTML styl dokumentu XML. Styl do XML dokumentu vložíme pomocí tohoto tagu:

```
<?xml-stylesheet type="text/xsl" href="styl.xsl"?>
```

13.3.2 JavaDoc

Javadoc je nástroj firmy Sun Microsystems sloužící ke generování API dokumentace ve formátu HTML ze zdrojových kódů Java programů. Řekli jsme si, že detailní dokumentaci softwarového systému zachycujeme v testovacích skriptech a ve zdrojových kódech ve formě komentářů. JavaDoc nám umožní část této práce automatizovat. Čitelné komentáře v Java kódu musíme samozřejmě psát sami ☺

Ukázka Java třídy:

```
public class Pr {
      String nm;
      long tn;

      public String gnm() {
            return nm;
      }

      public void sbd(long aaa) {
            tn = aaa;
      }
}
```

Tento zdrojový kód je zcela funkční, ale ignoruje pravidla pro psaní zdrojových kódů nejen v Javě. Název *Pr* nám o vlastní třídě neřekne skoro nic, nevíme jakou instanci vytvoříme. Atributy a metody jsou také nic neříkající,

názvy mají být výstižné, což víme, takže se pokusíme přepsat kód na lepší verzi.
Třída se bude jmenovat Person a instancí bude person (osoba), u které můžeme manipulovat s name (celé jméno) a telephoneNumber (telefonní číslo). Další otázka proč píšeme anglicky? Nikdy nevíme, zda nebudeme zdrojový kód publikovat nebo budeme spolupracovat s kolegy jiných národností. Nejen z tohoto důvodu je lepší mít jednoznačně vše v angličtině. Pokus o lépe strukturovaný a komentovaný kód.

```
public class Person {

      // list of attributes

      String name = "";

      long telephoneNumber;



      public String getName() {

      /*

      * getter method, returns name

      */
```

V kódu přibylo nejen srozumitelných textů, ale také komentářů. Ještě však není zdrojový kód okomentován, tak aby se z nějo dala vytvořit dokumentace pomocí JavaDoc, která by byla čitelná pro ostatní. JavaDoc komentáře značíme jako víceřádkový komentář, s jednou hvězdičkou navíc:

```
/**
 * toto je JavaDoc komentář
 * z tohoto textu se vytvoří dokumentace
 *
 */
```

JavaDoc komentář neslouží pouze ke komentování tříd, metod a kódu, ale dá se provázat s dalšíma JavaDoc komentáři pomocí speciálních příkazů (tagů). Seznam základních tagů JavaDoc:

- @author - autor třídy nebo metody,
- @version - verze zdrojového kódu,
- @since - od které verze je metoda přístupná,
- @param - parametry metody nebo konstruktoru,
- @return - návratová hodnota metody,
- @throws - popíšeme výjimku, kterou metoda vyvolá,
- @see - odkaz na jinou třídu (viz také),

- @deprecated - tímto termínem řekneme, že by se metoda neměla používat.

```
/**
 * Class creates object that has type Person
 *
 * @author  Pepa Jose Novak
 * @version 1.3
 */
public class Person {
        // class variables
    // normal comment, not included into documentation
        String name;
        long telephoneNumber;

        /**
         * @return the name
         */
        public String getName() {
                return name;
        }

        /**
         * @param telephoneNumber phone Nr. to set
         */
        public void setTelephoneNumber(long telephoneNumber) {
                this.telephoneNumber = telephoneNumber;
        }
}
```

Nyní můžeme vytvořit dokumentaci příkazem:

```
javadoc *.java -d Dokumentace
```

```
C:\WINDOWS\system32\cmd.exe
(C) Copyright 1985-2001 Microsoft Corp.

c:\Person>javadoc *.java -d Dokumentace
Creating destination directory: "Dokumentace\"
Loading source file Person.java...
Constructing Javadoc information...
Standard Doclet version 1.6.0_04
Building tree for all the packages and classes...
Generating Dokumentace\Person.html...
Generating Dokumentace\package-frame.html...
Generating Dokumentace\package-summary.html...
Generating Dokumentace\package-tree.html...
Generating Dokumentace\constant-values.html...
Building index for all the packages and classes...
Generating Dokumentace\overview-tree.html...
Generating Dokumentace\index-all.html...
Generating Dokumentace\deprecated-list.html...
Building index for all classes...
Generating Dokumentace\allclasses-frame.html...
Generating Dokumentace\allclasses-noframe.html...
Generating Dokumentace\index.html...
Generating Dokumentace\help-doc.html...
Generating Dokumentace\stylesheet.css...

c:\Person>
```

JavaDoc nám v adresáři Dokumentace vygeneruje HTML dokumentaci naší třídy. Vytvořená dokumentace slouží jako API dokumentace, čili popisuje jaké třídy, konstruktory a metody se vyskytují v našem programu (knihovně) a jak se mají používat.

Náhled vytvořené dokumentace:

Class Person

```
java.lang.Object
```

Person

```
public class Person
extends java.lang.Object
```

Class creates object that has type Person

Version:
1.3

Author:
Pepa Jose Novak

Constructor Summary
Person()

Method Summary	
java.lang.String	getName() **Deprecated.** *Do not use this method, it has been replaced by new.*
void	newMethod() new method
void	setTelephoneNumber(long telephoneNumber) klasicky setter

Constructor Detail

Person

`public Person()`

Method Detail

getName

`public java.lang.String getName()`

Deprecated. *Do not use this method, it has been replaced by new*
getter method
Returns:
the name
See Also:
novaMetoda()

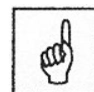
---

setTelephoneNumber

`public void setTelephoneNumber(long telephoneNumber)`

setter method
Parameters:

13.3.3 IEEE 1063-2001



Dalším z IEEE standardů a doporučení o kterém si něco povíme je Standard IEEE 1063-2001 je z roku 2007 a týká se dokumentace softwarových systémů. Tato verze aktualizuje starší dokument, který popisoval pouze použití tištěné dokumentace, tento nově zahrnuje i použití elektronických dokumentů. Standard opět, stejně jako doporučení IEEE 830 pro tvorbu požadavků na SW systém, definuje minimální požadavky na strukturu, obsah a formát uživatelské dokumentace. IEEE 1063 se omezuje pouze na softwarovou dokumentaci produktu, neobsahuje proces vývoje SW. Tento dokument může být obsažen v kontraktu, čímž dodavatel souhlasí, že bude doručovat dokumentaci v souladu se standardem.

Dokument IEEE 1063 obsahuje:

- Zaměření (scope) standardu popisující proč a co v případě software dokumentovat, pro koho je standard určen.
- Definice pojmů jako jsou kritické informace (critical information), ilustrace a jiné grafické elementy (illustration) či uživatel (user).
- Strukturu a formát SW dokumentace (přístupy, média, množství).

Součástí dokumentu je opět také doporučení týkající se formy a obsahu vlastní dokumentace. Konkrétně je zmíněno a vysvětleno, že dokumentace musí být complete (kompletní), accurate (přesná), jednoznačně identifikovatelná, apod. a to pro různé účely: tutoriály, uživatelská dokumentace, chybové hlášení. Jak již bylo řešeno v úvodu, týká se také elektronických verzí, navigace v tomto typu dokumentů, způsobu odkazování.

Komponenty uživatelské dokumentace podle IEEE 1063:

Component	See subclause	Required?
Identification data (package label/title page)	4.3	Yes
Table of contents	5.7.1	Yes, in documents of more than eight pages after the identification data
List of illustrations	5.7.2	Optional
Introduction	3.2	Yes
Information for use of the documentation	4.4	Yes
Concept of operations	4.5	Yes
Procedures	4.6, 4.7	Yes (instructional mode)
Information on software commands	4.8	Yes (reference mode)
Error messages and problem resolution	4.9	Yes
Glossary	4.10	Yes, if documentation contains unfamiliar terms
Related information sources	4.11	Optional
Navigational features	5.8	Yes
Index	5.7.3	Yes, in documents of more than 40 pages
Search capability	5.7.4	Yes, in electronic documents

Tabulka 13-1: komponenty uživatelské dokumentace podle IEEE 1063

Při tvorbě dokumentace nemusíme všechno psát znovu, jak jsme již viděli, lze znovupoužít komentáře v kódu (v případě Javy a JavaDocu) nebo také znovupoužít textové popisy scénářů use casů jako uživatelskou dokumentaci.

Kontrolní otázky:

1. Co a jak dokumentovat při vývoji SW?
2. Co je to XML?
3. Co je a k čemu slouží JavaDoc?
4. Co je doporučení IEEE 1063?

Úkoly k zamyšlení:

Zamyslete se nad dalšími možnostmi automatizace tvorby programátorské a uživatelské dokumentace, nad možným znovupoužitím existujících dokumentů, nad strukturou.

Korespondenční úkol:

V kapitole 5 jsme naznačili příklady scénářů pro use case ATM. Pokuste se vylepšit scénář a znovupoužít ho jako uživatelskou dokumentaci pro tyto funkce ATM. Pokuste se také v % zhodnotit jak velký díl práce jste museli vynaložit při transformaci scénáře na uživatelskou dokumentaci.

Shrnutí obsahu kapitoly

V této kapitole jsme se zabývali další důležitou disciplínou softwarového inženýrství, jíž je tvorba dokumentace. Řekli jsme si zásady pro její tvorbu a také možnou strukturu a nástroje. Opět jsme propagovali agile přístup, kdy se snažíme znovupoužít existující dokumentaci.

Literatura

Buchalcevová, A., Pavlíčková, J.: Základy softwarového inženýrství - objektově orientované programování Praha,VŠE, 2002,

Fowler M.: Destilované UML, Grada Publishing a.s., ISBN 978-80-247-2062-3

McConnell S.: Dokonalý kód: umění programování a techniky tvorby software, Computer press 2006, ISBN 9788025108499

Procházka J., Klimeš C.: Provozujte IT jinak, Grada Publishing a.s., 2011, ISBN 978-80-247-4137-6