INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ

INFORMAČNÍ SYSTÉMY 1

JAROSLAV PROCHÁZKA
JAROSLAV ŽÁČEK

ČÍSLO OPERAČNÍHO PROGRAMU: CZ.1.07
NÁZEV OPERAČNÍHO PROGRAMU:
OP VZDĚLÁVÁNÍ PRO KONKURENCESCHOPNOST

TVORBA DISTANČNÍCH VZDĚLÁVACÍCH MODULŮ PRO CELOŽIVOTNÍ VZDĚLÁVÁNÍ DLE § 60 ZÁKONA Č. 111/1998 SB. O VŠ NA PŘÍRODOVĚDECKÉ FAKULTĚ OSTRAVSKÉ UNIVERZITY

REGISTRAČNÍ ČÍSLO PROJEKTU: CZ.1.07/3.2.07/02.0033

OSTRAVA 2016

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

Recenzenti:

Ing. Roman Šmiřák – odborný oponent
RNDr. Martin Kotyrba – metodický oponent

Název: Informační systémy 1
Autor: Jaroslav Procházka, Jaroslav Žáček
Vydání: druhé, 2016
Počet stran: 120

Studijní materiály jsou určeny pro distanční modul: Informační systémy 1

Jazyková korektura nebyla provedena, za jazykovou stránku odpovídají autoři.

Obsah

1 Průvodce modulem Informační systémy 1

Modul je realizován v rámci projektu ESF VK **Tvorba distančních vzdělávacích modulů pro celoživotní vzdělávání dle §60 zákona č.111/1998 Sb. o VŠ na PřF OU**
Registrační číslo projektu: CZ.1.07/3.2.07/02.0033

1.1 K čemu slouží Průvodce modulem

Cílem Průvodce modulem je seznámit zájemce o studium s obsahovým zaměření a optimálním způsobem studia modulu distanční formou, dále poskytnout informace o tutoriálech, korespondenčních úkolech, seminární práci a podmínkách ukončení modulu, konkrétně bodové limity pro zápočet, zkoušku a výsledné hodnocení.	Cíl Průvodce modulem

1.2 Dvě stránky modulu – FORMA a OBSAH

Forma modulu je distanční, což představuje především samostudium s využitím PC připojeného k internetu, SW výukového řídicího systému Moodle. Tento systém slouží ke komunikaci s garantem modulu a lektory modulu (konzultace ke studované látce, plnění korespondenčních úkolů, semestrálních projektů aj.) a se spolužáky studujícími stejný modul. Presenční forma studia probíhá formou tutoriálů, které jsou věnovány organizačně technickým stránkám samostudia, a také ke konzultacím k dané problematice. Prezenční forma je doplněna vypsanými konzultacemi, přesné termíny jsou vždy aktualizovány v LMS Moodle, aby účastníci modulu s nimi byli seznámeni. Závěrečná zkouška/zápočet je vždy realizována prezenční formou na fakultě.	Forma modulu
Obsahově je modul zaměřen hlavně na základní teoretické znalosti v oblasti modelů a disciplín vývoje komplexních informačních systémů, částečně také na stránku praktického zvládnutí zpracování jednoduchých projektů (tato perspektiva je řešena v samostatném modulu Ročníkový projekt). Pro demonstraci klíčový principů je použit procesní framework IBM Rational Unified Process (RUP) a jeho odlehčená varianta OpenUP.	Obsah modulu

1.3 Formální charakteristika modulu Informační systémy 1

1.3.1 Rozsah modulu

• **Úvodní tutoriál** Seznámení se způsobem komunikace s garantem a lektorem modulu, způsobem komunikace s ostatními studenty. Student bude rovněž seznámen s obsahem vybraného modulu a se všemi požadavky na jeho úspěšné zvládnutí. Dále bude účastník modulu seznámen s termíny odesílání korespondenčních úkolů a jejich hodnocením. • **Samostudium** je důležitou součástí studia modulu. Jde o získávání znalostí a dovedností v oblasti vývoje komplexních IS podle moderních trendů (zvláště pochopení principů, smyslu fází a konceptu iterací), zpracování a odesílaní korespondenčních úkolů. • **Další tutoriály** V rámci tutoriálů budou řešeny vyslovené problémy se zvládnutím stěžejních oblastí studia modulu, se složitějšími kapitolami modulu. Tutoriály jsou převážně řešeny jako konzultační, z tohoto důvodu musí být účastníci modulu připraveni na dotazy, které se budou týkat zaslaných korespondenčních úkolů. • **Závěrečný tutoriál** Tento tutoriál bude věnován diskuzi k odevzdaným semestrálním projektům. • **Zkouška/zápočet** Zkouška/zápočet vždy probíhá prezenční formou.	Časový harmonogram

1.3.2 Komunikace v modulu

Komunikace mezi studentem a tutorem (lektorem), mezi studenty probíhá v prostředí Moodle. Ve výjimečných případech lze použít e-mail nebo telefon. Systém LMS Moodle slouží také k odevzdávání korespondenčních úkolů, semestrálních prací, aj.	Způsoby komunikace

1.3.3 Charakteristika účastníků – cílová skupina projektu

Cílovou skupinou jsou zájemci o studium v distančních formách výuky akreditovaných studijních programů na PřF OU nebo na dalších univerzitách s příbuznými obory. Zájemci dostanou příležitost začít studium právě s využitím § 60 zákona č.111/1998 Sb., a tím se na další studium v akreditovaných programu připravit absolvováním nabízených modulů, které jim mohou být uznány v případě přijetí ke studiu akreditovaných oborů. Předpokladem přijetí ke studiu nabízených modulů je středoškolské vzdělání ukončené maturitní zkouškou. Cílovou skupinou jsou například zaměstnanci počítačových i jiných firem bez VŠ vzdělání, případně se vzděláním v jiných oborech. Dále absolventi středních škol, kteří vstoupili na trh práce, ale uvažují o dalším vzdělávání distanční formou, zájemci o informační technologie. Moduly projektu jsou určeny i těm zájemců, kteří jsou vedeni na úřadě práce a snaží se získat zaměstnání získáním nových znalostí, absolvováním dalších rekvalifikací apod.	Cílová skupina
Absolvent modulu získá obecné znalosti životního cyklu vývoje IS a podrobnější znalosti procesu vývoje podle RUP/OpenUP (fáze, disciplíny, role, činnosti). Nedílnou součástí je také základní znalost UML a jeho použití v procesu vývoje.	Popis absolventa modulu

1.3.4 Kritéria hodnocení a způsoby prověřování znalostí v modulu

Vedoucí modulu hodnotí splnění základních parametrů úkolů s dvěma stupni splnění: splnil – nesplnil.	Jak a kdo hodnotí samostatné práce
Vypracování zadaných korespondenčních úkolů	Požadavky na zápočet/zkoušku
Diskuse nad vypracovanými korespondenčními úkoly včetně ověření teoretické znalosti	Jak probíhá zápočet/zkouška

1.3.5 Doplňující informace

Student bude ke studiu potřebovat PC s připojením na Internet. Dále doporučuji si zajistit doporučenou literaturu (alespoň k nahlédnutí v době přípravy na zkoušku). Průběžně si student bude muset nainstalovat na svůj počítač příslušný software, který je uveden v modulu (case nástroj pro modelování UML diagramů), jelikož je nutný k vypracování korespondenčních úkolů.	

1.4 Obsahová charakteristika modulu Informační systémy 1

1.4.1 Anotace modulu

Náplní předmětu je detailní zaměření na problematiku celého životního cyklu vývoje SW, počínaje specifikací požadavků až po testování, nasazení, tvorbu dokumentace a zajištění kvality jako součást práce. Probrány jsou také různé přístupy k vývoji SW (vodopádový, spirálový, iterativně inkrementální). Jako nosné metodiky, pomocí kterých jsou popsány role, činnosti, modely a dokumenty jsou brány IBM Rational Unified Process (RUP) a jeho minimalistická verze OpenUP. Pro vizualizaci modelů systému je použito UML. Pokud se v dané oblasti používá některých standardů ISO či IEEE jsou studenti seznámeni také s těmito standardy.	Anotace

1.4.2 Jaké jsou požadavky na předchozí znalosti a vybavení studujících

Tento předmět integruje dosud nabyté znalosti studentů v ostatních předmětech, resp. znalosti z rozličných oborů. Nutným předpokladem je alespoň základní znalost architektur počítačů a hardware, sítí, programování, aplikací software a databází. Nezbytné je pak objektově orientované myšlení a znalost objektově orientovaných principů jako základ objektově orientované analýzy a návrhu.	Požadavky na předchozí znalosti

1.4.3 Podrobná osnova modulu

Zaměříme se na následující témata: 1. Základní problematika, pojmosloví a cíle předmětu. 2. Hra na pochopení principů iterativně inkrementálních přístupů. 3. Základní modely vývoje SW: vodopádový, spirálový, iterativně inkrementální. 4. Proces vývoje SW podle RUP/OpenUP - fáze, iterace, milníky. 5. Byznys modelování podle RUP/OpenUP (účel, metody, notace, nástroje, formalismy). 6. Specifikace a sběr požadavků podle RUP (role, techniky, nástroje). Standard IEEE 830. 7. Analýza podle RUP/OpenUP (role, aktivity, modely), realizace use case modelů pomocí UML sekvenčních modelů. 8. Návrh podle RUP/OpenUP (role, aktivity, modely), možnosti SW architektur, vizualizace pomocí UML modelů nasazení/komponent. 9. Implementace podle RUP/OpenUP (strategie, role, aktivity), TDD, MDD. 10. Testování, testovací strategie a scénáře (validace,	Osnova

verifikace, testování komponent, integrační a regresní testování, white / black box, xUnit). 11. Nasazení podle RUP/OpenUP, řízení konfigurací a změnové řízení. 12. Zajištění bezpečnosti a kvality (QA) při tvorbě software - postupy, nástroje.	

1.4.4 Klíčová slova modulu

Metodiky vývoje SW, RUP, OpenUP, UML, architektury IS.	Klíčová slova

1.4.5 Doplňující literatura

PROCHAZKA, J., KLIMES, C. *Provozujte IT jinak. Agilní a štíhlý provoz, podpora a údržba informačních systémů a IT služeb.*. Praha Grada, 2011. Kroll, P., MacIsaac, B. *Agility and Discipline Made Easy: Practices from OpenUP and RUP*. Addison-Wesley, 2006. Arlow, J., Neustadt, I. *UML a unifikovaný proces vývoje aplikací. ComputerPress 2003*. Procházka, J., Vajgl, M. *Ročníkový projekt 1, 2*. Ostravská univerzita 2011.	Doplňující informace k modulu

2 Základní pojmy

V této kapitole se dozvíte:

- Zopakujeme základní pojmy.
- Jaké jsou problémy projektů?

Po jejím prostudování byste měli být schopni:

- Definovat informační systém
- Rozlišovat globální a dílčí architektury
- Porovnat pojmům metoda, metodika, technika a nástroj

Klíčová slova této kapitoly:

Informační systém, trendy, metodiky, metody, techniky, architektura.

Doba potřebná ke studiu: 3 hodiny

Průvodce studiem

Kapitola se věnuje opakování základních pojmů, které jsou nezbytné pro pochopení celého textu. Jedná se o trendy v oblasti IT, pojmy jako informační systém, metodika, architektura apod.
Na studium této části si vyhraďte 3 hodiny.

Tento text se zabývá nejen vývojem, ale také doručením, provozem a podporou (rozuměj údržbou a rozšířením) vytvořených softwarových systémů. Cílem celého týmu informatických profesí není pouze vyvinout software, který zákazníkovi přinese nějakou hodnotu, ale také tento systém provozovat a dále rozvíjet (opravovat objevené chyby, implementovat potřebné změny), a to vše s přijatelnými náklady a dostupnými zdroji podle ověřených, fungujících postupů/principů.

V první část textu představíme základní modely vývoje SW obecně, informační systémy nevyjímaje. Jádrem textu je výklad principů iterativně inkrementálního modelu, který je řízen riziky a Use Casy. Tento přístup zaručuje doručení hodnoty (spustitelného programu) zákazníkovi na konci každé iterace. Tento přístup je nastíněn s použitím metodiky RUP – Rational Unified Process (přesněji bychom měli říct procesního frameworku) a jeho zjednodušené modifikaci zvané OpenUP. Nebudeme se tedy věnovat popisu tohoto frameworku, který je možné nalézt ve spoustě literatury [Kr03a], [Kr03b] a také na Internetu [RE]. Zaměříme se spíše na problematiku iterativně inkrementálního vývoje, který je řízen riziky a Use Casy, na jeho principy, zásady a možné obtíže při jeho implementaci. Klíčovým faktorem je zde samozřejmě myšlení lidí, kteří se projektu účastní. Nelze dosáhnout úspěšné implementace v dohodnutém termínu a v dohodnuté ceně jen použitím RUP/OpenUP, pokud bude team myslet „vodopádově“. Proto se budeme bavit také o agilních principech a agilním manifestu, jelikož RUP je ve své podstatě také agilním přístupem.

Ještě než se do této problematiky ponoříme hlouběji, zmíníme některé důležité pojmy a vztahy.

2.1 Co v textu naleznete a co ne?

Text se věnuje problematice vývoje informačních systémů a iterativně inkrementálního vývoje komplexního softwaru s pomocí procesního frameworku RUP/OpenUP. Pro ilustraci některých aspektů vývoje použijeme procesní framework RUP (Rational Unified Process), popř. OpenUP. Oba představené procesní frameworky (RUP i OpenUP) jsou založeny na ověřených a úspěšných řešeních v daných oblastech, tzv. „best practices". Nejedná se tedy o konkrétní metodiky, které říkají jaké přesné kroky vykonávat, jaké dokumenty používat, ale o sady činností, rolí a dokumentů, ***které si tým vybere a přizpůsobí pro každý projekt***. Je tedy zřejmé, že kritickým faktorem pro tento krok bude zkušenost vedoucího týmu či metodika projektu, resp. mentora, který s výběrem vhodných artefaktů a činností pomůže.

V textu nenaleznete CASE study, jelikož je každý projekt odlišný a tudíž by toto nemělo velký význam. Spíše se zaměříme na demonstraci některých klíčových principů, jejichž samotná adaptace v projektu přináší výsledky.

2.2 Základní pojmy, opakování

Na úvod si zopakujeme některé pojmy, se kterými by měl být čtenář již seznámen. Jedná se hlavně o pojmy jako je informační systém, architektura či metodika vývoje SW a principy, které se v této oblasti používají (viz [Kli04], [Von02]). Dále také představíme pojmy nové.

2.2.1 Informační systém

Co je informační systém (dále IS) a z čeho se skládá? Informační systém je obecně řečeno ***soubor lidí, metod a technických prostředků zajišťujících sběr, uchování, analýzy a prezentace dat určených pro poskytování informací mnoha uživatelům různých profesí***. IS může a nemusí být podporován výpočetní technikou. My budeme uvažovat systémy podporované počítači. V takovém případě se IS skládá z následujících komponent:

- Technické prostředky (hardware) – počítačové systémy doplněné o periferní jednotky.
- Programové prostředky (software) – jsou tvořené systémovými programy, které řídí chod počítače, efektivní práci s daty, komunikaci počítačového systému s reálným světem a programy aplikačními.
- Datové zdroje – ke své práci je využívají programové prostředky.
- Organizační prostředky (orgware) – soubor nařízení a pravidel. Ty definují provozování a využívání informačních systémů a informačních technologií.
- Lidská složka (peopleware) – řeší otázky adaptace a účinného fungování člověka v počítačovém prostředí, do kterého je zasazen.
- Reálný svět (informační zdroje, legislativa, normy) – kontext informačního systému.

Informační systém musí mít prostředky sběru, kontroly a uchovávání dat. Data musí být zobrazitelná ve srozumitelné formě pro uživatele. Jinak potřebujeme data zobrazit pro ředitele, jinak pro návrháře výrobku či služby a jinak pro

skladníka. Toto zobrazení bývá častým problémem tvorby IS. IS je nástroj podporující jisté činnosti, proto ho není možno koupit jako obyčejný program, je třeba upravit již existující nebo vytvořit nový. K tomu je zapotřebí analýza potřeb a požadavků a ***z toho vyplývá spolupráce dodavatele se zákazníkem***. Je potřeba vědět proč, z jakého důvodu je IS zaváděn. Základními problémy, které mohou vést až k nedokončení vývoje, bývá nejasnost nebo nekomplexnost požadavků na systém, nedostatek zájmu a podpory ze strany budoucího uživatele nebo také nedostatek zdrojů (čas, peníze).

Teorie informačních systémů se proto musí soustřeďovat na řešení řady otázek:

- Metody ukládání a vyhledávání informací v paměťových podsystémech počítače.
- Jazyka a metody pro popis systémů a procesů.
- Problematika kontroly chyb a spolehlivosti.
- Principy učení se a heuristik.
- Procesy typu člověk – stroj.
- Typy zpracování informací – od dávkového, přes interaktivní až po plně automatizované.

Informatika je multidisciplinární obor, jehož předmětem je tvorba a užití informačních systémů v organizacích a společenstvích, a to na bázi moderních informačních technologií. Zmínili jsme spojení multidisciplinární obor. To znamená, že zahrnuje jak technické, tak také ekonomické, sociální, psychologické, právní a další aspekty.

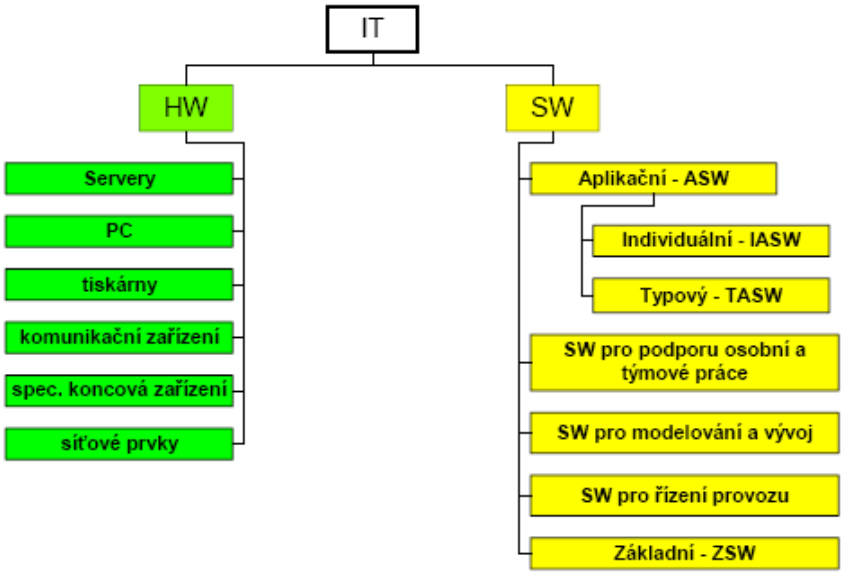
Systém chápeme jako uspořádanou množinu prvků spolu s jejich vlastnostmi a vztahy mezi nimi, jež vykazují jako celek určité vlastnosti, resp. „chování“. Pro naše účely zkoumání informačních systémů pak mají smysl jen takové systémy, u kterých je možno definovat účel, čili tzv. systémy s cílovým chováním. Jinak také řečeno systém je množina vzájemně propojených komponent, které musí pracovat dohromady pro celý systém tak, aby tento systém naplnil daný účel (daný cíl). Tzn. že i když každý jednotlivý prvek systému je dobře navržen a pracuje efektivně, jestliže tyto prvky nepracují dohromady, systém neplní svoji funkci.

Informací rozumíme data, kterým jejich uživatel přisuzuje určitý význam a která uspokojují konkrétní objektivní informační potřebu svého příjemce. Nositelem informace jsou číselná data, text, zvuk, obraz, případně další smyslné vjemy. Na rozdíl od dat (zvuků, obrázků apod.) nemůžeme informaci skladovat. Na druhé straně informací jako zdroj poznání jsou zdrojem obnovitelným, nevyčerpatelným. I když má informace nehmotný charakter, je vždy spojena s nějakým fyzickým pochodem, který ji nese.

Informační systém je soubor lidí, technických prostředků a metod (programů), zabezpečujících sběr, přenos, zpracování, uchování dat, za účelem prezentace informací pro potřeby uživatelů činných v systémech řízení. Abychom mohli zpracovávat dat, ze kterých posléze vzniknou informace, potřebujeme určité nástroje, metody a znalosti, které budeme dále nazývat informačními technologiemi.

Nyní již víme, co si pod pojmem informační systém představit. Zmíníme tedy rozdělení systémů a informačních technologií (IT) podle několika kategorií a klasifikací. Informační technologie jsou hardwarové a softwarové prostředky pro sběr, přenos, uchování, zpracování a distribuci informací. Hlavní rozdělení IT do dvou kategorií je:

- Technické prostředky (HW) – zařízení na pořizování, uchování, přenos, zpracování a prezentaci dat.
- Programové prostředky (SW) – algoritmizované postupy vyjádřené ve formě, v které jsou srozumitelné pro používaná technická zařízení.



Obr. 2-1: Rozdělení IT

Obr. 2-2: Aspekty IS

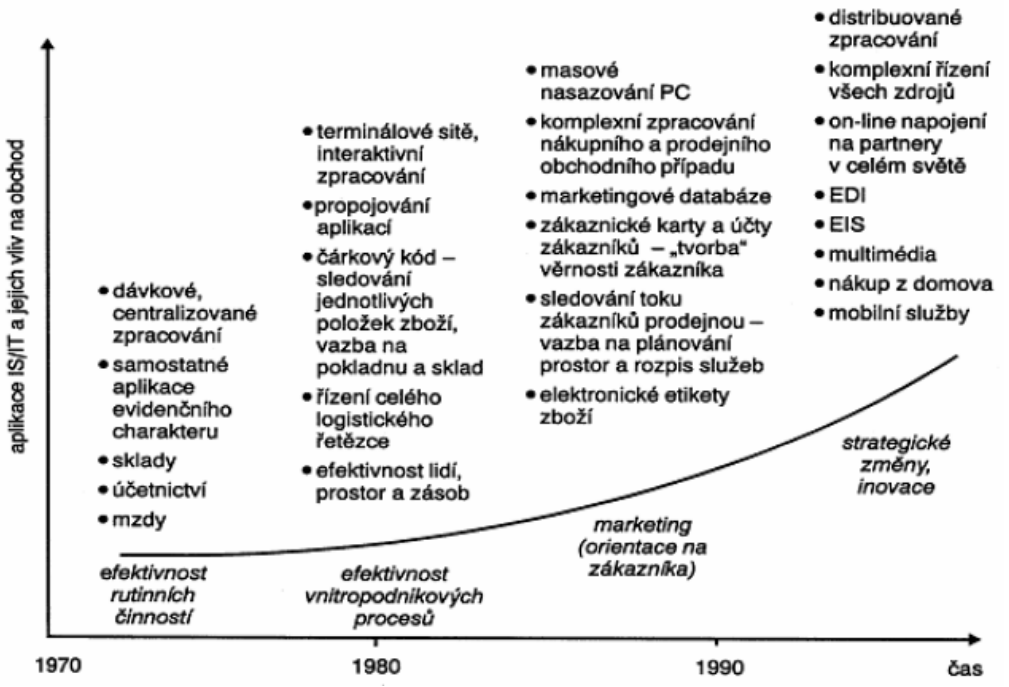
Ke komponentám infrastruktury IT patří:

- výkonný HW včetně síťových a komunikačních prostředků,
- vhodné a perspektivní operační a databázové systémy (základní SW),
- správné datové zdroje (dataware),
- dostatečná informační a počítačová gramotnost lidí (peopleware),

- adekvátní organizační uspořádání kompatibilní s informačními systémy a se systémem řízení podniku (orgware).

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 řadou 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).

Informatická služba (dnes je více používaným pojmem *IT služba*, přesná definice podle ITIL viz učební text SWENG) 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 čemu 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.



Obr. 2-3: Vývoj aplikací IS

Organizační struktura podniků a organizací v 70. letech byla převážně hierarchická. IS/IT bylo reprezentováno centrálním počítačem. V 80. letech vznikaly relativně nezávislé jednotky (divize) zaměřené na hlavní předmět činnosti. Struktura IT byla tvořena počítači propojenými přes síť LAN. 90. léta a první roky nového století jsou ve znamení flexibilních organizací, kde se struktura organizace pružně přizpůsobuje měnícím se podmínkám. Tvoří se dynamické týmy a IS/IT se vyznačuje distribuovaným a mobilním zpracováním dat. Sílí vazba mezi IS/IT a reengineeringem podnikových procesů. Podniky přizpůsobují IS/IT dynamice světového vývoje a změnám podnikových procesů. Vznikají virtuální pracovní týmy a začínají se

poskytovat mobilní služby IS/IT. Roste také význam informací o okolí a pro okolí podniku. Probíhá přesun priorit ke strategickému řízení a také posun zaměření IS/IT – snižování nákladů, zvyšování kvality při vzrůstající rychlosti reakce na kladené požadavky. Existuje rozdílná doba morální životnosti HW, základního SW (ZSW) (operační systémy, SŘBD) a aplikačního SW (ASW). Tato doba je u HW asi 2-3 roky, u ZSW 3-5 let a u ASW je doba životnosti dokonce 10-15 let (je běžné, že se setkáme s aplikacemi staršími než 20 let, např. v bankovním či telekomunikačním sektoru). Spousta z nás se alespoň jednou setkala s tím, že na počítači s instalovaným operačním systémem např. Windows 2000 vyměnila třeba grafickou kartu nebo rozšířila operační paměť RAM během dvou tří let.

Trendy v oblasti základního SW

Standardizují se funkce a uživatelské rozhraní operačních systémů (Windows, MacOS). Rozvíjí se distribuované systémy a s tím souvisí vznik platforem jako Java EE či .NET[1]. Rozvíjí se komunikační ZSW a s ním spojené služby. Trendy v oblasti databází se vyznačují přechodem od relačních k postrelačním databázím. Vznikají a již celkem běžně se používají objektově-relační, objektové, deduktivní, XML databáze či jejich hybridní verze (např. Caché).

Trendy v oblasti aplikačního SW

Trendy technologicky orientovaného SW jsou kancelářské balíky (včetně jejich online verzí např. Google Apps či Microsoft Cloud) a zaměření na workflow. Vzniká typový aplikační software s možností parametrizace. Takový software je komplexní a lze „nastavit“ pomocí parametrů dle potřeb uživatele. Aplikační SW má stavebnicovou architekturu, lze přidávat a odebírat různé části (díly) software. Vznikají otevřené systémy, které jsou standardizovány mezinárodními organizacemi a konsorcii (např. W3C, IEEE, ISO, OMG, …), členy takových skupin jsou často velcí a významní hráči na IT trhu (IBM, Microsoft, HP, Intel, Oracle, …). Díky těmto otevřeným standardům se upouští od proprietárních řešení výrobců. Týká se to převážně těchto oblastí:

- Počítačové sítě (OSI model, protokoly TCP/IP, IPX).
- Operační systémy.
- Objektové prostředí (CORBA, COM/COM+, UML).
- Komunikace s databází (SQL, ODBC, JDBC).
- Nezávislost uživatelského rozhraní na výpočetním rozhraní.
- Možnost výměny HW a ZSW bez vlivu na aplikaci.

Trendy v oblasti architektury IS

Aplikace jsou konstruovány tak, že se přechází od jednovrstvé architektury k třívrstvé či distribuované (CORBA, SOA, AJAX). V jednovrstvé architektuře jsou data, funkce i uživatelské rozhraní integrovány v jeden celek. V architektuře vrstvené jsou všechny tyto části odděleny; s tím souvisí i využívání klient/server architektury. V takovém případě jsou data a funkce uloženy na serveru a na klientském počítači je pouze uživatelské rozhraní –

[1] Java EE – Java Enterprise Edition od firmy SUN Microsystems (http://java.sun.com)
.NET – platforma firmy Microsoft (http://www.microsoft.com)

prezentační logika, pomocí které uživatel požaduje určité funkce po serveru. Server provede danou službu a zašle odpověď zpět klientovi, kde je reprezentována pomocí uživatelského rozhraní. Díky oddělení těchto tří logických celků lze jednoduše vyměnit určitou část systému, např. úložiště dat nebo uživatelské rozhraní, bez velkých zásahů do aplikace. Změní se pouze komunikační rozhraní částí, kterých se to týká. Přechází se od centralizovaného a decentralizovaného zpracování k distribuovanému a globálnímu zpracování. V počátcích počítačového zpracování (70. léta minulého století) bylo zpracování dat soustředěno kolem centrálního počítače. Centrální bylo rovněž řízení a kontrola. Později bylo zpracování dat decentralizováno na samostatné počítače – tím bylo dosaženo zvýšení produktivity dílčích prací. Tyto jednotky ale pracovali individuálně bez nějaké kooperace. S přechodem na distribuované zpracování dat založené na architektuře klient/server bylo dosaženo kooperace v rámci týmu (využití LAN sítě a centrálního serveru). Posledním stupněm vývoje je zpracování v rozsáhlých sítích, kde se využívá dynamické kooperace virtuálních týmů. Více o těchto komunikačních modelech a architekturách v předmětu Architektury OS.

Trendy v oblasti rozhraní člověk - stroj

Grafické uživatelské rozhraní (zkráceně GUI) se také mění a vyvíjí. Existují dokonce obory a disciplíny[2], které se zabývají vzhledem a hlavně použitelností GUI, využitím symbolů – např. koš jako místo pro vymazané soubory, apod. Uživatelské rozhraní různých aplikací se sjednotilo. Například textový editor využívá v hlavním menu nabídek jako Soubor, Úpravy, Nástroje, …, Nápověda. Ty samé nebo podobné nabídky najdeme také u grafických editorů, webových prohlížečů, ale také u RAD, CAD nástrojů nebo prostředí pro práci s mapami. Rozšířilo se také multimediální uživatelské rozhraní a multimédia vůbec. Můžeme si všimnout velké podpory multimédií u ZSW, konkrétně u operačních systémů. Za všechny můžeme jmenovat Linux či Windows 7, jež obsahují spoustu programků a utilit pro prohlížení fotek, přehrávání videa a hudby a usnadnění práce s nimi. Nejen multimédia, ale i virtuální realita má své využití v praxi, například u prezentace zboží. Rozvoj simulace přinesl možnost simulování převážně nevratných procesů ve zdravotnictví, armádě nebo automobilismu a stavebnictví. Z rozvojem multimédií a virtuální reality souvisí také rozvoj vzdělávacího, zábavního a filmového průmyslu (viz Youtube, Facebook apod.).

Trendy v oblasti metod a nástrojů vývoje IS/IT

Také v této oblasti proběhla standardizace. Ta se týká také tvorby a realizace informační strategie organizací a podniků. Vývoj částí IS se provádí jedním projektem, tento projekt je řízen, přičemž práce na projektu jsou standardizovány, standardní jsou také postupy implementace a údržby ASW (různé instalátory). Probíhá odklon od klasického sekvenčního vývoje IS/IT (etapy: specifikace požadavků, analýza, návrh, implementace a zavedení) a přechází se k iterativně inkrementálnímu (agilnímu) vývoji IS. Podobně se odchází od inženýrských přístupů, které abstrahovaly od vlivu lidského faktoru

[2] Oborem, který to vše zastřešuje je HCI – Human Computer Interaction.

na efektivnost užití IS. IS byl mnohdy implementován dokonale, ale lidem nesloužil. Trendy jsou následující:

- Podpora vícedimenzionálních metod vývoje IS.
- Posun od strukturovaného k objektovému přístupu (metodiky OMT, Booch, OOSE, UP a jeho komerční varianta RUP, OpenUP). Objektový přístup přináší přehlednost pro tvůrce i uživatele, jelikož více respektuje realitu.
- Vznik a další rozvoj unifikovaného modelovacího jazyka UML („The Three Amigos" Jacobson, Booch, Rambaugh), umožňující srozumitelný pohled na systém i pro běžného uživatele.
- Podpora tvorby softwarových systémů pomocí CASE nástrojů.

Trendy v organizaci a řízení IS/IT:

V organizacích se velmi významně využívá outsourcingu vývoje aplikací a také provozu HW, SW, resp. IT služeb. Využívá se také outsourcingu programátorských týmů, ať z důvodu nedostatečné kapacity organizace, nebo z důvodů spolupráce s kvalitnějšími odborníky. Důvodů k outsourcingu je několik. Provoz IS/IT vlastními silami stojí čas a peníze a ubírá podniku energii, kterou by jinak mohl věnovat vlastní oblasti činnosti a zájmu. Outsourcing je strategický organizační nástroj. Probíhá přesun odpovědnosti za provoz funkční oblasti (činnosti) podniku na externí specializovanou firmu – poskytovatele, zpravidla včetně zaměstnanců a vlastnictví aktiv. Účelem je především zaměření se na hlavní činnost, dosažení světové úrovně kvality v oblasti, případně úspory nákladů. Aplikuje se u oblastí, které nejsou hlavními činnostmi podniku, tedy nejsou motorem dlouhodobé konkurenceschopnosti podniku.

Outsourcing se používá i v oblasti podnikových informačních systémů. Zde je jeho aplikace složitá, protože informační systémy jsou s chodem podniku provázány a je obtížné specifikovat rozhraní (služby) mezi podnikem a poskytovatelem. Outsourcing se rozvinul hlavně ve dvou verzích:

- ***Near-shoring:*** geografická i kulturní blízkost týmu, stejné časové pásmo nebo pouze malý rozdíl, podobné kulturní aspekty, například zákazník v západních zemích jako je Švédsko, Německo, Velká Británie a výojové/provozní týmy v Česku, Rusku, Ukrajině, Polsku. Tato pracovní síla je o dost levnější než v zemi zákazníka, ale stále má podobnou mentalitu a těží z geografické blízkosti (doba letu například jen 2 hodiny).
- ***Off-shoring:*** vzdálenější varianta outsourcingu a to jak geograficky, tak mentálně, kulturně. Jedná se o varianty zákazník v USA, GB, Kanadě, Německu a vývojový/provozní tým v Indii, Číně, Mexiku, Filipínách, Brazílii. Komunikace je v takovém nastavení složitější nejen díky malému nebo žádnému časovému překryvu, ale také díky odlišné mentalitě a kultuře. Cenový model pak v praxi může být téměř totožný s nearshoringem, jelikož ještě nižší cenu pracovní síly v Indii či Číně dorovnává nutnost častějších vzájemných návštěv.

Pro vývoj a další rozvoj informačního systému organizace se využívá tzv. systémové integrace. Systémová integrace je proces, jehož cílem je vytvoření a

další rozvoj komplexního a taktéž zajištění bezpečnosti IS) a integrovaného informačního systému organizace. Tohoto cíle se dosahuje optimální kombinací a integrací vhodných hardwarových a softwarových komponent a informatických služeb. Firma nebo organizace, která provádí systémovou integraci se nazývá systémový integrátor. Systémový integrátor je zákazníkem na základě smlouvy pověřen komplexním řešením IS zákazníka. Zodpovídá za kompletní a kvalitní řešení integrovaného IS. K zajištění dodávek může systémový integrátor uzavírat smlouvy s jinými dodavateli a řešiteli.

Trendy v oblasti technologií podle společnosti Gartner

Společnost Gartner publikuje každý rok tzv. křivku humbuku, podle které vizualizuje dospělost, ale také popularitu nastupujících či medializovaných technologií. Podle Gartner se tedy v následujících letech blíží stavu dospělosti a běžného použití především následující technologie:

- cloud computing,
- mobilní aplikace,
- multimediální tablety,
- sociální média a komunikace.

Více o těchto tématech a predikci se můžete dozvědět na webu společnosti Gartner[3].

2.2.2 Projektování IS

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 tradiční rigorózní metodiky jako OMT, MDIS či agilní Extrémní programování, Scrum či OpenUP. *Metoda* nám říká, ***CO*** je třeba dělat v určité fázi nebo činnosti vývoje a provozu (např. objektový nebo relační model). 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 IS. Nástrojem jsou RAD systémy, CASE systémy, automatické testovací a buildovací nástroje, ale také tabule, tužka a papír.

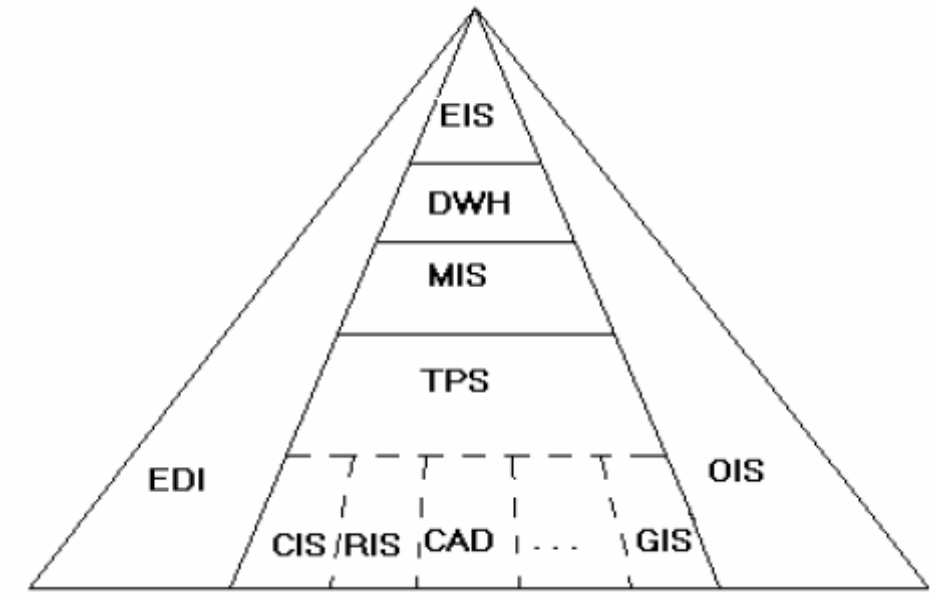
Definice říká, že metodika je doporučený souhrn principů, konceptů, dokumentů, metod, technik a nástrojů pro tvůrce softwarových (informačních) systémů, který pokrývá celý životní cyklus informačních systémů. ***Metodika určuje kdo, kdy, co, jak a proč má dělat během vývoje a provozu IS***. Metodika 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.

[3] http://www.gartner.com/it/page.jsp?id=1454221

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 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.

2.2.3 Architektury IS

Pojmy architektur jsme se zabývali v předmětu (X)SWENG. Jen pro zopakování uveďme, že ***globální architektura*** je hrubý návrh celého IS/IT. Je to vize budoucího stavu. Zachycuje 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 například aplikace typu EIS, DWH, MIS, TPS apod., viz následující obrázek:



Obr. 2-4: Příklad globální architektury podnikového systému.

Dílčí architektury se pak zabývají celkem nebo jen konkrétním blokem a popisují detailní návrh IS z hlediska různých dimenzí. Těmito dimenzemi IS/IT mohou být například:

- 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.

Na závěr ještě připomeneme, že pojmy architektura informačního systému a architektura software je 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 architektura SW je pouze podmnožina architektury informačního systému.

Více o architekturách informačních systémů lze nalézt například [Do97] a také v naších skriptech k předmětu (X)SWENG.

2.3 Problémy projektů

V této kapitole se stručně zmíníme o problémech spojených s vývojem software a o jejich příčinách, původech. Ukážeme výzkum Standish group za poslední roky, který ukazuje překvapivou skutečnost, která vysvětluje neúspěchy (nedodržení termínů, rozpočtů, interpretace požadavků, kvalita) vodopádového přístupu a jasně ukazuje, že problémem softwarových projektů není produktivita týmů a jejich členů. Podívejme se nyní blíže na to, co stojí v pozadí těchto neúspěchů.

Jelikož se systémy stávají složitějšími, rozsáhlejšími, jsou předmětem více změn, ať už ze strany zákazníka či z důvody změny zákonů, a je nutné se s nimi vypořádat. Problémy, se kterými se nejen z těchto důvodů běžně potýkáme, jsou následující:

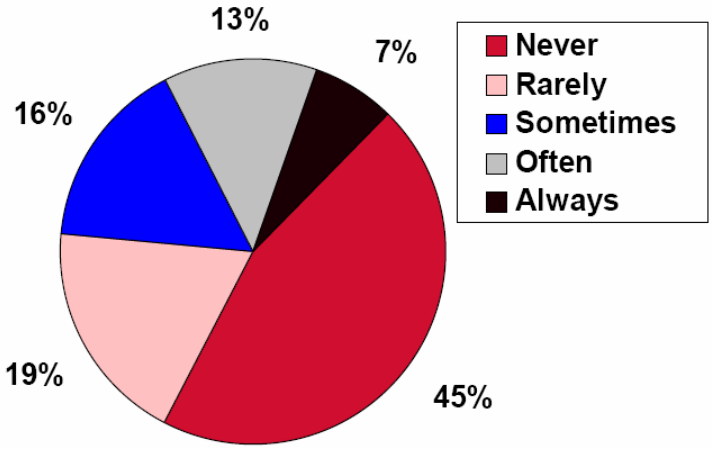
- Neporozumění klíčovým potřebám zákazníka (tzv. *core needs*).
- Neschopnost práce s měnícími se požadavky.
- Nemožnost či obtížná integrace SW modulů.
- Obtížná a drahá údržba či rozšíření SW systémů.
- Pozdní odhalení chyb, problémů.
- Špatná kvalita, výkonnost SW produktů.
- Problémy s buildy a releasy

Z různých důvodů, kterými je neznalost či ignorance vedoucích pracovníků (projektový manažeři, manažeři prodeje, ...), nedostatek času, tlaky zákazníka či pouhá neznalost či neochota použít nových postupů/metodik, jsou tyto problémy ještě prohlubovány, jelikož nejsou řešeny vlastní příčiny, ale jsou odstraňovány pouze následky problémů (viditelné symptomy) pomocí tzv. *workaroundů* (nestandardních řešení, obezliček). Opravdovou příčinou

zpoždění, chyb, větších nákladů a nespokojenosti zákazníka s výsledným produktem je většinou:

- Nevhodná, špatná specifikace požadavků.
- Nejasná a slabá komunikace (se zákazníkem, mezi členy týmu).
- Nevhodná či nedefinovaná architektura SW systému.
- Přílišná složitost systémů.
- Nekonzistence v požadavcích či v návrhu.
- Slabé a nedostatečné testování.
- Problém s riziky a neočekávanými událostmi.
- Nekontrolované a nestandardní změny.
- Nedostatečná automatizace.

Tyto problémy a hlavně jejich příčiny je třeba mít při vývoji software na paměti. Naštěstí existují praktiky a principy, které byly ověřeny na úspěšných projektech, a které když použijeme ve svých projektech, přináší zlepšení ve výše zmíněných problémových oblastech. Těm se budeme věnovat v jádru textu o iterativně-inkrementálních přístupech. Například problém specifikace požadavků, identifikace klíčových potřeb zákazníka byl předmětem výzkumu Standish group od roku 1998, viz např. [Sta02].



Obr. 2-5: Výzkum Standish Group: skutečné využití požadovaných funkčností SW systémů (zdroj: [Sta02]).

Graf ukazuje skutečnost, že pouhých 20% rysů softwarových aplikací je vždy nebo velmi často používáno. Právě těchto 20% jsou ony zmíněné klíčové potřeby zákazníka (*core needs*), ostatní nejsou pro zákazníka důležité. Pokud si uvědomíme, že se v průběhu projektu musíme věnovat i zbývajícím 80% požadavků, je nasnadě ptát se, zda je chyba v produktivitě vývojových týmů nebo v něčem jiném. Navíc je třeba si uvědomit, že těch 80% snahy analytiků, návrhářů, programátorů, testerů mohlo být soustředěno jinam (na větší kvalitu produktu, na jiné projekty) a také, že tuto práci zákazník „zbytečně“ platí!

Kontrolní otázky:

1. Co je to informační systém?
2. Co je to metodika?
3. Jaké jsou příčiny problémů projektů tvorby SW?

Shrnutí obsahu kapitoly

V této kapitole jsme stručně zopakovali pojmy z oblasti informačních systémů. Zmínili jsme problémy panující při vývoji software a také jejich příčiny. Součástí kapitoly bylo také shrnutí architektur IS.

3 Modely vývoje IS

V této kapitole se dozvíte:

- Jaké jsou základní modely vývoje IS.
- Výhody, nevýhody a specifika těchto modelů.
- Současné trendy a standardy v metodikách vývoje IS.

Po jejím prostudování byste měli být schopni:

- Popsat vodopádový, přírůstkový a iterativně inkrementální model.
- Vyjmenovat omezení těchto modelů.
- Nastínit obsah standardu ISO 12207.

Klíčová slova této kapitoly:

Model vývoje IS, vodopád, spirála, iterace.

Doba potřebná ke studiu: 3 hodiny

Průvodce studiem

Kapitola se věnuje představení základních modelů vývoje IS, na kterých jsou vystaveny konkrétní metodiky vývoje IS. Konkrétně se jedná o vodopádový, spirálový a iterativně inkrementální model (tím se budeme zabývat detailně v následujících kapitolách textu). Evoluce těchto modelů je pak představena včetně jejich omezení a dobového kontextu. V kapitole zmiňujeme také relevantní standard pro vývoj IS.
Na studium této části si vyhraďte 3 hodiny.

Následující kapitola se zabývá základními modely vývoje softwaru, respektive IS. Tyto modely mohou být používány různými metodikami a generalizují organizaci, posloupnost a počty vykonávání jednotlivých disciplín (plánování, analýza, návrh, programování, testování) v procesu vývoje IS. Základními modely jsou:

- Vodopádový neboli sekvenční,
- Spirálový,
- Iterativně-inkrementální
- Agilní jako specifická forma iterativně inkrementálního modelu.

Tyto modely si v následujícím textu popíšeme podrobněji a budeme se detailně zabývat iterativně-inkrementálním modelem, který je dnes již de facto standardem pro vývoj komplexních softwarových systémů, což jsou typicky právě informační systémy.

3.1 Vodopádový model

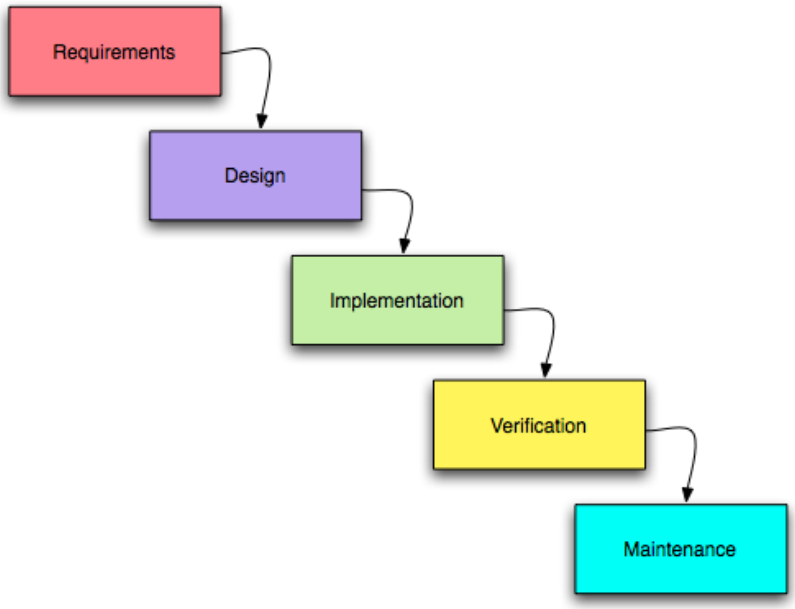
Hlavním poznávacím rysem tohoto modelu je sekvenční řazení disciplín vývoje SW s jediným průběhem modelu v rámci životního cyklu vývoje SW. ***Každá fáze vývoje softwaru tedy odpovídá právě jedné disciplíně.*** Typickými symptomy modelu jsou následující:

- Tvorba detailních plánů hned na úvod projektu, kdy ještě nevíme spoustu věcí a tudíž plán nepočítá 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. To trvá nějaký čas, ztrácí se tak spousta informací zachycených „mezi řádky“ a navíc mohou být tyto specifikace subjektivně nekorektně interpretovány. V menších projektech je problémem naopak nevhodná kombinace a kumulace rolí, kdy například jeden programátor vytváří analýzu i návrh určité funkčnosti. Tímto krokem redukujeme stohy dokumentace a čas nutný na její nastudování, ale výsledkem je řešení, které je technologické (vyhovuje nejlépe programátorovi) ale uživatelsky nepoužitelné. Také testování v těchto případech mnohdy provádí programátoři samotní a jelikož „ví, kam kliknout“, tak aplikace testy projde, ale zákazníkovi překvapivě neustále padá.

Vodopádový model:



Obr. 3-1: Vodopádový model vývoje SW.

Popis jednotlivých fází podle vodopádového modelu:

- Definice problému – pochopení záměru zákazníka, smyslu a účelu budoucího systému; identifikace všech zúčastněných na projektu a pochopení jejich potřeb; výstupem obvykle bývá úvodní studie.
- Analýza a specifikace požadavků (*Requirements*) – podrobné zkoumání a exaktní popis požadavků na systém; výstupem bývá specifikace požadavků (většinou velmi rozsáhlý dokument), kterou si dodavatel odsouhlasí se zákazníkem, z tohoto důvodu by měla být psána srozumitelným jazykem zákazníka.
- Návrh a architektura (*Design*) – na základě specifikace požadavků je navržena (pozor, ještě ne implementována) vhodná architektura

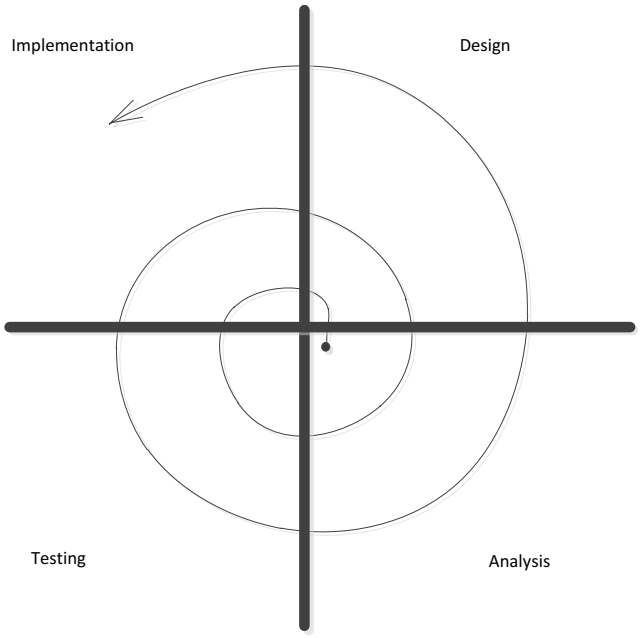
odpovídající potřebám, která je dále definována až na úroveň jednotlivých modulů a jejich závislostí, způsobu jejich komunikace či uložení a práce s daty. Součástí této fáze je také definitivní výběr technologií pro různé vrstvy budoucího systému, stejně jako vývojových nástrojů. Podle typu použitých metod a technik konkrétní metodiky jsou výstupem funkční modely datových toků, objektové modely tříd, ER diagramy, sekvenční diagramy apod.

- Implementace (*Implementation*) – jedná se vlastně „pouze“ o překlad návrhu do zdrojových kódů, v praxi je však spousta popisů nejasných a je možné, že je odhalena nesourodost v návrhu, což je třeba řešit celým tradičním kolečkem předchozích fází.
- Verifikace a testování (*Verification*) – ověření funkčnosti systému podle specifikace požadavků.
- Provoz a údržba (*Maintenance*) – instalace do provozního prostředí a neustálý proces úprav, vylepšování a ladění systému.

Vodopádový model byl odpovědí na první softwarovou krizi na konci šedesátých let. Vývoj softwaru se v té době stával komplikovanějším, jelikož se aplikace stávali komplexnějšími a již je nezvládal jeden programátor. Kvalita nebyla uspokojivá a rostla náročnost celé disciplíny nejen z pohledu programátorů (technologie a postupy), ale také z pohledu zákazníka (nutnost častější komunikace a formálnějších popisů potřeb). Tento model funguje a je vhodný pro menší a časově kratší projekty, kde je známá problémová doména i technologie a tudíž by neměly nastat velké překvapení a problémy, které logicky nemůžeme znát, když vytváříme počáteční detailní plán. Model není úplně vhodný pro komplexní projekty obsahující novější technologie, pokrývající neznámou doménu, vyžadující integrace na více dalších systémů a trvající delší dobu. V těchto případech model neposkytuje prostředky pro uchopení možných problémů.

3.2 Spirálový model

Dalším používaným modelem vývoje softwaru je tzv. *spirálový model*, který měl Barry Boehm na mysli, když představoval původní vodopádový model, ale byl pochopen jen z části. Spirálový model se snažil odstranit základní nedostatky vodopádového modelu tím, že v podstatě seřadil několik vodopádů za sebe, aby vývojáři získali zpětnou vazbu a aby se zaměřili na snížení rizik (což jsou v podstatě základní principy iterativně-inkrementálního modelu).



Obr. 3-2: Spirálový model vývoje SW.

Základem každého průchodu spirály je analýza rizik, prototypování a plánování dalšího průchodu spirály, které je následované provedením jednotlivých disciplín vývoje (analýza, návrh, …). Spirála samotná pak znázorňuje nejen průchod životním cyklem vývoje, ale také rostoucí časovou náročnost stejně jako náklady. Ve skutečnosti se ale ještě nejedná o iterativně inkrementální přístup, i když se zde objevují cykly a interakce se zákazníkem. Vlastní řešení je totiž uvolněno až na konci posledního cyklu a také tento model nepočítá se změnami. Prototypy mohou být jen v malé určité (například technologické) oblasti. Spirálový model tedy snižuje rizika a nepřesnosti v plánování, protože celý velký projekt je proveden v několika cyklech malých vodopádů s důrazem na rizika a prototypy, ale pořád nepojímá změny a neumožňuje nasazení prvních verzí.

Příklady (převážně vodopádových) metodik:

- Dodržování následujících metodik vyžaduje stát při státních zakázkách:
  - SSADM (*Structured Systems Analysis and Design Method*) – určena pro vývoj IS ve vládních úřadech a institucích státní správy ve Velké Britanii; nepokrývá celý životní cyklus IS (nepokrývá implementaci, zavádění a provozu) a základem metodiky a základním nástrojem je datový model a prověřování struktury dat vzhledem k požadovaným funkcím a výstupům
  - SDM (*System Development Metodology*) – Nizozemí; strukturovaná metodika uvažující vodopádový přístup k vývoji IS; metodika klade velký důraz na obecnost, tak, aby byla využitelná pro širokou třídu IS; věnuje se důkladně způsobům řízení projektů, zajišťování kvality vyvíjeného IS a zdůrazňuje souvislosti s organizační strukturou; základním nástrojem je prototypování.
- Mezinárodní:

    - Euromethod – EU; cílem této metodiky je nápomoci vzájemnému porozumění zákazníků a dodavatelů na mezinárodním trhu IS; uvažuje řízení smluvních vztahů mezi zákazníkem a dodavatelem IS „napříč“; je primárně zaměřena na provádění změn v IS, konkrétně na práce věcné i řídící.
- Firemní metodiky:
    - SE (*System Engineering*) – LBMS.
    - Oracle CASE Method.
- SIV – vyvinuta ve Stockholm *School of Economics* – specifická metodika vývoje IS pro customizaci typového aplikačního SW na základě přístupu analýzy aplikačního balíku z pohledu dodavatele (nabídka funkcí) a z pohledu zákazníka (skutečné potřeby).

Iterativně inkrementální model vývoje, který je řízen riziky a Use Casy, o němž se budeme bavit dále v textu, se snaží odstranit problémy předchozích modelů nejen identifikací a snahou o odstranění rizik, ale také 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. Iterativně inkrementální model tedy není jen jiným druhem pevně daného procesu, ale spíše způsobem myšlení a souborem principů, přičemž v každém projektu můžeme klást důraz na jiný princip. Hlavní rozdíly vodopádového a iterativně inkrementálního jsou shrnuty v následující tabulce:

Vodopádové principy	**Iterativně inkrementální (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 3-1: Vodopádový vs. Iterativně inkrementální přístup.

3.3 ISO/IEC 12207

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.

Procesy se dělí na aktivity, které se dále skládají z jednotlivých úkolů. Primární proces Údržba (5.5 Maintenance process) popisuje aktivity a úkoly, které vykonává podpora ve fázi údržby. Tento proces je aktivován v okamžiku, kdy systém podstoupí modifikaci kódu a týkajících se dokumentů z důvodu chyby, slabé výkonnosti, problému nebo potřeby vylepšení či přizpůsobení. Cílem procesu je upravit stávající systém při zachování jeho integrity. Kdykoliv je třeba existující software modifikovat, je vyvolán tento proces. Samozřejmě, že tento proces využívá k jejich provedení další procesy standardu ISO/IEC 12207, např. proces vývoje (5.3 Development process). Popis dalších procesů vypustíme (např. Management, Verification, …).

Stručně nastíníme, jakým způsobem je proces definován. Například součástí procesu údržba jsou následující tři aktivity:

- Analýza problému a modifikace (5.5.2 Problem and modification analysis) – obsahuje úkoly analýza požadavků na vylepšení a jejich dopad (typ, cena, potřebný čas, zda je kritický apod.), návrh možných řešení, dokumentace všech těchto výsledků a rozhodnutí, schválení vybraného řešení modifikace.
- Implementace změny (5.5.3 Modification implementation) – vývojář údržby se řídí analýzou a zjišťuje, které dokumenty a které části a verze SW mají být dokumentovány, poté spouští proces vývoje (musí být zajištěno, že se nezmění původní implementace a zároveň je zajištěna kompletní a korektní implementace nových požadavků).
- Revize/ schválení údržby (5.5.4 Maintenance review/acceptance) – skládá se ze dvou kroků: přezkoumání a autorizace změny z důvodu zajištění integrity systému, schválení implementované změny dle podmínek v kontraktu.

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.

Kontrolní otázky:

1. Vyjmenujte základní rozdíly mezi vodopádem a iterativně inkrementálním modelem.
2. Proč není spirálový model ještě iterativně inkrementální? Co mu chybí?
3. Co to je a co popisuje ISO 12207?

Úkoly k zamyšlení:

V kapitole byl nastíněn základní rozdíl mezi iterativně-inkrementálním 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 iterativně inkrementálním způsobu vývoje netvoříme na začátku detailní plán celého projektu, ale pouze jakousi *roadmap*. 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? Jaký je tedy hlavní rozdíl oproti detailnímu plánu ve vodopádovém modelu?

Shrnutí obsahu kapitoly

V této kapitole jsme se věnovali modelům vývoje IS, na kterých jsou vystaveny konkrétní metodiky vývoje IS. Konkrétně se jednalo o vodopádový, spirálový a iterativně inkrementální model (tím se budeme zabývat detailně v následujících kapitolách textu). Evoluce těchto modelů byla představena včetně jejich omezení a dobového kontextu. V kapitole jsme zmínili také relevantní standard pro vývoj IS, jímž je ISO 12207.

4 Principy iterativního vývoje

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 iterativně-inkrementálního vývoje řízeného riziky a use casy.

Klíčová slova této kapitoly:

Principy RUP, komunikace, motivace, iterativně inkrementální vývoj, řízení rizik, use case, kvalita.

Doba potřebná ke studiu: 4 hodiny

Průvodce studiem

Kapitola zmiňuje principy iterativně inkrementálního, riziky a use casy řízeného vývoje a podrobně se věnuje jejich popisu, včetně představení výhod, vzoru, jak princip aplikovat a anti-vzoru, který říká, čemu se vyvarovat.
Na studium této části si vyhraďte 4 hodiny.

Iterativně-inkrementální 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 (kde jsou částečně sloučeny a mírně přeformulovány), 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*).

Nyní si jednotlivé principy A až F představíme jeden po druhém. Jejich struktura je definována přínosy (*benefit*) použití daného principu ve vývoji software, dále vzorem (*pattern*), který zachycuje zkušenosti z úspěšných projektů, tzv. Best practices a anti-vzorem (*anti-pattern*), který popisuje nevhodné praktiky, které vedou k výše zmíněným problémům při vývoji SW, a tedy říká, jak věci nedělat. Pro demonstraci použijeme principy RUP, jelikož OpenUP principy nejsou pro začátečníky tolik přehledné. Bližší popis těchto principů lze nalézt v [Kr05], [RE] či přímo ve webové aplikaci RUP.

4.1 Adapt the process

Přínos (benefit): Větší efektivnost životního cyklu, lepší komunikace rizik.
Vzor (pattern): Preciznost a formálnost se vyvíjí od nízké po vysokou v průběhu životního cyklu tak, jak jsou odstraněny nejasnosti. Přizpůsobte proces velikosti a distribuci projektového týmu, složitosti aplikace. Neustále vylepšujte Váš proces.
Anti-vzor (anti-pattern): Precizní plány, odhady a postupování podle nich. Mít více procesu (artefaktů, aktivit, rolí) je vždy lepší. Vždy v průběhu životního cyklu udržujte stejnou úroveň formálnosti a preciznosti.

Pojmem více procesu máme na mysli tvorbu spousty artefaktů, detailní dokumentace není to nejdůležitější. Důležitější, než toto množství artefaktů či revizí, je správně nastavit proces potřebám projektu. Proces by měl být více formální, disciplinovaný, v případě distribuovaného vývoje, použití komplexní technologie či při jeho větší složitosti. V opačném případě, kdy je známá technologie, tým je malý a sedí v jedné místnosti, by měl být proces jednodušší, lehčí (s menším množstvím formalit, artefaktů a detailních specifikací).

Dále je třeba držet v každé fázi projektu jiný stupeň formálnosti. V úvodních fázích existuje spousta nejasností a rizik a potřebujeme velké množství kreativity, formální proces by nás svazoval. Naopak v pozdějších fázích, kdy jsou významná rizika odstraněna a vývoj je poměrně předvídatelný, je nutné proces více svázat, aby se vývojáři nezabývali nepodstatnými věcmi. Také je třeba mít pod kontrolou změny (je třeba proces či disciplína který bude sloužit pro potřeby změnového řízení).

Nepřetržité zlepšování procesu je zaručeno zhodnocením na konci každé iterace, formulováním ponaučení (tzv. *lessons learnt*) a jejich promítnutí do následných prací a plánů. Z toho také vychází poslední bod tohoto principu, jímž je tvorba a úprava plánů. Na začátku, kdy je míra rizika a neznalosti vysoká, vytváříme pouze jakousi roadmap, velmi hrubý plán. Kratší časové úseky – iterace – plánujeme podrobněji, až když známe více detailů a jsme schopni kratší úseky přesněji odhadovat.

4.2 Balance competing stakeholder priorities

Přínos (benefit): Vývoj aplikace podle potřeb uživatelů, redukce vývoje na zakázku, optimalizace přínosů pro byznys.
Vzor (pattern): Definuj a porozuměj podnikovým procesům a potřebám uživatelů; snaž se porozumět, které komponenty můžeš znovupoužít.

Anti-vzor (anti-pattern): Zachyť precizně veškeré požadavky předtím, než začnou veškeré práce na projektu. Definuj požadavky tak, aby byl celý vývoj zákaznický (bez znovupoužitých komponent).

Jádrem tohoto principu jsou dvě roviny:

- Identifikace klíčových potřeb/požadavků uživatelů.
- Zakázkový vývoj versus znovupoužití existujících komponent.

Většina uživatelů chce od budoucí aplikace přesně takové chování, jaké chtějí oni. Použitím komponent jsme však schopni radikálně snížit cenu vývoje a zkrátit jeho dobu, i když může být chování komponenty mírně odlišné od očekávání uživatelů. Pokud uživatelé či zákazník trvají na všech rysech a vývoji aplikace na zelené louce, pak musí počítat s delší dobou vývoje a také s vyšší cenou.

Jelikož uživatel řídí vývoj a našim cílem je vyvinout aplikaci, která mu přinese užitek, je nutné pochopit problémy a potřeby, které chce zákazník aplikací řešit. Díky pochopení jeho potřeb jsme schopni požadavky prioritizovat a také na základě těch prioritních definovat architekturu.

Anti-vzor říká, že máme precizně a detailně specifikovat požadavky předem, nechat je schválit zadavatelem a pak vyjednávat jejich každou změnu. Další anti-vzor říká, že máme brát v úvahu pouze požadavky nejhlasitějších uživatelů.

4.3 Collaborate across teams

Přínos (benefit): Produktivita týmu, lepší spojení mezi byznysem, vývojem a provozem SW.
Vzor (pattern): Motivuj lidi, aby pracovali co nejlépe. Upevni spolupráci mezi jednotlivými funkčními celky; analytik, vývojář a tester pracují dohromady. Ujisti se zda byznys, vývojové a provozní týmy pracují efektivně jako integrovaný celek.
Anti-vzor (anti-pattern): Vychovávej heroické jednotlivce a vybav je mocnými nástroji.

Software je produkován talentovanými a motivovanými lidmi, kteří úzce spolupracují. Při vývoji složitých projektů je třeba spousty lidí s různými dovednostmi, proto je komunikace nezbytným a kritickým faktorem a proto také mluvíme o tzv. „*soft skills*“. Tyto dovednosti jsou také základem agilního vývoje a proto jsou silně zdůrazněny v Agilním manifestu [AgM].

Prvním krokem pro efektivní spolupráci je vytvořit tzv. samo-řízené týmy (*self-managed teams*). Takový tým seznámíme s našim cílem, co chceme zákazníkovi doručit a poté mu dáme také odpovědnost, aby mohl rozhodovat o věcech, které konkrétně ovlivňují celkový výsledek. Pokud lidé cítí opravdovou odpovědnost za výsledek, cítí se více motivováni, než v případě, kdy jim úkoly přiděluje nadřízený, navíc bez celkového obrazu.

Jak již bylo řečeno, vývoj software je týmový sport. Iterativní vývoj zdůrazňuje nutnost těsné a neustálé spolupráce jednotlivých rolí / členů týmu. Je třeba zbořit zdi, které jsou po vzoru klasických metodik vystavěny mezi analytiky, vývojáři a testery. Tito lidé spolu pracují v těsném kontaktu v průběhu celého projektu (v průběhu všech fází a jejich iterací). Navíc každý z členů týmu musí znát a rozumět cílům a vizi projektu.

Anti-vzor říká, že máme vychovávat heroické jedince schopné pracovat extrémně dlouho každý den, včetně víkendů. Snažit se mít specialisty pro každou oblast, vybavené mocnými nástroji, kteří pracují odděleně a jejichž nástroje nejsou vzájemně integrované. Předpokladem tohoto anti-vzoru je, že pokud každý dělá svou práci, výsledek se dostaví a bude dobrý.

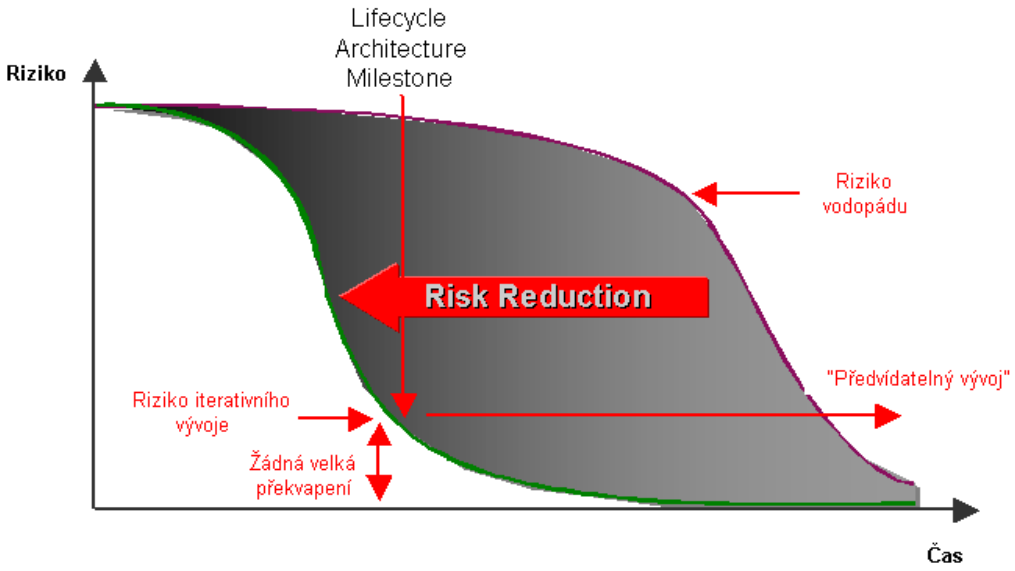
4.4 Demonstrate value iteratively

Přínos (benefit): Brzké zmírnění rizik, vyšší předvídatelnost vývoje, důvěra mezi všemi účastníky projektu.
Vzor (pattern): Adaptivní management s použitím iterativního vývoje. Atakuj významná technická, byznys a programátorská rizika co nejdříve. Získej zpětnou vazbu od uživatele tím, že v každé iteraci doručíš nějakou hodnotu.
Anti-vzor (anti-pattern): Naplánuj detailně celý životní cyklus, sleduj změny proti tomuto plánu. Detailní plány jsou lepší plány. Posuzuj status projektu revizí specifikací.

Pod tímto principem se skrývá několik bodů. Prvním z nich je, proč chceme doručovat inkrementálně nějakou hodnotu zákazníkovi – proto, abychom od něj brzy dostali zpětnou vazbu. Toho dosáhneme rozdělením projektu do několika iterací (vždy ale proběhne společně analýza, návrh, implementace, integrace, testování), kterými se přibližujeme výsledné podobě produktu. Díky této zpětné vazbě od zákazníka a uživatelů jsme schopni zjistit, zda se ubíráme správným směrem, zda je třeba změnit některé rysy nebo si zákazník uvědomí zapomenutý rys a chce ho přednostně zařadit místo jiného. Tímto nejen doručujeme zákazníkovi hodnotu, ale také budujeme vzájemnou důvěru tím, že zákazník vidí, zda aplikace opravdu řeší jeho problém, vyhovuje jeho potřebám. Nepřidáváme spoustu rysů podle našeho mínění, které uživatel nikdy nepoužije a přitom nám přidávají spoustu práce.

Druhým bodem tohoto principu je úprava plánů podle zpětné vazby od uživatele. Zaměřujeme se na spustitelný a testovatelný kód, který ukazujeme zákazníkovi, místo nepodstatného hodnocení specifikací. Tímto jsme schopni zjistit, jak moc či málo naplňuje aplikace představy zákazníka a podle toho aktualizovat celkové plány projektu a definovat detailní plán pro příští iteraci.

Třetím bodem principu je zahrnutí a řízení změny. Jelikož je změna v požadavcích v dnešních podmínkách trhu nevyhnutelná, je třeba s ní v projektech (procesu) počítat. Iterativní přístup nám poskytuje způsob, jak inkrementálně implementovat změny. Pro efektivní správu, řízení a implementaci změn je nutné mít definovaný proces a využívat podpůrný nástroj (např. JIRA).



Obr. 4-1: Úroveň rizik pro různé přístupy vývoje (zdroj: RUP)

Posledním, čtvrtým bodem principu je brzké snížení, odstranění rizik projektu (viz Obr. 4-1). Je nutné se věnovat snížení významných technických, byznys a programátorských rizik tak brzy, jak je možné. Tento přístup je výhodnější než odsunutí rizik na konec projektu. Jak víme, oprava chyby či implementace změny je 100-1000krát dražší v případě provozované aplikace, než v raných fázích vývoje. Právě nutnost opravy či implementace nového požadavku může být způsobena špatným odhadem rizik. Proto se snažíme neustále vyhodnocovat různá rizika a snažíme se je v následující iteraci snížit.

Anti-vzorem, dříve běžně používanou softwarovou praktikou, která přispívala k selhání projektů, je detailně plánovat celý životní cyklus předem, kdy je spousta nejasností a rizik v projektu, které určitě nastanou. Dalším anti-vzorem je posuzovat stav projektu na základě revizí a hodnocení specifikací, než na základě demonstrace fungujícího software (implementovaného a otestovaného v několika iteracích).

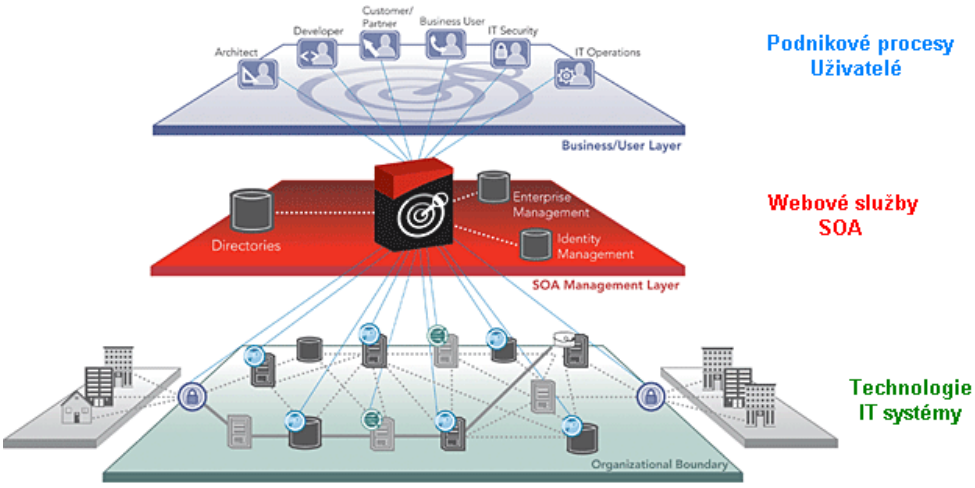
4.5 Elevate the level of abstraction

Přínos (benefit): Produktivita, nižší komplexnost.
Vzor (pattern): Znovupoužij existující komponenty, redukuj objem ruční práce použitím nástrojů a jazyků vyšší úrovně, navrhuj pružnou, kvalitní a srozumitelnou architekturu.
Anti-vzor (anti-pattern): Jdi přímo od vágních, vysokoúrovňových požadavků uživatelů k psaní kódu celé aplikace.

Možná jedním z největších problémů dnešního vývoje software, je jeho přílišná složitost. Snížení této složitosti má velký dopad na produktivitu. Práce na vyšší úrovni abstrakce snižuje složitost a usnadňuje komunikaci.
Jedním z efektivních přístupů redukujících složitost je znovupoužitelnost existujících „komponent“, jedná se o existující komponenty, existující systémy, podnikové procesy, vzory či Open source software. Příkladem takového znovupoužití, které mělo velký vliv na celou oblast IT je middleware,

konkrétně databáze, web servery, portály, v posledních letech pak rozličný Open source.

Příklad:
Dalším příkladem z dneška jsou webové služby (WS – Web Services) a architektura SOA. S použitím WS jsme schopni znovupoužít existující aplikace, které vystavíme jako webovou službu nebo pouze rozličné komponenty, které již jako webová služba existují. Potom je možné pomocí katalogu služeb UDDI skládat výslednou aplikaci jako z kostek (jednotlivých funkčností), bez znalosti implementace komponenty/aplikace.



Obr. 4-2: Příklad vyšší abstrakce pomocí webových služeb

Tento princip mluví také o snížení složitosti a zlepšené komunikaci díky použití nástrojů a jazyků vyšší úrovně. Standardní jazyky (např. UML) jsou schopny vyjádřit pojmy vyšší úrovně, jako jsou např. podnikové procesy, komponenty a jejich komunikaci, a naopak skrýt v tu chvíli nepodstatné detaily. Návrhové a implementační nástroje (CASE) mohou pomoci přejít z vyšší úrovně ke kódu nejen modelováním různých modelů, ale také automatickým generováním kódu či testů z těchto modelů.

Třetí bod tohoto principu hovoří o zaměření na architekturu. Jak si řekneme později, je našim cílem při vývoji software navrhnout, implementovat a otestovat architekturu produktu v brzkých fázích projektu. To znamená brzy definovat vysokoúrovňové bloky a nejdůležitější komponenty, jejich odpovědnosti a rozhraní, způsob komunikace, ukládání dat. Definicí architektury vytvoříme kostru systému, která definuje základní mechanismy. Tím usnadníme zvládnutí složitosti v projektu, když se přidá více vývojářů, vznikne více komponent, či napíší se tisíce řádků kódu. Samozřejmě ne pro všechno můžeme znovu použít existující části, je třeba také uvažovat nad tím, které části a jak budeme vyvíjet zákaznickým vývojem.

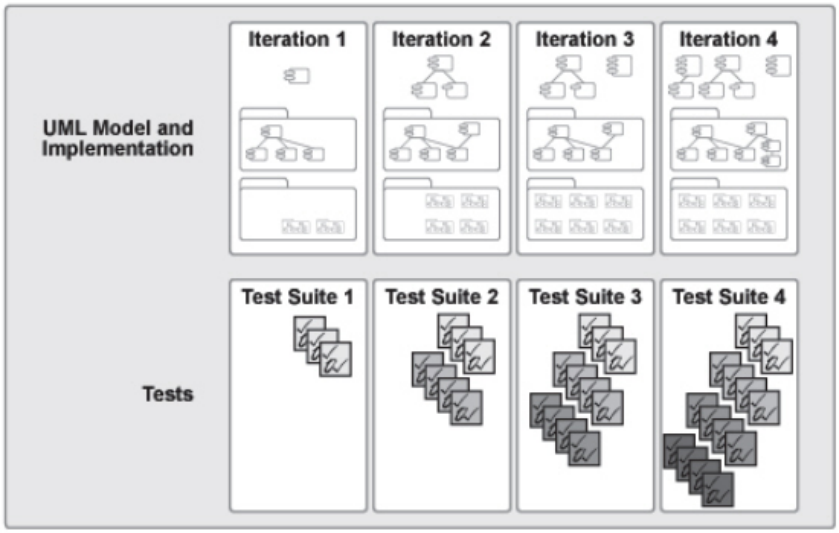
Anti-vzor říká, že máme jít přímo od vágních, vysokoúrovňových požadavků uživatelů k psaní kódu celé aplikace zákaznickým vývojem, bez tvorby abstraktních modelů, které nám umožní identifikovat místa, kde můžeme znovupoužít existující komponenty. Neformálně zachycené požadavky

vyžadují spoustu rozhodnutí a ověření pochopení v několika iteracích. Díky tomu, že se minimálně zaměřujeme na architekturu, jsme nuceni v budoucnu k rozsáhlému a neustálému přepracovávání aplikace.

4.6 Focus on quality

Přínos (benefit): Vyšší kvalita, rychlejší pokrok.
Vzor (pattern): Odpovědnost celého týmu za výsledný produkt. Testování a průběžná integrace se stávají prioritními. Inkrementálně zlepšuj testy a automatické testování.
Anti-vzor (anti-pattern): Posuň integrační testování až do fáze, kdy je celý produkt naprogramovaný a otestovaný unit testy. Prováděj revizi všech artefaktů raději než ověřování a testování částečně implementovaného řešení za účelem nalezení problémů.

Zajistit kvalitu produktu znamená mnohem více, než jen účast testovacího týmu na projektu. Zajistit kvalitu znamená, že každý člen týmu vlastní kvalitu, je za ni zodpovědný v průběhu celého životního cyklu. Analytik je zodpovědný za zajištění testovatelnosti požadavků, za jasnou specifikaci požadavků pro potřeby testů, které je třeba vykonat. Vývojáři by měli při návrhu aplikace mít na mysli její testovatelnost a jsou (musí být) odpovědní za testování vlastního kódu unit testy. Management zajišťuje vhodné testovací plány a zdroje pro tvorbu, spouštění a analyzování testů. Testeři jsou experti na kvalitu. Provádí zbytek týmu a vysvětlují hlavní body týkající se kvality software a jsou také odpovědní za funkční, systémové a výkonnostní testování.



Obr. 4-3: Testování, jako vlastní produkt, je v každé iteraci rozšiřováno (zdroj: RUP)

Jedním z největších přínosů iterativního vývoje je brzká a neustálá testovatelnost produktu. Jelikož nejdříve implementujeme důležité vlastnosti produktu a postupně přidáváme další méně důležité, je aplikace provozována s nejdůležitějšími rysy několik měsíců, a co je důležitější, je také několik měsíců průběžně testována (viz Obr. 4-3)! Proto je nejviditelnějším přínosem iterativního přístupu vyšší kvalita výsledného produktu.

Stejně jako inkrementálně budujeme výslednou aplikaci, tak bychom měli zvyšovat automatizaci testování. Cílem je nalézt chyby co nejdříve a minimalizovat námahu a náklady do toho vložené. V průběhu návrhu (design) aplikace je třeba uvažovat i nad její testovatelností, některá rozhodnutí v průběhu návrhu totiž mohou výrazně zvýšit možnost automatického testování. Navíc opět připomeneme automatické generování kódu z modelů, jenž může výrazně snížit počet chyb a defektů v kódu. Přístupem, který je populární v Agile komunitě je tzv. Test-first approach, kdy nejdříve píšeme testy a až poté vlastní řešení. Díky dřívějšímu psaní testů vlastně už rozmýšlíme vlastní návrh, který je pak kvalitnější a hlavně píšeme řešení, které je lépe testovatelné.

Anti-vzor tohoto principu zdůrazňuje detailní revize artefaktů (modely, specifikace), což je kontraproduktivní, jelikož nás zdržuje od testování spustitelné aplikace, kde jedině můžeme identifikovat závažné problémy. Druhý anti-vzor zdůrazňuje provést veškeré unit testy před integračním testováním, což opět způsobuje zpoždění v identifikaci závažných problémů.

Kontrolní otázky:

1. Co říká, co zahrnuje princip A?
2. Co říká, co zahrnuje princip B?
3. Co říká, co zahrnuje princip C?
4. Co říká, co zahrnuje princip D?
5. Co říká, co zahrnuje princip E?
6. Co říká, co zahrnuje princip F?
7. Jaká je struktura principů (3 části)?

Úkoly k zamyšlení:

Zmínili jsme 6 základních principů RUPu. Zamyslete se nad tím, jak byste jednotlivé principy implementovali ve vaší dennodenní práci programátora, analytika či projektového manažera. Myslíte si, že je možné tyto principy přijmout všechny najednou? Pokud ne, jaký časový rozsah by byl vhodný?

Korespondenční úkol:

Napište několik důvodů a tyto zdůvodněte, proč není vhodné vytvořit detailní plán pro celý životní cyklus projektu na úvod.

Shrnutí obsahu kapitoly

V této kapitole jste se seznámili se základními principy iterativně-inkrementálního vývoje podle RUP/OpenUP. Pro demonstraci jsme použili přehlednější principy RUP a některé byly pro lepší pochopení doplněny příklady. Principy říkají, jaká je výhoda jejich implementace a představují vzor a anti-vzor (typické chování vedoucí nás na scestí). Vzor říká, jakým způsobem princip implementovat, anti-vzor říká, čemu se naopak vyvarovat.

5 Fáze RUP/OpenUP

V této kapitole se dozvíte:

- Jaké jsou RUP/OpenUP fáze?
- Co je to iterace?
- Rozdíl mezi vodopádovou fází a fází RUP/OpenUP.

Po jejím prostudování byste měli být schopni:

- Pochopit strukturu RUP/OpenUP projektu.
- Pochopit životní cyklus RUP/OpenUP projektu.
- Porozumět náplni RUP/OpenUP fází.
- Znát milníky RUP/OpenUP fází.

Klíčová slova této kapitoly:

Fáze RUP/OpenUP, iterace, milníky.

Doba potřebná ke studiu: 4 hodin

Průvodce studiem

Kapitola vysvětluje rozdíly mezi fází a iterací, stejně jako mezi fází vodopádu a fází iterativního vývoje. Detailně je pak vysvětlena náplň každé fáze RUP/OpenUP, její cíle a milníky.
Na studium této části si vyhraďte 4 hodin.

V této kapitole se budeme věnovat náplni jednotlivých fází projektu podle RUP/OpenUP. Na úvod ukážeme rozdíl mezi klasickými fázemi vodopádu a fázemi v iterativně-inkrementálním 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. 5-1). 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:

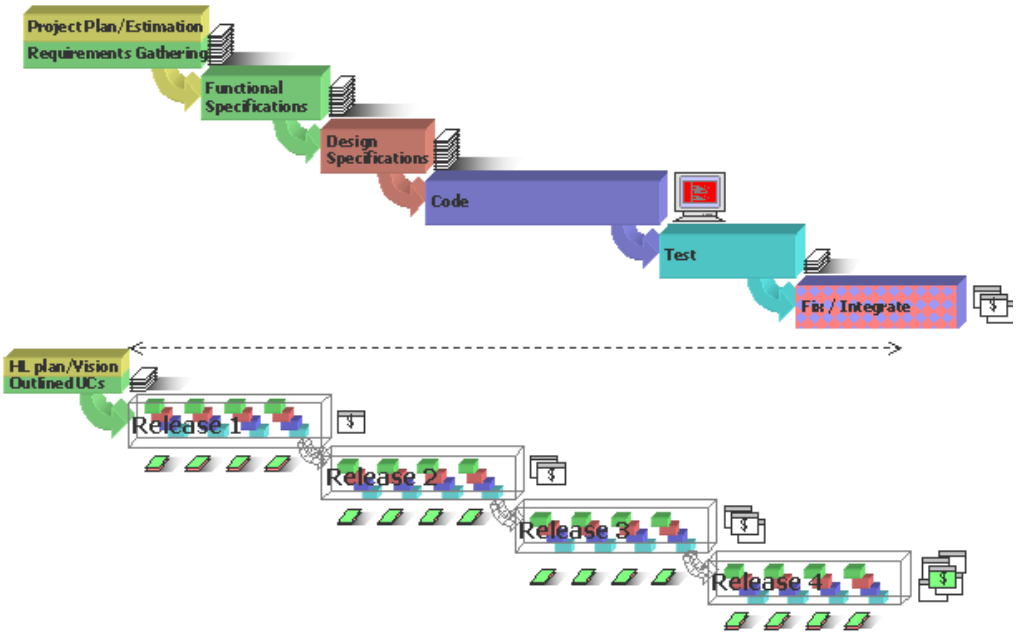
- Systém by měl mít toto ...
- Systém by měl mít tamto ...
- Systém by měl dělat toto ...
- Systém by měl splňovat standard ....
- Systém by měl umožnit ...

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 – s 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 Obr. 2-5 (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. 5-1: 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.

5.1 Iterace

Jak je patrné z Obr. 5-1, mezi fázemi RUP/OpenUP, resp. iterativně-inkrementálního modelu a vodopádu, je zásadní rozdíl. Iterativně-

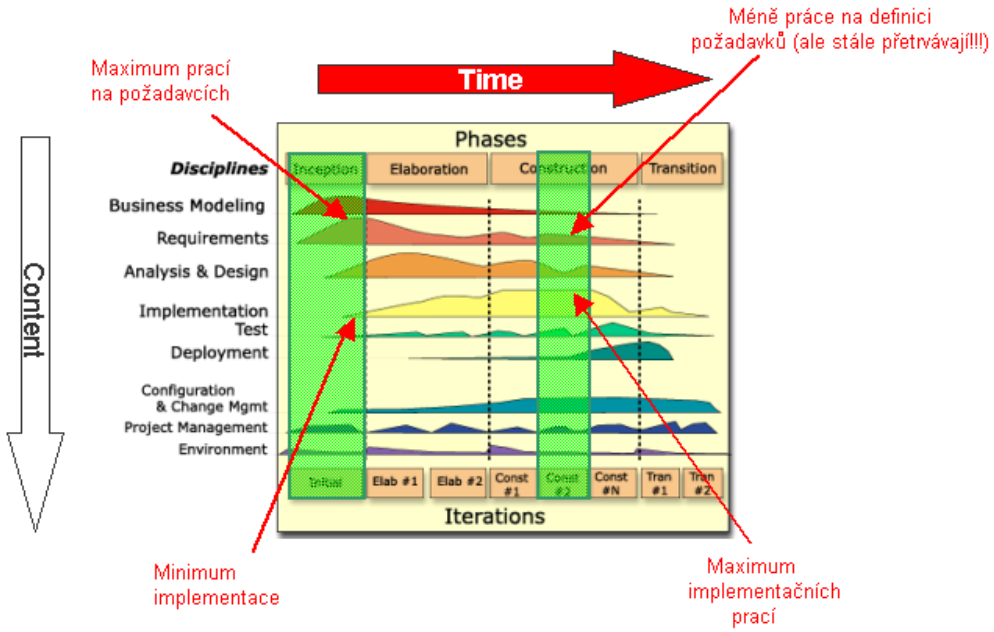
inkrementální vývoj probíhá v několika tzv. iteracích (opakováních). Takových zásadních rozdílů je hned několik:

- 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 2-4 týdny) – 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! Nejen jednu jako ve vodopádovém modelu.
- 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. 5-2 jsou 2 z iterací naznačeny zeleně.

5.2 RUP/OpenUP fáze

Jak již bylo naznačeno výše, RUP/OpenUP 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. Iterativně inkrementální projekt není jen sadou zřetězených iterací. Fáze seskupují iterace do určitých celků, přičemž každý takový celek (fáze) má jiný cíl, smysl. Například smyslem iterací v Inception fázi je pochopit cíl navrhovaného systému, jeho základní potřeby a use casy a definovat cestu (rizika, architekturu a základní milníky-plán) k dosažení systému. Cílem iterací v Construction je pak vyvinout zbývající use case aplikace ke spokojenosti uživatelů. 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. Každá fáze vodopádového projektu provádí pouze jednu disciplínu, kdežto fáze iterativně-inkrementálního modelu seskupují iterace v nichž provádíme vždy všechny disciplíny vývoje IS.

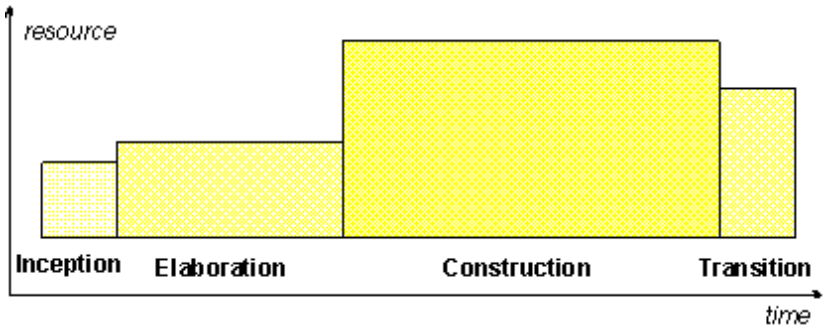


Obr. 5-2: Dvojrozměrný model RUP - fáze a disciplíny

Výsledkem Inception je pochopení problematiky, vize projektu, identifikovaná rizika a základní plán postupu včetně možné architektury. 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 také potřebného hardware, reklamní kampaně, médií. 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. 5-2)

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. 5-3: 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 5-1: Průměrné časy trvání jednotlivých fází RUP (zdroj: RUP)

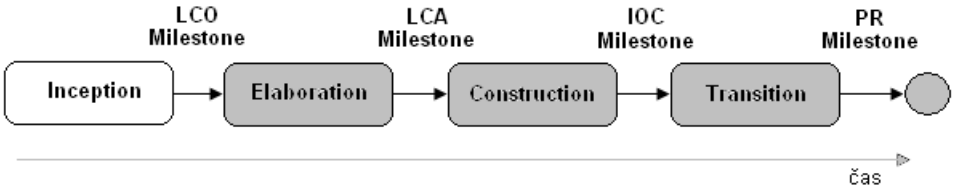
Elaboration obsahuje o málo více zdrojů a časově je její trvání zhruba 30% celkového času. Z obrázku i tabulky je patrné, že nejvíce času a nejvíce zdrojů vyčerpá Construction, kdy je v několika iteracích implementován výsledný produkt. Časově to znamená přibližně 50% celkového času projektu. Transition již pak zabírá 10% času, ale zdrojů je využíváno pořád hodně, jelikož dodělávám 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.

5.3 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. 5-4: 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.

Pochopení potřeb uživatele je často největším problémem softwarových projektů (viz graf Standish –Obr. 2-5). Každý ze zúčastněných na projektu pod nějakým cílem může chápat něco jiného. Něco jiného chápe manažer, něco jiného člověk, který danou práci vykonává denně, něco jiného pak analytik, který nemusí detailně znát podnikové procesy dané organizace. Správné pochopení potřeb uživatele, zákazníka, je náplní analytika. Výsledkem této práce by měla být vize. 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 (šablona vize z OpenUP [OUP]):

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.
Souhrn vize 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 - jednotlivý vývojáři - pracovníci administrativy - projektový manažer
Use Case - Změř čas aktivity - Sesbírej týdenní data - Sluč / konsoliduj data pro každý projekt - Nastav nástroj a databázi pro projekt

Tabulka 5-2: Příklad vize

Dalším bodem Inception fáze je identifikace kritických Use Casů (většinou se jedná o 20 – 30% všech Use Casů). Kritické jsou z důvodu technologického

(tvoří kostru aplikace – rozhraní) nebo z důvodu nutných funkčností, které musí aplikace obsahovat. Kritické UC mají významný dopad na architekturu systému, jinými slovy ji tvoří. Analytiky je vytvořen jejich detailnější popis, je možné posunout zpodrobnění některých alternativních toků těchto UC do dalších fází, pokud nemají významný dopad na architekturu. Poté, co jsou identifikovány, jsou kritické UC představeny zbytku týmu a vysvětleny, proč jsou kritické. Zbytek týmu by s nimi potom měl souhlasit.

Kritické UC systému jsou potom vyjmenovány v SAD – Software Architecture Document.

Cílem Inception fáze je 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.

Na konci Inception bychom měli být obeznámeni s riziky, kterým budeme vystaveni dále v projektu, hlavně se jedná o rizika spojená s vybranou technologií či znovupoužitými komponentami.

Pro celý projekt je kritické pochopení toho, co chceme vytvořit. Stejně tak kritické je ale také vědět, jak toho dosáhnout a s jakými náklady. 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):

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

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

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

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 prvních iteracích bychom měli také nastínit proces a prostředí, v druhé iteraci nasadit a upravit podle okamžité zpětné vazby ode všech zúčastněných a poté samozřejmě stále iterativně vylepšovat a upravovat. Pokud jsme vybrali a

nastavili proces, můžeme se věnovat výběru nástrojů. Pokud pro tuto oblast existují firemní standardy, měli bychom je samozřejmě vzít v potaz. Pokud ne, uvažujeme následující nástroje:

- IDE – integrované vývojové prostředí (př. Eclipse, Visual Studio).
- Nástroj pro správu požadavků (př. Rational Requisite Pro, Jira).
- Vizuální modelovací nástroj (př. Magic Draw, Borland Together)
- Nástroje pro správu konfigurací a změn – Configuration and Change Management Tools (CVS, SVN, Jira).

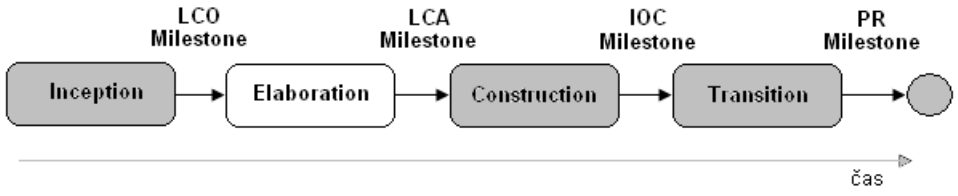
5.3.1 Milník LCO

Na konci Inception fáze následuje první milník projektu nazýván Lifecycle Objective Milestone (LCO). 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. LCO 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.

5.4 Elaboration phase

Po úspěšném dokončení první fáze 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. 5-5: Fáze Elaboration

Pokud jsme dosáhli prvního milníku (Lifecycle Objective Milestone – LCO), 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. Architekturou tedy myslíme implementaci základních kritických use case, ne instalaci aplikačního serveru a databáze! 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?)

Většinu rizik snížíme tím, že v této fázi ***vyprodukujeme spustitelnou architekturu našeho výsledného řešení (tj. implementujeme kritické scénáře, ne že instalujeme či nastavíme prostředí)***. 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*** (například jednoho use case), 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 (***opět na konkrétních scénářích/use case***).
- 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.

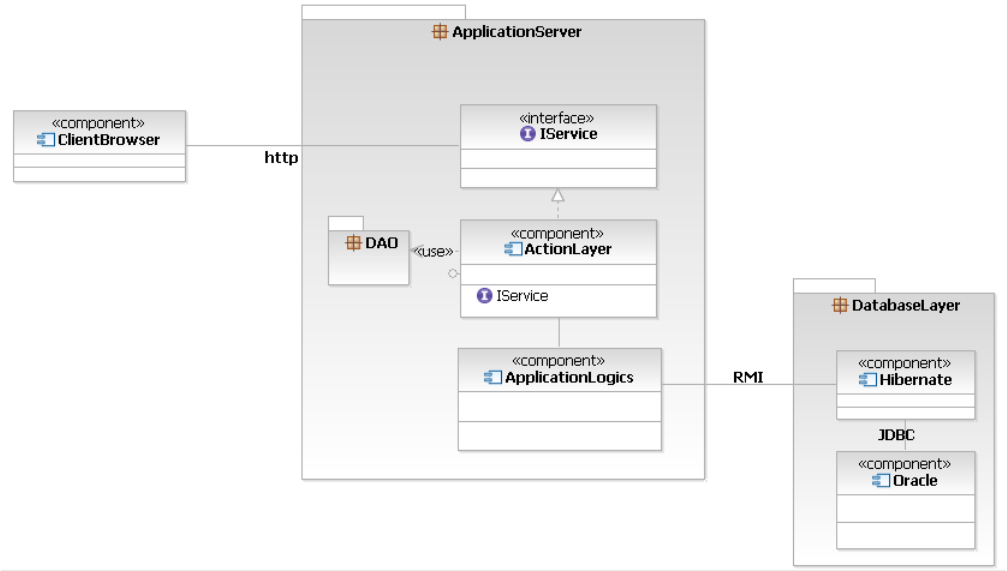
- Návrh a implementace první verze databáze.
- Detailní popis zbývající poloviny Use Casů, které chceme blíže specifikovat v Elaboration.
- Testování, ověření a úpravy architektury do její stabilní verze, na ni pak můžeme dále navěsit další funkčnosti systému.

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).

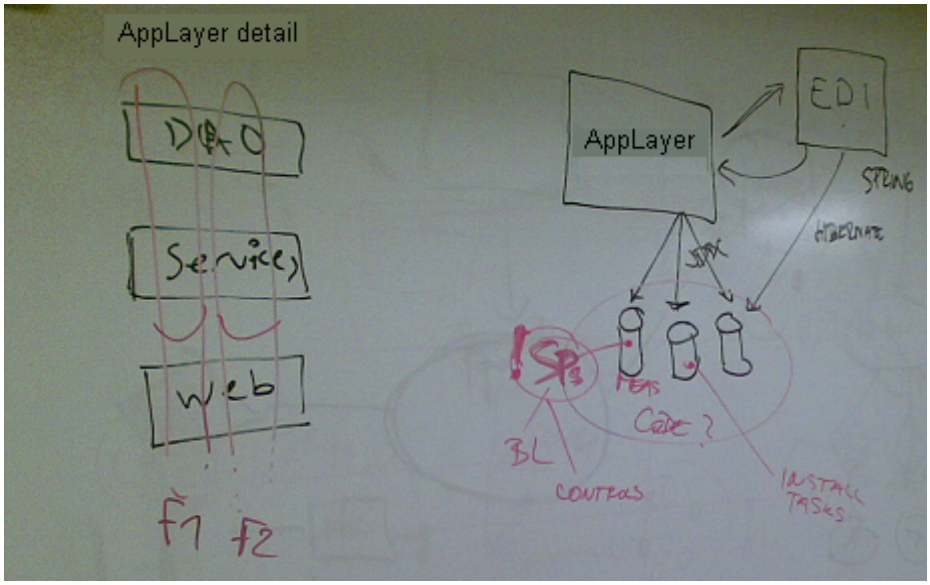
Při tvorbě architektury uvažujeme následující:

- 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).

Následující dva příklady ukazují možnosti popisu architektury více a méně formálním způsobem:



Obr. 5-6: Varianta Klient-server (C/S) softwarové architektury pomocí UML modelu komponent a balíčků.



Obr. 5-7: Méně formální varianta modelu softwarové architektury. Vrstvený model vlevo je detailem aplikační vrstvy vysokoúrovňového modelu.

Je třeba si uvědomit, že proto abychom byli schopni ověřit správnost a proveditelnost navržené architektury, potřebujeme více než jen revidované modely či stránky papíru nebo nainstalovaný aplikační a databázový server. Potřebujeme spustitelnou architekturu, kterou můžeme testovat, tj. ***spustitelný kód ve formě implementovaných scénářů kritických use casů. Toto je jediný způsob, kterým ověříme vhodnost architektury***, jinak nevíme, že je opravdu vhodná pro naše řešení.

5.4.1 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.

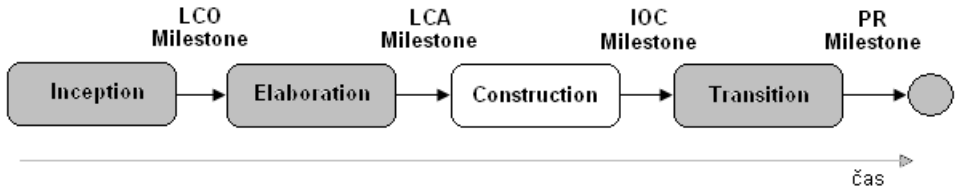
Zda jsme dosáhli tohoto milníku nám pomůže zjistit následující kontrolní seznam:

- 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.

5.5 Construction phase

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

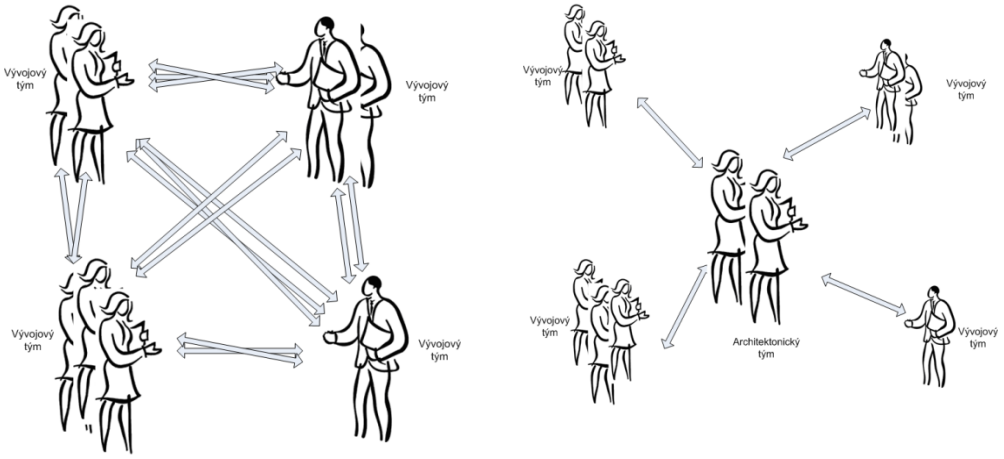


Obr. 5-8: Fáze Construction

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 s pravidelnými přírůstky funkčností. Výstup Construction je připravený k doručení uživatelské komunitě (beta release – první funkční verze finální aplikace).



Obr. 5-9: Organizace kolem architektury minimalizuje přílišné komunikační zatížení.

Hojně využívaným konceptem této fáze je organizace týmů kolem architektury. Tento koncept se snaží efektivně nahradit komunikaci tváří v tvář, která je důležitá, ale v případě velkého vývojového týmu by neúměrně narostla (geometricky!) a snížila efektivitu vývojového týmu. Tento potenciální problém 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).

Pro zajištění nepřetržitého postupu při vývoji nové aplikace v Construction fázi je třeba naplnit krátkodobé cíle, mezi něž patří:

- Vytvoření jednoho týmu s jedním úkolem – je třeba předejít funkčním týmům, kde jsou analytici v jednom týmu a vytvořenou dokumentaci „hodí přes zeď“ vývojářům atd. Cílem je mít funkční týmy obsahující kombinace rolí, kde každý cítí odpovědnost za aplikaci a za postup, k tomu pomohou mimo jiné také denní krátké schůzky, kde tým diskutuje současný stav a problémy a také na co se zaměřit v příštích iteracích.
- Nastavení jasných cílů pro vývojáře, co dokončit v této iteraci (tzv. objektivní evaluační kritéria).
- Průběžně demonstrovat a testovat kód – toto je jediné měřítko postupu, ne říci 90% hotovo, pouze demonstrovatelná aplikace je brána jako měřítko postupu.
- Průběžná integrace – pokud je to možné, je vhodné dělat denní buildy.

V Elaboration jsme rozdělili systém na subsystémy a definovali jsme klíčové komponenty a jejich rozhraní a také mechanismy architektury. V každé iteraci v Construction se zaměřujeme na dokončení návrhu určité skupiny komponent a subsystémů a určité skupiny Use Case. V prvních iteracích Construction se zaměřujeme opět na snížení či odstranění nejrizikovějších věcí (jedná se hlavně o rozhraní, výkonnost, požadavky a použitelnost). V pozdějších iteracích v Construction se zaměřujeme na úplnost, kdy navrhujeme, implementujeme a testujeme veškeré nutné či možné scénáře vybraných Use Case.

Důležitým faktorem v Construction fázi je průběžné ověřování chování implementovaných komponent/systémů. Pro tento účel je většinou využíváno unit testů a v další fázi integračních a systémových (funkčních) testů. Cílem je co největší automatizace testování, jelikož tím omezíme chybovost, zvýšíme produktivitu a vývojáři dostanou okamžitou zpětnou vazbu o kvalitě svého kódu. Při tvorbě unit testů využíváme útržků kódu, které mohou simulovat další komponenty či interakci s nimi[4]. Ty mohou být automaticky generovány vizuálními modelovacími nástroji. Automatizace a znovu-použitelnosti je využíváno také v případě Test Casů, které jsou odvozeny z Use Casů. Jaká omezení (např. výkonnost) je třeba testovat najdeme v nefunkčních požadavcích. Při testování bychom měli pamatovat několik zásad:

- Cíle testování identifikujeme analýzou iteračního plánu, je třeba vědět, co je jeho cílem, abychom toto mohli následně otestovat.
- Identifikace způsobu testování.
- Analýza způsobů a výběr oblasti, pro kterou vzniknou testovací scénáře (Test Case).
- Implementace testů pro každý Test Case a jejich provedení.
- Analýza testů, které selhali a návrh změn.

Stejně jako je pro vývojáře důležitá zpětná vazba pomocí unit, systémových či integračních testů, je pro celý tým důležitá zpětná vazba od uživatele, který si „hraje“ s jednotlivými releasy aplikace, ověřuje její správné chování a poskytuje důležitou zpětnou vazbu a to již v průběhu Elaboration, kde implementujeme základní koncepty interakce aplikace s uživatelem.

Na konci Construction fáze provádíme také tzv. beta-relase, jehož předmětem je testování, do něhož jsou zahrnuti vybraní uživatelé. 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.

5.5.1 Milník IOC

Milník IOC (Initial Operation Capability) 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:

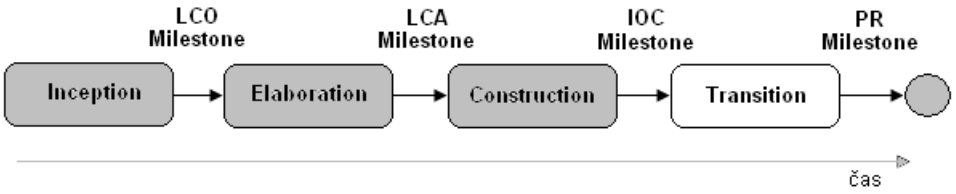
- Je produkt dostatečně stabilní a vyzrálý, aby mohl být rozeslán mezi komunitu uživatelů (výsledky testů)?
- Jsou aktuální výdaje na zdroje oproti plánovaným stále akceptovatelné?

5.6 Transition phase

Poslední fáze představovaného iterativního způsobu vývoje se nazývá Transition. Jejím cílem je především finální vyladění produktu a to nejen

[4] Tyto komponenty/systémy ještě nemusí být hotové a proto bychom jinak nemohli testovat naši část systému (třídy, komponenty či moduly).

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), ale je možné je mít ve formě fotek náčrtků z tabule, či na něm přímo ponechané, dále ve formě papírových náčrtků nebo je mít jen v hlavě, samozřejmě vše podle rozsahu, formálnosti projektu a velikosti a distribuce týmu.



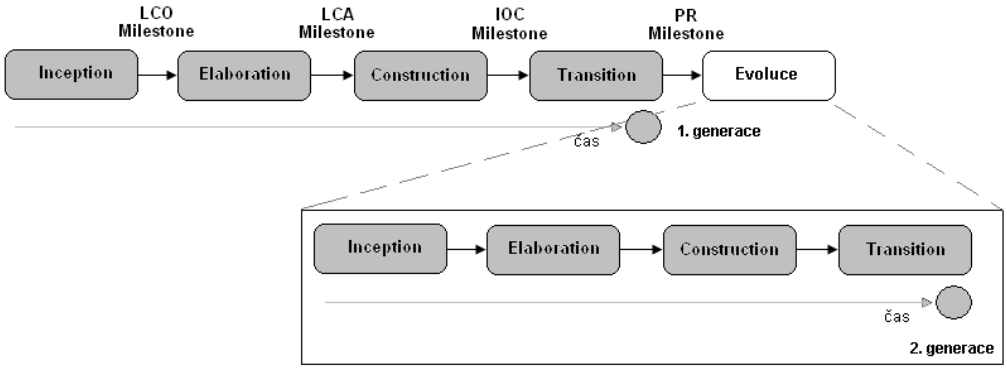
Obr. 5-10: Fáze Transition

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 a use casy). 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.

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 zbývajících chyb (převážně odhalených beta-testováním) 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). V případě evoluce, čímž je rozuměn vývoj další verze, jsou většinou překryty fáze Transition právě dokončované verze a Inception životního cyklu verze nové.



Obr. 5-11: Vývojové cykly více verzí produktu a jejich návaznost. Typicky se mohou v takovém případě překrývat Transition předchozí verze a Inception fáze verze nové.

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 prováděné zákazníkem. Pokud je vývoj ukončen, chyby opraveny a vytvořen build, je 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.

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.

5.6.1 Milník PRM

Posledním milníkem je tzv. PRM – Product Release Milestone, který ukončuje čtvrtou a poslední fázi životního cyklu RUP/OpenUP. 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 (včetně výsledků akceptačního testování)?
- Jsou aktuální výdaje versus plánované akceptovatelné; pokud ne, jaké akce mohou být v příštích projektech provedeny, abychom tomuto problému předešli?

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 fází vodopádu a fází iterativně inkrementálního přístupu?
6. Kdy, v které fázi se snažíme odstranit technická rizika projektu?
7. Jaké jsou nejdůležitější principy, na kterých stavíme ve fázi Elaboration?
8. Co vše by měla obsahovat Vize (Vision) vytvořená v Inception?

Úkoly k zamyšlení:

Pokuste se zamyslet nad finančními přínosy iterativně-inkrementálního a vodopádového modelu. 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 iterativně-inkrementálního modelu vývoje podle metodiky RUP/OpenUP a také s milníky, které tyto fáze uzavírají, respektive ověřují jejich naplnění. 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.

6 Disciplíny vývoje podle RUP/OpenUP

V této kapitole se dozvíte:

- Jaké jsou obecné disciplíny vývoje software?
- Jaké jsou definovány disciplíny podle RUP/OpenUP?
- Jaká je náplň jednotlivých disciplín?
- Jak probíhá spolupráce mezi disciplínami?
- Vztah k disciplínám v iteracích?

Po jejím prostudování byste měli být schopni:

- Porozumět disciplínám a jejich náplni.

Klíčová slova této kapitoly:

Disciplíny vývoje, cross-functional tým, RUP, OpenUP.

Doba potřebná ke studiu: 10 hodin

Průvodce studiem

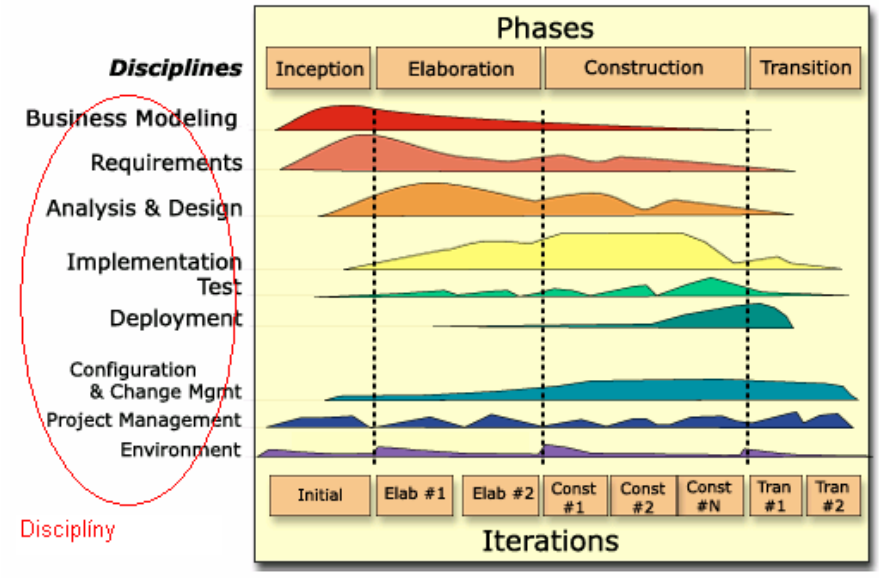
Kapitola představuje jednotlivé disciplíny vývoje software podle RUP/OpenUP, dále představuje jejich náplň a spolupráci jednotlivých disciplín v průběhu iterativně-inkrementálního projektu. Připomenut je iterativní způsob, kdy jsou vykonávány všechny disciplíny paralelně.
Na studium této části si vyhraďte 10 hodin.

Tato kapitola se zabývá nezbytnými disciplínami vývoje software. Každá z disciplín je nejdříve popsána obecně a poté také v kontextu RUP/OpenUP. Zaměříme se tedy na následující disciplíny:

- Byznys modelování,
- Specifikace požadavků,
- Analýza a návrh,
- Implementace,
- Testování,
- Doručení (Deployment)
- Projektové řízení (*Project management*),
- Správa konfigurací a změn (*Configuration and Change management*),
- Prostředí (*Environment*).

Toto jsou základní disciplíny, jejichž aktivity bychom měli vykonávat téměř v každém projektu vývoje software bez ohledu na použitý model vývoje (vodopád, spirála, iterativně-inkrementální). Rozdíl bude jen v jejich mapování na fáze životního cyklu daného modelu. Jinými slovy řešeno, zda je provádím jednou či vícekrát, za sebou nebo paralelně apod. V předchozích kapitolách jsme nastínili, v jakém množství jsou dané disciplíny vykonávány v rámci iterací. Pozor tedy, neplést si disciplíny (jen seskupení aktivit vykonávaných v různých fázích projektu) s fázemi vodopádu, každou (či většinu) z těchto disciplín provádíme v každé iteraci každé fáze iterativně založeného projektu.

Obsah metodického frameworku RUP i OpenUP je, kromě rozdělení na iterace a fáze s rozdílným zaměřením, organizován také do disciplín. Disciplíny jsou souhrnem, kolekcí úkolů a aktivit týkajících se konkrétní oblasti zájmu. Objem prací z každé disciplíny v daných fázích RUP definuje množství plochy pod křivkou v modelu. Disciplíny v modelu RUPu zachycuje následující obrázek.



Obr. 6-1: Disciplíny podle RUP (zdroj [RUP large]).

Tak například nejvíce aktivit spojených s disciplínou Requirements (definice požadavků) je zjevně ve fázi Inception a Elaboration, jelikož zde je plocha pod křivkou největší. Je ale nutné připomenout, že i v dalších fázích provádíme některé činnosti této disciplíny! Například detailní popis požadavků ve fázi Construction a částečně také v Transition. Podobně je vidět, že disciplína testování (Test) je prováděna v každé iteraci v každé fázi, což jsou viditelné vlnky tvarované danou křivkou. Toto je logické, protože výsledkem každé iterace je sestavený a otestovaný spustitelný build. Nejvíce práce spojené s testovací disciplínou je pak na konci Construction, jelikož tvoříme a spouštíme nejvíce Unit testů, stejně jako systémových, integračních, regresních a výkonnostních. Výsledkem Construction je pak téměř finální verze aplikace, testovaná, obsahující téměř všechny rysy, tzv. Beta release.

Jak už bylo zmíněno v textu několikrát, mezi disciplínami iterativního vývoje a disciplínami, resp. fázemi, vodopádového vývoje je zásadní rozdíl. Ve vodopádovém modelu jsou jednotlivé disciplíny prováděny sekvenčně. V jedné fázi provedeme veškeré aktivity jedné disciplíny, výstupy zrevidujeme, odsouhlasíme, fázi uzavřeme a pokračujeme do další fáze, kde provádíme aktivity další disciplíny. V praxi to znamená, že nejdříve provedeme definici požadavků, posléze celou analýzu, pak předáme podklady architektovi pro návrh apod. Výsledkem je možná ztráta informace a rozdílné interpretace informací a požadavků, pokud tito lidé dále při vývoji těsně nespolupracují. Typicky od fáze návrhu nejsou ve vodopádových projektech dostupní analytici, jelikož se již účastní dalšího projektu.

V iterativně-inkrementálním způsobu vývoje provádíme všechny disciplíny paralelně v každé fázi a v každé její iteraci. To znamená že v rámci jedné iterace provádíme zároveň detailnější rozbor relevantních požadavků (ty co nyní nebo v následující iteraci budeme implementovat), realizaci komunikace a rozpad na komponenty na čemž se podílí analytik, architekt, tester i programátor a mezitím mohou být zároveň kódovány testy na dané požadavky, pokud používáme techniku TDD (*Test Driven Development* – nejdříve píšeme testy, až potom vlastní kód, více viz Informační systémy 2). Klíčovým pojmem je zde tzv. *cross-functional* tým. Jedná se o tým, jež obsahuje po celou dobu vývoje/iterace všechny potřebné role, které spolu těsně spolupracují. Navíc lidé hrající tyto role nejsou detailně specializovaní, ale ovládají všechny dané disciplíny, které je třeba v iteraci provést (od plánování iterace, přes analýzu a návrh požadavků až po testování a kódování), nebo alespoň ovládají jejich většinu (v případě některých jsou větší odborníci, v případě jiných zase menší, ale mají povědomí o více disciplínách než o té své vlastní). Iterace tedy není provedení malého vodopádu, ale opravdu provádíme všechny disciplíny paralelně. Každý z členů může být zároveň částečně návrhář i programátor (spíše technický specialista) nebo analytik a tester (spíše doménový specialista).

6.1 Disciplína Business Modeling

Tato disciplína definuje aktivity, které je nutné provést pro pochopení problémové domény, pro které připravujeme řešení – SW produkt. Pokud bude našim cílem vyvinout program sloužící pro burzu a makléře, budeme pravděpodobně muset nejdříve pochopit danou doménu, tj. pojmy používané makléři (definovat slovník), způsob obchodování a oceňování. Jen tak budeme schopni vytvořit softwarový produkt, který bude zákazníkovi přínosem a bude řešit jeho problémy a komunikovat s ním v jeho odborné řeči.

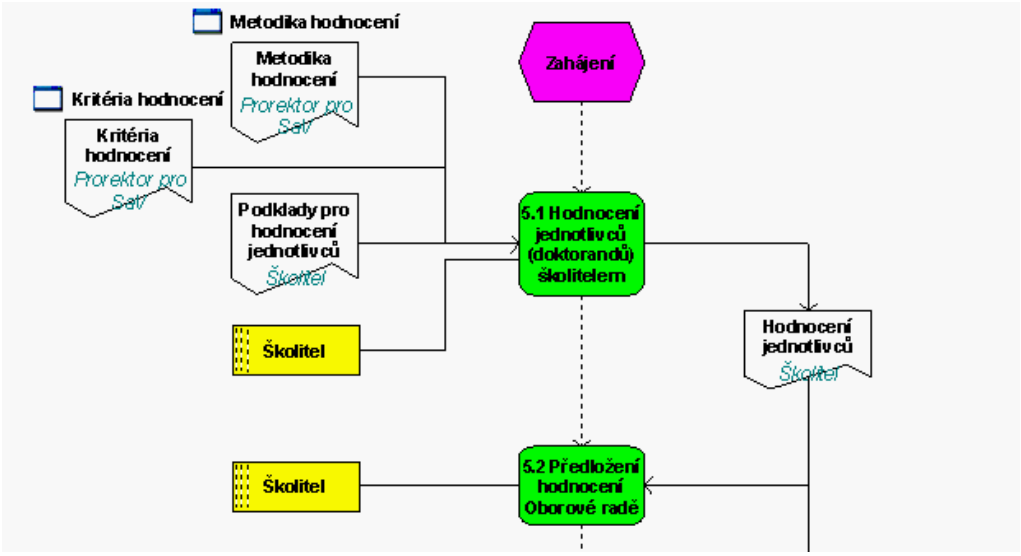
Aktivity definované touto disciplínou není třeba vykonávat v rámci každého projektu. Je-li cílem projektu vytvoření další verze produktu nebo rozšíření stávající verze, pochopení domény uvnitř týmu již většinou existuje. Naopak důležité je, se na tyto aktivity zaměřit, vytváříme-li pouze analýzu proveditelnosti (včetně prototypu) pro budoucí produkt, který jsme ještě nevytvářeli. Procesní či doménový model (včetně slovníku pojmů a synonym) je velmi důležitý také pro potřebu údržby (*maintenance*). Vývojáři pracující na údržbě a vylepšení systému potřebují rozumět, co a jak byznys uživatelé vykonávají, aby mohli aplikaci udržovat a spravovat. Toto je často přehlížený aspekt pramenící v příkazy typu: *„oprav tuto třídu“*, *„přidej tuto metodu“* bez pochopení přínosu pro uživatele, což může vést k neefektivní implementaci, k různým výkladům podle vývojářova (ne)pochopení – nechápeme proč má aplikace dělat přesně to či ono, či k redundanci kódu.

Jak však dostat pochopení problémové domény do hlavy? Nejlepší cestou je danou věc opravdu dělat. V případě burzy se stát makléřem či kupujícím. Toto samozřejmě není vždy možné, proto existují také další možnosti:

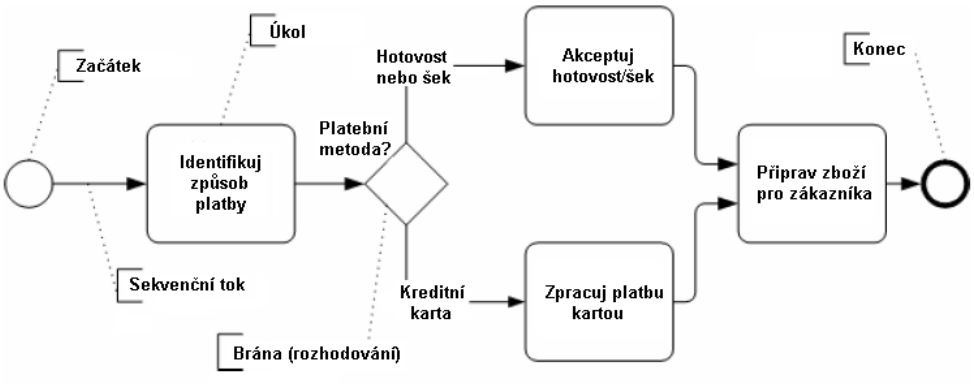
- Stínování zaměstnanců – sledování jejich práce po určitý čas v pravidelných intervalech a dotazování se na nejasné věci.

- Použití byznys konzultantů – specialistů na danou problémovou doménu.
- Tvorba jednoduchých prototypů a jejich demonstrace a diskuse nad nimi s budoucími uživateli či s byznys konzultanty.

Velmi rozšířeným nástrojem pro modelování podnikových procesů jsou tzv. EPC (Event-Driven Process Chain) diagramy popisující tok aktivit, vstupní a výstupní artefakty a zúčastněné osoby pro každou aktivitu (viz následující obrázek). Další rozšířenou metodou, která je i standardem OMG je BPMN (Business Process Modeling Notation). Tyto metody či nástroje jsou podporovány řadou nástrojů a mohou být automatizovány, avšak nejsou součástí RUP.



Obr. 6-2: Příklad notace procesu pomocí ECP



Obr. 6-3: Příklad notace procesu pomocí BPMN

Již tedy víme, k čemu obecně slouží byznys modelování, jaký je jeho účel. Nyní se zaměříme na byznys modelování podle RUP/OpenUP. RUP/OpenUP definuje účel disciplíny Business Modeling následovně:

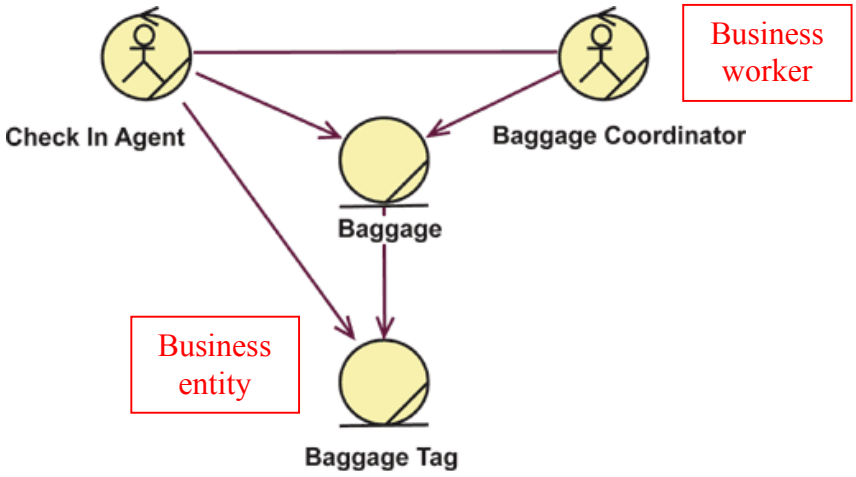
- Pochopení problémů organizace a identifikace oblastí vhodných ke zlepšení.
- Ohodnocení dopadu změny na organizaci.

- Zajištění porozumění potřebám organizace mezi všemi (zákazníci, koncoví uživatelé, vývojáři, …).
- Vyvození požadavků na softwarový systém podporující potřeby organizace.

K tomu nám pomáhají artefakty jako vize podniku, byznys slovník, podniková pravidla, procesní modely apod. Následující obrázek ukazuje aktivity této disciplíny.

Obr. 6-4: Aktivity a artefakty přiřazené roli Business Process Analyst disciplíny Business Modeling (zdroj [RUP large]).

Dalším možným modelem je tzv. *Business Object Model* <<Business Object Model>>. Business use case model popisuje, **co** zákazník (uživatel) provádí. Business Object Model pak říká, **jak** toto provádí. K tomu nám slouží dva typy entit – aktivní (Business worker) a pasivní (Business entity), viz následující obrázek.



Obr. 6-5: Business Object Model (zdroj [RUP large])

6.2 Disciplína Requirements

Tato disciplína definuje aktivity týkající se identifikace aktorů, nastínění a také detailního popisu požadavků ve formě scénářů, agregovaných do Use Casů (v případě RUP, OpenUP a UP). Scénář je popsaný mírně formalizovaný postup, jakým uživatelé pracují s aplikací. Tato disciplína, stejně jako role, které vykonávají jednotlivé aktivity, je velmi důležitá, neboť slouží k pochopení budoucí aplikace a k distribuci tohoto pochopení mezi všechny účastníky projektu (hlavně architekty, vývojáře, testery). Proto musí být

analytici v průběhu celého životního cyklu aplikace k dispozici, jelikož neustále synchronizují pohled architekta, vývojářů a testerů na systém. Přenos znalosti nemůže proběhnout pomocí dokumentů, ale pouze pomocí neustále komunikace a vyjasňování. ***Use case či scénář tedy není kompletní dokumentace, ale spíše jen pozvánka ke komunikaci!***

Obecně můžeme říci, že existují dvě základní kategorie požadavků na systém:

- ***Funkční požadavky*** – popisují, co má systém dělat, jeho budoucí chování (use case systému – základní způsoby použití, ne detaily).
- ***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, ..).

Dalším pojmem na úvod, který osvětlíme je SRS. Tato zkratka znamená *Software Requirements Specification*, česky: specifikace softwarových požadavků. Jedná se o dokument, model či elektronický artefakt, který zachycuje požadavky. Forma a způsob uchování tedy není opět tolik důležitá jako vlastní obsah.

Tradiční přístup k SRS

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, jak již víme, 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 těžko 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 6-1: Funkční specifikace požadavků

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

FUPRS+
Tento přístup popisuje, jaké kategorie požadavků bychom neměli opomenout spíše než formu jejich zápisu, a je také doporučován v RUP. 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)
- **S**upportability (podporovatelnost)
- + znamená, že bychom neměli zapomenout ani na návrhové, implementační a fyzické požadavky.

Standard IEEE 830
Jelikož se zabýváme vývojem software, existuje i v oblasti specifikace požadavků spousta doporučení a standardů. Představíme si tedy alespoň 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 následující:

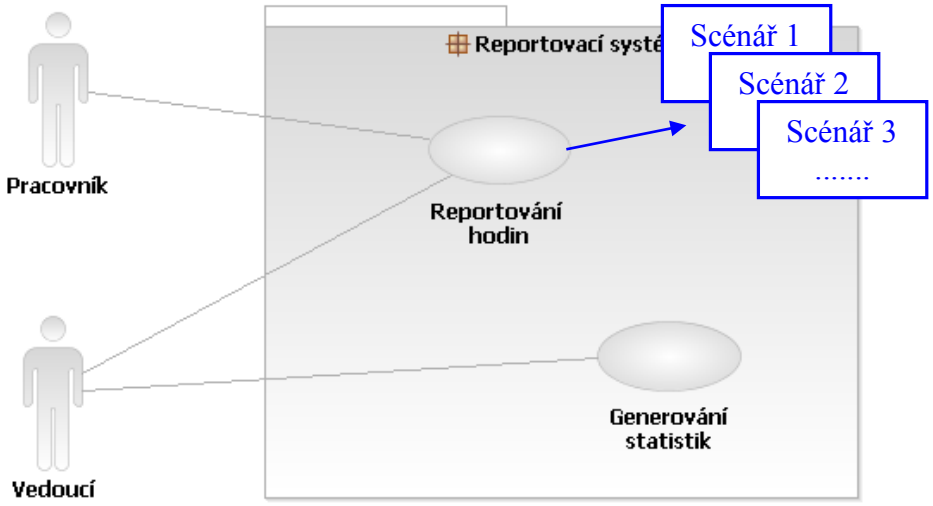
- Zaměření (*scope*) tohoto doporučení, jímž je popis praktik SRS za účelem vývoje SW, částečně také za účelem výběru SW.
- Odkazy na relevantní standardy IEEE (např. IEEE 610.12 – Standard Glossary of Software Engineering Terminology; IEEE 1042 – Guide to SW Configurations Management).
- Definice používaných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

Součástí dokumentu je také dporučení 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).

Use case
Jako poslední zmíníme techniku zvanou use case, česky někdy nazývány případy užití nebo modely jednání. Jak jste mohli vidět v kapitole věnující se životnímu cyklu, je RUP *„Use Case driven“* – tedy řízen use casy. To znamená, že vývoj, analýza, plánování, testování je prováděno podle use case, ty tvoří rozsah mini-projektu iterace a jsou v průběhu životního cyklu projektu znovupoužity. Jak již dále 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 a navíc v byznys řeči zákazníka, bez nějakých technologických detailů.

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



Obr. 6-6: Use case model zaštiťující scénáře systému a scénáře jednoho UC na dokreslení kontextu (pozor, ty nejsou součástí UC modelu! Nekreslí se!).

Use casy popisují chování, které očekává uživatel od vytvářeného systému. Toto chování – use case – je pak detailněji popisováno scénáři. Jedná se o mírně formalizovaný (strukturovaný) příběh, který zachycuje, jakým způsobem uživatel využívá konkrétní funkčnost systému.

Jak use case řídí vývoj, jak podle nich plánujeme, jak je realizujeme, či jak je můžeme znovupoužít pro dokumentaci či testy, je zmíněno v celém textu těchto skript, hlavně pak ale v učebním textu Ročníkový projekt. Blíže k detailnímu popisu a tvorbě use case a také ke specifikaci požadavků viz například [Co05].

Nefunkční požadavky

V rámci SRS podle IEEE 830 a také podle FURPS+ jsme zmínili různé druhy nefunkčních požadavků, mezi nimi:

- použitelnost (*usability*) – týká se lidského faktoru, estetičnosti, jednoduchosti a intuitivnosti aplikace; důležitou složkou je také konzistence nejen GUI, ale také dokumentace či tréninkových materiálů; v neposlední řadě musíme zmínit následování standardů, např. standardy ovládání okenních aplikací (3 hlavní ovládací tlačítka vpravo nahoře) apod.,
- spolehlivost (*reliability*) – týkající se frekvence a dopadu výpadků, jejich předpověditelnosti, obnovy (rychlost, způsob) a přesnosti,

- výkonnost (*performance*) – mají dopad na funkční požadavky, omezují či specifikují dobu zpracování požadavku, či transakcí, využitelnost paměti pro danou operaci, přístupové rychlosti, přesnost a další,
- schopnost podpory (*supportability*) – tyto požadavky se týkají udržitelnosti aplikace v chodu. Jedná se o testovatelnost, rozšiřitelnost atd. Netýkají se pouze požadavků či systému jako takového, ale také procesů a standardů kolem této aplikace!
- Bezpečnost (*security*) – týkají se bezpečného uložení dat, přístupu k aplikacím či jejich některým funkcím: zabezpečený vs. nezabezpečený přístup, zobrazení pouze relevantních informací či dokumentů apod.

Víme-li, jak tyto nefunkční požadavky rozdělit, je vhodné si také něco říct o způsobu jejich identifikace a dokumentace. Přitěžující okolností těchto typů požadavků je, že často prochází přes všechny či vícero use casů a jejich scénářů, proto je dobré je držet mimo a pouze na ně odkazovat (ano, opět). Jednou z forem dokumentace je opět scénář. 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], [SEI3].

Struktura scénáře kvalitativních atributů podle QAW je následující:

Stimul (Stimulus)	Podmínka, která ovlivňuje systém.
Zdroj (Source)	Entita, která generuje stimul.
Prostředí (Environment)	Podmínky za kterých se daný stimul objeví.
Artefakt (Artefakt)	Artefakt, který je stimulován.
Odezva (Response)	Odezva, kterou dostaneme jako odpověď na daný stimul.
Míra pro měření odezvy (Response measure)	Míra, jíž je daná systémová odezva měřena, ohodnocena.

Tabulka 6-2: Scénář kvalitativních atributů.

Takový scénář tedy budeme vytvářet pro všechny důležité atributy, které mají dopad na architekturu systému. Zdrojem možných kategorií jsou výše zmíněné: použitelnost, spolehlivost, výkonnost a mnoho dalších. Plný seznam možných viz [SEI3]. Pro zachycení a identifikaci těchto scénářů opět použijeme minimalistický princip, kdy v Inception některé tyto scénáře pouze identifikujeme a v průběhu iterací v Elaboration, jak začnou být dané nefunkční požadavky potřebné, resp. naplněné, je detailně popíšeme a průběžně implementujeme, resp. architekturu navrhujeme podle nich. Pro lepší obrázek ještě přikládáme příklad jednoho scénáře.

Scenario Refinement for Scenario N		
Scenario(s):		When a garage door opener senses an object in the door's path, it stops the door in less than one millisecond.
Business Goals:		safest system; feature-rich product
Relevant Quality Attributes:		safety, performance
Scenario Components	Stimulus:	An object is in the path of a garage door.
	Stimulus Source:	object external to system, such as a bicycle
	Environment:	The garage door is in the process of closing.
	Artifact (If Known):	system's motion sensor, motion-control software component
	Response:	The garage door stops moving.
	Response Measure:	one millisecond
Questions:		How large must an object be before it is detected by the system's sensor?
Issues:		May need to train installers to prevent malfunctions and avoid potential legal issues.

Obr. 6-7: Scénář kvalitativních atributů (zdroj [SEI3]).

RUP/OpenUP a správa požadavků

Již tedy víme, k čemu obecně slouží specifikace požadavků, jaké metody a nástroje můžeme použít, jakých chyb bychom se měli vyvarovat. Nyní se zaměříme na identifikaci a správu požadavků podle RUP/OpenUP. Tyto frameworky definuje účel disciplíny Requirements následovně:

- Vytvoření a udržování dohody mezi zákazníkem a dalšími účastníky projektu o tom, co by měl systém dělat.
- Poskytnout vývojářům lepší pochopení požadavků na systém.
- Definovat hranice systému.
- Poskytnout základ pro plánování technické náplně iterací.
- Poskytnout základ pro odhad nákladů a času potřebných k vývoji systému.
- Definovat uživatelské rozhraní systému se zaměřením na potřeby a cíle uživatele.

Systémový analytik v průběhu všech fází těsně spolupracuje nejen se zákazníkem, ale také se systémovým architektem, projektovým manažerem a dalšími členy vývojového týmu (vývojáři, testeři). Dále je také zodpovědný za tvorbu nefunkčních požadavků (výkon, vzhled, bezpečnost, …) ve spolupráci s architektem. Úkolem analytika je také sbírat a zpracovávat zpětnou vazbu od uživatelů, jelikož to bývá on, kdo prezentuje řešení (vytvořený build), a asistuje uživatelům při jejich hře s aplikací, aby věděli, které funkčnosti jsou nové a které zatím nefungují a jsou pouze ukázány, jejich možnosti a uživatelské spuštění a podobně.

Získávání požadavků

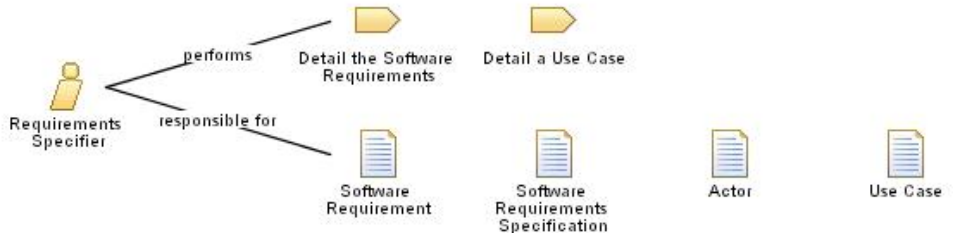
Způsobů získávání požadavků je několik. Můžeme využít IT strategii firmy, stejně jako provést analýzu existujícího systému. Pokud vytváříme systém nový, je třeba tyto představy získat od budoucích uživatelů. To je zdrojem problémů, jelikož lidský mozek je dobrý v popisu hmatalených věcí, tj. při hře 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 principu iterativního vývoje, kdy v častých intervalech ukazujeme zákazníkovi, co jsme vytvořili. Mezi způsoby, jak identifikovat požadavky 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) pro demonstraci,
- Uživatelské příběhy (*user stories*) + post-it lístečky – kreslený vzhled GUI a jeho přetváření pomocí nalepovacích štítků (tzv. Post-it).

Jednotlivá sezení by měla být časově omezená, abychom se opravdu věnovali tomu, čemu máme (tj. time-boxed, stejně jako samotné iterace). V případě potřeby je pak možné naplánovat další sezení. Více o náplni a některých aktivitách disciplíny Requirements bylo napsáno v kapitolách o Inception a Elaboration fázi a v úvodní kapitole o use case.

Požadavky přichází z různých zdrojů, výše byla zmíněna strategie podniku a uživatelé, dále jsou to podniková pravidla, trh, konkurence či další zdroje. Důležitá je také forma požadavků. Velmi výhodné v iterativním modelu vývoje je definovat požadavky pomocí use case a jejích scénářů, a detaily, které nejsou nezbytné pro pochopení požadavku, ale musíme je testovat (verifikovat), zachytíme ve formě spustitelných testů, test skriptů. Znamená to opět ušetření práce, kdy detailní specifikace požadavků slouží zároveň jako testovací artefakt. Navíc je tento přístup velmi tolerantní ke změnám. Každá změna, která se dotkne požadavku se musí promítnout také do testu. Díky popisu detailů požadavků ve formě spustitelných testů upravujeme jedním krokem specifikaci požadavků a zároveň související test.



Obr. 6-8: Aktivity a artefakty přiřazené roli Requirements Specifier disciplíny Requirements (zdroj [OpenUP]).

V rámci disciplíny Requirements je třeba zdůraznit aplikaci C principu (*Collaborate Across Teams*, viz kapitola 4.3) a spolupráci rolí této disciplíny s rolemi vývojářskými a testerskými. Některé aktivity disciplíny byly představeny výše, je však také třeba zmínit důležitost některých samostatně v kontextu C principu:

- Vytvoření slovníku – pojmy, synonyma, zkratky, pomáhá nově příchozím se rychle zorientovat. Definuje společnou řeč pro obě strany business lidi a IT lidi, vývojáře. Pojmy definované ve slovníku potom

konzistentně používáme v dalších (převážně textových) dokumentech, specifikacích, hlavně však ve scénářích use casů. Aktivita je typicky vykonávána několikrát v průběhu Inception a Elaboration podle potřeby.

- Vytvoření plánu pro správu požadavků (*Requirements management*) – jedná se hlavně o dohodnutí formy dokumentování požadavků, jejich atributů, průvodců, šablon a způsob jejich uchování a správy. V neposlední řadě je řešeno trasování od potřeb k vlastnímu kódu aplikace, respektive a minimálně od potřeb k požadavkům. Tedy cílem není vyprodukovat nějaký dokument, toto může být popsáno ve formě wiki a přiložených šablon či zakódováno v nástroji pro správu požadavků (*Rational Requisite Pro*, *Enterprise Architect*, *ObjectiF*, *Teamcenter*).
- Vytvoření vize – cílem této aktivity je pochopení problému, identifikace všech zúčastněných, definice hranic systému a popis hlavních rysů aplikace. Za vykonání těchto činností je odpovědná role *System Analyst*, ale do počátečních rozhodnutí musí být rozhodně zapojeni také architekt spolu s projektovým manažerem. Konečný souhlas a vysvětlení potřeb a hlavních rysů je pak aktivita, do které by měli být zahrnuti všichni členové týmu a zainteresovaní účastníci projektu.
- Správa závislostí – opět leží hlavní zodpovědnost na roli *System Analyst*, ale blíže zahrnuti musí opět být i architekt a projektový manažer, jelikož vazby mezi scénáři a scénáře samotné mají dopad na pořadí implementovaných věcí, na možná rizika a dokonce i technologii.

Mluvíme-li o aplikaci C principu, je vhodné zmínit také kombinace rolí. Vhodné je kombinovat analytické a testerské role, jelikož má analytik nejlepší pochopení potřeb a požadavků uživatelů (a nevidí dovnitř aplikace) a je tedy nejvhodnějším adeptem na jejich testování. Dívá se na aplikaci jako na černou skříňku, nevidí a nezná řešení. Nevhodné je naopak kombinovat analytické role s developerskými. Daný člověk potom vidí již při specifikaci požadavků technologii a její implementaci a tudíž nevědomky dělá velmi brzy spoustu návrhových rozhodnutí, což může přinést těžkosti, zjistíme-li, že jiná technologie by byla vhodnější (srovnej např. rozdílnost AJAX vs. tlustý klient vytvořený v Delphi). Analytik se snaží navrhnout nejlepší řešení z pohledu byznys domény, naopak architekt, vývojář se snaží realizovat nejlepší technologické řešení. Jedná se tedy o ideový rozpor.

Důležité artefakty, které tato disciplína zahrnuje jsou tedy:

- vize,
- slovník,
- doplňující specifikace (Supplementary specification),
- use case model,
- storyboards a UI prototypy.

Jelikož jsme zmínili artefakty, zmíníme na závěr ještě jednu, nám již známou věc (minimálně z předmětu Softwarové inženýrství), týkající se formy dokumentace. ***Forma*** (dokument, skript, komentář v kódu, model, fotka, …)

zachycení informace není v softwarovém inženýrství často rozhodující. Rozhodující je, 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, distribuci týmu a zkušenostech členů, množství zainteresovaných stran a dalších faktorech, použijeme různou formu a množství dokumentace. V případě malého, zkušeného týmu, který sedí pohromadě, a kterému je technologie a problémová doména 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.

Na závěr opět připomeneme, že analytik (resp. lidé hrající analytické role) v rámci své práce vykonává také aktivity definované disciplínou Business modeling, Requirements a také Analysis & Design.

6.3 Disciplína Analysis and Design

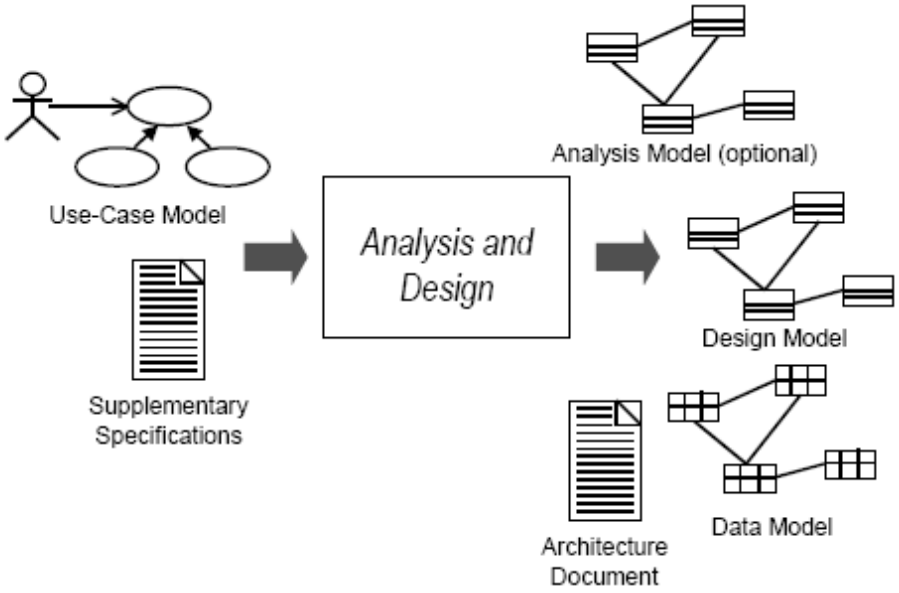
Tato disciplína je zaměřena na to, co a jak máme vytvořit. Základním cílem této disciplíny je transformovat požadavky do specifikace popisující, jakým způsobem implementovat daný systém. Budeme se tedy zabývat tím, jak z popisů jednotlivých scénářů identifikovat analytické objekty, způsob komunikace a tyto analytické objekty transformovat do návrhových v závislosti na vybrané technologii (návrh architektury). V počáteční fázi projektu je naším cílem vytvoření stabilní architektury za účelem jednoduché implementace a rozšiřitelnosti daného systému. Dalším krokem je mapovat vytvořený design na implementační prostředí a navrhovat systém s ohledem na výkonnost, robustnost, škálovatelnost, testovatelnost a další kvalitativní atributy. Aktivity definované v rámci této disciplíny jsou vykonávány analytiky ale také architekty, návrháři systémů.

Rozdíly mezi analýzou a návrhem

Pokud mluvíme o disciplíně Analysis & Design, je nutné nejdříve vysvětlit, jaký je mezi nimi rozdíl. Design je v kontextu softwarového inženýrství (a tedy i RUP/OpenUP) chápán v českém významu jako návrh, ne jako vytváření použitelného uživatelského prostředí – to je pouze malinká podmnožina celého návrhu! Takto omezeně význam tohoto slova často chápou studenti, jelikož si nedokážou představit, co všechno návrh obnáší, ale znají návrh vzhledu webových stránek a myslí si, že toto (tvorba GUI) je všechno, co musíme dělat i při návrhu softwarových systémů. Pak omezují návrh na tvorbu GUI a databáze, čímž také často začínají implementaci, aniž by vůbec uvažovali, sbírali a analyzovali požadavky na systém!

Čím se tedy zabývá analýza a čím návrh? Analýza se zaměřuje na sběr a pochopení funkčních požadavků na systém, zjednodušeně by se dalo říci, že opomíjí nefunkční požadavky, implementační omezení a prostředí. Výsledkem analýzy je téměř ideální obraz budoucího systému ve formě množiny tříd a

subsystémů, ale bez vazby na technologii (neuvažujeme knihovny, kolekce, databáze apod.). Kdežto v rámci návrhu se snažíme napasovat výsledky analýzy na implementační prostředí a na omezení plynoucí z nefunkčních požadavků. Jedná se tedy o další krok k výsledné aplikaci, o další zjemnění, propracování analýzy. Cílem návrhu je mít optimálně navržený systém pokrývající veškeré požadavky, a to jak funkční, tak nefunkční.



Obr. 6-9: Use casy řízená analýza a návrh včetně analytického a návrhových modelů (vpravo).

Analýza a návrh podle RUP/OpenUP

Tyto frameworky definují účel disciplíny Analysis and Design následovně:

- Transformace požadavků do návrhu budoucího systému.
- Vývoj robustní architektury systému.
- Přizpůsobení návrhu implementačnímu prostředí.

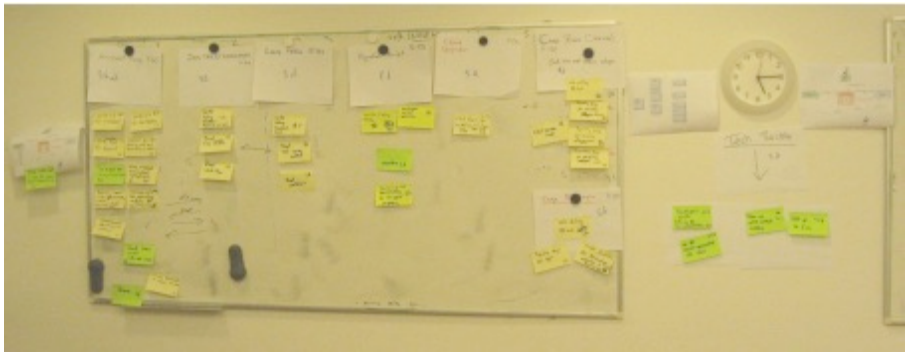
Tato disciplína, jak již bylo zmíněno výše, se týká hlavně definice architektury. Víme, že architekturu definuje použitá technologie (operační systém, programovací jazyk, běhové prostředí), komunikační mechanismy, definovaná rozhraní, způsob ukládání dat, rozvrstvení komponent apod. Architektura je také definována a tvořena základními, kritickými Use Casy (zhruba 20 – 30% ze všech Use Casů). Architektura je potom využívána všemi programátory, jako základ, na který nabalujeme funkčnosti aplikace. Využíváme definovaných cest, způsobů komunikace mezi komponentami či po síti, ukládání, předdefinované frameworky či API.

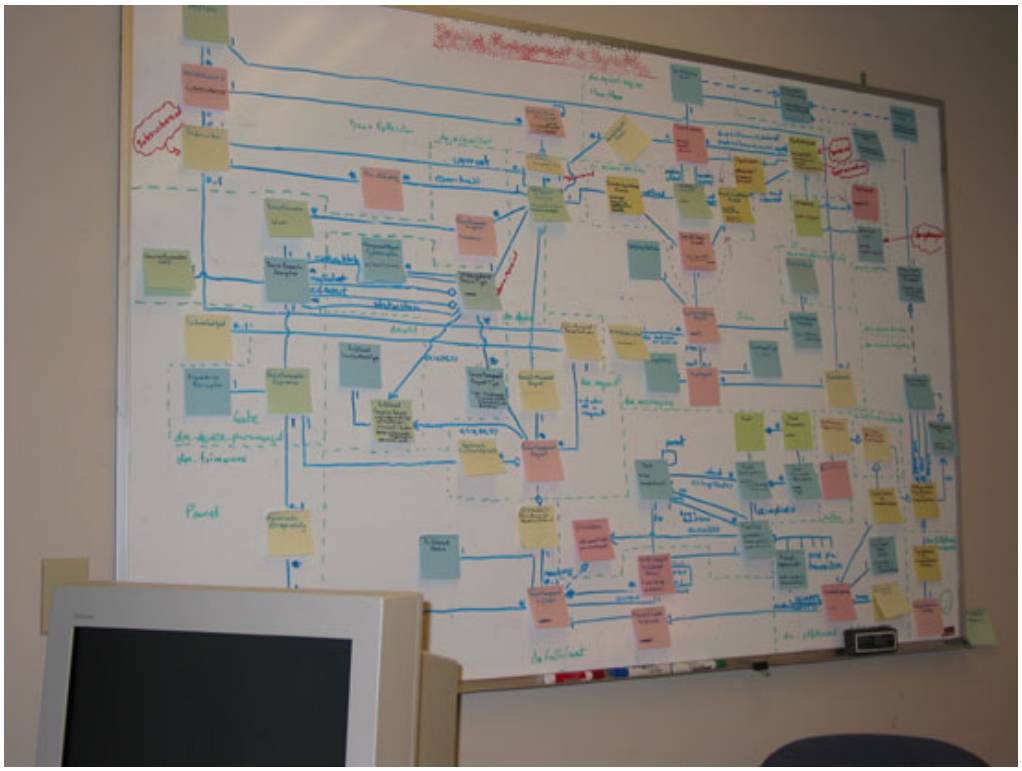
Jak ukazuje samotné workflow disciplíny, sled aktivit prostupuje životním cyklem vývoje již od počáteční fáze Inception a je zhruba následující. Nejdříve se architekt podílí na definici a prioritizaci use case, jelikož nejvíce kritické scénáře z pohledu byznysu, a také ty nejvíce riskantní z pohledu technologie, tvoří základ architektury. Právě pro ně definujeme nejvhodnější architekturu. Pokud se jedná o nový projekt v neprozkoumané oblasti, je možné vytvořit prototyp architektury, abychom ověřili proveditelnost naší volby či snížili

rizika. Dále disciplína prostupuje fází Elaboration, kde provádíme aktivity zaměřené na identifikaci návrhových mechanismů (většinou dáno výběrem infrastruktury, middleware), návrhových elementů (pozor na přílišnou detailnost a rozsáhlost) a znovupoužitelných komponent. Důležitým krokem je také struktura implementačního modelu, kde budou uloženy implementační artefakty (zdrojové kódy, spustitelné soubory, dokumentace, modely), jak spravovat znovupoužitelné komponenty, co bude pod správou verzí. Cílem architekta není architekturu jen definovat, ale také zajistit využívání definovaných mechanismů. Proto je třeba tuto znalost rozprostřít mezi všechny vývojáře (guidelines, workshopy), aby nedefinovali jiné boční cesty, které přispějí k horší udržovatelnosti a čitelnosti aplikace, tedy k nižší kvalitě a vyšší ceně údržby. Posledním krokem, který je možné provést v rámci disciplíny Analysis and Design, je revize architektury. Cílem je detekovat chyby v návrhu, ať již technického rázu nebo doménového rázu (řešení nesplňuje požadavky zákazníka). Revize architektury je součástí LCA milníku na konci fáze Elaboration. Je však nutné připomenout, že tato revize se nebude týkat pouze dokumentů a jejich ohodnocení mimo veškerou realitu, ale jedná se o ověření spustitelného kódu a také demonstrování základních rysů zákazníkovi. Revize navíc probíhá neustále ve formě párového programování, unit testů či neustále integrace (*Continuous Integration*).

Kritickým bodem disciplíny je opět spolupráce architektů s analytiky, kteří mají nejlepší představu o zákazníkových požadavcích, s test manažerem a návrhářem testů, který se stará o testovatelnost řešení a také s projektovým manažerem. Spíše ale opět zdůrazníme cross-functional týmy, kdy všichni umí vše od každé disciplíny (samozřejmě s určitými omezeními) a spolupracují v dennodenním kontaktu.

Pro potřeby disciplíny Analysis and Design lze použít vizuální modelovací jazyk UML. Konkrétně RUP a OpenUP UML doporučuje a využívá. Je třeba však mít opět na paměti rozdílnou možnost formálnosti artefaktů vytvářených v rámci této disciplíny (formální a detailnější dokumenty, modely, či jen hrubé skici, náčrty). V méně rozsáhlém projektu, kdy spolu členové týmu pracují v jedné místnosti a využívají známou technologii, je dostačující, když architekturu definujeme návrhem na tabuli (*whiteboard*) a snímek návrhu uchováme jako návrhový dokument v repository či jako součást dokumentu popisujícího architekturu systému. Příklad neformálních, ale velice účinných a výkonných neformálních nástrojů pro modelování různých pohledů na software zachycují následující obrázky (v pořadí dokumentace požadavků a návrh architektury).





Obr. 6-10: Příklad neformálních modelů architektury systému (využití whiteboardu, nástěnek, flipchartů).

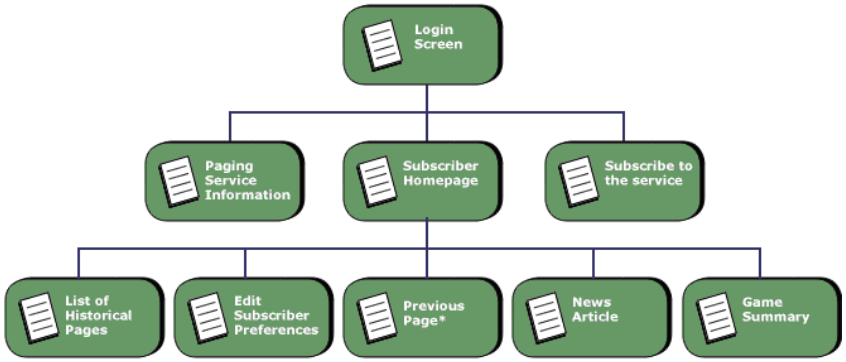
Samozřejmě bychom si měli dávat pozor na úroveň abstrakce, abychom nepopisovali příliš detaily, když jsme například v úvodní fázi projektu. Proto je třeba mít na paměti E princip (Elevace the Level of Abstraction) a v případě potřeby (neznámé či předpokládané složité chování, nutnost popsání komunikace s jiným systémem apod.) se z obecnější úrovně ponořit do detailů řešení. Hlavní náplní návrhu a tvorby architektury je rozdělení funkcí do vrstev a subsystémů a následná definice rozhraní jednotlivých komponent a subsystémů, abychom předešli problémům s integrací. Architekturu také definujeme společně s Test Manažerem či Test Analytikem, abychom byli schopni vybrat a zaručit testovatelnou architekturu.

Příklad:
Pokud mluvíme o testovatelnosti řešení, aplikace, musíme si uvědomit, že už výběr a definice architektury k tomuto také přispívá a to dost kriticky.

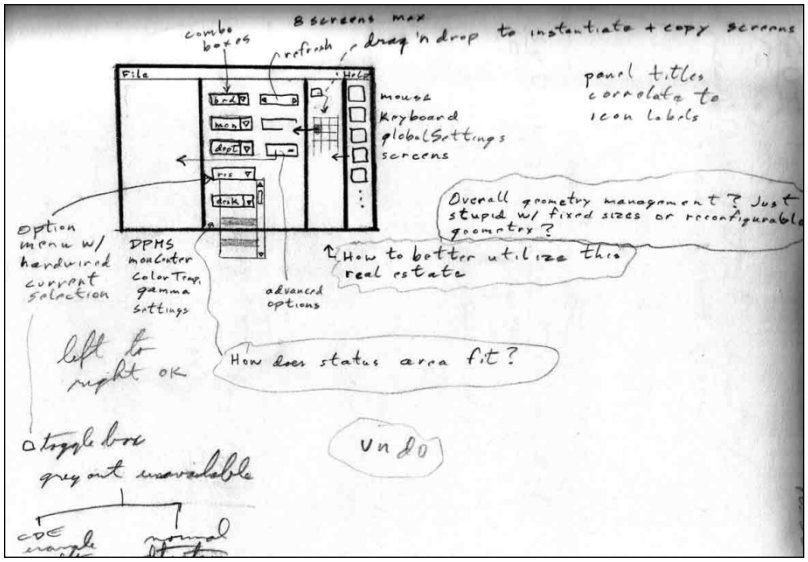
- Hůře se například bude testovat monolitická aplikace, jelikož nejsem schopen sledovat například pouze chování (aplikační logiku) systému.
- Strukturovaný jazyk COBOL a IBM mainframe jako hlavní technologie bude asi také hůře testovatelná než například aplikace napsaná pomocí Java EE, pro které existují množství testovacích komerčních i open-source nástrojů, z nichž některé jsou přímo integrované s vývojovým prostředím.

Proto je nutné mít na paměti, že už i správný výběr architektury přispívá k testovatelnosti aplikace a může přinést významné úspory práce a nákladů díky automatizaci či zvýšenou kvalitu aplikace.

Součástí této disciplíny jsou také aktivity spojené s definicí uživatelského prostředí. Výsledkem mohou být formální prototypy uživatelského rozhraní, navigační mapy nebo opět jen skici a náčrty.

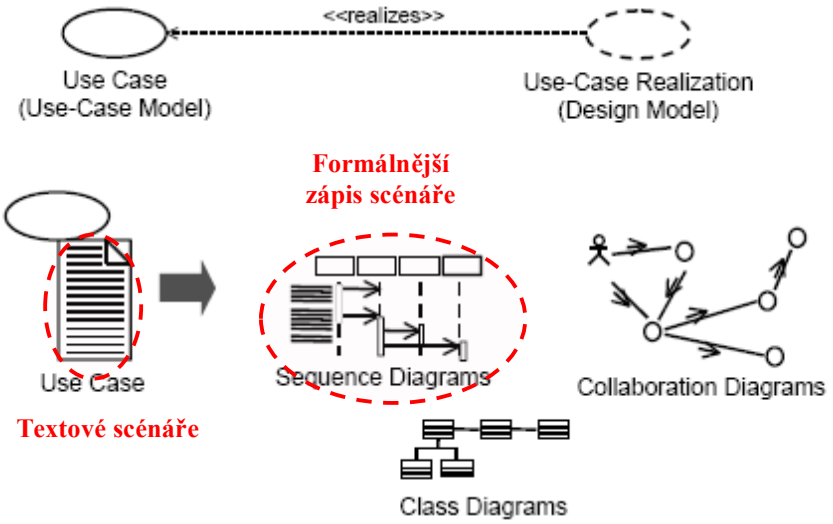


Obr. 6-11: Příklad navigační mapy webového rozhraní.

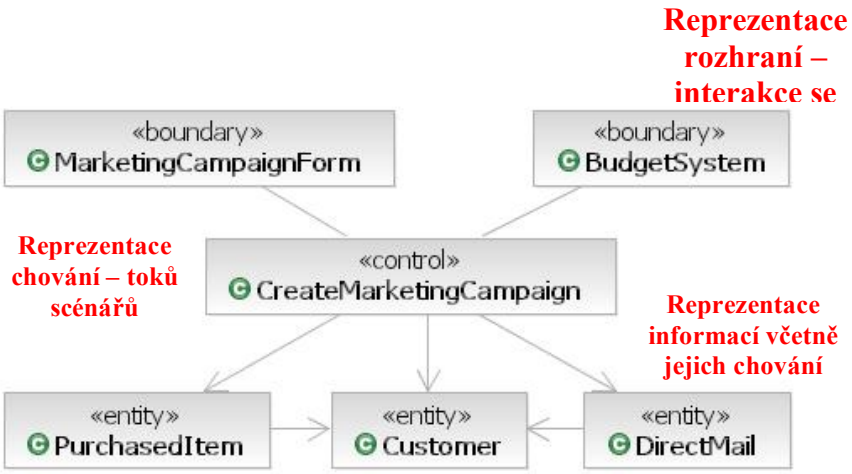


Obr. 6-12: Příklad náčrtku uživatelského rozhraní.

Jedním z hlavních kroků v rámci disciplíny Analysis and Design je tzv. **use case realizace** (use case realization). UC realizace popisuje, jakým způsobem je konkrétní use case realizován v návrhovém modelu z hlediska spolupracujících objektů. Use casy definují chování systému, objekty toto chování implementují. Za běhu spolu objekty spolupracují (komunikují), aby naplnily to, co je od systému očekáváno. Use case scénář může být chápán jako black-box pohled na systém a UC realizace jako white-box pohled ukazující, jak je daný UC vykonáván z pohledu interakcí mezi objekty a třídami. UC realizace tedy linkuje dohromady use casy z use case modely s třídami a vazbami návrhového modelu. Use case realizace specifikuje, jaké třídy je třeba vytvořit pro naplnění implementace daného use case. Pro use case realizace používáme UML sekvenční diagramy popisující kroky scénáře více formálněji (viz následující obrázek) a následný Entity-Control-Boundary model (viz Obr. 6-14) specifikující více odpovědnosti identifikovaných objektů.



Obr. 6-13: UC realizace

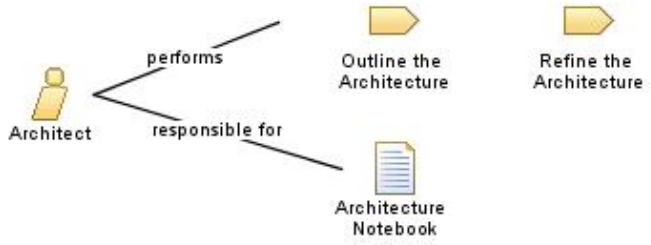


Obr. 6-14: Entity-Control-Boundary model (zdroj [OpenUP]).

Důležité artefakty, které vznikají v rámci této disciplíny, jsou následující:

- Analytický model (*analysis model*).
- Návrhový model (*design model*).
- *Software architecture document* – popisuje architekturu systému pomocí několika rozdílných pohledů (*view*) na systém, cílem je zachytit různé aspekty systému – statický pohled (vrstvy, komponenty, subsystémy), dynamický pohled (jak spolu tyto komunikují) či pohled rozmístění komponent/částí systému na HW komponenty.
- Rozhraní (*interface*) – specifikace rozhraní pro jednotlivé vrstvy a komponenty systému, jedná se o sadu operací bez detailů jejich implementace (pouze název, popis a požadované chování).
- Model nasazení (*deployment model*).
- Datový model (data model) – je-li nezbytně nutný, například pokud jej neřešíme pomocí frameworku typu ORM (objektově relační mapování objektů na relace relační databáze). Jak je zřejmé, tento model není natolik důležitý, jak studenti často vnímají, natož abychom s ním začínali celý návrh!

Forma těchto artefaktů, modelů či dokumentů je opět na týmu a zvolíme ji podle potřeby. Provádíme a vytváříme pouze dané artefakty a kroky, které nám přináší nějaký užitek. Například některé opakující se scénáře či jejich jednoduché varianty není třeba popisovat pomocí use case realizací.



Obr. 6-15: Role architekta podle OpenUP (zdroj [OpenUP]).

Důležité role disciplíny jsou tedy:

- Architekt – zodpovědný za architekturu systému a technická rozhodnutí ovlivňující celkový návrh a implementaci projektu.
- Designer – zodpovědný za detailní návrh přiřazené části systému, tento návrh musí být konzistentní s architekturou a musí využívat definovaných mechanismů.

Pohybujeme se v oblasti modelování, jednoduše se tedy může stát, že budeme modelovat a modelovat a vytvoříme spoustu bezcenných artefaktů, jejichž údržba nás navíc bude stát obrovské úsilí. Jak daleko tedy máme zajít? Do jakých detailů? Do jaké úrovně? Úroveň designu musí být dostatečná, abychom mohli daný systém jednoznačně implementovat, opět se tedy bude velmi lišit pro každý projekt. V některých projektech bude úroveň návrhu taková, že bude možné návrhové modely automaticky a systematicky přetransformovat do kódu. V jiných případech zase vytvoříme pouze jednoduché náčrtky, abychom si byli jisti, že programátor pochopil požadavky a je schopný je naimplementovat. Tato hranice záleží na znalostech a zkušenostech daného programátora, který bude přetvářet design model v kód, dále na složitosti návrhu a návrhových rizicích.

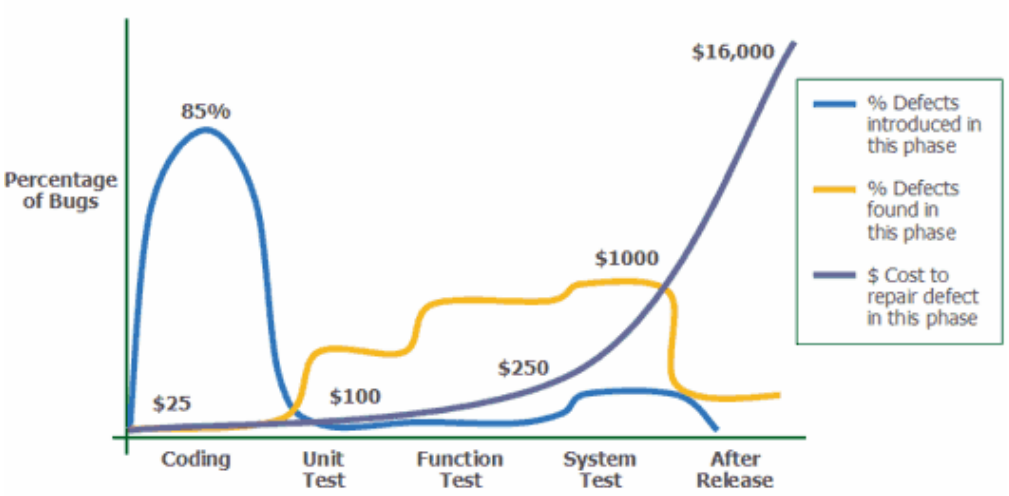
Více o modelech (UML) v samostatné kapitole, více také o aktivitách analýzy a návrhu viz kapitoly o Elaboration a Construction fázi.

6.4 Disciplína Implementation

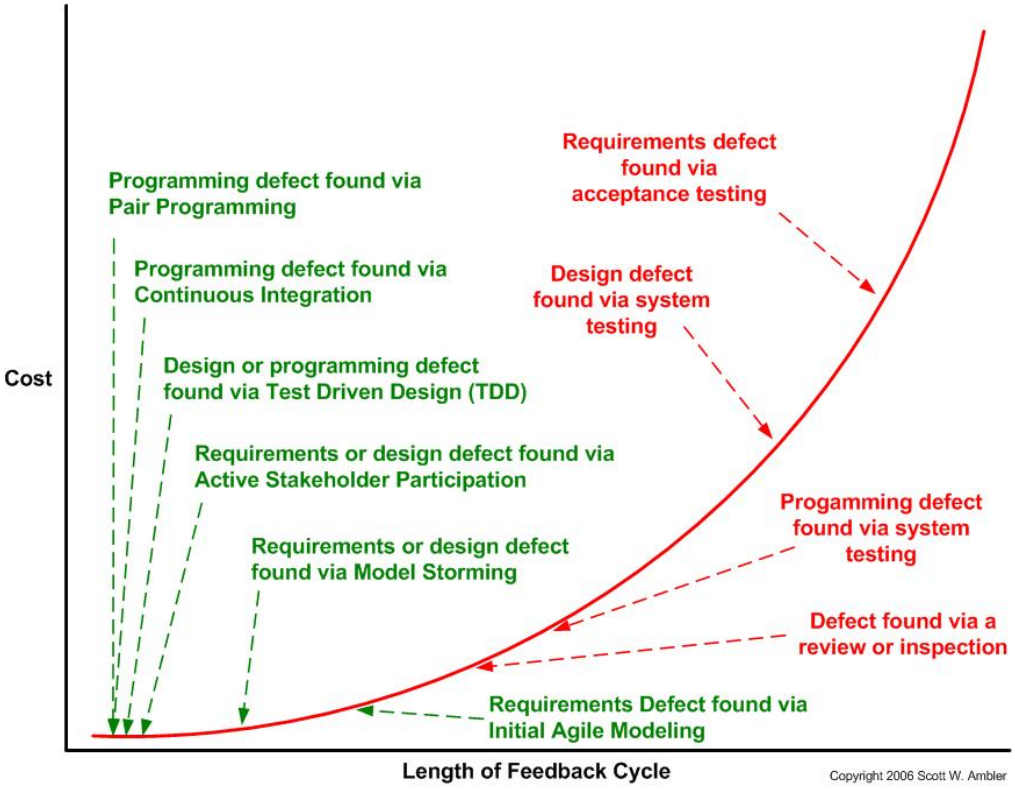
Tato disciplína je zaměřena nejen na vlastní psaní kódu! Detailně řečeno, Implementation znamená realizaci navržených subsystémů v jednotlivých vrstvách, implementaci návrhových elementů (zdrojové, binární, spustitelné soubory), unitové testování vyvinutých komponent[5], integrace samostatných vyvinutých částí do jednoho celku – spustitelného systému. Testování tříd a komponent je omezeno pouze na unitové testování. Systémové a integrační testování (tedy odstíněné od technologie) je předmětem disciplíny test.

[5] Ano, vývojáři testují svoji práci, ale implementačně na úrovni kódu, ne manuálními funkčními testy.

V této kapitole se tedy budeme zabývat hlavně implementací a možnostmi testování a předcházením chyb v disciplíně implementace, jelikož z následujícího obrázku je zřejmé, že největší procento chyb (85 %) je zaneseno právě v jejím průběhu.



Obr. 6-16: Procentuální rozložení zanesení chyb a jejich odhalení a cena v průběhu různých fází životního cyklu aplikace (zdroj [Ag06]).



Obr. 6-17: Cena chyby v průběhu životního cyklu aplikace (červená křivka) a techniky minimalizující cenu opravy chyby v různých fázích životního cyklu aplikace (zdroj [Am06]).

Obr. 6-17 zachycuje cenu chyby, respektive cenu jejího odstranění v průběhu životního cyklu aplikace. Je zřejmé, že tato cena exponenciálně roste čím více se blížíme ostrému provoze aplikace. Co to znamená? Jak to vysvětlíme?

Pokud udělá programátor chybu či zvolí nevhodný návrh (špatná technika, neuvolnění zdrojů, zapomenutá kontrola na null objekt, ...) a ta je ihned odhalena kolegou při párovém programování či zachycena unit testem, může být během několika minut odstraněna. Programátor zná kontext, má v paměti, co a proč dělal, má daný kód „rozdělaný“. Kdyby se tato chyba projevila až v ostrém provozu, bude cena a čas o dost vyšší. Uživatel by nemohl pracovat s aplikací (např. zpracovávat objednávky), což může podniku generovat ztráty, uživatel musí chybu reportovat a popsat, někdo na straně IT ji musí ohodnotit, přidělit k řešení, programátor ji musí odhalit a nalézt dané místo v kódu, pochopit strukturu kódu a jeho význam. Někdo další musí řešení otestovat, musí je schválit zákazník a poté je musí někdo nasadit do provozního prostředí. Čas a úsilí těchto lidí stojí peníze a někdo je musí zaplatit, ***proto je odhalení a opravení chyby v provozním prostředí o hodně dražší.*** Pokud se zaměříme na prevenci vnesení chyb, či jejich brzké odhalení již v průběhu implementace, můžeme významně snížit cenu údržby aplikace (což při mnoha letech provozu aplikace může činit i miliony korun) a současně zvýšit její kvalitu a spokojenost uživatelů.

Veškeré techniky, které přispívají k prevenci před zanesením defektů, se nazývají *„Defect preventing techniques“* [Kr06] a lze je rozdělit do tří základních kategorií, podle fáze vývoje, ve které je vykonáváme:

- Před psaním kódu:
  - bereme v potaz komplexnost návrhu pro danou aplikaci (nepotřebujeme EJB pro jednoduché fórum a naopak v PHP5 bychom asi elektronické bankovnictví nepsali), nejlepší cesta ověření proveditelnosti požadavků a návrhových rozhodnutí je psaní kódu;
  - dále bereme v potaz jednoduchost instalace, migrace, používání aplikace (změny vzhledu a ovládání, nové funkčnosti, dopad na integrované aplikace);
  - posledním důležitým uvážením je testovatelnost aplikace, debugging by měl být prováděn v průběhu psaní kódu, nejen při hledání chyby.
- Při psaní kódu:
  - defenzivní programování (popsáno níže);
  - mysleme při používání/testování aplikace stejně jako náš zákazník – zákazník použije aplikaci způsobem, který nás nikdy ani nenapadl, aktivní účast zákazníka při demonstracích, demu, pilotech stejně jako obecné generátory chyb používané v celé aplikaci (ne natvrdo psané řetězce) pomůže takové chyby identifikovat a odstranit;
  - inspekce kódu – je možné použít formální inspekce kódu (časově nejnáročnější), párové programování či jen kontrolu nástrojem vůči best practices pro danou technologii.
- Při testování kódu:
  - defenzivní testovací techniky jako statická analýza kódu pomocí nástrojů (pomáhá odstranit hlavně hraniční a null pointer výjimky, problémy s pamětí a dalšími zdroji), testování výkonnosti a identifikace úzkých míst;

- testovací „falešné“ a „zárodečné“ objekty – mocks a stubs – reprezentující chování některých částí systému či celých systémů; proto, abychom mohli otestovat chování určité spolupracující části;
- testy řízený vývoj (TDD) – popsán níže.

Cílem této disciplíny je tedy používaní velmi jednoduchých technik, které sníží na minimum zanesení chyby či přispějí k jejímu brzkému odhalení. Jedná se hlavně o následující techniky (více viz text Informační systémy 2):

- Defenzivní programování (*Defensive coding*),
- Párové programování a navrhování (*Pair Programming*),
- Unit testování a testy řízený vývoj (*Test Driven Development*),
- Neustálá integrace (*Continuous Integration*).

Disciplína implementace podle RUP/OpenUP představuje hlavně dva koncepty:

- Buildování – definice způsobu integrace a sestavení spustitelné aplikace nebo její části.
- Mapování návrhu na kód – popisuje způsob transformace návrhových modelů do zdrojového (spustitelného) kódu. Může se lišit podle použité technologie či architektury.

V rámci této disciplíny bychom měli v každé iteraci vykonávat následující:

- Plánování, který subsystém/scénář budeme implementovat a pořadí, v jakém budou integrovány do výsledného celku.
- Implementaci tříd a objektů podle návrhového modelu – což zahrnuje psaní kódu, unitových testů, linkování knihoven, kompilace, integrace komponent a spuštění.
- Implementaci záplat pro nalezené chyby a spuštění unit testů pro ověření, revizi a kontrolu podle konvencí kódu.
- A další nezbytné aktivity.

Obr. 6-18: Role Developer a minimální artefakty a prováděné aktivity (zdroj [OpenUP]).

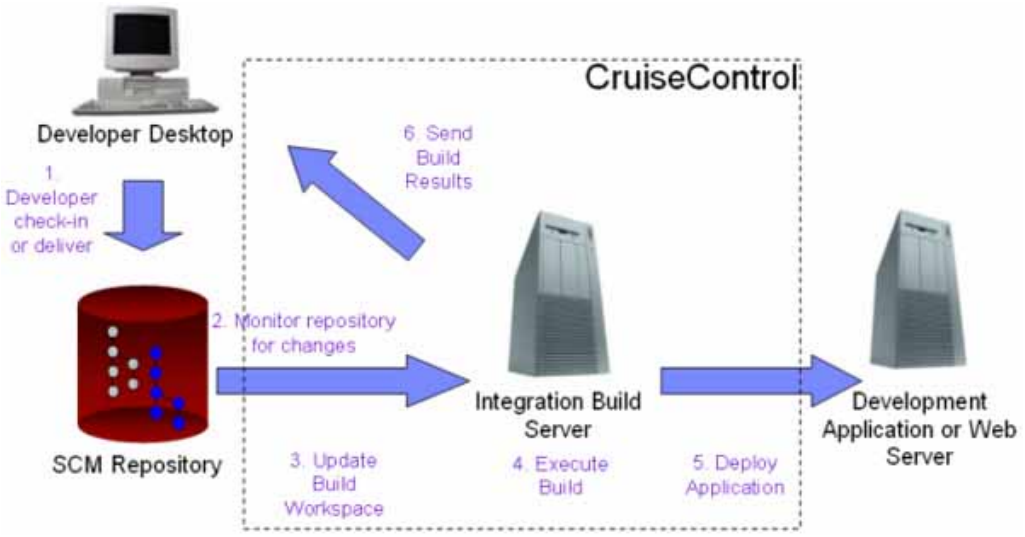
6.4.1 Buildování a integrace

Buildování je integrace a sestavení spustitelné aplikace nebo některé její části. Tento proces je v případě iterativního vývoje důležitým prvkem. Build slouží k neustálé demonstraci nových funkčností a také k demonstraci postupu na projektu. Každý build je pod správou konfigurací, abychom ho mohli v případě potřeby použít.

Pokud jsou naše iterace krátké a tudíž potřebujeme buildovat a testovat často, je automatizace tohoto procesu kritickým prvkem celého vývoje. Důležité je také integrovat a testovat jednotlivé části vyvinuté samostatnými vývojáři nebo týmy. V této souvislosti se často zmiňuje technika či koncept zvaný neustálá, častá integrace (*Continuous Integration* – CI), což je soubor známých praktik používaných při vývoji software, nyní velmi propagovaný agilními přístupy.

Martin Fowler a Matthew Foemmel definují neustálou integraci jako plně automatizované, opakované buildování, které zahrnuje také testování a je spouštěno několikrát denně. To umožňuje vývojářům denně integrovat svou práci a předejít tak integračním problémům či jejich dopad redukovat a hlavně získat rychlou zpětnou vazbu o kvalitě své práce.

Jednoduše lze říct, že neustálá integrace se skládá z repozitory pro všechny členy týmu obsahující (nejen) poslední kód a spustitelné soubory a také z automatického procesu buildování a testování, jenž může být spouštěn několikrát denně a je soběstačný, není třeba zásahu člověka. Je důležité říci, že ačkoliv neustálá integrace hodně závisí na nástrojích, není to jen použití nástrojů, ale spíše postoj k vývoji SW, který snižuje rizika plynoucí z pozdní integrace jednotlivých částí produktu.



Obr. 6-19: Neustálá integrace s pomocí nástroje Cruise Control.

Blíže a více o neustálé integraci viz například článek na Rational Edge[6] či množství článků od Martina Fowlera[7].

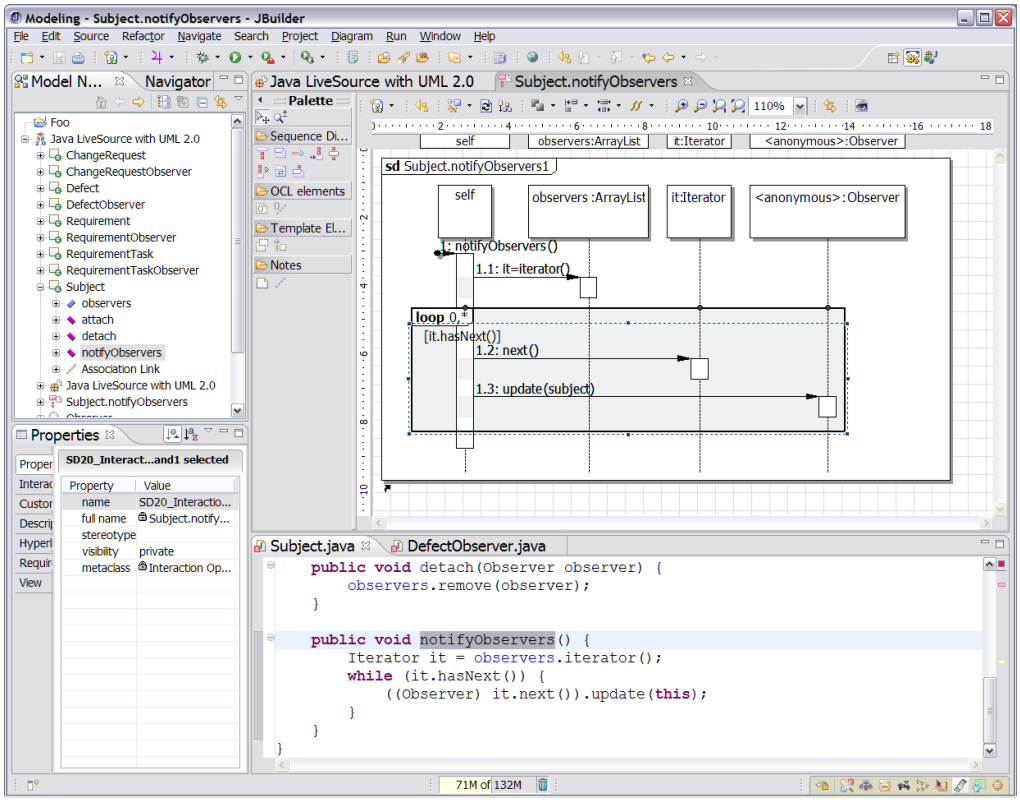
6.4.2 Mapování návrhu do kódu

Pro transformaci návrhu do zdrojového kódu jsou běžně využívány dva přístupy. My se zmíníme ještě o třetím, který začíná být také významným, hlavně však přispívá k vyšší kvalitě kódu. Jedná se o testy řízený vývoj (TDD).

[6] http://www.ibm.com/developerworks/rational/library/sep05/lee/
[7] http://www.martinfowler.com/articles/continuousIntegration.html

První z nich na úrovni návrhu vytvoří vysokoúrovňový návrhový model, podle kterého je poté přímo tvořen kód. Návrhové třídy většinou slouží jako hlavní třídy v kódu, k nimž jsou doprogramovány pomocné či obslužné třídy. Vysokoúrovňový návrhový model je poté udržován ručně. Stejný přístup může být aplikován také na uživatelské rozhraní a další vrstvy aplikace.

Druhým přístupem je tzv. round-trip engineering (adekvátní český překlad, který není paskvilem, se nepoužívá). S použitím automatizovaných nástrojů je vytvořen návrhový model až do té úrovně, kdy vlastně představuje vizuální reprezentaci kódu. Visuální reprezentace kódu a vlastní kód jsou synchronizovány právě pomocí nástrojů.



Obr. 6-20: Vizuální návrhový model provázaný s kódem v IDE.

Třetím způsobem, který doporučují nejen agilní přístupy, je tzv. testy řízený vývoj (TDD – Test Driven Development). Princip tohoto přístupu, který pochází z XP, je velmi jednoduchý. Nejdříve napíšeme unitový test a až poté vlastní kód. V průběhu psaní testu rozmýšlíme vlastní kód a většinou nás napadne několik řešení. Výhod má tento přístup několik:

- výběr nejvhodnějšího implementačního řešení – při psaní testů promyslíme několik variant, vybereme podle nás tu nejvhodnější,
- ihned po napsání zdrojového kódu jsme schopni ověřit funkčnost kódu unitovým testem,
- máme okamžitou zpětnou vazbu o funkčnosti a kvalitě kódu,
- v případě refaktoringu, implementace změny či rozšíření okamžitě vidíme, jestli jsme nezanesli do programu chybu.

Postup podle TDD je následující:

1) Vytvoříme nový test – test nemůže projít, jelikož neexistuje implementace, kterou má testovat; důležité je mít pochopení problematiky (požadavků) např. z detailních scénářů Use Casů.
2) Spuštění všech unit testů a zjištění, že nový nebyl úspěšný – ověření, že nový nemůže projít, jelikož ještě neexistuje kód, který má testovat; ověření jeho správnosti (odhalení případné chyby v testu).
3) Vytvoření nějakého kódu – napsání kódu za účelem úspěšného průchodu testem, nejedná se o finální implementaci!
4) Spuštění automatických testů a zjištění úspěšného průchodu novým testem – kód splňuje požadavky, můžeme začít psát finální implementaci.
5) Refaktoring kódu – zlepšení kvality kódu, jeho vyčištění, doplnění funkcionality, nahrazení kouzelných čísel či řetězců v kódu; zanesení chyby je kontrolováno spouštěním automatických testů.
6) Opakování postupu – iterativně tento postup opakujeme, jak přidáváme nové funkčnosti, opravujeme chyby či implementujeme změny.

Pro unitové testování je možné využít různé automatizované nástroje či frameworky. Nejznámějším je pravděpodobně xUnit, který existuje ve variantách pro různé programovací jazyky, např. původní verze SUnit pro Smalltalk, JUnit pro Javu, PHPUnit pro PHP, CUnit pro jazyk C apod.

6.4.3 Defenzivní programování

Defenzivní programování se v podstatě týká definice a následování týmových pravidel pro zápis zdrojového kódu aplikace. Příkladem těchto technik mohou být následující [Kr06]:

- Inicializace všech proměnných před jejich použitím.
- Konsistentní zápis návratových hodnot – častý problém při debuggingu je překrytí této části kódu.
- Použití pouze jednoho bodu opuštění každé procedury/metody – několik možných cest návratu je častým zdrojem problémů s uvolňováním zdrojů („Jak to, že nemám dostatek paměti?“) a nejasných návratových hodnot („Odkud se tady proboha bere ten null?“).
- Použití assert, který má smysl – jedná se o jednoduchou kontrolu podle zdravého rozumu („Je tato hodnota, kterou předpokládám true, opravdu true?“).
- Psaní čitelného kódu – jsme schopni pochopit za půl roku z daného kódu, co jsme tímto zápisem původně zamýšleli? Jsou jména všech proměnných a metod dostatečně popisná a reprezentují dané chování či vlastnost? Jsou používány dostatečné komentáře a to konzistentním způsobem (například metody komentujeme v definici rozhraní)?

Definicí a dodržováním těchto jednoduchých pravidel můžete skutečně předejít velkému množství chyb či přispět k jejich brzkému nalezení při debugování napsaného kódu. Společně s dalšími technikami jako je TDD, revize kódu či párové programování pak přispívá k mnohem kvalitnějšímu a lépe čitelnému kódu aplikace.

6.4.4 Konvence pro zápis kódu

Velmi důležitým bodem této disciplíny je také definice konvencí pro zápis kódu. Tato konvence popisuje strukturu kódu, pojmenování balíků kódu, způsob zápisu proměnných, konstant, překrývání metod, volání metod, způsob zápisu komentářů apod. Konvence je možné definovat pro každý projekt, vhodné je však používat obecné konvence definované výrobcem technologie nebo komunitou sdruženou kolem této technologie. Například konvence pro Java kód lze nalézt přímo na stránkách společnosti Oracle[8].

Příklad zápisu Java kódu podle konvencí

```
public class Class {

    // vsechny tridni a instanci atributy na zacatku definice tridy
    Connection m_con = null; // nezbytna inicializace
     /*
      * Comment
      */
       public String method() {
        int t_count = 0;// nezbytna inicializace
        String t_name = "";
        Boolean t_rozhod = true;

        if( t_name.equals("Adam") ) {
        } else {
        }
        if( t_rozhod == true ) {
        } else {
        }
       }
}
```

6.4.5 Prototypy

Důležitou náplní disciplíny Implementation je tvorba kódu, toto se týká také prototypů. Cílem prototypů je demonstrovat určité řešení, že jsme schopni použít danou technologii a připojit se pomocí ní k určité databázi či staršímu systému, zpracovat XML data, publikovat reporty apod. Je to tedy přímá cesta, jak brzy redukovat riziko. Prototypy vytváříme hlavně v případě zjištění/ověření:

- zda je produkt životaschopný na trhu,
- stability či výkonnosti klíčové technologie,
- pochopení požadavků,
- vzhled a použitelnost produktu.

Prototypy mohou mít rozdílné cíle a účel, podle nich pak definujeme několik typů. Podle zaměření prototypu – co prozkoumává, se jedná o prototypy chování (zkoumání specifického chování, algoritmu) a strukturální prototypy (zkoumání možností architektury či technologie). Podle toho, jak tento prototyp dále přežívá či nikoliv se bavíme o evolučních a ověřovacích prototypech. Ověřovací slouží pouze ověření myšlenky a poté jsou zahozeny. Kdežto evoluční jsou nadále rozvíjeny a stává se z nich finální systém.

[8] http://www.oracle.com/technetwork/java/codeconv-138413.html

6.5 Disciplína Test

Disciplína Test slouží jako poskytovatel služeb pro ostatní disciplíny. Testování je zaměřeno hlavně na ověřování a hodnocení kvality jednotlivých produktů. K tomu jsou využívány následující praktiky:

- Nalezení a dokumentace defektů v kvalitě software.
- Sdělování očekávané kvality.
- Verifikace a ověření předpokladů vytvořených v návrhu a specifikaci požadavků formou konkrétní demonstrace.
- Verifikace softwarového produktu podle návrhu.
- Ověření správné implementace požadavků zákazníka.

Narozdíl od ostatních disciplín zaměřených na úplnost (Requirements, Analysis & Design, Implementation), je disciplína Test zaměřena také na neúplnost, ptáme se: Jak můžeme tento software shodit? V jakých situacích, jakými kroky je možné způsobit nepředvídané chování software?

Testování bude probíhat jinak u systémů pro běžný provoz (uživatelé „pouze“ nemohou pracovat s aplikací) ***a jinak u kritických systémů*** (možnost ztráty lidských životů, vliv na ekonomiku, velké finanční ztráty), které řídí či obsluhují velmi nákladné (kosmonautika, zbraňové systémy, burza) ***či životně důležité systémy*** (letecký provoz, nemocnice).

6.5.1 Úrovně testování

V popisu předchozí disciplíny Implementation jsme zmínili unitové testování, které provádí samotní vývojáři na svém kódu, aby měli okamžitou zpětnou vazbu o kvalitě a chybovosti svého kódu. Ostatní testy, které ověřují integrované komponenty dohromady a funkčnost systému proti požadavkům zákazníka provádí testeři. Každý druh testu má jiný cíl, zaměření a může se provádět v jiné fázi životního cyklu.

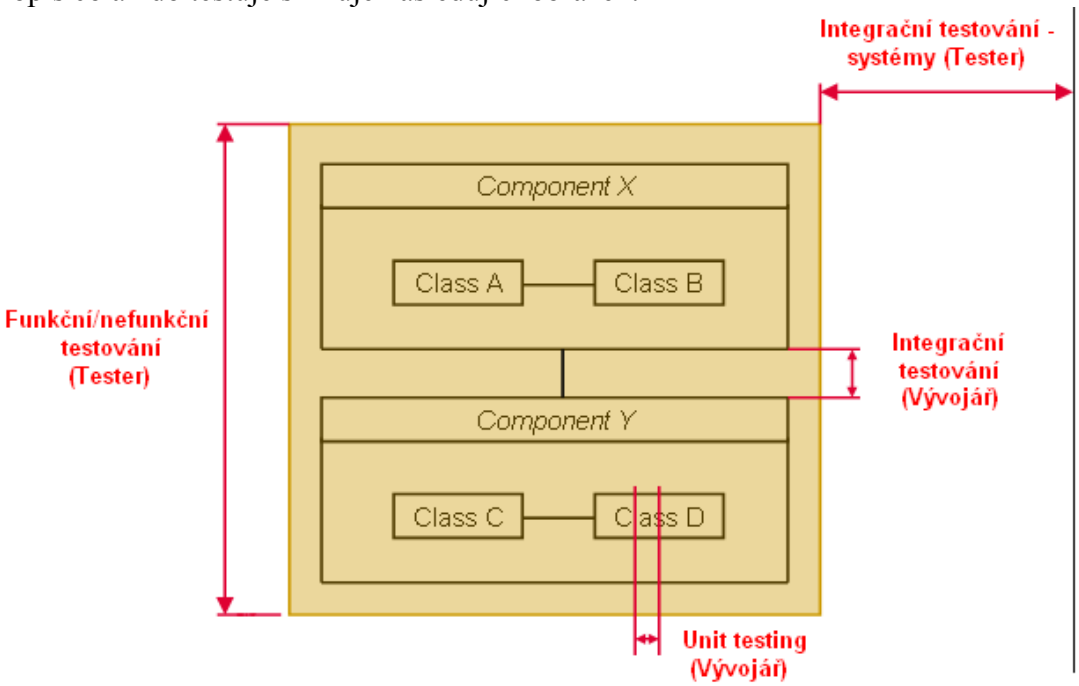
Jedním z možných a případných rozdělení druhů a variant testování může být následující (podle úrovně testování):

- Unitové – zaměřuje se na ověření nejmenších testovatelných jednotek systému, typicky je zaměřeno na komponenty a ověření jejich správných toků dat a procesů. Vykonání unit testu probíhá v rámci implementace (disciplína Implementace).
- Integrační – slouží k ověření, zda komponenty zahrnuté do implementačního modelu (balíček – package nebo subsystém) pracují a komunikují správně při provádění daného use case. Nedostatečná integrace bývá častým zdrojem problémů a selhání SW systémů. Vhodné je zahrnout testery i vývojáře (s rozdílnými strategiemi, aby se testování nepřekrývalo).
- Systémové – je zaměřeno na testování funkčnosti aplikace, většinou je prováděno s fungujícím systémem jako celkem, v případě iterativního vývoje je možné už od počátku vývoje pouze s určitými implementovanými funkčními celky.
- Akceptační – prováděno uživateli, jedná se o poslední testovací akci před vlastním nasazením software. Cílem akceptačního testování je

ověřit, zda je systém hotový a může být používán uživateli k naplnění cílů, pro které byl vytvořen.

Jak a co tedy máme testovat a co je čí náplní? Náplní implementace komponenty, třídy je i její testování. Testujeme chování nejmenší jednotky, pro simulaci okolních jednotek (databáze, UI, jiné komponenty či systémy) používáme tzv. stubs a mocks. Jedná se o jednoduché třídy či zásobníky dat (nebo naopak rozsáhlejší implementace imitující chování podle vstupních dat), které vrací předdefinované sekvence dat a tím imitují očekávané chování okolí.

Cílem je otestovat na těchto datech chování vlastní jednotky, komponenty. Následně testujeme rozhraní a chování mezi jednotkami a komponentami. Posledním krokem je pak testování funkční (zda systém dělá co má) a nefunkční (zda jsou technické parametry dostatečné – dostupnost, počty zpracovaných transakcí za vteřinu, rychlost zpracování jednoho dotazu, přítulnost a použitelnost UI a použití standardních ovládacích prvků apod.). Popis co a kdo testuje shrnuje následující obrázek.



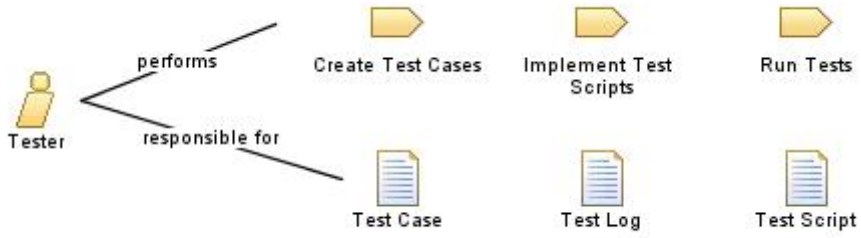
Obr. 6-21: Předmět testování

Připomeňme, proč je dobré aplikaci testovat a proč dělat už i unitové testy (nejsou předmětem Test disciplíny nýbrž Implementation disciplíny, jelikož je píší a vykonávají sami vývojáři). V průběhu životního cyklu vývoje software prochází software evolučním vývojem, nejvíce kódu je napsáno v Construction fázi. Proto zde vzniká i nejvíce chyb (až 85%, viz obrázek Obr. 6-16). Pokud chyby odhalíme hned v této fázi, je jejich odstranění relativně levné (jeden z důvodů, proč se vyplatí psát unit testy). Pokud však chybu odhalíme až při provozu aplikace, její cena exponenciálně roste. ***V krátkodobém hledisku je tedy psaní testů náklad, ale v dlouhodobém*** (rozuměj celý životní cyklus IS zahrnující i provoz a údržbu) ***se jednoznačně vyplatí, zvlášť ve srovnání s cenou chyby v produkčním prostředí***.

Tato kapitola se točí hlavně kolem kvality a slovo ***kvalita*** také často zmiňujeme, co si tedy pod tímto pojmem představit? Kvalita systému je často chápána a zmiňována pouze v kontextu bezchybného systému. Důležité je ale zmínit, že je sice pěkné, pokud máme nějakou věc, která je bez chyb, ale pokud nedělá to, co potřebuje či chceme, je nám to vlastně k ničemu, stejně jako produkt plných chyb. ***Kvalitní systém tedy bude systém s minimem chyb, který vykonává přesně to, co zákazník potřebuje***. Cílem testování je pak ověřit kvalitu produktu a komponent, ze kterých se produkt skládá, k čemuž používáme definovaná akceptační kritéria.

Pokud se bavíme o kvalitě a testování v kontextu RUP, je třeba zmínit F princip, který (kromě jiného) říká, že každý je odpovědný za kvalitu. Není možné, aby testeři jediní korigovali špatnou práci analytiků, návrhářů a programátorů. Pouze jejich snažením se produkt kvalitním nestane. Pokud není produkt navrhován s ohledem na kvalitu od začátku, není možné ji „tam přidat silným tlakem managementu či zákazníka později“. Je tedy třeba zajistit, aby všechny role přispívali k výslednému kvalitnímu produktu od počátku jeho vývoje. ***Cílem testovacích rolí tedy není zajistit kvalitu, ale ohodnotit ji!*** Dále pak také poskytovat častou zpětnou vazbu ostatním rolím, aby bylo možné problémy s kvalitou odstranit za rozumnou cenu v rozumném čase.

Testování tedy není jedinečná aktivita či fáze testování kvality produktu v průběhu projektu. Vývojáři by měli dostávat včasnou zpětnou vazbu o kvalitě produktu a také sami určitý způsob takové vazby využívat (unitové testování), proto musí testování probíhat v průběhu celého životního cyklu. Měli bychom testovat již rané prototypy, stabilitu a výkonnost architektury v době, dokud je ještě jednoduché toto opravit a dokud na to máme čas. Na konci samozřejmě musíme ještě také testovat, zda je produkt opravdu připraven k doručení zákazníkovi. Opět tedy připomínáme aplikaci F principu – za kvalitu výsledného produktu je odpovědný každý člen týmu a přispívá k ní v průběhu celého projektu.



Obr. 6-22: Aktivity a artefakty přiřazené roli Tester disciplíny Test (zdroj [OpenUP]).

Důležitým pojmem v případě iterativního vývoje je ***regresní testování***. Jedná se o strategii testování, kdy dříve provedené testy provádíme opět na nové verzi aplikace s cílem zajistit, aby kvalita produktu nešla dolů, nesnížila se díky přidání nových funkčností. Cílem regresního testování je tedy:

- zajistit, že dříve odhalené chyby jsou již odstraněny,
- zajistit, že změny provedené v kódu nezanesly nové chyby či se znovu neobjevily chyby předchozí.

Artefakty Test disciplíny jsou následující:

- Testovací scénář (*test case*) – množina testovacích dat, podmínek a postupu provedení testu a předpokládaných výsledků testů pro specifické cíle; test case může být derivován z use case, návrhových dokumentů či z vlastního kódu aplikace.
- Test skript – počítačově zpracovatelný sled testovacích procedur, který je možné automatizovat.
- Testovací třídy a komponenty (*mocks* a *stubs*) – třídy a komponenty realizující navržené testy, zahrnují také ovladače a různé částečné implementace (*mocks* a *stubs*) imitující části aplikace či jiné spolupracující systémy, které ještě nejsou naprogramovány, ale je třeba je mít k dispozici k otestování naší části aplikace.

Důležité role této disciplíny jsou:

- Test designer – odpovědný za plánování, návrh, implementaci a vyhodnocení testování; má tedy také slovo při výběru technologie pro daný projekt, jelikož tato technologie musí být testovatelná, musí existovat nástroje podporující automatizaci testů apod.
- Tester – odpovědný za provádění systémových testů, to zahrnuje nastavení prostředí a testů a jejich spouštění, vyhodnocení testů a jejich výsledků, obnovení po chybách, zapisování požadavků na změny.

Testovací scénáře a skripty vznikají již v raných fázích každé iterace. Test analytik, Test designer (návrhář testů) a Tester spolupracují s analytiky na pochopení požadavků (musí mít pochopení toho, co má systém dělat, když jej budou testovat) a formě jejich testování (co, jakým způsobem a v jakém rozsahu). Cílem je rychlá zpětná vazba pro vývojáře v rámci iterace, znova použitelnost use case jako test case a automatizace testovacích skriptů. Příklad znovupoužití use case a díky tomu zjednodušení test casu ukazuje následující příklad test case:

Use case	Scénář	Data a podmínky		
		Hodiny	Uživatel	Výstup
Reportuj čas	BF	5	user	OK
Reportuj čas	AF#1	-4	user	Chybové hlášení
...	...	...	...	...

Tabulka 6-3: Testovací scénář a znovupoužití use case.

Na závěr zmíníme, že disciplína Test se zabývá verifikací (*verification*) systému. Verifikace znamená ověření aplikace vůči specifikaci, zda implementuje to, co je definováno ve formě požadavků. Neřešíme validaci, validace se zabývá tím, zda aplikace přináší uživateli užitek.

6.6 Disciplína Configuration and Change Management

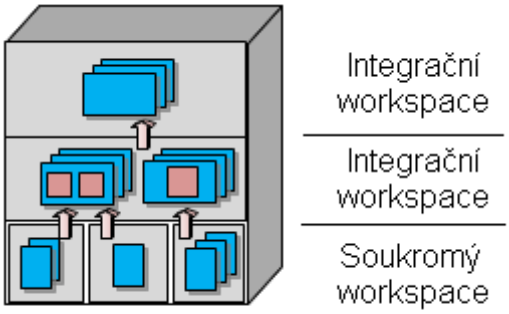
Configuration (správa konfigurací) a *Change Management* (změnové řízení) jsou velmi důležité disciplíny, které se zabývají správou artefaktů (zdrojové kódy, modely, dokumentace, sestavená aplikace či její části, konfigurační soubory) spojených s vývojem software a jejich následnou změnou. Cílem této

disciplíny je sledovat a udržovat integritu vyvíjejícího se projektu, resp. jeho artefaktů. Toto je kritické zvláště u iterativních projektů, kdy je vytvářeno a v průběhu iterací měněno spoustu artefaktů jako jsou modely, samotný kód, test casy, use casy, test skripty. Každý člen týmu musí ale být schopný vždy nalézt daný artefakt, jeho konkrétní verzi, či procházet historii z důvodu porozumění změnám. Současně je třeba zajistit správu nových požadavků, požadavků na změny a jejich konzistentní implementaci v rámci všech artefaktů. Posledním cílem je dodání informací o statusu jednotlivých artefaktů a metrik pro potřeby projektového řízení.

V kapitole se budeme zabývat dvěma částmi:

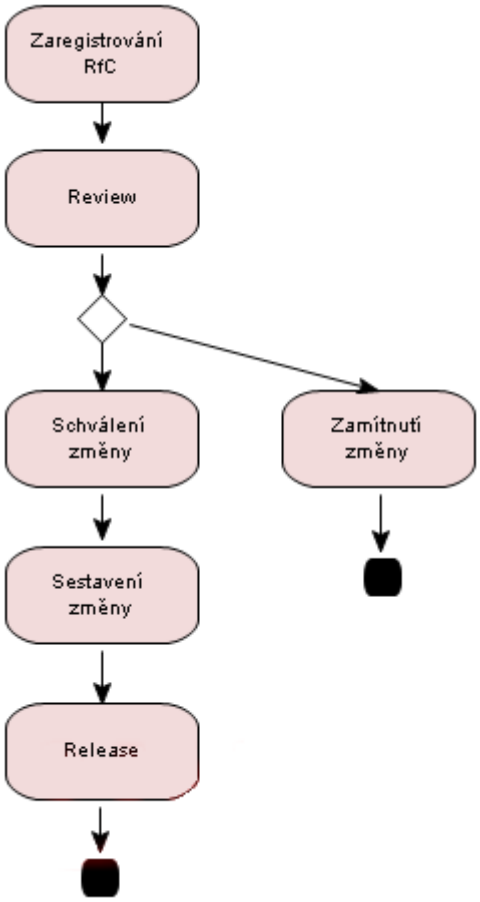
- Správa konfigurací (*Configuration management*) – správa konfigurací, verzí a historie změn jednotlivých artefaktů projektu; artefakty jsou na sobě závislé – například modifikací jednoho artefaktu musí být znovu sestaven build (automaticky či ručně).
- Změnové řízení (*Change management*) – je proces pro registraci, ohodnocení, schválení či zamítnutí a implementování změn; požadavek na změnu (*RfC – Request for Change*) je popsaný návrh změny jednoho nebo více artefaktů; RfC může být vyvolán z mnoha důvodů:
    - oprava chyby,
    - zlepšení kvality produktu (výkonnost, použitelnost),
    - přidání nového požadavku.

Workspace, česky bychom mohli říct asi privátní pracovní prostor, poskytuje jednotlivým vývojářům či malým týmům prostředí, ve kterém mohou pracovat jako by pracovali izolovaně, odděleně. V rámci tohoto prostoru mají samozřejmě vývojáři přístup ke všem potřebným artefaktům. Workspace tedy slouží pro základní vývoj malých dílků a pro jejich postupnou integraci. Výsledkem integrace několika workspace je další inkrement aplikace.



Obr. 6-23: Skládání jednotlivých workspace do výsledné aplikace.

Požadavek na změnu prochází v rámci změnového řízení (*Change Management*) určitým životním cyklem a jeho stavy se mění. Při změně každého stavu jsou doplňovány specifické informace jako důvod pro změnu, dopad změny (které artefakty a jak moc se změní), dopad na návrh/architekturu, odhad ceny změny a doby implementace. Na základě analýzy dopadu změny můžeme požadavek schválit či zamítnout, schválené je třeba podle priorit naplánovat do vývoje.



Obr. 6-24: Příklad životního cyklu požadavku na změnu, resp. změny.

Z výše zmíněného zaměření této disciplíny je zřejmé, co je její náplní, jaké jsou jednotlivé aktivity:

- Plánování konfigurací projektu a řízení změn – konfigurační plán (*CM plan*) obsahuje základní body a aktivity: popisuje nezbytné procedury a politiky, jmenné konvence, zabezpečení přístupů, archivování a další; CM plán také obsahuje popis procesu pro registrování a řízení změn (je třeba informovat zúčastněné, znát náklady a dobu implementace).
- Vytvoření CM prostředí – CM prostředí by mělo napomáhat a usnadňovat vývoj produktu (poskytnout přístup ke správným artefaktům podle potřeby, zálohovat je umožňovat integraci, znova použitelnost); cílem této aktivity je vytvoření takového prostředí.
- Změna a doručení konfiguračních položek/artefaktů – pomocí workspace může každý člen týmu přistupovat k jednotlivým artefaktům projektu a měnit je; doručení, kontrola a integrace těchto změn probíhá v integračním workspace; sestavení produktu a vytváření verzí probíhá taktéž nad tímto workspacem a jsou poté k dispozici vývojovým týmům.
- Správa releasů a popisů verzí (*baseline*) – baselines se vytváří většinou na konci iterací, projektu či projektových milníků, většinou tedy vždy při doručení zákazníkovi (při reportované chybě se můžeme vždy vrátit na původní baseline a chybu opravit).

- Správa požadavků na změnu (*change request*) – cílem standardního, dokumentovaného procesu změnového řízení je konzistentní implementace změn, informování zúčastněných na projektu o stavu produktu a jeho změnách a v neposlední řadě o cenovém a časovém dopadu těchto změn.

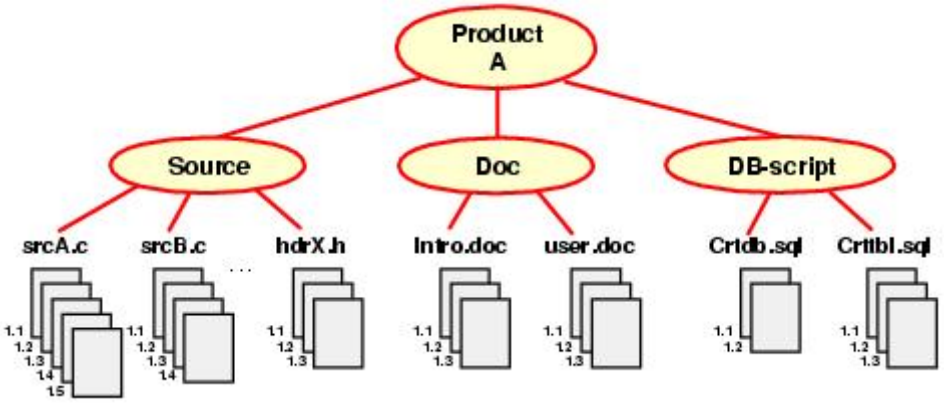
Následuje přehled důležitých rolí a artefaktů (podle RUP/OpenUP), které vystupují ve výše zmíněných aktivitách. Začněme rolemi:

- *Configuration manager* (manažer konfigurací) – odpovědný za nastavení struktury produktu v CM systému (IBM Rational Clear Case, SVN, CVS, ...).
- *Change control manager* (manažer změn) – dohlíží na proces řízení změn a přijímají/zamítají změny; role je většinou hrána tzv. CCB (změnový výbor – *Change Control Board*), který se skládá ze zástupců všech zainteresovaných stran (uživatelé, vývojáři, support, manažeři, sponzoři, ...).
- Integrátor – akceptuje změny v integračním prostředí a sestavuje produkt.

Artefakty:

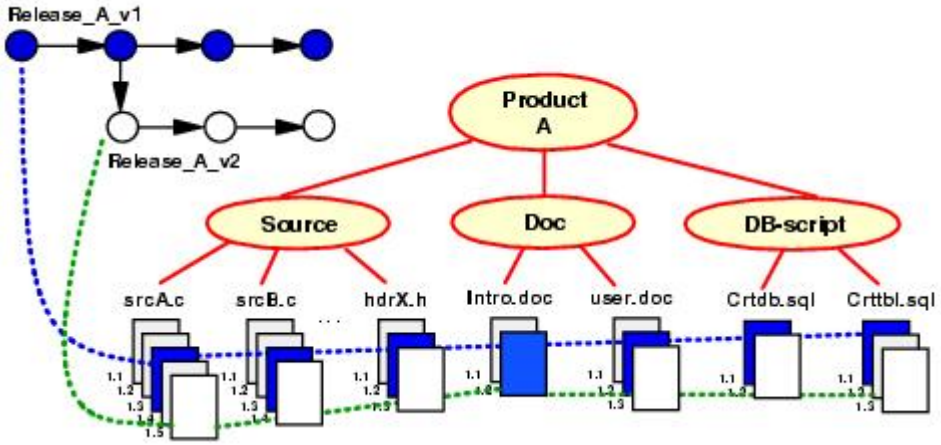
- *CM plan* (konfigurační plán) – popisuje praktiky a politiky projektu týkající se verzování, workspace, procesu změn, sestavování buildů a releasů; plán definuje také odpovědnosti a pravidla CCB (změnového výboru).
- *Change request* (požadavek na změnu) – může být rozličných druhů (defekt, změna v požadavku, dokumentaci) a pro každou změnu bychom měli identifikovat důvod či účel a také ohodnotit její dopad a cenu/dobu implementace; je-li změna implementována, mění se průběžně její stav.
- *Repository* – zabezpečené týmové úložiště všech artefaktů projektu spravované CM nástrojem (udržující různé verze artefaktů a historii změn).
- Další možné: implementační model, metriky a status report.

Následující obrázek ukazuje příklad typické konfigurace produktu. Produkt A obsahuje několik adresářů, kdy každý adresář pak obsahuje několik verzí různých souborů.



Obr. 6-25: Typická konfigurace SW produktu (zdroj [IRB1]).

Release je specifická konfigurace produktu dekující podmnožinu/výběr jedné verze z různých existujících verzí těchto produktů (viz následující obrázek). Jedná se o logickou organizaci či mapování artefaktů aplikace A či produktu A. Fyzická struktura a uložení artefaktů se nijak nemění. Release je tedy určitý filtr, který zahrnuje artefakty, které musí být sestaveny, testovány a doručeny společně. Samozřejmě, jeden artefakt může být součástí několika releasů.

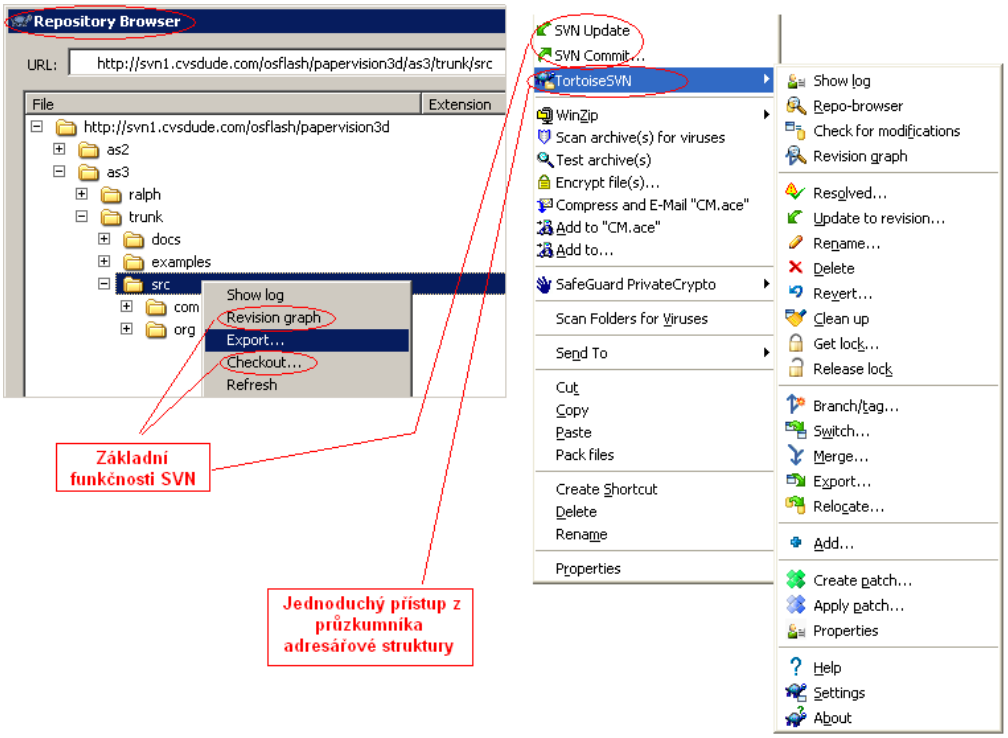


Obr. 6-26: Release SW produktu (zdroj [IRB1]).

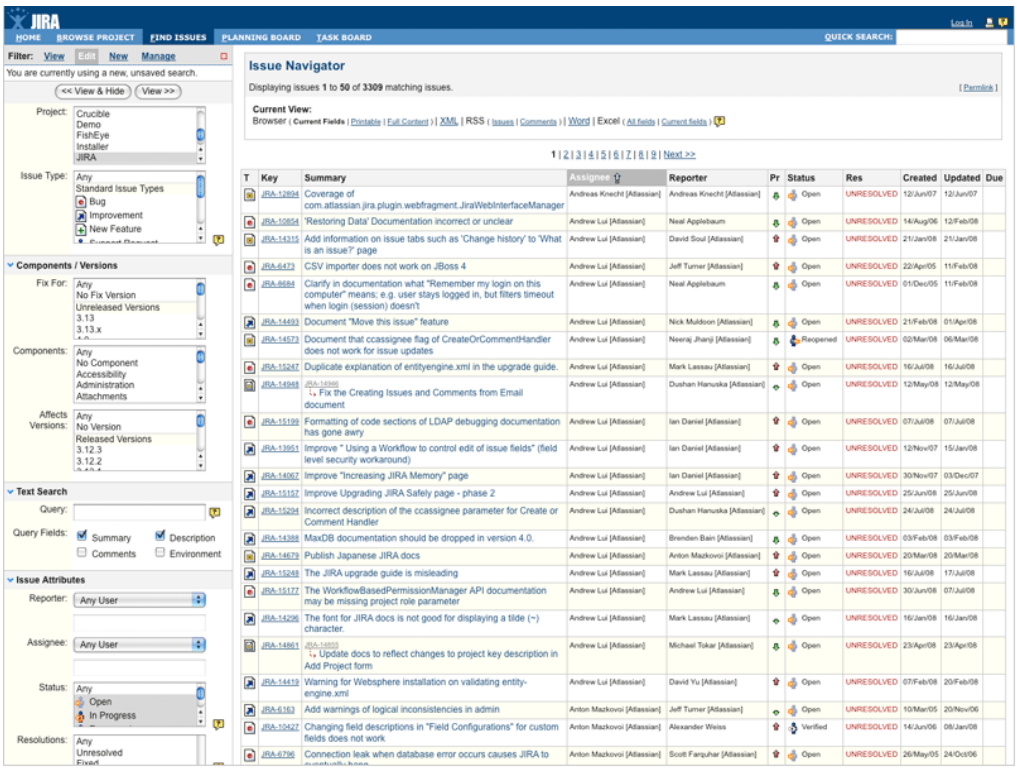
6.6.1 Nástroje

Podpora nástroji je v případě evidování a implementace změn a také sestavování aplikace, buildů a releasů kritická. Komplexnost daného procesu či aktivit roste exponenciálně s tím, jak roste velikost týmu, produktu, resp. počet artefaktů, geografická vzdálenost týmů či jak se blíží termín jednoho z releasů. Ruční sestavování či vedení evidencí je značně chybové, proto v tomto případě můžeme hodně získat díky automatizaci. V rámci této disciplíny mluvíme především o:

- Repository pro uložení artefaktů a umožnění paralelní práce programátorů (CVS, SVN, IBM Rational ClearCase) + možnost napojení na build mechanismus (Ant, Maven, ClearCase).
- Podpoře změnového řízení pomocí nástrojů pro evidování změn a defektů (issue tracking tool) jako jsou Jira, IBM Rational ClearQuest.



Obr. 6-27: SVN prohlížeč repository a možné akce.

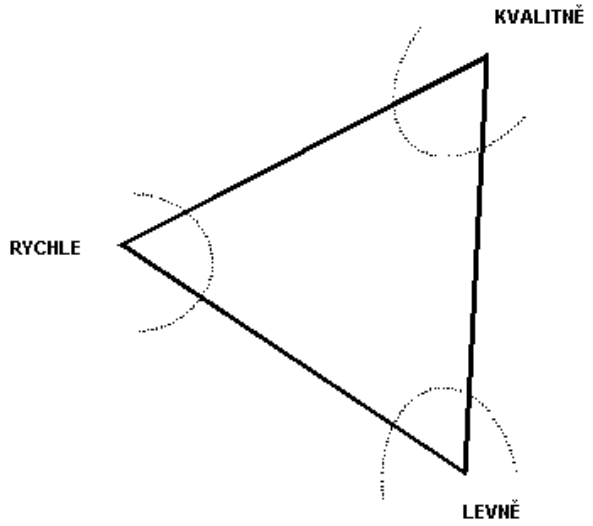


Obr. 6-28: Nástroj Jira pro evidování a sledování změn, požadavků a defektů.

6.7 Disciplína Project Management

Na úvod stručně připomeneme, co to vlastně projekt je a jaké základní fáze má disciplína projektového řízení, ať již se jedná o stavbu domu, strojírenskou výrobu či tvorbu software. Projekt je plánovaná, řízená, časově ohraničená skupina činností, která má dané vstupy a výstupy a spotřebovává určité zdroje (lidské, technologické, finanční). Definice PMBOK[9] říká, že projekt je dočasné úsilí s cílem vytvořit unikátní produkt nebo službu.

Z obou definic vyplývá, ***že bez přesně stanovených časových hledisek*** (počátek, konec) ***a jasných výstupů (produkty, služby) se nejedná o projekt!*** Nedefinování nebo pouze vágní definice těchto aspektů bývá častou chybou projektů. Při řízení projektů musí být brán v potaz tzv. magický trojúhelník (viz Obr. 6-29), 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.



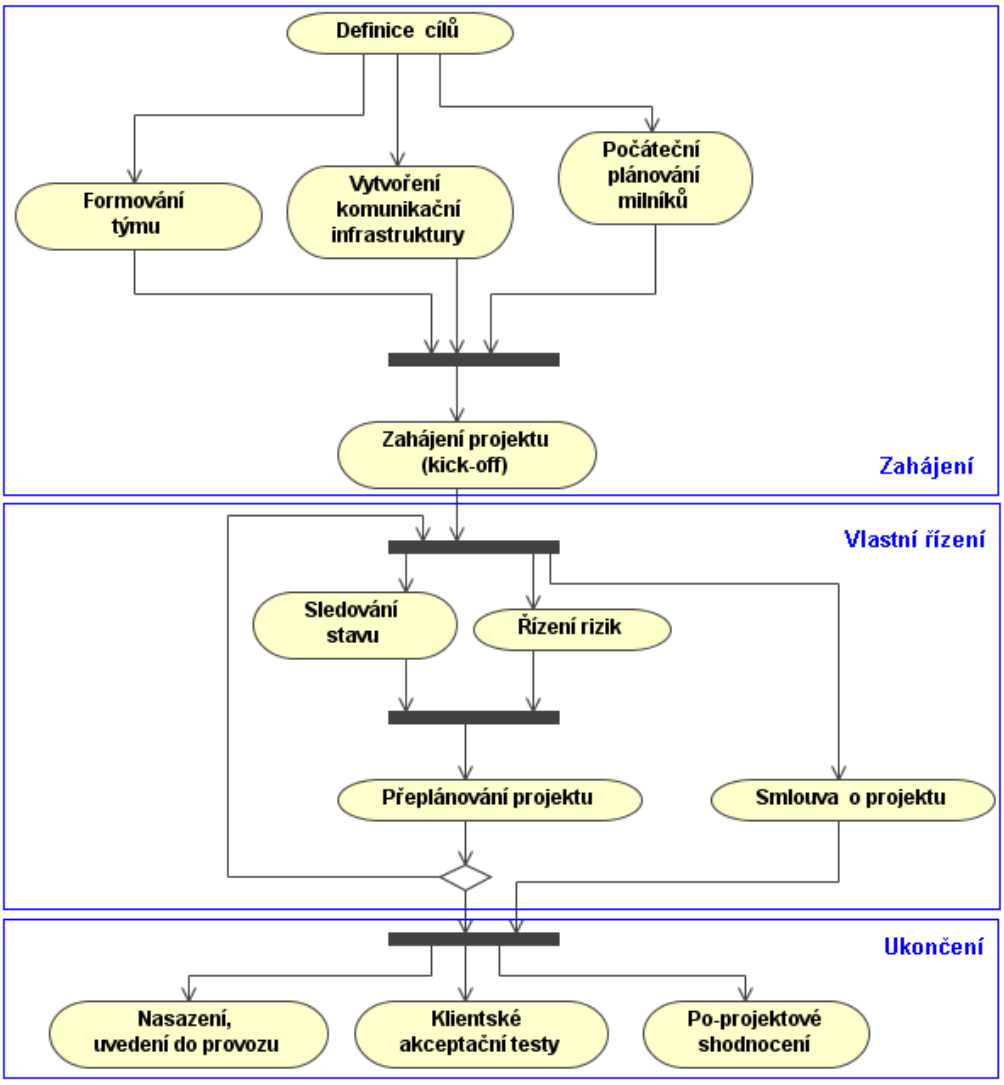
Obr. 6-29: Magický trojúhelník kvality: kvalita je výsledkem těchto tří hran.

Řídící rovinu projektu lze rozdělit do několika fází, jak ukazuje následující obrázek. Jedná se o tyto obecné fáze:

1. Příprava a plánování projektu – definice cílů projektu, ustavení týmů, definice komunikačních kanálů, počáteční plánování milníků, alokace zdrojů, výběr nástrojů a prostředků pro realizaci projektu, schválení plánu projektu. Obsahuje také zahájení projektu – tzv. kick-off meeting.
2. Řízení postupu projektu – řízení projektového týmu vedoucím projektu, koordinace projektového týmu a subdodavatelů, školení, provedení předmětných činností a aktivit, změnová řízení, testování, návrhy na změny.

[9] PMBOK je celosvětově uznávaná procesně orientovaná metodika řízení projektů

3. Vyhodnocení a závěr projektu - monitorování a vyhodnocení dosažených výsledků a splnění měřitelných kritérií, podání hodnotící zprávy zadavateli a projektovému týmu, analýza vyskytnuvších se problémů projektu (zdroje problémů, postupy pro jejich odstranění). Součástí je také předání a akceptace – předání výstupů jednotlivých etap, jejich akceptace zadavatelem a zkušební provoz, předání nezbytné dokumentace.



Obr. 6-30: Činnosti a fáze řízení projektů.

6.7.1 Projektové řízení a RUP/OpenUP

Co o této disciplíně říká RUP respektive OpenUP? Disciplína projektového řízení (*Project Management*) je nejdůležitější pro pochopení iterativního vývoje. ***Základním konceptem je iterace a dvou úrovňové plánování***. Jak již víme, projekt je sled iterací, v nichž tým tvořený vývojáři, analytiky, architekty, testery produkuje hmatatelný výstup ve formě spustitelné aplikace. Vždy v každé iteraci vytvoří tým jednu nebo více kompletních funkčností, které nabaluje na předchozí. Pro celý projekt vytváříme mapu postupu s cíli a milníky, neobsahuje detailní cíle ani aktivity. Detailní cíle a aktivity vytváříme pro každou iteraci zvlášť.

Projektové řízení v případě software je podle RUP balancování mezi protichůdnými cíli projektu, umění řízení rizik a překonávání omezení projektu s cílem doručit produkt, který naplňuje očekávání zákazníka (ten, kdo platí) a koncových uživatelů.

Cílem projektového řízení v RUP/OpenUP je:

- Poskytnout framework pro řízení softwarově orientovaných projektů.
- Poskytnout praktické průvodce pro plánování, obsazení pozic, vykonávání a monitorování projektu.
- Poskytnout nástroj pro řízení rizik.

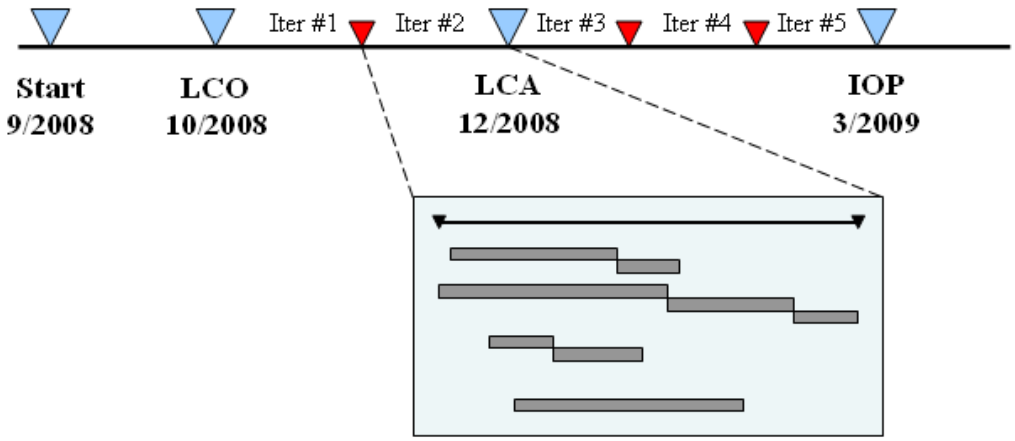
Je třeba si uvědomit, že smyslem PM disciplíny v RUP/OpenUP je zaměření se hlavně na produkování a doručení hodnotného software zákazníkovi. Proto jsou záměrně vynechány některé oblasti řízení projektů jako je řízení lidských zdrojů (najímání, tréninky), řízení rozpočtů (definice, alokace), řízení kontraktů s dodavateli a zákazníky[10].

Základních problémů při realizaci iterativního řízení projektu je hned několik: Kolik iterací potřebuji? Jak dlouhé by měly být? Jak určím, co má být obsahem? A spousta dalších praktických otázek. Kolik iterací v které fázi projektu budeme mít záleží na doméně, schopnostech a znalostech týmu, stabilnosti prostředí (jak moc se požadavky mění) a dalších aspektech, viz rozložení prací popsané v kapitole 5.2.

V úvodních kapitolách tohoto textu jsme si řekli, jaké jsou problémy SW projektů a proč detailní plánování SW projektů na jeho počátku nefunguje. Proto, abychom vytvořili realistický plán na počátku projektu, je třeba mít pochopení toho, co máme vytvářet, mít stabilní požadavky a stabilní architekturu a také nějakou znalost, zkušenost (podobný projekt), ze které můžeme derivovat jednotlivé detailní úkoly. Problémem je, že je těžké naplánovat Petrovi tvorbu modulu grafů na týden 37, když o takovém modulu nemáme ani ponětí. V průběhu projektu se učíme, zákazník také, technologie se může změnit, stejně jako trh a s ním i požadavky zákazníka. Jak tedy tyto problémy řeší RUP? Základním konceptem jsou ***dvě úrovně plánování***:

- Hrubý, vysokoúrovňový plán pro celý projekt či významné části (***Project plan***, ***Phase plans***) – existuje pouze jeden pro celý projekt a obsahuje základní údaje jako cíle iterací, data milníků, potřebu lidských zdrojů v čase, data konců iterací (demonstrací aplikace). Ačkoliv je tento plán vytvořen na začátku Inception fáze, je ***pravidelně aktualizován***, kdykoliv je to třeba!
- Série detailních plánů pro krátké časové úseky (***iterační plány***) – může trochu připomínat klasický podrobný plán (Ganttův graf), jejich obsahem je definice podrobných cílů a úkolů nutných k jejich dosažení, přiřazení úkolů členům týmu, data jednotlivých buildů (když ne denní buildy), data počátku a konce a také evaluační kritéria iterace.

[10] Více o řízení rizik při vývoji software viz autorův článek v Systémové integraci 1/2009 (http://www.cssi.cz/cssi/riziky-rizeny-vyvoj-software nebo http://www.differ.cz/?p=84).



Obr. 6-31: Hrubý plán (road mapa – projektový plán) a detailní plán (iterační plán).

Více se touto problematikou plánování zabývá kapitola 5, prakticky pak učební text Ročníkový projekt.

Jak bylo zmíněno v textu již několikrát, velmi důležitým konceptem v RUP/OpenUP, potažmo v disciplíně Project Management je řízení rizik. Proces vývoje software je založen hlavně na *známých* aspektech (tvorba plánu, definice a přiřazení úkolů). *Neznámými* aspekty se zabývá právě řízení rizik (tzv. *Risk Management*). Ještě připomeneme, co je to riziko. Riziko je v softwarovém procesu *událost*, jejíž výskyt může zabránit úspěšnému doručení software v dohodnutém čase. Rizika jsou většinou nejistá, či dokonce neznámá, proto je nutné se jimi zabývat. Důležité je věnovat se akcím na snížení rizika a také definovat eventuelní plán pro případ, že by riziko opravdu nastalo.

Řízení rizik (a risk driven approach) je zaměřeno hlavně na snížení či úplné odstranění rizik co nejdříve v projektu, když máme ještě volnější ruce a můžeme například vyměnit technologii, člena týmu, či změnit architekturu a způsob komunikace s externími programy. Koncept *„make your hands dirty“* je hlavní, který je aplikujeme. Říká, že nestačí riziko pouze identifikovat a sledovat, cílem je opravdu něco udělat: vyzkoušet použití neznámé technologie na části projektu/prototypu, pokusit se spojit se starým systémem, číst a ukládat data s pomocí neznámé DB. Je dokonce vhodné mít v Elaboration fázi (pokud je třeba ověřovací prototyp, tak už i v Inception) dvě verze v rozdílné technologii/architektuře, abychom si ověřili, zda jsme schopni implementovat funkce s definovanými omezeními.

Název rizika	Popis	Dopad	Pravděpodobnost	Důležitost	Vlastník
Nedostatečné zapojení všech stakeholderů	Předchozí zkušenost nám říká že lidé z oddělení X nemusí pochopit či souhlasit s požadavky, výsledkem budou požadavky na zásadní změny po Beta-releasu.	3 vysoký	90%	2.7	Analytik Honza
Integrace se systémem X	Není zřejmé, jak integrovat naši aplikaci s historickým systémem X.	3 vysoký	80%	2.4	Architekt Petr
Tréninkové materiály	Nemáme oprávnění vytvořit kvalitní tréninkové materiály, což může vést k nekvalitnímu tréninku.	2 střední	100%	2.0	Manažerka Anička
Nedostatečná zkušenost s Java EE	Je riziko, že vytvoříme druhořadé, méně technicky kvalitní řešení v důvodu nezkušenosti s platformou Java EE.	2 střední	60%	1.2	Vývojář Tom

Tabulka 6-4: Příklad seznamu rizik (risk list).

Dalším konceptem, který zmiňuje RUP v kontextu projektového řízení SW projektů, jsou metriky. Metriky slouží pro monitorování a měření postupu na projektu (náklady, čas, kvalita, naplnění požadavků zákazníka). Měřit musíme také dopady změn; zda je možné je implementovat, zda to není příliš drahé (donutí nás např. změnit architekturu). Není možné měřit vše a pořád, proto je důležité definovat a shromažďovat pouze potřebná data, která se týkají definovaných cílů. Příkladem takových cílů může být:

- Monitoring postupu podle relativního plánu.
- Zvýšená spokojenost zákazníka.
- Zvýšená produktivita.
- Zlepšené odhady týmu.
- Zvýšená znuvupoužitelnost komponent.

Na tomto místě opět zdůrazníme, že nejdůležitější metrikou sledování projektu by měla být implementovaná aplikace a spokojený zákazník.

Nejdůležitější rolí, která vystupuje v rámci disciplíny projektové řízení, je projektový manažer, ale je nutné si uvědomit, že většina aktivit je prováděna v týmu! Mezi ty nejzásadnější patří:

- Plánování iterace – zahrnuje plánování rozsahu a odpovědností další iterace; architekti a vývojáři tuší mnohem lépe než manažer pracnost

a možná skrytá rizika daného řešení; testeři zase, co vše musí udělat pro efektivní testování apod.; náplň plánování je zhruba následující:
    - prioritizace položek (požadavků zákazníka) - vysokoúrovňové,
    - definice cílů iterace,
    - definice jednotlivých detailních úkolů (většinou v rozsahu půl den až dva dny),
    - identifikace a revize rizik,
    - definice evaluačních kritérií (testování, úspěšná demonstrace, dokumentace).
- Hodnocení iterace – tým prezentuje zákazníkovi, co v rámci dané iterace vytvořil; rozhoduje, zda byla iterace úspěšná či ne; aplikuje ponaučení, případně zlepší současný proces. Součástí iteračního hodnocení je i retrospektiva – viz text Informační systémy 2.

Naopak, některé akce jsou čistě v režii manažera, například monitorování postupu na projektu/v iteraci (např. pomocí Burn down chartu – viz Informační systémy 2). Co ještě dělá projektový manažer a za co je zodpovědný? V zásadě je jeho náplní následující:

- Je zodpovědný za tvorbu a úpravu projektového plánu a iteračních plánů (pozor, zde je však zapojen celý tým!).
- Je zodpovědný za řízení, plánování (nejrizikovější věci nejdříve) a monitorování rizik – akce navrhovány a schvalovány v rámci týmu.
- Pomáhá odstraňovat zbytku týmu problémy a překážky, řeší jejich problémy s připojením do sítě, problémy uvnitř týmu, atd.
- Průběžně kontroluje průběh projektu (např. pomocí burn down chartu), může být hlídačem na daily meetingu týmu (abychom ho příliš nenatahovali, říkali pouze co máme).
- Má zodpovědnost za řízení lidských zdrojů a kompetencí na projektu.
- Řídí retrospektivu a shromažďuje návrhy na zlepšení procesu.

V rámci celého textu zmiňujeme vhodné kombinace jednotlivých rolí. Podle velikosti projektu je možné kombinovat roli projektového manažera s Testerem (nezná řešení, ale zná požadavky, potřeby zákazníka – ideální kombinace) či Test manažerem. Ostatní kombinace nejsou příliš vhodné. Navíc na větších projektech je tato role vlastně již funkčním místem, tj. je zastávaná jedním člověkem na plný úvazek.

6.7.2 Délka iterace a jejich počet

Délku iterace ovlivňuje několik aspektů. Základním pravidlem je délka iterace od 2 do 6 týdnů. Pokud tým není seznámený s konceptem iterací a je zvyklý pracovat vodopádovým způsobem, budou delší iterace (klidně 2 měsíce) v úvodu vhodnější. Druhým zásadním bodem je nutná režie potřebná na iteraci z hlediska týmu. Tým se účastní plánování a hodnocení iterace, což mohou být klidně 2 dny (1 den plánování iterace + příprava detailního plánu, druhý den hodnocení, demonstrace a retrospektiva). Pokud by iterace trvala týden, jsou na vlastní práci k dispozici pouze 3 dny, což pro vývoj není opravdu mnoho.

Například tým 5 lidí může stihnout plánování v pondělí dopoledne, při každodenním společném obědě monitorovat postup, přehodit úkoly atd. Ve

čtvrtek večer sestavit build, v pátek dopoledne opravit nalezené problémy a v odpoledne demonstrovat a provést retrospektivu. S 20 lidmi v jednom týmu či více týmech již tento scénář nebude proveditelný, jelikož si distribuce práce vyžaduje více času, stejně jako synchronizace týmů a výsledné sestavení buildu. V tomto případě je vhodné mít iteraci tří až čtyř týdenní. Ještě jedno měřítko, které lze použít pro stanovení délky iterace týmu ukazuje následující tabulka:

Počet tříd	Počet lidí	Délka iterace
500	4	2 týdny
2000	10	2-4 týdny
10.000	40 (sub-týmy)	4 týdny
100.000	150 (sub-týmy)	4 týdny

Tabulka 6-5: Délka iterace podle týmu a projektu.

Problematika počtu iterací byla již detailně probrána průběžně v kapitole 5 Fáze RUP, více lze nalézt také v textu Ročníkový projekt.

6.8 Disciplína Deployment

Vývoj aplikace je v celém životním cyklu časově méně významnou částí. Nejdelší dobu aplikace běží v produkčním prostředí (kde pomáhá uživatelům a vydělává peníze), běžně jsou to u aplikačního software jednotky až desítky let. Například v bankovnictví, pojišťovnictví a telekomunikacích se můžeme dnes setkat se systémy, které byly vyvinuty a nasazeny v 80. létech minulého století – to je více než 20 let existence a provozu. Proto, aby vyvinuté systémy mohly být provozovány, musíme je do provozu tzv. nasadit. Nasazení (*deployment*) obecně zahrnuje spoustu aktivit týkajících se nejen vlastního spustitelného kódu aplikace, ale také testování na produkčním prostředí (testovací se od produkčního bohužel často liší), vlastní distribuce software po síti, poštou, fyzicky, školení uživatelů či úprava a migrace dat. Po tomto výčtu už Deployment rozhodně nevypadá jako nepodstatná disciplína.

Disciplína Deployment není důležitá či použitá pouze jednou, při prvotním nasazení. Tato disciplína vstupuje do hry při jakémkoliv doručení buildu či části aplikace zákazníkovi.

Aktivity disciplíny v případě RUP/OpenUP vstupují do hry nejpozději v Elaboration fázi. Hlavní slovo pak mají v Construction a vrcholí v Transition fázi a jsou jakýmsi posledním krokem k předání aplikace zákazníkovi ke ***spokojenému*** užívání. Proto jsou součástí této disciplíny i nezbytné aktivity zahrnující beta-testování (testování s vybraným okruhem zákazníků) v průběhu několika iterací, školení, distribuce do cílového místa, jednoduchá instalace apod.

Z rozsahu aktivit v úvodu zmíněných můžeme vytušit, že tato disciplína definuje úkoly pro větší množství rozdílných rolí, konkrétně jde o:

- *Deployment manager* (manažer zodpovědný za nasazení) – plánuje a organizuje nasazení, zajišťuje a kontroluje vhodnou strukturu balíčků.

- *Project manager* – hlavní rozhraní mezi vývojovým týmem a zákazníkem, je odpovědný za schvalování nasazení na základě zpětné vazby (neúplné, netestované či zákazníkem neschválené funkčnosti nemohou být instalovány).
- Tvůrce technické dokumentace – plánuje a píše uživatelskou dokumentaci a dokumentaci pro podporu koncových uživatelů.
- *Configuration manager* a administrátorské role.
- Vývojář – vytváří implementační skripty a podpůrné artefakty, které pomáhají uživatelům při instalaci.
- Grafik.
- Analytik testů, tester.

Obr. 6-32: Role technical writer a její artefakty a aktivity (zdroj [RUP large]).

RUP dělí artefakty Deployment disciplíny podle toho, zda nasazujeme customizovaný systém či zda instalujeme typový aplikační software stažený z webu. Podle toho jsou také vyžadovány specifické artefakty. Typický release by měl obsahovat následující artefakty:

- spustitelný software,
- instalační artefakty – skripty (např. pro tvorbu databáze), nástroje, soubory, licenční informace, instalační průvodce,
- release notes popisující release pro koncového zákazníka,
- podpůrné materiály jako provozní či uživatelské příručky a manuály,
- tréninkové materiály.

Základní aktivity týkají se artefaktů potřebných pro úspěšné nasazení produktu u koncového zákazníka jsou následující:

- Plánování nasazení – nasazení by mělo být především řízeno přáním zákazníka; navíc musíme brát v potaz, že zákazník musí mít k dispozici všechny informace, aby o doručení vědět a také mohl doručený produkt využívat. Kromě pravidelných demonstrací na konci iteraci a těsné kooperaci s analytiky probíhá beta testování v rámci několika posledních iterací (hlavně Construction, možno také v Transition fázi).
- Tvorba podpůrných materiálů – jedná se o veškeré materiály, které zákazník potřebuje k tomu, aby mohl produkt nainstalovat, provozovat, používat a udržovat, tyto materiály zahrnují také tréninkové materiály.
- Tvorba releasů – tvorba instalačního balíčku, který obsahuje všechny potřebné artefakty a navíc ve správných verzích, aby mohl uživatel daný produkt používat.

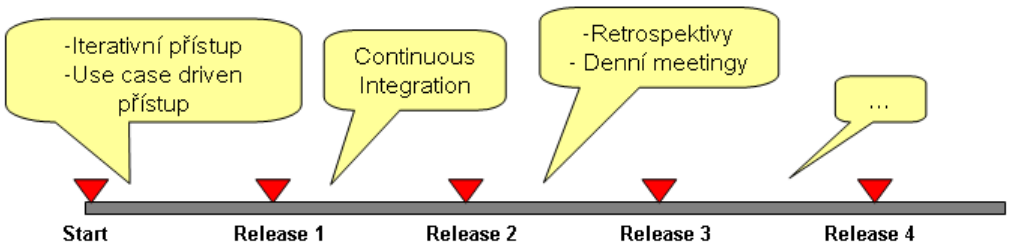
- Další aktivity jsou Beta test release, balíčkování produktu, poskytnutí přístupu k webu včetně podpory 24x7.

6.9 Disciplína Environment

Tato disciplína je v rámci RUP disciplínou velmi důležitou, i když povědomí o ní je velmi malé. Disciplína zahrnuje elementy a aktivity procesu týkající se podpůrného prostředí vývojového týmu, čímž jsou myšleny jak procesy (check-list, průvodce, šablony), které tým následuje, tak nástroje, které tým využívá. Hlavní úkoly, které tato disciplína definuje jsou:

- Definice popisu stylů pro tvorbu příruček, průvodců, šablon.
- Konfigurace procesu – uvedení vývojového procesu do projektu (úprava standardního podle potřeb, publikace procesu, tréninky, podpůrné materiály) a jeho aktualizace, úpravy a zlepšování podle potřeby.
- Příprava vlastních šablon a průvodců procesu.
- Výběr a zajištění (nákup) nástrojů pro potřeby projektu, případně vývoj vlastních nástrojů či jejich integrace.
- Instalace a nastavení nástrojů.

Pro vlastní realizaci těchto úkolů je doporučeno zahrnutí mentorů do projektu a inkrementální implementace procesu a nástrojů. Díky inkrementální implementaci nebudou lidé zahlceni novými věcmi a budou schopni nadále vykonávat svoje úkoly a přitom se po částech učit nové postupy či pracovat s novými nástroji. Po tom, co si členové týmu zažijí tyto nové věci, můžeme přidat další postup, techniku či využití nového nástroje a zase nechat týmu určitý čas na jejich zažití. Tímto dosáhneme postupného naučení nových, efektivnějších postupů a přivedeme tým na vyšší úroveň automatizace pomocí nástrojů.



Obr. 6-33: Inkrementální implementace procesu v průběhu několika iterací jednotlivých releasů.

Funkce mentorů je pak zásadní v zavádění procesních postupů, metod a technik. Mentor je zkušený expert v oblasti procesů, má zkušenosti z různě velkých projektů, různě distribuovaných, s různými nástroji i technologiemi, s vývojovými projekty, s vývojem a údržbou a také s podporou aplikací. Mentor nerozhoduje za tým, jen pomáhá na denní či méně časté bázi (ale pravidelně!), navrhuje varianty a vysvětluje důsledky. Pomáhá objasnit, jak přesně danou techniku provádět (například jak modelovat pomocí use case a psát scénáře, jak párově programovat, jak zlepšit komunikaci mezi členy týmu apod.).

Důležitým aspektem je také neustálé informování všech členů týmu (vývojáře, analytiky, projektové manažery, testery, ...) a jejich zahrnutí do tvorby a úpravy procesu. Je důležité poslouchat jejich zpětnou vazbu – pokud je nějaký přístup či technika v daném projektu neefektivní, nebudeme trvat na jejím provádění.

Jedná se zde hlavně o aplikaci principu A vysvětleného v úvodních kapitolách tohoto učebního textu (viz aplikace tohoto principu na Obr. 6-33). Jde tedy o to použít z celého procesního frameworku jen to, co je na daném projektu potřeba a implementovat tyto věci krok po kroku, ne najednou, abychom členy týmu nezahltili učením se nových věcí a na samotnou práci by jim nezbyl žádný čas. Dále je v této disciplíně jasně viditelný princip C a také E. C mluví o spolupráci a motivaci členů týmu. Pokud jim umožníme spolupráci (sezení ve společné místnosti, pravidelné workshopy, společná repozitury, Instant Messaging, automatické buildovací nástroje) a snazší komunikaci, nebudou muset řešit jak sehnat informace, jak ručně na server nakopírovat a sestavit soubory, ale mohou se věnovat své práci – tvorbě hodnotného software. Požadavky na proces navíc přichází od jednotlivých členů týmu, musí naplňovat jejich potřeby. E princip se pak projeví hlavně v nástrojích, které nám umožňují pracovat s danou mírou abstrakce, jaká je potřebná. V kontextu celého produktu to může být abstraktní digram tříd či komponent, v kontextu jedné komponenty detailní diagram tříd či samotný zdrojový kód.

6.9.1 RUP a Environment

Hlavní záběr této disciplíny je v úvodu projektu, kdy definujeme vhodný proces, vybíráme a nastavujeme nástroje a pak také na úvod každé fáze. Každá je zaměřena na něco jiného, tudíž potřebujeme jiné nástroje, aktivity a také například těsnost procesu. V případě nástrojů vždy uvažujeme, jaký je přínos v porovnání s investovanou snahou, a také musíme počítat s nutnosti údržby detailního modelu vytvořeného pomocí mocného nástroje, což stojí čas a tedy peníze.

V první iteraci bychom měli nastínit proces a prostředí, v druhé iteraci nasadit a upravit podle okamžité zpětné vazby ode všech zúčastněných a poté samozřejmě stále iterativně vylepšovat a upravovat (opět aplikace principu A – Adapt the process). V zásadě v Inception již začneme nástroje podle potřeb procesu a vývojářů připravovat, v Elaboration je tým ověří „v boji“, aktualizujeme a přednastavíme podle potřeb tak, aby v Construction celý tým měl nástroje, guidelines a ostatní potřebné artefakty k dispozici.

Pokud jsme vybrali a nastavili proces, můžeme se věnovat výběru nástrojů. Dané nástroje by měly podporovat zvolený proces, metodu! Příkladem je například sada nástrojů Rational a integrace s RUP, existence tzv. tool mentorů (popisy krok za krokem, jak dané aktivity procesu vykonávat pomocí určitého nástroje). Pokud pro tuto oblast existují firemní standardy, měli bychom je samozřejmě vzít také v potaz. Pokud ne, uvažujeme následující oblasti:

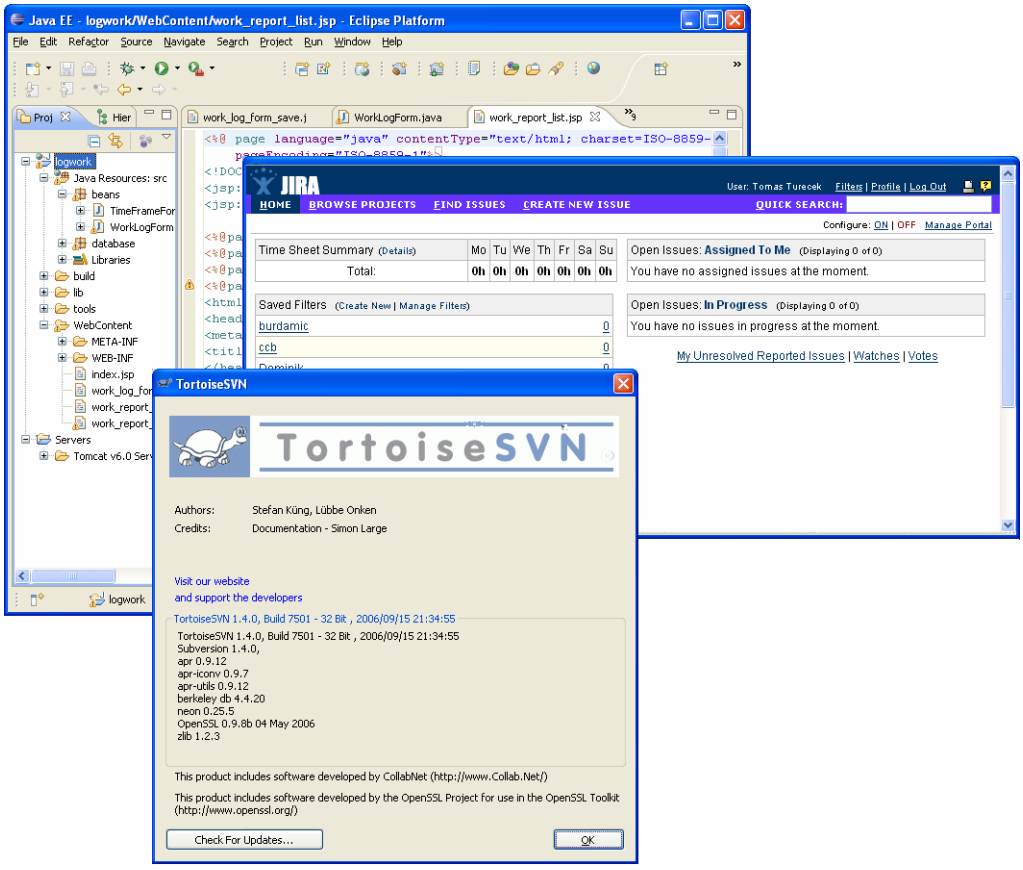
- IDE – integrované vývojové prostředí (př. Eclipse, Visual Studio, NetBeans, …).
- Nástroj pro správu požadavků, chyb (př. Rational Requisite Pro, Jira, Rational Team Concert).

- Vizuální modelovací nástroje (př. Magic Draw, Borland Together, Eclipse pluginy, …)
- Nástroje pro správu konfigurací a změn – Configuration and Change Management Tools (CVS, SVN, Jira).

Jelikož je vývoj software týmovým sportem, kde několik vývojářů, analytiků, testerů neustále spolupracuje, je třeba podpořit také paralelní práci a sdílení kódu nástroji. Jedná se hlavně o následující oblasti:

- Úložiště zdrojových kódů a dokumentace (repozitory) – cílem by mělo být dosáhnout možnosti sestavení kompletní aplikace z repozitory.
- Automatizace sestavení aplikace (build) a testování – pomocí nástrojů make, Ant (nUnit testy jako task).
- Ukládání (commit) pouze hotových věcí do repozitory – nehotové věci mohou způsobit problémy a chyby.

Dohodnutá či psaná pravidla jsou důležitá, ale lze je obejít. Pokud chceme jejich dodržování implicitně vynutit, pomohou nám právě nástroje. Neumožní nám commit pro neotestované či chybové třídy, nutí nás nejdříve vytvořit test, kontrolují čitelnost kódu a překrytí metod vůči standardu a spousta dalších.

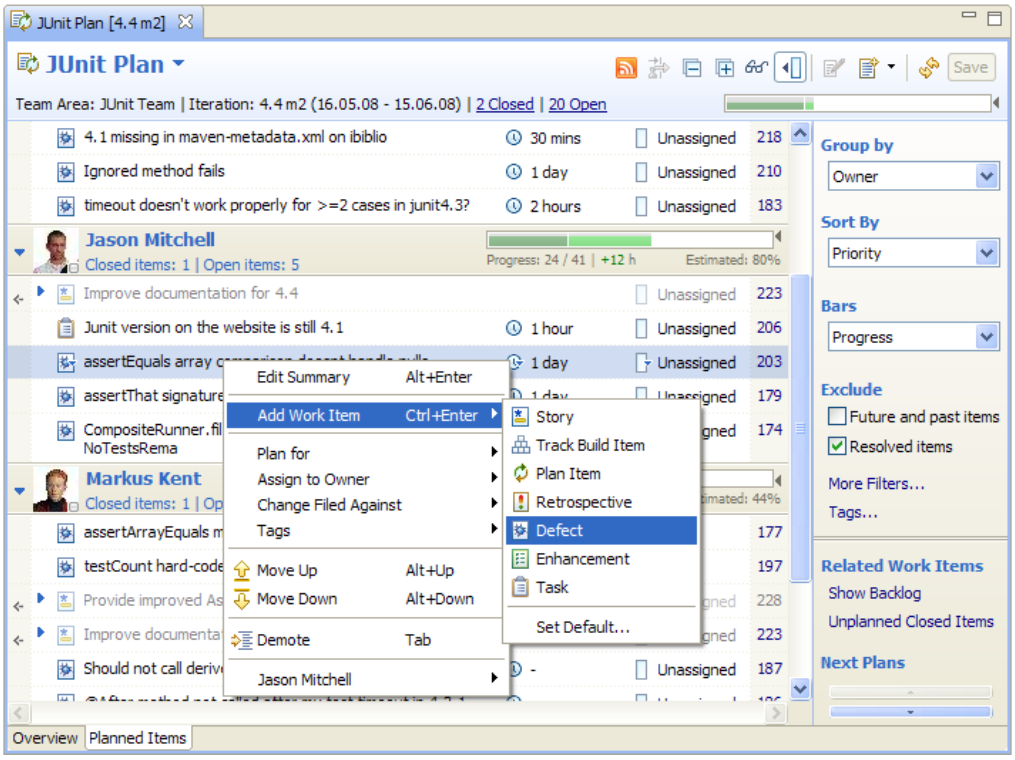


Obr. 6-34: Eclipse IDE, SVN repository, Jira issue tracking tool.

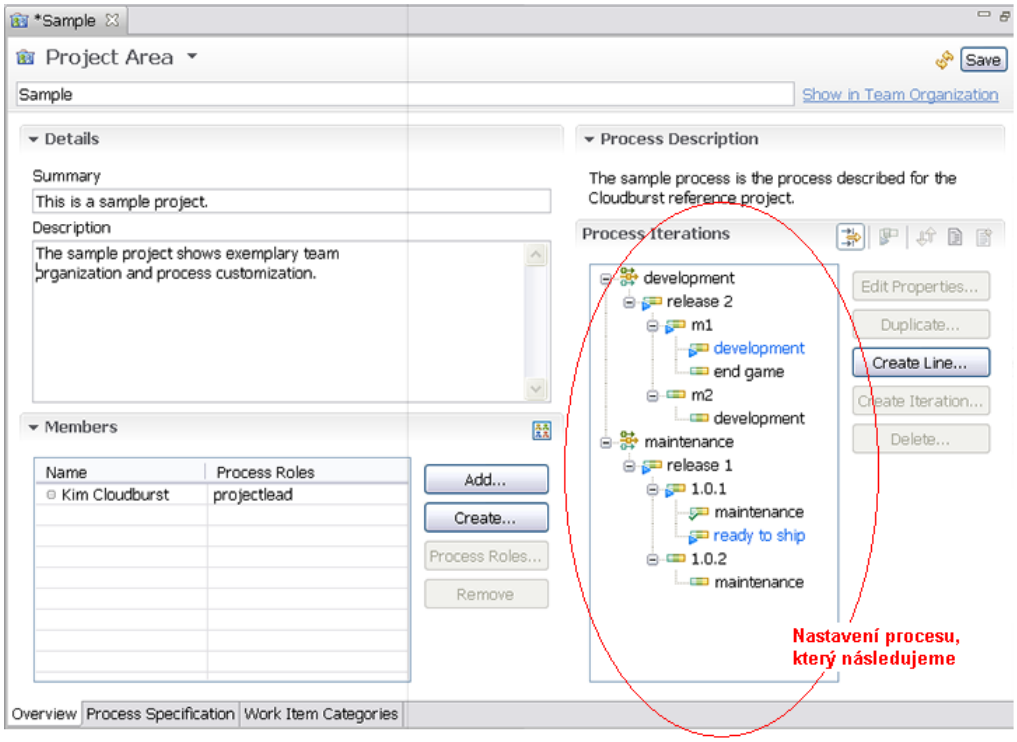
Výborným, i když ne zcela levným počinem, na poli nástrojů je nový produkt IBM zvaný Rational Team Concert (RTC) běžící na platformě Jazz. Tento nástroj v sobě integruje definici procesu, podle kterého při vývoji postupujeme (viz Obr. 6-36), dále agilní plánování, wiki, repozitory, buildovací

mechanismus, správu členů týmu, prostě veškeré potřebné nástroje pro vývoj. Na rozdíl od výše zmíněné kombinace nástrojů (viz Obr. 6-34:) je toto jeden nástroj (není třeba odkazovat a integrovat, vše je provázáno) a hlavně není orientován pouze na jednoho člena týmu, neobsahuje pouze pohledy na jeho práci, nýbrž základním pohledem je pohled na celý tým.

Následující obrázek ukazuje tento pohled, kde vidíme jednotlivé položky, tzv. *work items* (přiřazené úkoly, chyby, požadavky, rizika, …) každého člena týmu, jeho odhady pracnosti a stav dané položky (otevřená, uzavřená, právě řešená). Navíc tento pohled obsahuje také týmové informace (kolik z celkového počtu položek je již uzavřeno, kolik zbývá). Pohled umožňuje jednoduchou manipulaci (*drag and drop*) a také vytvoření položek.



Obr. 6-35: Team Central View IBM Rational Team Concert, zdroj [Jazz].



Obr. 6-36: Process View - definice procesu, který tým následuje, zdroj [Jazz].

Měl jsem možnost několik roků s RTC pracovat na denní bázi a často jsem spolupracoval s autory produktu (zpětná vazba, nové požadavky, reportování chyb), a už bych se nechtěl vracet k původně používané kombinaci nástrojů (CVS, Jira, Ant, ...). V RTC je možné vidět, kdo na čem dělá a mít tak přehled zda neděláme dva na stejné věci, výborná je podpora agilního plánování a redukce generování dokumentů (wiki, provázanost položek), při commitu do repozitory je automaticky k dané sadě přiřazen otevřený úkol, nástroj si přímo vynucuje pracovat podle námi definovaného procesu, je možné definovat RSS a různé pohledy, grafy. Více informací o platformě Jazz a souvisejících aplikacích na [Jazz].

Důležité role disciplíny environment jsou tedy následující:

- *Process engineer* – zodpovědný za úpravu procesu pro daný projekt podle jeho potřeb, tvorba specifických guideline a zahrnutí firemních standardů.
- *Tool specialist* – zodpovědný za podpůrné nástroje pro vývoj, jejich instalaci, nastavení, provoz.
- Architekt/Analytik/Test designer – zodpovědný za tvorbu guideline pro návrh a zápis kódu/pro zápis use case/pro tvorbu testů.

Aktivity jsou v podstatě již popsány několikrát výše, jedná se o:

- Příprava prostředí pro projekt.
- Příprava prostředí pro iteraci.
- Příprava guideline pro iteraci.

Kontrolní otázky:

1. Jaké disciplíny RUP znáte?
2. Definujte jejich účel a základní role.
3. Která disciplína je nejkritičtější z technického hlediska pro co nejvyšší automatizaci pracovního prostředí (testy, buildování, repository)?
4. Která disciplína zmiňuje iterativní, inkrementální vývoj pro řízení projektu?

Úkoly k zamyšlení:

Zamyslete se nad agilní technikou zvanou párové programování. Najděte si dodatečné informace a zkuste popřemýšlet nad jejím nejefektivnějším využitím. Její využití po celý den asi nebude u všech akceptovatelné (introverti, přílišná ochrana soukromí, lenost apod.); zamyslete se nad tím, jak ji provádět efektivněji a pro jaké oblasti projektu a v jaké formě je to vhodné.

Korespondenční úkol:

Pokuste se předchozí úkol k zamyšlení rozdělit do kategorií a vysvětlit.

Shrnutí obsahu kapitoly

Tato kapitola byla poměrně rozsáhlá a představila všechny disciplíny vývoje IS podle RUP, respektive OpenUP. Pro dokreslení rozdílné povahy RUP a OpenUP byly místy ukázány artefakty z obou frameworků. Většina disciplín byla vysvětlena v kontextu celého projektu, respektive iterace a byla podrobně popsána pomocí rolí, aktivit a artefaktů včetně příkladů.

7 UML a jeho využití při tvorbě software

V této kapitole se dozvíte:

- Co je to UML?
- Jaké diagramy UML nabízí?
- Zapojení UML do procesu vývoje software.

Po jejím prostudování byste měli být schopni:

- Pochopit k čemu UML slouží a jaké prostředky nám nabízí.

Klíčová slova této kapitoly:

Modelování, UML, modely.

Doba potřebná ke studiu: 3 hodiny

Průvodce studiem

Kapitola se zabývá modelovacím jazykem UML, jeho historií, použitím na projektu vývoje software, výhodami a nevýhodami.
Na studium této části si vyhraďte 3 hodiny.

Při vývoji informačního systému je výhodné postupovat snižováním abstrakce. Pro rozdělení systému na více vrstev abstrakce je zapotřebí mít k dispozici unifikovaný jazyk, který bude pro člověka srozumitelný a zároveň dostatečně vhodný k tomu, aby mohl na konkrétní abstraktní vrstvě vyjádřit vše potřebné pro správné popsání systému. Tyto požadavky daly impuls ke vzniku jazyka UML (Unified Modeling Language).

„UML je vizuální modelovací jazyk, který se používá ke specifikování požadavků, vizualizaci a dokumentaci artefaktů softwarového systému.“

UML je jako jazyk definován obecně a lze jej použít i k jiným účelům, než je popis vývoje software. My se ovšem omezíme pouze na diagramy a příklady, které vývoj software přímo podporují. UML je standardizován sdružením Object-Management Group (OMG) a obsahuje kromě syntax a sémantiky také jednoduché návody ke správnému použití. Standard lze rozdělit na tři základní oblasti a to struktura, dynamika a správa modelu. Pro každou oblast UML obsahuje specifické pohledy a diagramy. Při vytváření diagramů se kladl velký důraz, aby diagramy mohly být použity jako vstup pro interaktivní vizuální nástroje, které umí generovat části zdrojového kódu. Specifikace UML nedefinuje žádný proces vývoje a nedá se použít jako metodika vývoje software. Naopak slouží jako podpůrný nástroj pro vodopádové a agilní metodiky vývoje. Diagramy UML jsou koncipovány převážně pro objektově-orientované systémy.

UML umí popsat statické i dynamické chování systému. Model systému je tvořen kolekcí diskrétních objektů, které spolu reagují a vykonávají tak nějakou činnost. Statická struktura definuje druhy objektů a jejich implementaci. Také popisuje vazby mezi objekty (podobně jako objektově-orientovaný přístup zavádí dědičnost, generalizaci atd.). Dynamická struktura

pak popisuje historii chování objektů v čase a komunikaci mezi nimi. Z výše uvedeného lze vytušit, že standard UML je velmi silný nástroj. Proto je potřeba se důkladně seznámit se základní filosofií nejčastěji používaných diagramů. Pokud jsou tyto diagramy nesprávně využity, tak vývoj software neulehčí, ale naopak zbrzdí.

V současné době existuje dle standardu UML pouze jeden diagram, který je schopný vytvořit spustitelný model – Executable UML (ExUML). Tento diagram v praxi není velmi rozšířen a také existují alternativy jako doménově specifické jazyky (DSL), proto zde nebude zmíněn. Zájemce o tvorbu spustitelných modelů mohu odkázat na knihu [Fo10].

7.1 Cíle a historie UML

Již v 80-letech se začínaly objevovat jednoduché objektově-orientované aplikace. Současně s rozvojem vyšších programovacích jazyků vznikaly komplexnější objektově-orientované systémy a vznikla potřeba je vizualizovat. Proto začaly vznikat jednotlivé samostatně dle požadavků z praxe. Postupně vznikly metody Object-Oriented analysis (OOA) a Object-Oriented Design (OOD). První zlom nastal v roce 1994, kdy společnost Rational Software uvedla techniku Rumbaugh's Object-Modeling Technique (OMT) jako lepší alternativu k OOA a metodu Booch jako lepší alternativu k OOD. V roce 1995 se k Rational Software připojil Ivar Jacobsen a přispěl technikou Object-Oriented Software Engineering (OOSE). Tito lidé jsou označování v literatuře jako „*3 Amigos*" a jejich cílem bylo sjednotit dosud navržené přístupy pod jeden společný standard. V roce 1996 vytvořili koncept jazyka a ten byl zaslán OMG k posouzení, v roce 1997 bylo UML schváleno jako OMG standard ve verzi 1.0.

UML prošlo za svou existenci určitým vývojem, který se snažil reflektovat požadavky vývojářů, analytiků, návrhářů a dalších, kteří UML využívají. Výsledkem největší změny byla nová verze 2.0, která přidává nové diagramy umožňující modelování dosud chybějících aspektů. Tak jako software, tak i UML používá zažitý způsob verzování:

- Malé změny a opravy verzí jsou za tečkou (např. verze 1.4 či 1.5, 2.1).
- Významné změny a revize jsou reprezentovány prvním číslem (např. verze 2.0).
- V roce 2008 byla schválena verze 2.2, která je v současnosti (rok 2012) verzí nejnovější. Dokumenty podrobně popisující poslední změny i standard jako celek jsou dostupné na webu OMG[11].

Příkladem novinek v nové verzi 2.X je následující:

- Modelování chování – v UML 1.X byly modely chování nezávislé, ve verzi 2.X jsou derivovány ze základní definice chování.
- Lepší vazba mezi strukturálními modely a modely chování.

[11] http://www.omg.org/spec/UML/2.2/

Rozdíl mezi UML verze 1 a UML verze 2 je značný, hlavně v oblasti definice sémantiky. Naopak v rámci standardu UML 2 jsou revize spíše kosmetické. Zaměříme se tedy přímo na verzi 2.X bez rozlišení revize.

Jak již bylo naznačeno v úvodu, UML poskytuje výrazový vizuální modelovací jazyk pro tvorbu a výměnu smysluplných modelů a také mechanismus pro rozšiřování jádra UML. UML je nezávislé na programovacím jazyku, implementačním prostředí a procesu vývoje (modely lze použít jak ve vodopádovém procesu, tak při iterativním inkrementálním modelu). Modely tedy poskytují analytický obraz, tzv. „blueprint“ budoucí aplikace bez ohledu na proces či technologii.

Diagramy UML jsou v praxi při vývoji software velmi často používány. Částečně kvůli historickému vývoji a částečně kvůli nepochopení samotného účelu nástroje jsou ovšem diagramy vývojáři špatně interpretovány. Proto musíme uvést nejčastější oblasti špatného použití UML:

- Programování - čisté UML není programovací jazyk, přestože obsahuje definici syntax i sémantiky. Existují sice vizuální nadstavby pro tvorbu spustitelných modelů, ovšem tyto nástroje nemají stanovenu obecnou standardizovanou sémantiku a tedy jejich použití není univerzální.
- Řízení vývoje software - UML se nedá použít jako procesní model či metodika vývoje. Jednotlivé diagramy sice můžou vyjádřit analytickou myšlenku, ale neříkají nám kdy a co dělat. Pouze poslouží jako nástroj pro metodiku vývoje.

Standard UML obsahuje „pouze“ pohledy a diagramy, proto nám nedokáže odpovědět ani na následující otázky:

- Jaký diagram a kdy použít?
- Jaký model vývoje software (vodopád, agilní) bychom měli použít?
- Jak transformovat UML modely na schéma relační databáze či do ER diagramů?
- Mohu použít UML pro modelování procesů a vláken aplikace (toto již odpovězeno částečně UML ve verzi 2)?

UML dále neřeší problematiku návrhu grafického uživatelského rozhraní, mapování architektury na vybranou technologii, distribuci procesů a dat, objektově relační mapovaní kódu do relační DB. Na druhou stranu existuje spoustu nástrojů, které jsou schopny automatizovaně vytvořit např. kostru kódu z UML diagramu a je moudré je při vývoji využít.

Vidíme, že stěžejním bodem efektivního využití UML v procesu vývoje a údržby software je jeho vhodné zařazení do našeho procesu vývoje (toto za nás již udělal např. Unified Process či RUP) a výběr vhodných modelů pro každý případ/projekt (to musíme bohužel udělat vždy v rámci každého projektu sami).

7.1.1 Diagramy a pohledy v UML

UML nabízí několik pohledů na systém. Můžeme modelovat strukturu systému či jeho části, chování systému či komponenty, třídy, interakce mezi třídami,

změny stavů apod. Pro různé pohledy nabízí UML čtyři základní oblasti uvedeny v následující tabulce. První sloupeček uvádí výčet oblastí, které UML pokrývá. Následuje popis jednotlivých pohledů v oblastech následovaný konkrétním názvem diagramu. Pro některé názvy diagramů se neustálily české ekvivalenty, proto jsou všechny uvedeny v původních názvech dle specifikace. Poslední sloupec tabulky naznačuje koncepty, které je možno vyjádřit diagramy UML.

Oblast	Pohled	Diagram	Koncept
Statické chování systému	statický pohled	class diagram	třída, asociace, generalizace, závislost, realizace, rozhraní
	příklady užití	use case diagram	příklad použití, asociace, rozšíření, generalizace použití
	implementace a nasazení	component diagram	komponenta, rozhraní, závislost, realizace
		deployment diagram	stanice (uzel), komponenty, závislost, lokalita
Dynamické chování systému	stavy systému	statechart diagram	stav, akce, událost, přechod
	aktivity systému	activity diagram	stav, aktivita, dokončení přechodu, příznak
	vzájemné interakce	sequence diagram	interakce, objekt, zpráva, aktivační impuls
		collaboration diagram	spolupráce, interakce, role spolupráce, zpráva
Správa modelů	pohled správy	class diagram	balíček, subsystém, model
Možnost rozšíření	všechny	všechny	architektonické vzory, omezení

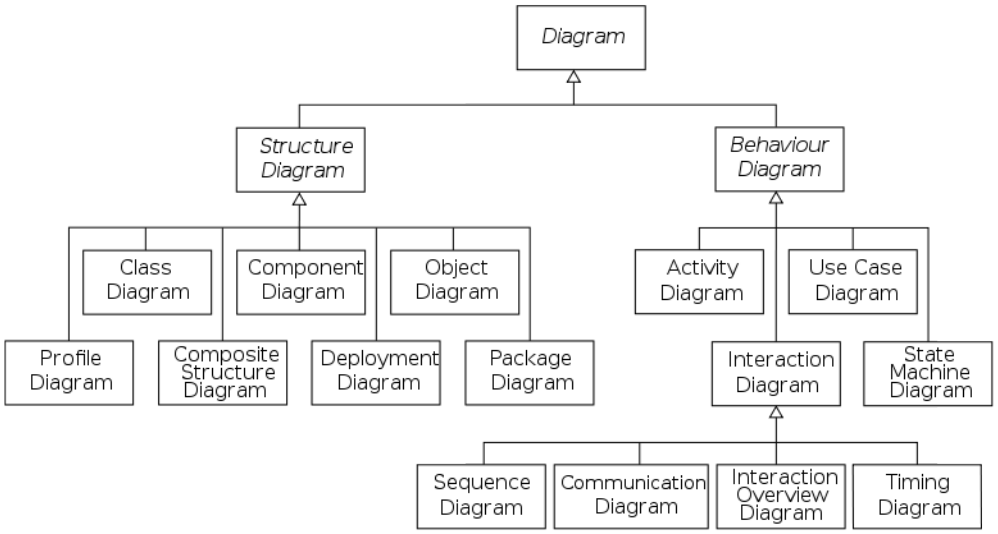
Tabulka 7-1: Pohledy a diagramy dle UML.

Při bližším prostudování tabulky zjistíte, že některé diagramy jsou schopny popsat více konceptů, ovšem v jiném pohledu. Je tedy zřejmé, že nelze udělat ostrou hranici mezi jednotlivými diagramy a modelováním chování systému. To je jedním z důvodu oblíbenosti použití UML v praxi při popisu návrhu softwarových systémů.

Kromě statického a dynamického popisu systému potřebujeme i možnost vytvářet hierarchické struktury mezi jednotlivými modely. Proto je přítomna oblast „správa modelů“, která tento akt umožňuje. Model se tak může skládat s jednotlivých subsystémů a vzájemně na sebe odkazovat.

Posledním pohledem je možnost rozšíření UML, které je samozřejmě limitováno schváleným standardem. Zde je prostor pro hledání architektonických vzorů, nebo vytváření vlastního slovníku pojmů (označují se stereotypy).

Standard UML obsahuje celkem 14 diagramů a jejich struktura je znázorněna na následujícím obrázku:



Obr. 7-1: Struktura UML modelů

Běžně se při vývoji software nesetkáte se všemi a ani není podmínkou použití UML vytvořit při vývoji všech 14 diagramů. Proto se dále zaměříme na nejčastěji používané diagramy a vysvětlíme si jak jejich základní atributy a možnosti, tak jejich správné použití při vývoji software.

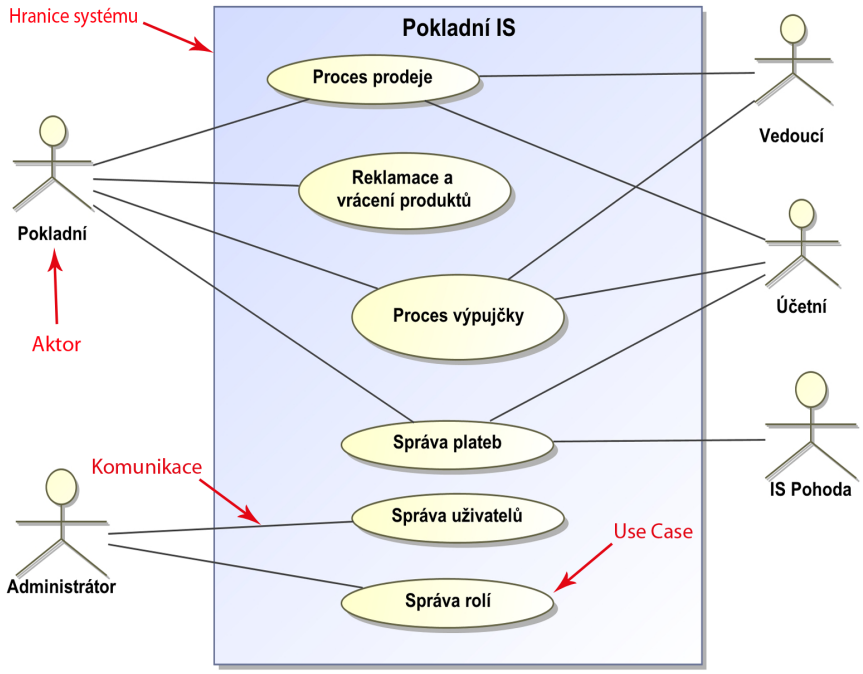
7.2 Diagramy UML a jejich použití

Pro efektivní a přínosné použití UML modelů je velmi důležitá zkušenost daného vývojáře. Je třeba si uvědomit, že udržování diagramů stojí čas a úsilí (= peníze) a proto více detailních diagramů rozhodně neznamená větší přínos a kvalitnější výsledný produkt. Je třeba se ptát: není lepší mít některé detaily spíše v dokumentovaném kódu či ve formě spustitelných testovacích skriptů a test case?

Jak je již zřejmé z výše zmíněného, neexistuje jeden správný a špatný model (use case, analytický či návrhový model, use case realizace), každý z nás vytvoří model mírně odlišný. Je však pravdou, že jeden model bude lépe čitelný, druhý hůře, ať už z důvodu přílišných detailů nebo třeba špatné struktury. Proto je dobré mít modely popisující pouze vyšší úroveň systému (use case, základní architekturu apod.), případně určitou složitou část, komponentu, kde model přispívá k pochopení složitosti např. problémové domény. Místo všech 14 diagramů si zde vysvětlíme pouze ty nejpoužívanější, se kterými se určitě setkáte při vývoji software. Začneme, stejně jako postupujete při vývoji, od nejvyšší úrovně abstrakce.

7.2.1 Use Case diagram – případ užití

Nejvyšší abstrakci, kterou můžeme v UML dosáhnout je tzv. Use case, česky někdy nazývány případy užití. 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. Znalost Use Case a pochopení techniky jejich tvorby je důležitá také kvůli faktu, že některé metodiky vývoje software jsou řízeny pomocí Use Case.



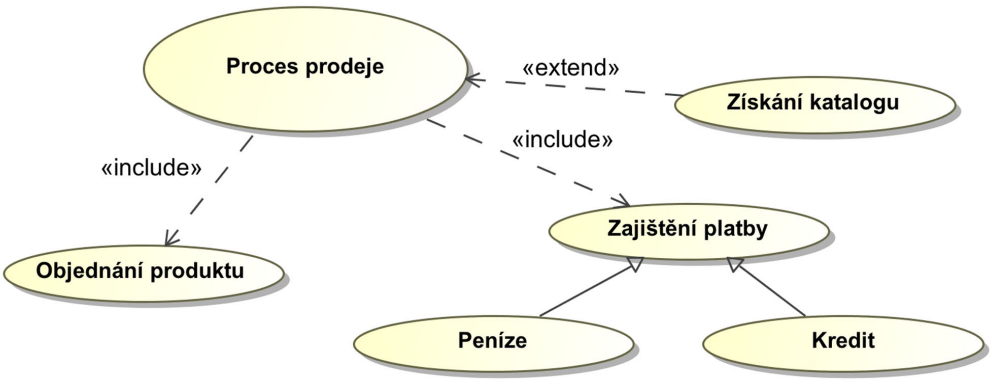
Obr. 7-2: Ukázka použití Use Case

Na Obr. 7-2 je uveden příklad použití Use Case diagramu. Jsou zde zobrazeny základní stavební prvky|:

- aktor,
- Use Case,
- komunikace,
- hranice systému.

Aktor zde zastupuje možnosti interakce se systémem. Jedná se tedy o vnější interakci, která je spojena s konkrétní případem užití. Všimněte si, že v příkladu je kromě „standardních“ uživatelů systému, které si představíte jako konkrétní zaměstnance firmy, také aktor *IS Pohoda*. V tomto případě se nejedná o běžného uživatele, ale o jiný informační systém, který reaguje s navrhovaným pokladním systémem. Z hlediska diagramu případu užití je ovšem na stejné úrovni jako uživatel. Jestli je aktor vlevo či vpravo od hranice systému z hlediska standardu nehraje žádnou roli. V některých literaturách se ovšem můžete setkat s doporučením, že primární a tedy důležití uživatelé jsou umístěni vlevo a uživatelé, kteří tvoří podporu a nejsou pro systém klíčoví se umísťují vpravo.

Use Case (konkrétní případ užití) zastupuje promyšlenou souvislou logickou funkční jednotku. Lidsky můžeme říci, že se jedná o soubor funkcí, které mají něco společného a jsou součástí námi navrhovaného systému. V příkladu je uveden Use Case *Proces prodeje*, pod kterým si můžete představit veškeré činnosti vztahující se k prodání konkrétní věci zákazníkovi – vytvoření nabídky, založení objednávky, vyřízení objednávky atd. V korektním Use Case diagramu je jednotlivý případ užití unikátní. To znamená, že pokud jsou vytvářeny nové detailnější pohledy na systém, je využito již dříve vytvořených případů užití. Například profesionální CASE nástroje (Enterprise Architect, MagicDraw) při vytváření nového případu užití automaticky nabízejí již vytvořené případy užití v rámci jednoho projektu. Využijeme toho a vytvoříme detailní rozpracování případu užití *Proces prodeje* tak, jak je uvedeno na následujícím obrázku.



Obr. 7-3: Možné vazby diagramu Use Case

Pro vytváření složitějších struktur jsou k dispozici vazby *extend*, *include* a *generalizace*. Významově jsou shodné s objektově-orientovaným přístupem. Časté chyby při vytváření Use Case jsou funkční dekompozice a vytváření tzv. CRUDL diagramů. Funkční dekompozice je snaha popsat vnitřní detailní funkcionalitu jako nový případ užití. Pokud bychom vytvářeli textový editor

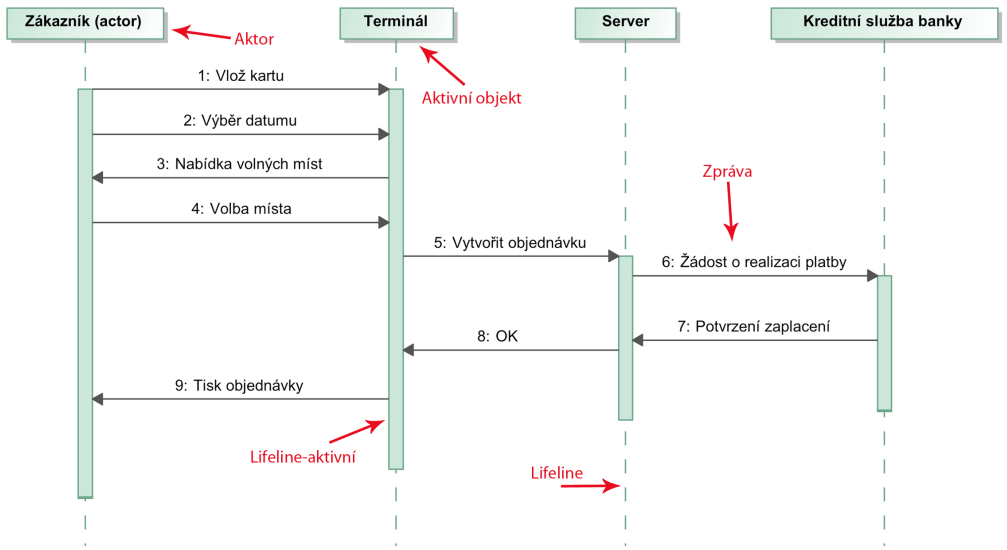
tak příklad dekompozice je tento: Use Case Správa textu by obsahoval případy užití: přepni tučné písmo, přepni kurzívu, podtrhni slovo atd. Takový přístup je samozřejmě špatný a jde proti filosofii použití diagramu. Druhý problém odráží myšlení vývojářů. V databázi jsou základní operace – Create, Read, Update, Delete a List. Proto Use Case „Správa objednávek“ dle vývojáře vypadá následovně: Vytvoř objednávku, Vypiš objednávku, Aktualizuj objednávku, Smaž objednávku a Zobraz všechny objednávky. Tento postup ovšem také není správný, protože se nejedná o abstraktní návrh, ale spíše o databázové operace.

Komunikace je v diagramu znázorněna jednoduchou úsečkou bez naznačení směru komunikace (interakce probíhá obousměrně). V tomto pohledu nejsou specifikována žádná rozhraní ani komunikační protokoly, k tomuto vyjádření slouží jiné diagramy.

Hranice systému zde slouží k jednoznačnému oddělení Use Casů. V jednom diagramu může existovat více systémů s rozdílnými případy užití, které ovšem využívají stejného aktora. Proto je výhodné z hlediska přehlednosti tyto hranice zobrazit.

7.2.2 Sequence diagram – sekvenční diagram

V předchozí kapitole se Use Case používal pro popsání základních funkčních oblastí systému. Toto znázornění je ale příliš abstraktní a proto je pro detailní popis potřeba využít dalšího diagramu – sequence. Sekvenční diagram lze použít více způsoby, jeden z nich je právě snížení abstrakce Use Case a tedy popis jednotlivých kroků v případu užití. Účelem sekvenčního diagramu je popsání komunikace uvnitř systému v čase a tím získání popisu, jak objekt spolupracuje s jinými objekty a identifikace událostí probíhajících mezi objekty. Pro případ užití „Prodej letenek“ by sekvenční diagram pro popis jednotlivých kroků mohl vypadat následovně:



Obr. 7-4: Sekvenční diagram pro rezervaci letenky

Základní myšlenka příkladu je rezervace letenky zákazníkem na terminále letiště. Terminál má dotykovou obrazovku a čtečku kreditních karet. Pro

popsání procesu rezervace byly využity čtyři objekty a celkem devět zpráv pro popsání interakce mezi nimi. Obecně je sekvenční diagram složen z těchto prvků:

- objekt,
- zpráva,
- životní dráha.

Objekt slouží k popsání určité entity, se kterou mohou jiné objekty komunikovat prostřednictvím zasílání zpráv. Na této úrovni abstrakce se ovšem nejedná o objekt z pohledu programátora (např. třída jazyka Java), ale o obecnou entitu. V příkladu je toto patrné u prvního objektu – Zákazník. Ten je zde v roli aktora a tedy není přímou součástí tvorby systému, pouze s ním reaguje. Některé CASE nástroje dovolují objekt aktora znázornit stejně jako v Use Case diagramu, interpretace je ovšem stejná jako v případě objektu.

Zpráva v diagramu znázorňuje interakci s ostatními objekty. Pokud je objektu odeslána zpráva, stane se aktivním. Zprávy mohou být parametrické i bezparametrické. Na této úrovni abstrakce nelze zprávu považovat za volání metody objektu, ale případný parametr může posloužit později při implementaci systému. Zpráva nemusí vyjadřovat pouze interakci s jinými objekty, lze vytvořit rekurzivní zprávu v rámci jednoho objektu.

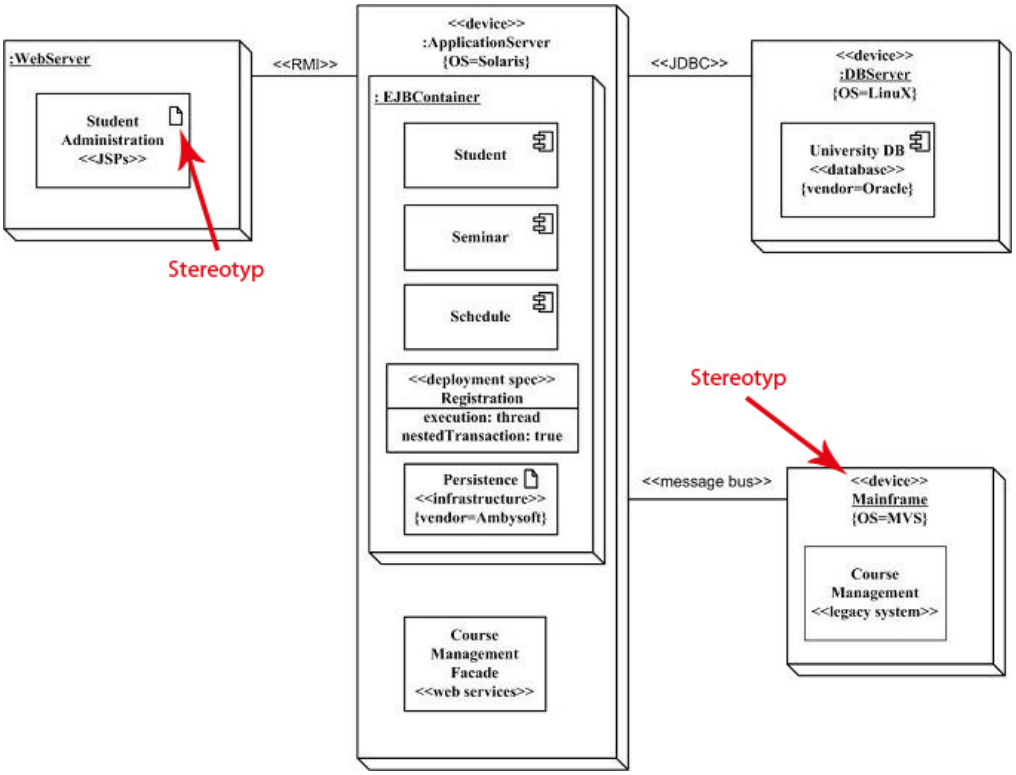
Životní dráha objektu je prostředek pro naznačení možné délky zpracování požadavku. Po odeslání zprávy z jednoho objektu na druhý se příchozí objekt stává aktivním. Vertikální osa je tedy osou časovou.

7.2.3 Deployment diagram - diagram nasazení

Pokud jsou popsány základní dynamické prvky systému, lze využít dalšího diagramu pro snížení abstrakce. Diagram nasazení slouží k zobrazení hardwarové architektury systému, je modelem implementační úrovně a řadí se mezi statické. Jinými slovy ukazuje hardware navrženého systému, software který je na hardware nainstalován a prostředky pro komunikaci komponent mezi sebou. Diagram obsahuje dva základní prvky:

- uzly,
- spojení mezi uzly.

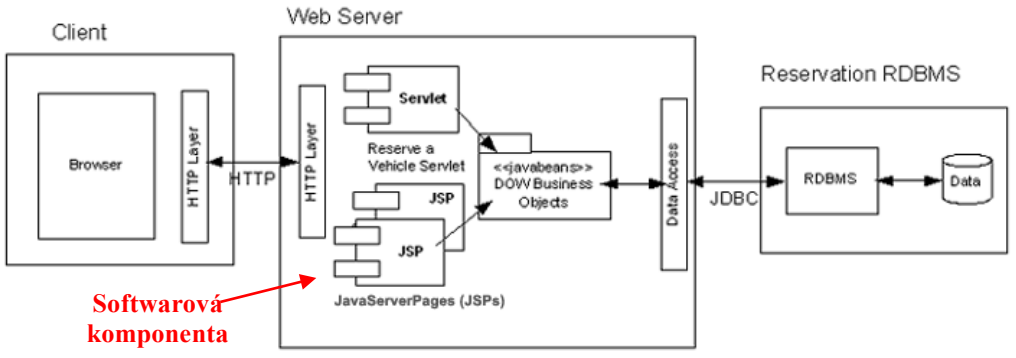
Uzly mohou obsahovat další uzly, nebo softwarové artefakty. Na následujícím obrázku je uveden příklad diagramu nasazení pro univerzitní informační systém. Systém je složen z aplikačního, webového a databázového serveru. Dále je přítomen mainframe pro běh celého systému. Aplikační server pak obsahuje EJB kontejner jehož součástí jsou komponenty a parametry pro spuštění kontejneru. Spojení mezi uzly charakterizují použité technologie pro komunikace (RMI, JDBC, zasílání zpráv). Softwarové artefakty jsou označeny takzvanými stereotypy. Jedná se o dodatečná metadata, které konkretizují daný uzel nebo spojení. Vizuální podoba stereotypu je buď textová, nebo ve formě ikony.



Obr. 7-5: Diagram nasazení univerzitního informačního systému [Am02].

7.2.4 Component diagram – diagram komponent

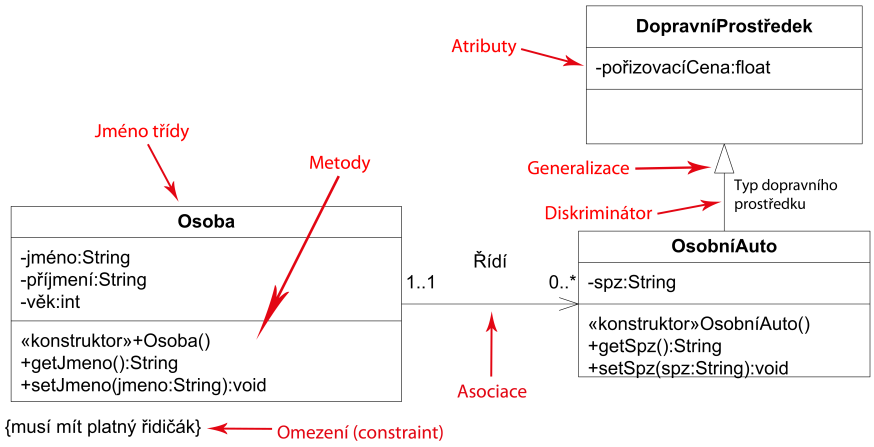
Diagram nasazení nám sice dává pohled na jednotlivé uzly systému, ovšem nespecifikuje rozmístění softwarových komponent. Proto byl vytvořen diagram komponent, který zachycuje rozmístění skutečných softwarových komponent do jednotlivých uzlů či vrstev a jejich interakce. Tyto vrstvy mohou být ve skutečnosti na jednom hardwarovém stroji, například webový server může běžet na stejném stroji jako relační databáze. Následující obrázek ukazuje použití diagramu komponent pro popsání systému s aplikačním serverem a databází. Při bližším prostudování zjistíte, že některé informace, které diagram vyjadřuje, se překrývají s diagramem nasazení. Běžně se oba diagramy doplňují a notace (např. použití stereotypů) je podobná.



Obr. 7-6: Diagram komponent

7.2.5 Diagram tříd – class diagram

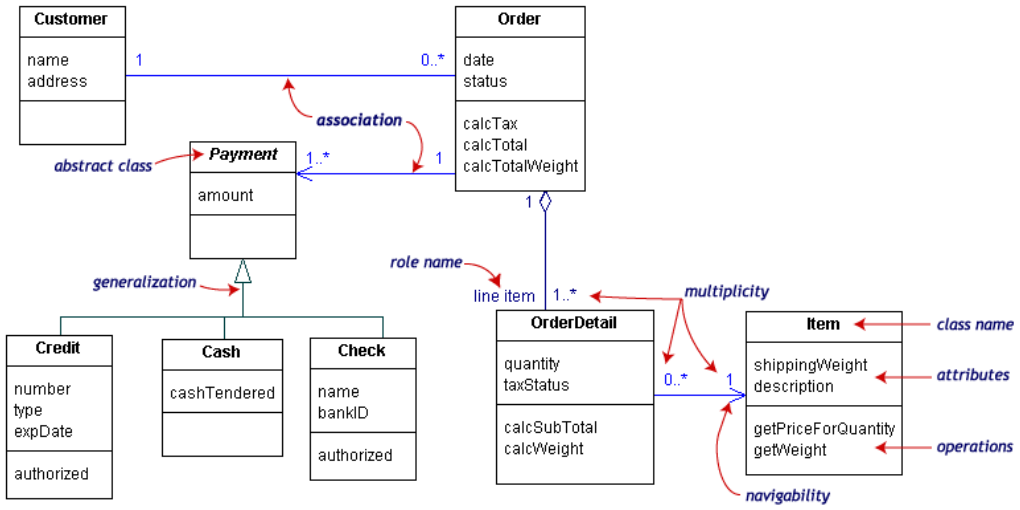
Abychom mohli začít vytvářet informační systém dle přístupu UML, potřebujeme snížit abstrakci na úroveň pohledu implementátora. K tomu využijeme diagram tříd, který je nejnižší úrovní abstrakce UML.
Diagram tříd bývá považován za základní model objektově orientovaného přístupu k analýze a návrhu IS. Poskytuje základní přehled o objektech, které budou v rámci vyvíjeného IS vystupovat. V dalších fázích analýzy a návrhu systému je model postupně rozpracováván až do potřebné podrobnosti. Na následujícím obrázku můžete vidět jednoduchou ukázku použití UML diagramu.



Obr. 7-7: Ukázka použití diagramu tříd

Účelem diagramu tříd je zobrazit statický pohled na vyvíjený systém, což znamená zobrazit vnitřní objekty systému, vlastnosti těchto objektů a vztahy mezi objekty. Objekt je v tomto případě konkrétní prvek vyskytující se v systému, který má definovány vlastnosti (pomocí atributů a jejich hodnot) a chování (pomocí operací), třída je poté celá skupina objektů, které mají podobné vlastnosti (atributy) a stejné, nebo alespoň podobné chování, kde atributy jsou položky sloužící k popisu vlastnosti třídy resp. objektu. Operace jsou následně algoritmy popisující reakce objektu na vzniklé situace a na další objekty. Diagram se nemusí omezovat pouze na názvy tříd, atributy a metody, v ukázce je například použit stereotyp pro označení konstruktoru. Pro objekty můžete definovat také omezení - v diagramu je specifikováno, že osoba, která řídí auto, musí mít platné řidičské oprávnění.

Důležitým krokem pro tvorbu diagramu tříd je dědičnost. Daný objekt je instancí (tj. konkrétním výskytem) určité třídy. Objekt ze třídy dědí atributy a operace dané třídy. Atributy přitom u objektu nabývají konkrétních hodnot. Dědičnost existuje i mezi třídami. Třída nižší úrovně v hierarchii tříd dědí vlastnosti (atributy a operace) třídy vyšší úrovně. Atributy a operace, které třída zdědila ze strany třídy vyšší úrovně lze doplnit dalšími atributy a operacemi, které již budou charakterizovat danou třídu na úrovni nižší. Vztahům mezi objekty následně říkáme vazby. Tímto postupem lze vytvářet složité objektové struktury.
Následující diagram tříd znázorňuje možný návrh systému pro objednávku zboží z webových stránek:



Obr. 7-8: Ukázka použití diagramu tříd (Zdroj: [Mi09])

Diagram se skládá z většiny základních objektově-orientovaných prvků. Obsahuje sedm běžných tříd, jednu abstraktní a vazby tvoří asociace klasické i navigační, kde v obou případech je použita násobnost a jedna vazba je pojmenovaná. Dále je zde obsažena generalizační vazba a agregace. Třídy v návrhu neobsahují datové typy, ani parametry a návratové hodnoty operací. Diagram neslouží pouze pro analytický pohled nad navrhovaným systémem. Pokud bychom doplnili atributům datové typy a také parametry a návratové hodnoty operacím, lze z diagramu automatizovaně generovat kostru zdrojového kódu. Generátory zachovají i agregační a generalizační vazby.

Kontrolní otázky:

1. Jaké tři základní modely UML byste použili v projektu?
2. Jaký UML model byste použili při popisu budoucího nasazení aplikace (rozmístění komponent na sítí, serverech, klientech)?
3. Jsou UML modely svázány s nějakou konkrétní metodou či metodikou?

Úkoly k zamyšlení:

Zamyslete se nad hodnotou různých UML modelů v různých projektech a nad jejich formálností. Je nezbytné, abychom vytvářeli v týmu o 3 lidech všechny modely UML verze 2.X a všechny vytvářeli pomocí sady nástrojů IBM Rational? Jaké by byly přínosy, výhody, popřípadě problémy tohoto přístupu? Znáte lepší přístup, který by byl v dané situaci vhodnější?

Korespondenční úkol:

Vytvořte Use case model a popis některých scénářů daných use case pro poskytovatele letenek. Jako inspiraci si zvolte např. servery: www.csa.cz, www.gtsint.cz, www.letuska.cz. Pozorně si ještě jednou prostupujte z dostupných zdrojů (hlavně studijní materiály pro předměty Ročníkový projekt a Softwarové inženýrství) základní chyby a pokuste se jich vyvarovat.

Shrnutí obsahu kapitoly

Tato kapitola představila modelovací jazyk UML, v úvodu bylo zmíněno, k čemu jazyk slouží, co je, co není a jaké jsou jeho silné a slabé stránky. Dále byly stručně představeny některé modely UML a jejich smysl, účel a využití v projektu.

8 Pokročilý návrh v UML

V této kapitole se dozvíte:

- Jak přesněji pracovat s vybranými diagramy.
- Jak přiblížit procesu návrhu do procesu implementace v UML diagramech
- Jak obecně používat ostatní diagramy s využitím konvencí pojmenování a stereotypů.

Po jejím prostudování byste měli být schopni:

- Korektně namodelovat danou problematiku ve vybraných diagramech.
- Rychle pochopit princip a způsob práce s dalšími diagramy UML.

Klíčová slova této kapitoly:

UML

Doba potřebná ke studiu: 3 hodiny

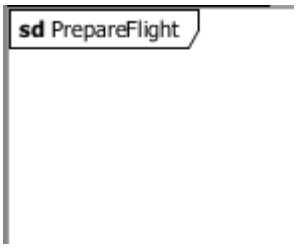
Průvodce studiem

Kapitola představuje problematiku modelování pokročilejších požadavků a a jejich konstrukcí do vybraných UML diagramů. Představuje také obecné pojmy týkající se UML diagramů.
Ke studiu se doporučuje vyzkoušet si ukazované příklady ve vlastním, vhodně zvoleném modelovacím nástroji.
Na studium této části si vyhraďte 3 hodiny.

Problematika UML diagramů byla již částečně představena ve studijní opoře pro předmět Informační systémy 1. Protože spolu se získanými znalostmi je třeba vědět, jak korektně zapisovat i pokročilejší konstrukce v UML diagramech a získat obecnější přehled o jejich fungování, některé principy a vybrané diagramy zde budou představeny blíže. Jedná se ale stále o obecný přehled, zájemce o bližší studium odkazujeme například na [IBM1, IBM2, IBM3, IBM4, Gr], pro podrobnější informace pak na [Pe].

8.1 Diagramy

Každý diagram v UML je podle standardu 2.0 je reprezentován rámcem, který vlevo nahoře v malém čtverečku obsahuje zkratku typu diagramu a název diagramu.
Rámce jsou obecnou strukturou UML, která umožňuje zahrnovat v sobě jiné artefakty a v levém horním rohu vždy obsahuje název nebo očekávaný popis rámce.



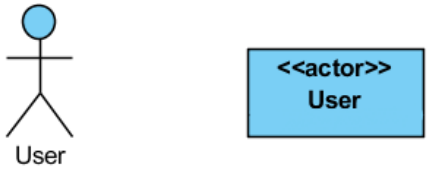
Prefix **sd** říká, že tento diagram je sekvenční. Každý z diagramů v UML má svou vlastní zkratku. Za prefixem následuje název diagramu.

Obecně se však jedná o rámec, který lze použít kdekoliv pro „zabalení“ skupiny artefaktů, které k sobě patří. Tehdy se samozřejmě prefix typu diagramu vynechává a vkládá se pouze název rámce. S takto představenými rámci se setkáme i v dalších diagramech.

8.2 Stereotypy

Stereotyp je technika, která v UML slouží k upřesnění významu artefaktů. Cílem je blíže specifikovat chování daného objektu, nebo jeho vlastnosti. V UML se diagramy zakreslují pomocí artefaktů – značek, které reprezentují zachycované objekty. UML pro každý artefakt definuje obecný vzhled, jak má vypadat. Někdy ale designer/analytik potřebuje zachytit určité odlišnosti, fakta, specifika, které chování daného artefaktu mírně upravují nebo odlišují od běžného chování. Toto se provádí pomocí tzv. stereotypů.
Stereotypy se zakreslují typicky do artefaktu nahoru nad název popisující daný artefakt, nebo nad daný artefakt (typicky u čar), a to do dvojice lomených závorek: <<název_sterotypu>>. Obecně použití stereotypu není omezeno, ale jejich nadměrné používání může diagramy značně znepřehlednit. Z toho důvodu, že určité stereotypy se používají často, byly nahrazeny zvláštním symbolem artefaktu, který sám o sobě stereotyp reprezentuje. Tehdy je samozřejmě důležité používat tyto nahrazující grafické artefakty, z důvodu jejich zavedenosti a snadné čitelnosti. Nejběžnějšími příklady stereotypů a jejich nahrazení uvádí následující obrázky.

Kurzíva v názvu typicky značí abstraktní objekt. Můžeme jej ale také zapsat běžným fontem, a doplnit stereotyp *abstract*.



Postavička v UML značí aktora. Můžeme ale použít obecný tvar a připojit stereotyp *actor*.

8.3 Identifikace objektů

V diagramech UML typicky potřebujete identifikovat artefakty k nejrůznějším objektům – třídy potřebují své názvy, stejně jako aktoři, případy užití a další. V rámci UML 2.X se můžete odkazovat třemi různými způsoby. Následující příklady budou uvažovat použití v diagramu tříd (protože je nejsnáze pochopitelný i pro začínající OOP programátory a UML designéry), ale lze je samozřejmě využít obecně v libovolném diagramu, kde toto použití bude dávat smysl.
Prvním způsobem je klasický název:

```
Flight
```

Takto uvedeno se jedná o obecnou šablonu skupiny objektů, nejblíže asi pojmu „název třídy“. Jedná se o klasický text (nepodtržený, ne kurzívou)

reprezentující název. V diagramu tříd tedy budeme vědět, že artefakt popisuje třídu nazvanou *Flight*.
Druhým způsobem je odkaz na instanci:

```
EZY5495
```

Podtržený text znamená, že se nejedná o šablonu (tedy ne třída), ale přímo o jednu konkrétní instanci. Tato jedna konkrétní instance má své **jedinečné jméno**. Je-li třeba (a pokud je to známo), lze za dvojtečku doplnit název šablony (třídy), ke které se instance vztahuje.[12] Pro zdůraznění se doplňuje dvojtečka „za“ i v případě, že třída není známa (její název se pak nepíše). Po doplnění bude zápis vypadat například:

```
EZY5495 :
EZY5495 : Flight
```

Poslední variantou je tzv. „role“. Role reprezentuje skupinu instancí stejného chování, ale neodkazuje nutně na konkrétní objekt. Od zápisu instance se liší nepoužitím **podtržení**. Abychom byly schopni odlišit role od tříd, u rolí vždy za název doplníme dvojtečku, i přesto, že nevíme, jaký název třídy za roli uvést. Následují příklady rolí reprezentující „lety společnosti easyJet s označením letu začínající na EZY“:

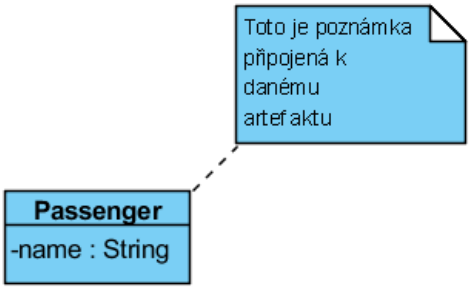
```
EZYFligth :
EZYFlight : Flight
```

Pro doplnění – někteří návrháři naopak pro zdůraznění, že určitý název není rolí, ale přímo typem, přidávají dvojtečku „před“ zápis:

```
: Flight
```

8.4 Obecné poznámky

Někdy je třeba do UML zapsat informaci, která nejde klasickým způsobem pomocí standardních prvků zaznačit, ale její prezentace je důležitá. Tehdy lze využít obecného mechanismu poznámek, který umožňuje k libovolnému artefaktu pomocí přerušované čáry připojit obdélník s přehnutým rožkem, do kterého je možno vložit libovolný text.



Připojení poznámky má úplnou volnost – lze ji připojit k jakémukoliv jinému grafickému symbolu ve všech UML diagramech, měly by se však **používat co nejméně**, protože:

[12] Není to však nutné. Zejména v úvodních fázích návrhu může analytik chtít specifikovat, že existuje konkrétní instance, které bude mít určitý stav nebo určité chování, ale ještě nemůže vědět, do jaké třídy (zda vůbec) bude patřit.

- Značně snižují přehlednost UML diagramu, zvláště, pokud se jich v daném diagramu vyskytuje větší množství;
- Jsou neformálním mechanismem. Zatímco ostatní stavební bloky jsou formální a dají se tedy automatizovaně zpracovávat (a převádět například na zdrojový kód ve vybraném jazyce), poznámky formálně zpracovat nelze, protože můžou obsahovat cokoliv.

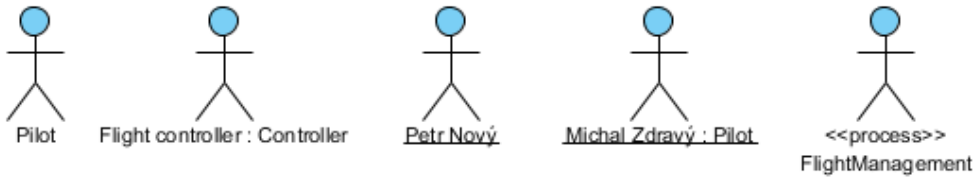
Cílem tedy je použití poznámek minimalizovat a nahrazovat je jinými, standardními a formálními přístupy, nejlépe stereotypy.

8.5 Diagram případů užití – Use case diagram

Diagram případů užití již byl poměrně podrobně představen v předchozí studijní opoře věnované předmětu Softwarové inženýrství. Tedy jenom stručně: případy užití ukazují funkcionalitu nabízenou systémem. Hlavním cílem je vizualizovat tuto funkcionalitu vzhledem ke tzv. aktorům – typicky lidským uživatelům, a také zachytit vazby mezi jednotlivými případy užití.
Důležité oblasti, které je třeba si zapamatovat při tvorbě obecného diagramu případu užití:

- Přestože **aktor** bývá typicky graficky reprezentován symbolem postavičky, a některé zdroje uvádějí, že se jedná o lidského uživatele, obecná definice říká, že „aktor je subjekt, který je v nějakém vztahu komunikace se systémem“. **Aktor tedy vůbec nemusí být člověkem**, ale může jím být například webová služba, nebo jiný systém, který bude využívat služeb modelovaného systému. Aktor také nemusí aktivně vyvolávat komunikaci, ale může být konzumentem služeb vyžadovaných modelovaným systémem – například, při modelování systému bankomatu bude tento bankomat muset komunikovat s bankou. Banka, ač sama aktivně komunikaci s bankomatem nevyvolá (vždy je bankomatem zavolána), bude i v tomto případě v systému aktorem. Typicky tedy aktorem bývá uživatel, organizace, stroj nebo jiný externí systém.
- V každém UC diagramu je velmi nutné modelovat **hranice systému**, typicky jako obdélníkové orámování UC, které do systému patří. Čtenář diagramu bude okamžitě vědět, které věci jsou součástí systému a které jsou již vně.
- Vzhledem k vztahům mezi případy užití (vztahy budou uvedeno dále) je důležité hledat a analyzovat případy užití z pohledu jejich užitelnosti. V diagramu by se neměl vyskytovat případ užití, který není spojen s žádným aktorem a ve vztahu s některým z ostatních případů užití. Stejně tak aktor, který nevyvolává žádný případ užití, není aktorem vůči modelovanému systému.

U aktorů můžeme využívat identifikaci objektů, jak bylo uvedeno výše.

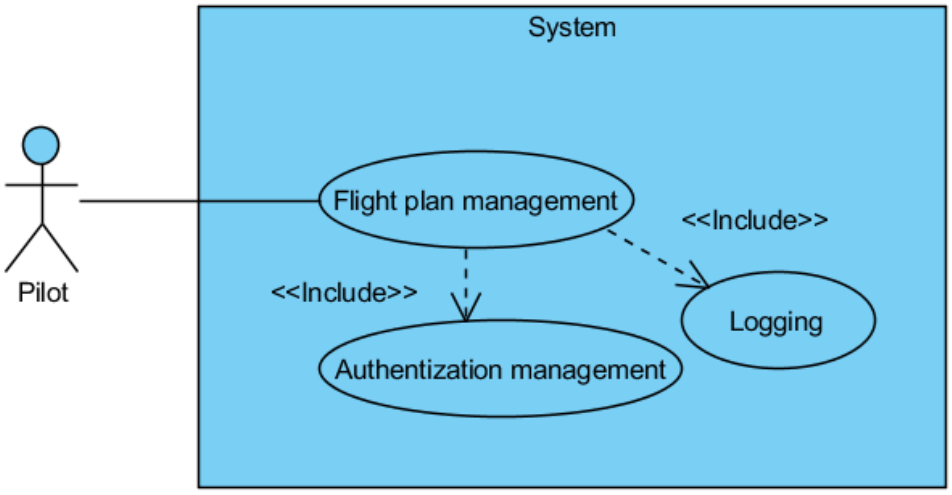


Pilot je obecným aktorem. Flight Controller je nějaký letový řídící (ne konkrétní instance, ale obecná role) – navíc instance této role je povinně řídícím. Petr Nový je konkrétní instance – aktor. Michal Zdravý je také konkrétní instance – aktor a navíc je pilotem. FlightManagement je aktorem, není to ale osoba, nýbrž nějaký vykonávaný proces.

V některých nástrojích lze specifikovat i obecný tvar pro aktora (například obdélník), jinak je nutno explicitně (ideálně stereotypem) určit, o jaký typ aktora se jedná.
Aktoři se se svými případy užití (tj. s případy, které používají, nebo kterými jsou využíváni) spojují plnou čarou. Je-li třeba explicitně naznačit jednosměrnou komunikaci, může být čára doplněna šipkou.
Kromě vztahu „aktor – případ užití“ mohou vznikat vztahy i mezi aktory a stejně tak mezi případy užití.

8.5.1 Vnoření případů užití – include

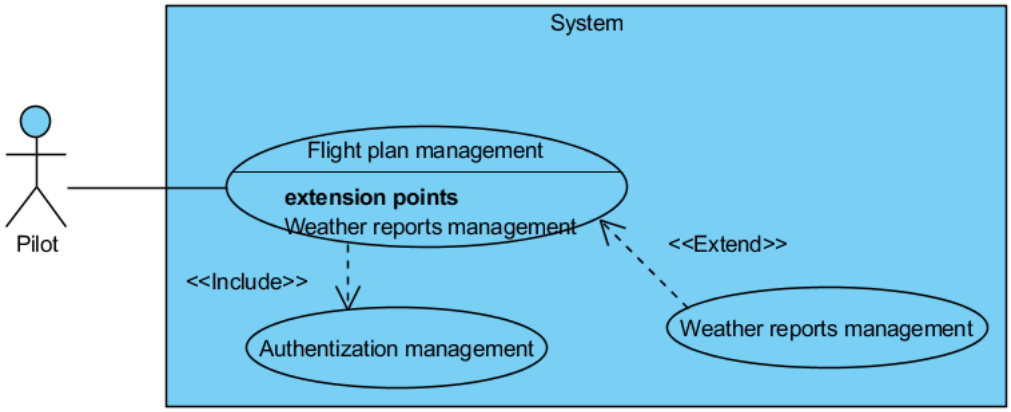
„Vnoření“ případů užití je jedním z nejčastějších a nejběžnějších vztahů mezi případy užití. Závislost *include* od prvního případu užití (nazývaného také *base use case*, dále pojmenovaného jako A) k druhému případu užití (*inclusion use case*, dále pojmenovaného jako B*)* značí, že **první případ užití A bude zahrnovat (nebo volat) druhý případ užití B**. Jeden případ užití může mít vnoření do libovolného počtu dalších případů užití. Vnoření se zaznačí přerušovanou šipkou s přidaným stereotypem <<include>>. Případ užití A je se svým popisem zodpovědný za korektní určení „kdy a jak“ bude daný vnořený případ užití B volán. Tento popis ale již typicky není součástí UML diagramu, ale značí se do textové části popisu projektu.



Ve výše uvedeném případě může pilot pracovat se správou letových plánů. Před práci s ní se ale typicky bude muset autentizovat (povinně) a práce s letovými plány se bude také monitorovat do logů.
Důležité je uvědomit si vztahy obou případů užití. Případ užití A uvnitř **nutně potřebuje vyvolat** případ užití B, a tudíž o případu užití B ví. Samostatně bez něj nemůže fungovat. Oproti tomu případ užití B nic neví o případu užití A. Typicky, vnořované případy užití (v našem případu B) nejsou schopné fungovat samostatně. **Typickým použitím *include* je vytknutí společných funkcionalit** do nového případu užití, který si budou původní případy užití připojovat.

8.5.2 Rozšíření případů užití – extends

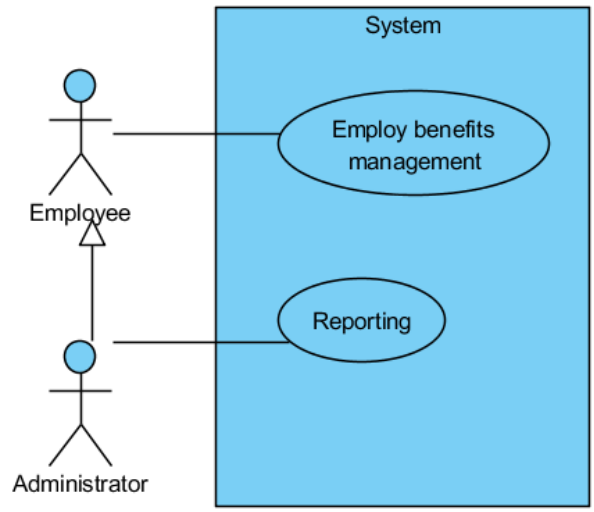
Rozšíření je další varianta vztahu případu užití. Tentokráte máme první případ užití (nazývaný *extensit use case*, dále označený jako B (!)), který rozšiřuje původní případ užití (nazývaný *base use case*, dále označený jako A). Důležité je povšimnout si **opačné orientace vztahu** oproti variantě *include*.
Rozšíření umožňuje rozšířit případ užití A o nové funkcionality reprezentované případem užití B, které původní případ užití neobsahuje. Do definice případu užití A se zavádějí tzv. *extension points* (body rozšíření), které říkají, že v určitých místech/fázích případu užití A lze jeho chování rozšířit.



Výše uvedený příklad říká, že případ užití *Flight plan management* může být rozšířen o část umožňující zjišťovat a pracovat s informacemi o počasí. Toto rozšíření není povinné – pokud modul správy počasí nebude existovat, pilot může pořád pracovat se správou letových plánů, jenom si informace o počasí bude muset zjistit někde jinde.
Závislost případů užití je zde tedy **opačná** oproti *include*. Základní (rozšiřovaný) případ užití A neví nic o rozšiřujícím případu užití B (vyjma *extension points*) a **je schopen plně fungovat i bez jeho existence**. Případ užití B volitelně rozšiřuje případ užití A a typicky o jeho existenci ví a odkazuje se na něj. Opět, rozšiřující případ užití B není typicky schopen fungovat samostatně.

8.5.3 Dědičnost mezi aktory

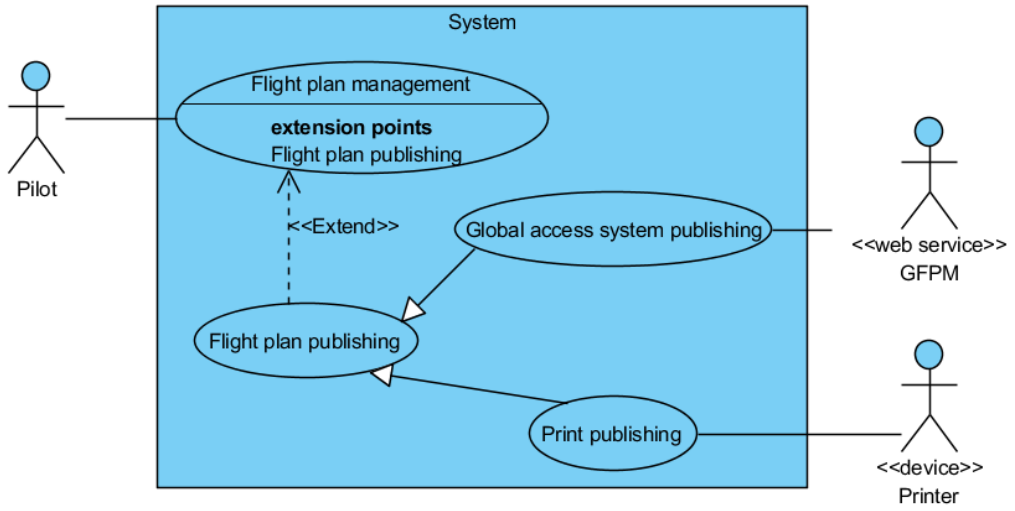
Dědičnost lze chápat jako klasický vztah generalizace/specializace, kdy chceme při modelování naznačit, že určitý aktor (potomek) sdílí všechno, co „umí“ jiný aktor (předek) a navíc může mít své vlastní další schopnosti. Potomek má dostup na všechny případy užití předka, navíc může mít své případy užití, na které se předek nedostane.



Zaměstnanec může pracovat v systému se zaměstnaneckými benefity. Administrátor může totéž, navíc však může nahlížet do sestav a reportů.

8.5.4 Dědičnost mezi případy užití

Dědičnost mezi případy užití je poměrně složitější v případě složitých nebo nevhodně namodelovaných případů užití. Opět, principem je získat od případu užití (předka) do případu užití (potomka) dědit aktory, sekvence chování a body rozšíření (*extensit points*, bude vysvětleno dále). U složitějších případů užití se dědičnost používá hlavně ke zpřehlednění aktorů, které budou daný případ užití používat.



Výše uvedený příklad ukazuje, že letové plány lze obecně publikovat – publikování konkrétně bude umožňovat systém buď do globální celosvětové databáze letových plánů, nebo na lokální tiskárnu.

8.6 Diagram aktivit – Activity diagram

Diagram aktivit v UML je základní, poměrně jednoduchý diagram, určený k zachycení chování které závisí na výsledcích vnitřních procesů. Diagram aktivit je reprezentován tokem mezi očekávanými operacemi (aktivitami) – k další aktivitě se přechází po splnění aktivity předchozí. Diagramy aktivit jsou vhodné při definici operací jako doplnění pro diagramy případů užití, kdy

umožňují zjistit a definovat toky dat, které mezi sebou propojují právě jednotlivé případy užití.
Každý diagram aktivit se skládá minimálně ze 3 základních bloků:

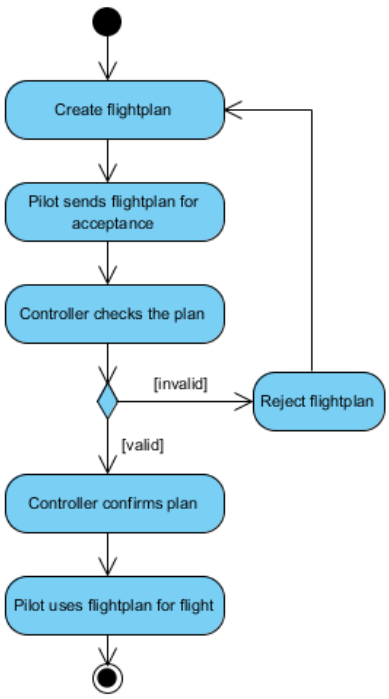
- Počáteční uzel – reprezentován černým kolečkem;
- Aktivitou (akcí) – reprezentován obdélníkem s oblými rohy a popisem akce;
- Koncovým uzlem – reprezentován černým kolečkem s černým orámováním.

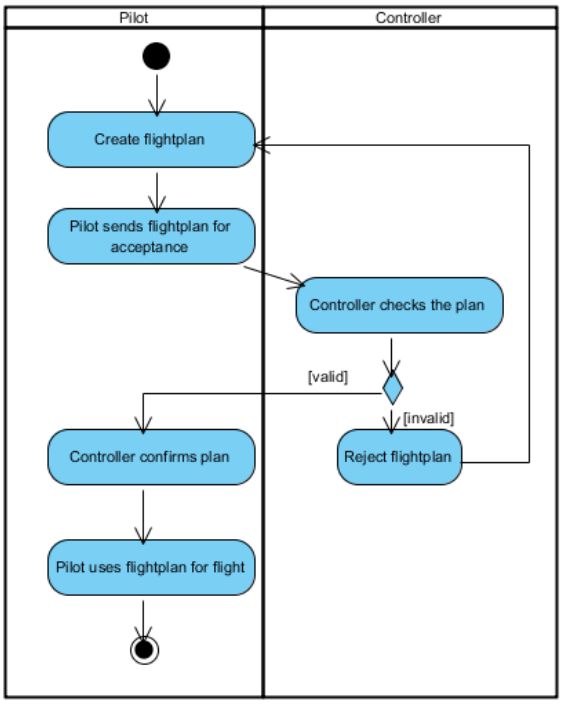
Tyto bloky jsou mezi sebou spojeny orientovanou jednosměrnou šipkou.

Počáteční uzel říká, kde aktivita vzniká. Aktivity popisují jednotlivé akce, které se v průběhu diagramu aktivit dějí. Koncový uzel značí konec běhu aktivity v diagramu. Diagram může obsahovat maximálně jeden počáteční uzel, ale může obsahovat více koncových uzlů. Z bloku akce nemůže vycházet více než jedna odchozí šipka (popis by potom nebyl deterministický).

Klasickým blokem typickým v diagramu aktivit je rozhodování. Místo rozhodnutí je reprezentováno kosočtvercem, ze kterého může vycházet více odchozích hran. U každé hrany se do hranatých závorek zapisuje podmínka, která má být splněna, aby se danou hranou mohlo pokračovat.

Další volbou zpřehledňující diagram aktivit jsou tzv. „swim-lanes“ – plavební dráhy. U složitějších diagramů, nebo je-li to třeba, lze definovat, které konkrétní objekty jsou s danou aktivitou spojeny. V diagramu se následně vymezí obdélníkové prostory, kam se její aktivity zapisují. Představené formalismy ukazují následující obrázky.

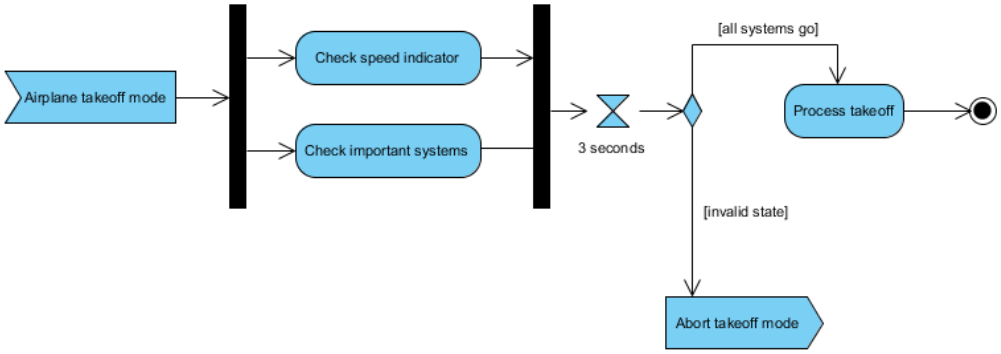




Jen ve stručnosti další stavební bloky, které lze použít:

- Paralelní vykonání lze zachytit uzavřením bloku aktivity do dvou svislých čar.

- Je-li třeba zdůraznit časové hledisko, lze použít artefakt přesýpacích hodin s doplněním časového intervalu.
- Další běžnou oblastí jsou tzv. signály. Aktivity nemusí nutně začínat počátečním uzlem – aktivitu může vyvolat i vnější vstup – tzv. signál (událost). Zakresluje se obdélníkem s vnořenou stranou. Diagram aktivit může obsahovat několik signálů, které mohou do procesu diagramu aktivit vstoupit. Obdobně, aktivita může vyvolat výstupní signál (který se předává dále). Výstupní signál se zakresluje obdélníkem s vystouplou stranou (viz následující obrázek).
- Speciálním případem hrany je zalomená šipka do tvaru blesku. Taková šipka upozorňuje na nestandardní výjímečné volání způsobené typicky chybou (tzv. výjimky).



8.7 Diagram tříd – Class diagram

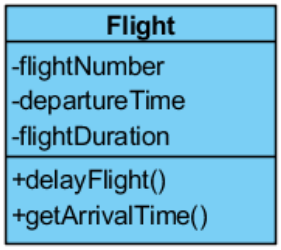
Diagram tříd je, jak již bylo představeno dříve, základním diagramem pro reprezentaci typů, které budou modelovány ve vytvářeném systému. Typicky se modelují čtyři základní definice: třídy, rozhraní, datové typy[13] a komponenty. Obecně budou nejdříve příklady ukázány na diagramu tříd, na případné odchylky bude ukázáno později.
Na začátek trochu opakování.

8.7.1 Základní modelování třídy

Základní reprezentace třídy je obdélník. Nutně musí obsahovat tučný text definující název třídy.

V pokročilejších fázích návrhu již potřebujeme do třídy doplnit budoucí modelované členy tříd. Reprezentace třídy se změní na obdélník rozdělený horizontálně na tři části. V horní části se nachází pojmenování. Prostřední část obsahuje názvy třídních proměnných, v nejnižší části se nacházejí metody. I pokud ještě nejsou známy parametry metod, k názvům metod se doplňují (prázdné) kulaté závorky.

[13] Ve smyslu obecnějšího pojmu, než třída, například referenční, hodnotové, výčtové aj, dle zvolené technologie.



Na výše uvedeném obrázku již lze vidět ještě jeden doplněný artefakt – jedná se o definici viditelnosti. UML umožňuje modelovat základní viditelnost pomocí symbolů:

- + public
- # protected
- - private
- ~ package

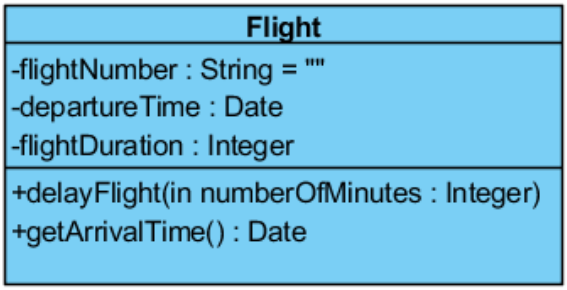
V pokročilejší fázi návrhu již můžeme doplnit další důležité poznámky k chování třídy. Zejména se samozřejmě doplňují datové typy, parametry funkcí, návratové hodnoty a výchozí hodnoty. Obecná syntax pro úplný zápis třídní proměnné může vypadat:

```
<visibility> <name> : <dataType> = <defaultValue>
```

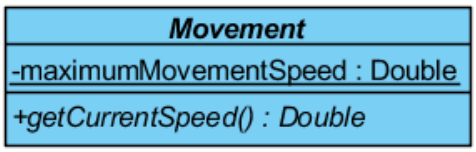
Obdobně u funkcí můžeme zapsat obecnou syntax – všimněte si, že u jazyků, které to podporují, můžeme explicitně ujasnit chování parametrů (vstupní, výstupní, explicitně předávané referencí apod.):

```
<visibility> <name> (<in|out|ref> parameterName :
parameterType) : returnType
```

Návratový typ funkce doplňujeme pouze tehdy, pokud není *void*.

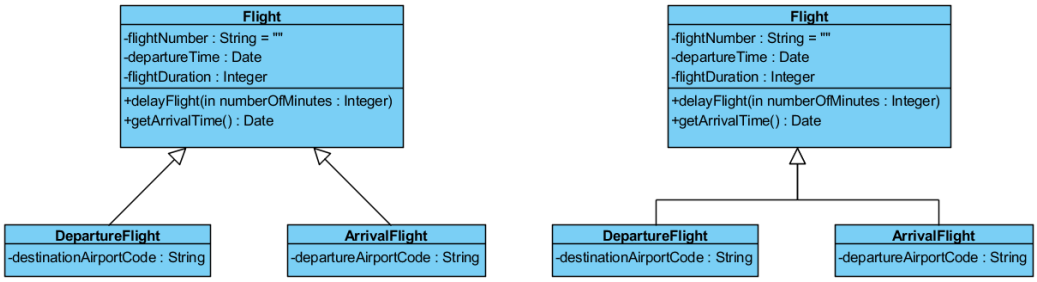


Pokud chceme vyjádřit, že třída nebo některý z jejich členů je abstraktní, zapíšeme jej pomocí *kurzívy*. Pokud chceme ve třídě zmínit statického člena, podtrhneme jej.



8.7.2 Vztahy mezi třídami – dědičnost

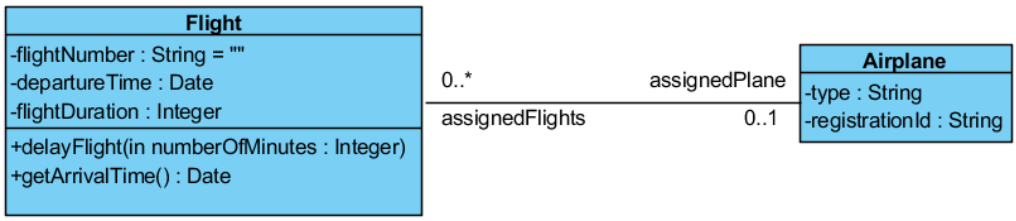
Základním vztahem mezi třídami je dědičnost. Zakresluje se pomocí šipky s plnou čarou a prázdnou špičkou. Je možno využít šipky přímé, i zalomené do stromu (které jsou typicky přehlednější).



8.7.3 Vztahy mezi třídami – asociace, agregace, kompozice

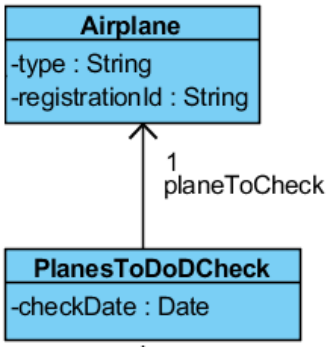
Problematika asociací je poměrně rozsáhlá, budou proto představeny alespoň ty nejzákladnější varianty.

Základní variantou je obecná asociace identifikující nějaký vztah mezi dvěma třídami. Obě třídy vědí, že jsou ve vztahu s jinou třídou. Zakresluje se pomocí plné čáry jako spojnice, doplňují se povinně názvy asociací a typicky kardinalita.



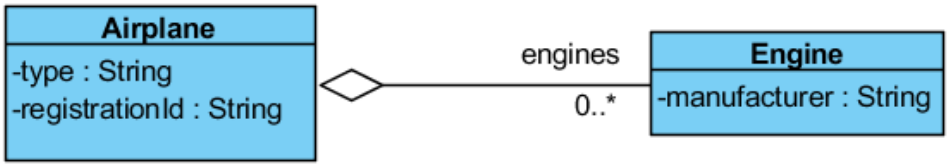
Názvy asociací se zapisují **napříč** k druhé třídě, tedy v našem případě třída *Flight* bude mít jeden z atributů *assignedPlane* a naopak třída *Airplane* bude mít třídní proměnnou *assignedFlights*.

Dalším případem je jednosměrná asociace – tehdy jedna ze tříd neví, že je ve vztahu s druhou třídou.

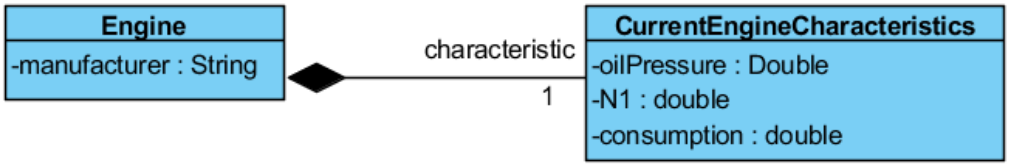


Název asociace a kardinalita se tedy nastavuje pouze u jedné strany asociace, čára je navíc typicky doplněna šipkou (případně kardinalitou 0), zápis se opět provádí napříč.

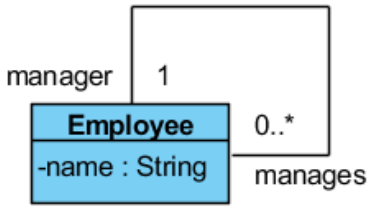
Silnější variantou asociace je **agregace**. Agregace naznačuje těsnější vztah mezi objekty, a zapisuje se pomocí kosočtverce bez výplně u nadřazeného objektu. Typicky se opět doplňuje o kardinalitu. Jedná se však stále o volnou vazbu, takže objekty mohou smysluplně existovat bez nadřazeného objektu – v našem případu motor patří k letadlu, ale lze jej demontovat a může existovat samostatně bez letadla.



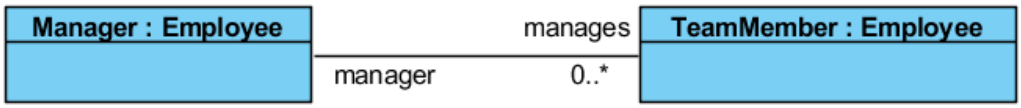
Poslední základní variantou asociace je **kompozice**. Ta se zakresluje pomocí kosočtverce s výplní a používá se v případě, kdy podřízený objekt nemůže smysluplně existovat bez svého nadřazeného objektu – v našem případě popisné charakteristiky motoru nemohou existovat bez motoru, který popisují[14].



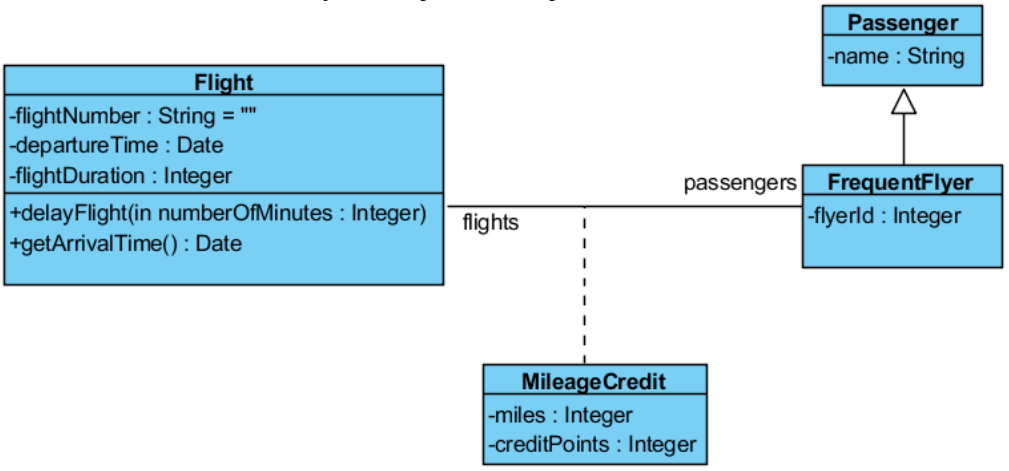
Další běžný modelovaný příklad jsou reflexivní asociace, kdy třída odkazuje sama na sebe. Jejich nevýhodou je nízká přehlednost.



Lze si povšimnout, že tento zápis není moc přehledný. Mnohem vhodnější je použít rolí (viz předchozí kapitola) a rozepsat jej pomocí vztahu mezi dvěma rolemi třídy *Emplyee*.



Názvy nejsou podtrženy, jedná se tedy o role a nikoliv o instance třídy.
Poslední ukázanou vlastností budou asociační třídy. Používají se v případech, kdy je třeba dekomponovat asociace M:N, nebo pokud potřebujeme k asociaci dodat nějaké atributy, které ani jednomu z asociovaných objektů nepřísluší. Vizualizaci asociační třídy ukazuje následující obrázek.

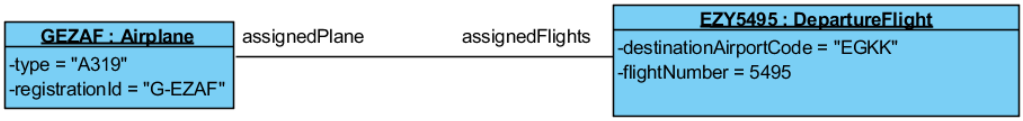


8.7.4 Reprezentace chování instancí

V určitých případech je potřebné ukázat na určité vlastnosti konkrétních rolí nebo objektů daných tříd. Tehdy se UML řídí výše uvedenými principy – pokud chceme naznačit, že informace uvedené v diagramu patří ke konkrétní instanci, potrhneme název instance (tam kde se původně vyskytoval název

[14] Jiným názorným příkladem je firma a její oddělení. I zde se jedná o kompozici, protože pokud firma zanikne, oddělení nemohou existovat dále.

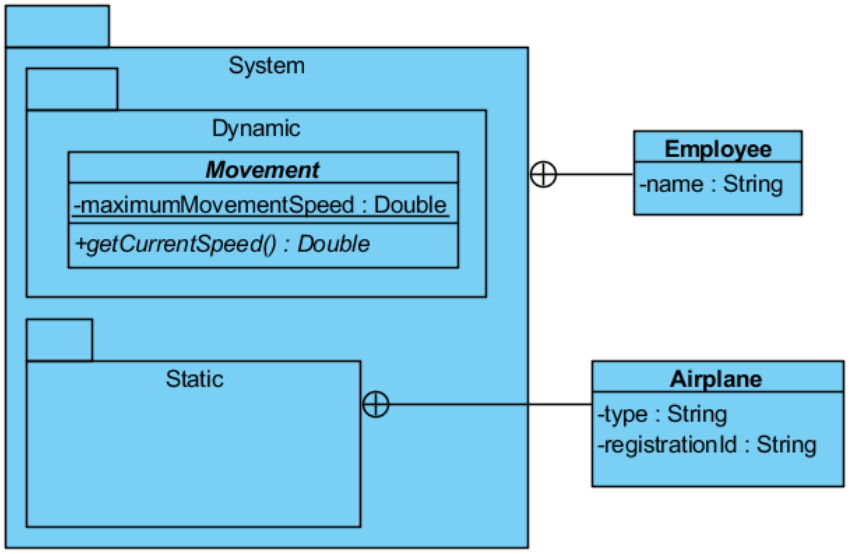
třídy) a doplníme i typ třídy, od které je instance odvozena. V dalších blocích, kde jsou uvedeny očekávané hodnoty třídních proměnných. **Doplňujeme pouze důležité hodnoty** – nedoplňují se hodnoty atributů, které nejsou známé, nebo důležité, přestože u obecného popisu třídy se vyskytují. Kombinací lze naznačit i vztahy mezi instancemi tříd, jak ukazuje následující obrázek.



Tato technika funguje až v UML 2.0 a ne všechny editory ji umí korektně zaznačit. V případě problémů je nutno využít stereotypů nebo poznámek.

8.7.5 Packages

Na závěr pouze stručně – kvůli přehlednosti se samozřejmě dají jednotlivé třídy umísťovat do balíčků, které lze rekurzivně zanořovat. Typicky lze použít dva přístupy – první spočívá v uvedení tříd dovnitř artefaktu reprezentujícího balíček – příkladem bude na níže uvedeném obrázku třída *Movement* v balíčku *System.Dynamic*. Druhou variantou je odkazování pomocí spojnic doplněných znakem „+“ v kolečku – příkladem je třída *Employee* v balíčku *System* nebo třída *Airplane* v balíčku *System.Static*. Obě formy zápisu jsou si ekvivalentní.

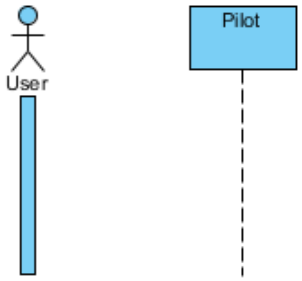


8.8 Sekvenční diagram

Dalším důležitým diagramem je sekvenční diagram, zachycující interakci mezi objekty v daném pořadí. Sekvenční diagram typicky v úvodních fázích návrhu zahrnuje obecné objekty a obecné postupy jejich komunikace, spolu s upřesňováním návrhu a prováděním implementace se upravují nebo transformují do diagramu popisujících typicky konkrétní třídy nebo objekty odpovědné za provádění určitých operací, a jejich metod.

8.8.1 Život objektu

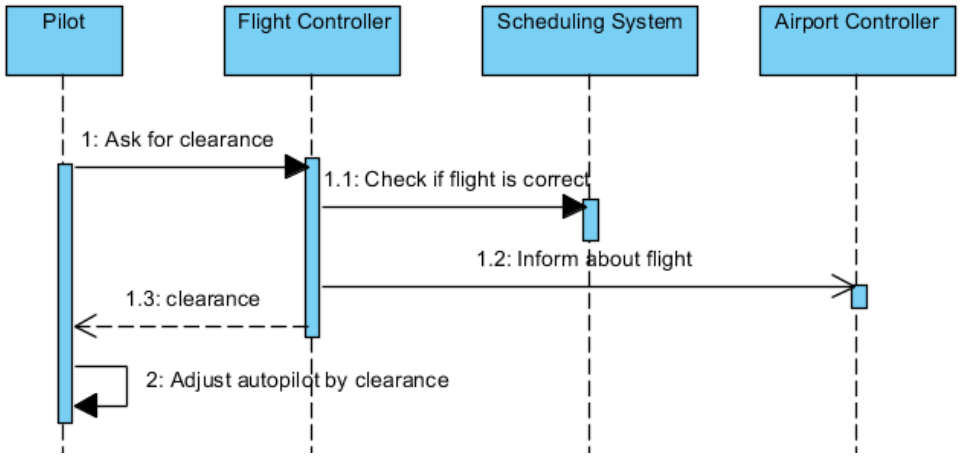
Jak bylo zmíněno, sekvenční diagram zachycuje objekt a jeho interakci s okolím. Proto má každý objekt v rámci sekvenčního diagramu (kromě svého názvu) i tzv. životní čáru. Ta se zakresluje pod objekt přerušovanou čarou a její existence reprezentuje **existenci konkrétní instance** vykonávající daný sekvenční diagram.



V záhlaví je opět uveden název objektu (třídy/ instance/role, dle pravidel uvedených na začátku kapitoly). Přerušovaná čára reprezentuje konkrétní instanci dané třídy. Objektem sekvenčního diagramu nemusí být něco, co si návrhář představuje jako třídu, ale samozřejmě také aktoři i jiné stavební prvky (databáze, rozhraní, webová služba apod.).

8.8.2 Zasílání zpráv

Cílem sekvenčního diagramu je reprezentovat zasílání zpráv. Proto typicky objekty interagují mezi sebou.
Základním mechanismem je zaslání zprávy. Zpráva má zdroj a cíl, zakresluje se typicky vodorovnou šipkou, nebo šipkou s plnou výplní.
Volání vytvoří nad životní linií přijímacího objektu modrý svislý pruh – tento pruh reprezentuje zpracovávání přijímaného požadavku (nereprezentuje tedy existenci instance)[15].



Ve výše uvedeném příkladu tedy jednotlivé body znamenají:

1	Pilot aktivně požádá letového řídícího o udělení povolení k provedení letu
1.1	Letový řídící aktivně zkontroluje přes objekt plánovače letů, zda je let připraven v pořádku.
1.2	Letový řídící informuje o budoucím letu letištního řídícího.
1.3	Letový řídící zašle zpět pilotovi povolení k letu.
2	Pilot zapíše získané povolení do počítače letadla.

Důležité poznámky:

- Nad šipky reprezentující komunikaci se dopisuje popisný text, který říká, jaká operace se provádí (jaká zpráva se zasílá). Opět, v úvodních

[15] Zjednodušeně si to lze programátorsky představit jako blok volané funkce { }, když je zpracováván. Toto přirovnání ale nemusí platit vždy.

fázích návrhu se může jednat o obecný text, později tyto texty typicky reprezentují volání metod volaných objektů. Volání se běžně pro přehlednost i číslují.

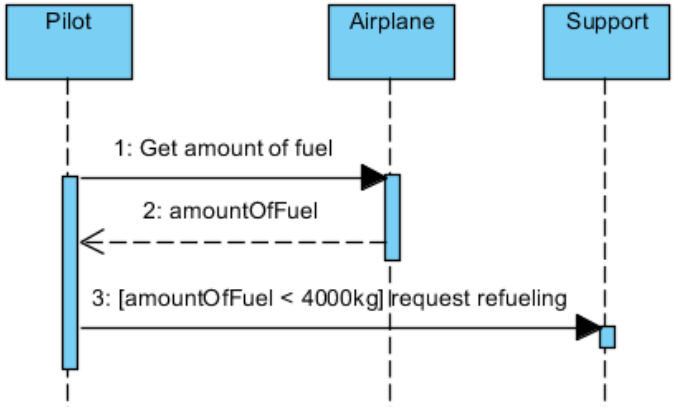
- Z pohledu pilota se jedná o jeden požadavek a **čekání na výsledek**. Bez výsledku pilot neví, zda může pokračovat. Protože se čeká na vyhodnocení volání, jedná se o **synchronní volání**. Tato volání jsou reprezentována plnou šipkou u volaného objektu. Synchronní volání je i volání 1.1.
- Vrácení informace o výsledku pilotovi se provádí v bodu 1.4 pomocí **přerušované** šipky vedoucí zpět k objektu, který interakci vyvolal. Tato zpětná šipka se ale zakresluje pouze tehdy, pokud chceme explicitně zdůraznit, že je volání ukončeno, nebo upozornit na vracenou hodnotu (kterou potom můžeme dále v diagramu používat). Velké množství těchto zpětných šipek značně zesložiťuje diagram. Ve volání 1 explicitně upozorňujeme na vracení objektu *clearance*, který dále můžeme využít. Oproti tomu u volání 1.1 nepožadujeme explicitně zpět navrácení hodnoty (můžeme například předpokládat, že v případě problému celý sekvenční diagram skončí chybou) a zpětnou přerušovanou šipku tedy neděláme.
- Volání 1.2 se provádí **asynchronně** – letový řídící informuje letištního řídícího o budoucím letu, ale nečeká dále na jeho odpověď a hned provádí další operace (zde 1.3). Šipka proto nekončí plným tvarem. V těchto případech samozřejmě nemá smysl kreslit navrácení hodnoty, protože nevíme, kdy se tato hodnota vrátí (za jak dlouho řídící letiště na zaslanou informaci zareaguje). Povšimněte si ještě jednou odlišností mezi synchronním a asynchronním voláním. U synchronního volání 1 pilot čeká na vrácení hodnoty 1.3 (i kdyby tam tato šipka nebyla, čeká na uběhnutí zpracování požadavku 1) a teprve potom pokračuje bodem 2. U asynchronního volání by pilot nečekal, odeslal by informaci letovému řídícímu a hned by pokračoval bodem 2 (což by samozřejmě nedávalo smysl). Je proto důležité tyto typy volání odlišovat.
- Bod 2 reprezentuje volání sebe sama – pilot může zaslat zprávu i sám sobě. Je-li třeba demonstrovat rekurzivní volání, vytvoří se u dané šipky další životní linie objektu (objekt má pak dvě překrývající se životní linie).

Tyto základy typicky stačí k modelování sekvenčního diagramu pro základní scénáře v brzkých fázích projektu. Postupně, jak se projekt a jeho řešení stává více konkrétním, potřebuje návrhář zachytit více a více programátorské konstrukce, které blíže objasňují, jak se bude daný kód provádět, jako jsou podmínky, cykly a další.

8.8.3 Guards

Tato konstrukce umožňuje vytváření jednoduchých podmínek (true/false), za kterých se dané volání provede či nikoliv, či specifikaci dalších vlastností

spojených s daným artefaktem. *Guard* se zapisuje do hranatých závorek k odpovídajícímu artefaktu (typicky před popis volání).
Jak bylo zmíněno, nejběžnější použití konstrukce *Guard* je určeno pro jednoduché pomínky; neumožňuje však větvení, jedná se vlastně o nejjednodušší variantu příkazu *if* bez *else*. Podmínka se zapisuje do hranatých závorek před popis volání.



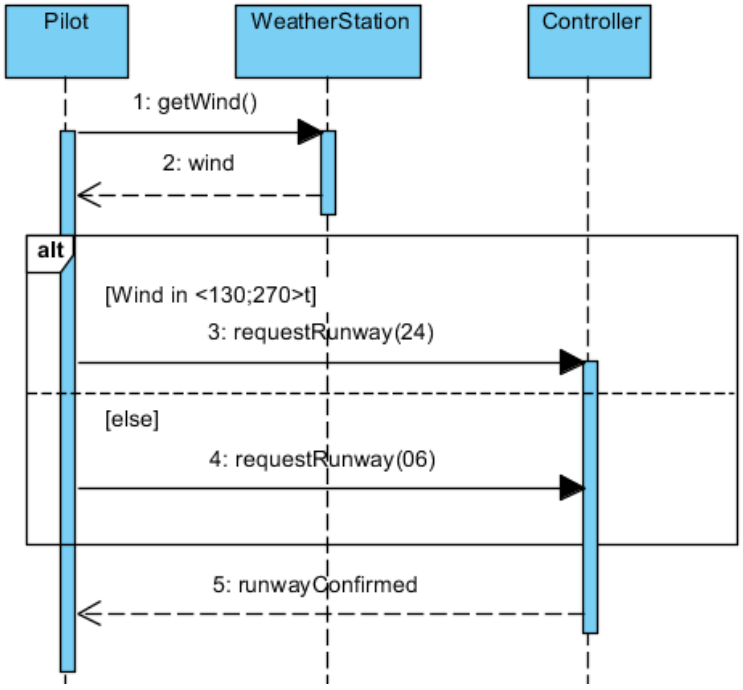
Další možnou variantou je jednoduchý zápis cyklu – například:

```
[for each airplane in airplanes]
```

8.8.4 Kombinované fragmenty – podmínky

Složitější variantou, kterou lze využít při modelování sekvenčního diagramu, je použití tzv. *kombinovaných fragmentů*. Kombinovaným fragmentem se rozumí jednoduchý **rámec**[16], který překryje životní čáry zainteresovaných objektů. Každý rámec má svůj odpovídající štítek, který říká, o jaký typ fragmentu se jedná.
Základním typem reprezentující výběr z několika možností je fragment *alternativa*. Jedná se o zápis, kdy může sekvenční diagram probíhat několika různými větvemi podle vyhodnocení podmínky. Jednotlivé části jsou alternativy od sebe odděleny přerušovanou čarou a stejně jako u prvku *Guard*, podmínky se zapisují do hranatých závorek. Rámec *alternativy* obsahuje štítek **alt**.

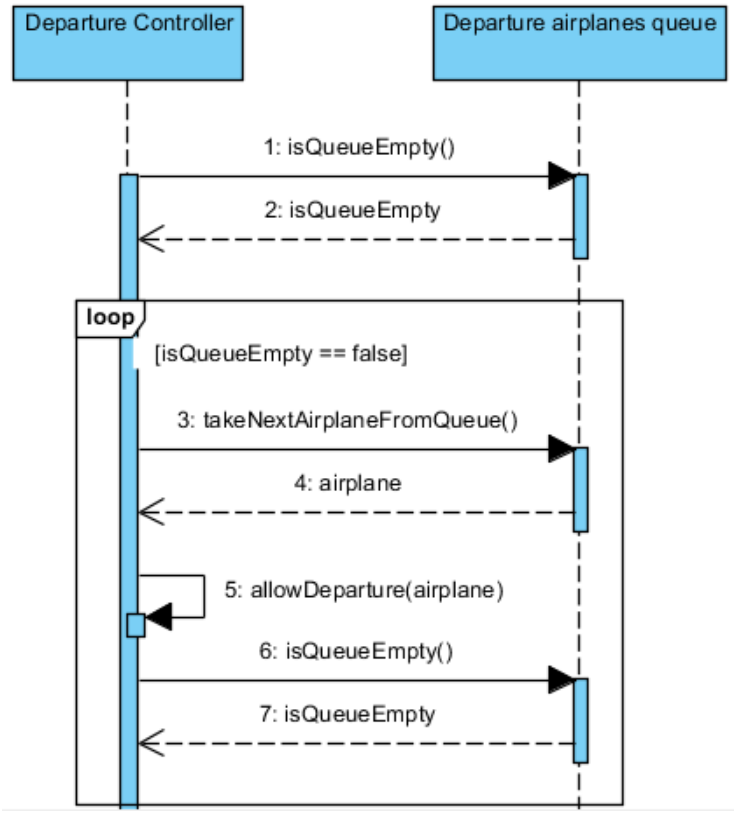
[16] Jedná se o stejný rámec, který byl představen při základním popisu každého diagramu. Tentokrát ale vlevo nahoře v rámci není uveden typ a název diagramu, ale jiný popis.



Ve výše uvedeném příkladu pilot požádá o ranvej 24, pokud jde vítr ze směru 130 – 270°, jinak požádá o ranvej 06.
Jednodušší obdobou alternativy je *option*, možnost – která nemá větev *else*. Jedná se o složitější *guard*, kdy do rámečku můžeme zapsat sadu různých interakcí (kdežto *guard* je vždy spojen pouze s jedním voláním). Rámec možnosti obsahuje štítek **opt**.

8.8.5 Kombinované fragmenty – cyklus

Další typickou návrhovou záležitostí sekvenčních diagramů jsou cykly. Využívá se stejného artefaktu jako u alternativ, tedy rámce, který zahrnuje všechny operace, které se mají provádět v cyklu. Rámec cyklu má štítek **loop**, navíc lze, stejně jako u rámce *alt*, do hranatých závorek specifikovat podmínku, která musí být splněna, aby se rámec provedl. Pokud podmínka není splněna, cyklus se opouští.



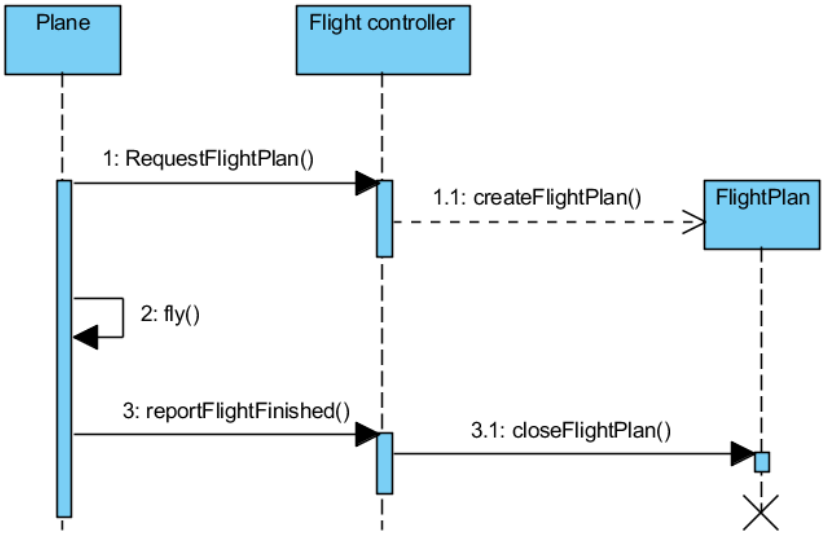
8.8.6 Kombinované fragmenty – ostatní

Kombinovaných fragmentů je více druhů a popis všech přesahuje rozsah této studijní opory. Běžně používané, které se mohou ještě hodit, jsou:

- Break (štítek *break*) – je poměrně podobný typu *alt* nebo *opt*. Obsahuje podmínku, která říká, kdy se do fragmentu vstoupí a operace v něm uvedené se provedou. Liší se ale tím, že pokud se příkazy ve fragmentu provedou, po ukončení fragmentu se již **neprovádí další příkazy** za tímto fragmentem (funguje tedy obdobně jako příkaz *break* v jazce Java).
- Parallel (štítek *par*) – reprezentuje jednoduchý fragment pro realizaci paralelních operací. Opět, jednotlivé paralelní bloky jsou od sebe odděleny vodorovnou přerušovanou čarou, jako varianty u fragmentu *alt*.

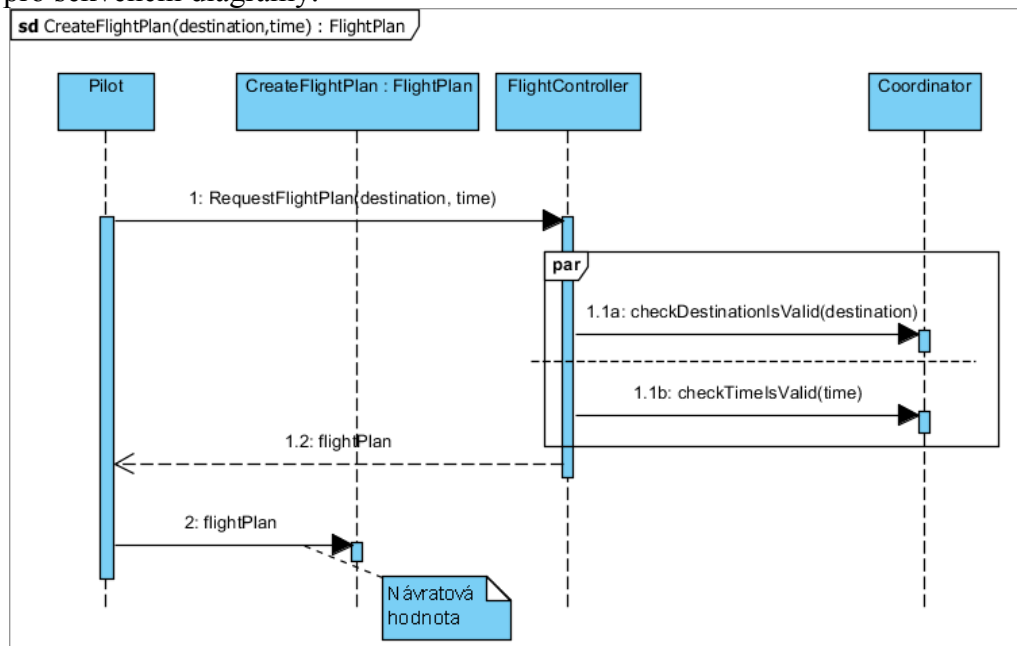
8.8.7 Vytváření a rušení objektů

Ne všechny objekty musí nutně v sekvenčním diagramu existovat od začátku. Typicky existují pouze byznys objekty nebo klíčové objekty daného sekvenčního diagramu. Voláním však libovolné objekty mohou jiné objekty vytvořit a posléze zrušit. Odpovídající syntax ukazuje následující obrázek.



8.8.8 Volání sekvenční diagramů, vstupní a výstupní parametry

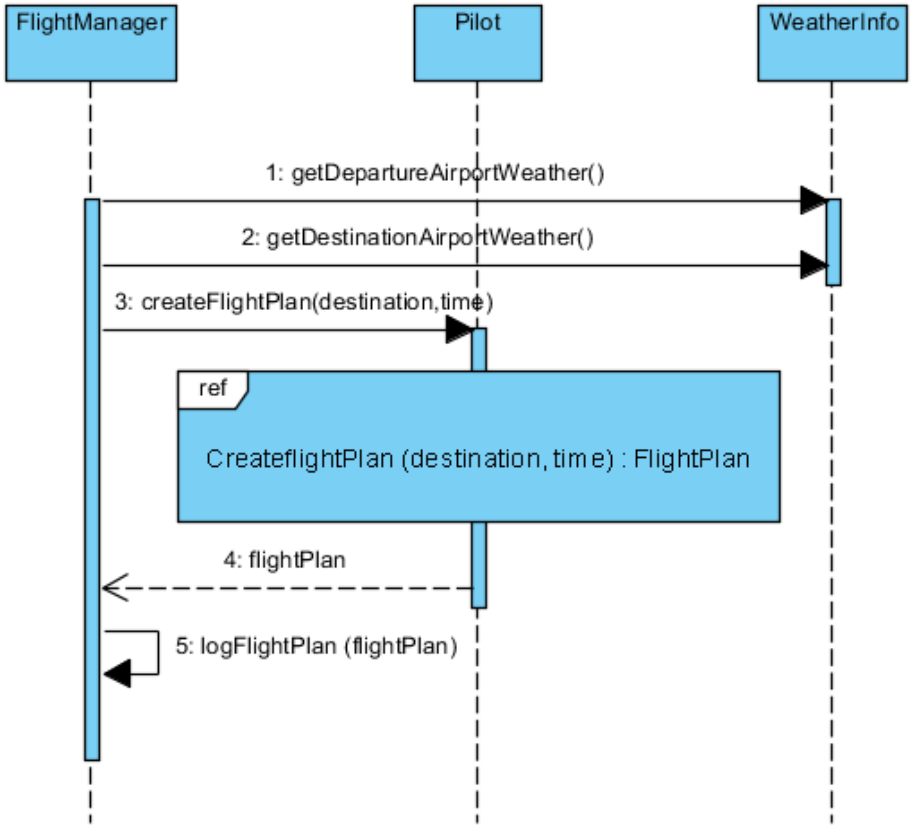
Sekvenční diagramy se již jen v lehce komplikovanějších případech mohou velmi rychle stát nepřehlednými. Proto se používají techniky, které umožňují zavolat jiný sekvenční diagram z aktuálně tvořeného. Celá technika funguje hodně podobně jako u programování a volání funkcí. Protože funkci při programování jsme schopni předat data ve formě parametrů – a stejně tak nám potom funkce může vrátit hodnotu, bude nejdřív představena stejná technika pro sekvenční diagramy.



Prvním bodem je název sekvenčního diagramu. Do závorek (podobně jako u funkcí) uvádíme seznam parametrů. Lze doplnit i datový typy, případně výchozí hodnoty, je-li třeba. Za dvojtečku uvádíme očekávaný typ, který sekvenční diagram vrací. Pokud neočekáváme návratovou hodnotu, blok opět neuvádíme.

Parametry lze zapisovat v sekvenčním diagramu jako vlastní objekty s životní čarou, vhodnější je ale přímo je používat při popiscích jednotlivých vytvářených volání (viz krok 1) – samozřejmě, **názvy parametrů uvedených v pojmenování sekvenčního diagramu musí odpovídat názvům uvedeným**

v popiscích. Jako návratová hodnota slouží opět zavedený objekt v rámci sekvenčního diagramu, který **je pojmenován stejně, jako sekvenční diagram.** V rámci sekvenčního diagramu mu potom požadovanou hodnotu ke vrácení vložíme (viz bod 2).
Nyní jsme tento sekvenční diagram schopni vyvolat a zjednodušit tak původní diagram.



K volání se opět využije kombinovaný fragment, se štítkem **ref**. Jako nápis do fragmentu se následně vloží text popisující, který jiný sekvenční diagram se má zavolat.

8.9 Diagram komponent

Diagram komponent představuje vztahy mezi jednotlivými komponentami systému. Byl dostatečně představen v předchozí studijní opoře. Blíže bude nyní představen pouze mechanismus rozhraní.

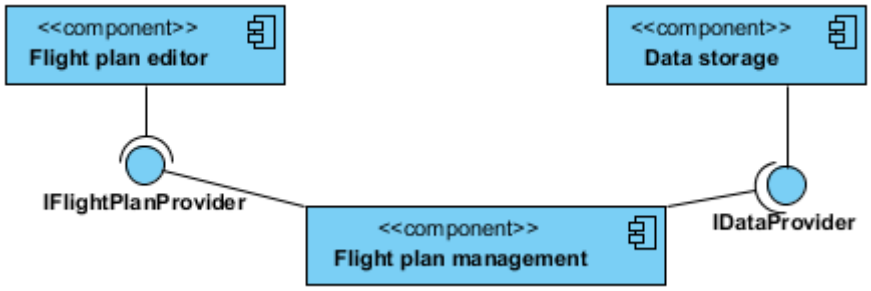
8.9.1 Rozhraní

Obecný mechanismus propojení komponent mezi sebou uvažuje, že komponenty se (ve smyslu závislostí) mezi sebou znají a vědí o sobě. Často se však dostaneme do problému cyklické závislosti, která se v programátorském prostředí při použití běžných principů řeší dost nesnadně Proto je pro doplnění vhodné zmínit mechanismus rozhraní pro demonstraci nezávislosti jednotlivých komponent na implementaci.
Technika rozhraní vychází z klasických programovacích principů, kdy potřebujeme, aby určitý blok mohl využívat nabízené funkcionality jiné komponenty, ale zároveň byl zcela odstíněn od konkrétní implementace. U rozhraní komponenta pouze říká, jaké služby (operace, funkcionality) nabízí, či jaké funkcionality vyžaduje. Komponenty v diagramu komponent tedy mohou

rozhraní jak nabízet (tj. implementovat), tak vyžadovat pro připojení. Komponenty představují rozhraní ve dvou přístupech:

- **Required** (tedy požadované) – značí se půlkruhem připojeným čarou k dané komponentě, typicky doplněný názvem rozhraní.
- **Provided** (tedy poskytované) – značí se kolečkem připojeným čarou k dané komponentě, typicky doplněný názvem rozhraní.

Pokud se dvě komponenty propojují přes rozhraní, zakreslí se kolečko dovnitř půlkruhu. Správné zakreslení zajistí korektní připojení správných typů rozhraní na sebe.



8.10 Shrnutí

(Zejména) pokročilé možnosti modelování však nemusejí nabízet všechny nástroje a v některých je třeba vystačit si s tím, co daný nástroj umí. Některé nástroje určitou problematiku mohou řešit pomocí vlastních značek či řešení. Důležité je, že **u diagramů ale není cílem precizní formalizace**, ale podání problematiky tak, aby i nově příchozí jednoznačně a pokud možno snadno pochopil, co se mu daný diagram snaží sdělit.

Kontrolní otázky:

1. Co je to stereotyp v UML a k čemu se využívá?
2. V jakém diagramu se mohou vyskytovat artefakty reprezentující třídy?
3. Čím se liší diagram komponent od diagramu tříd?

Úkoly k zamyšlení:

V kapitole byla uvedena problematika zaznačení programátorských konstrukcí do sekvenčních diagramů. Zdůvodněte, zda/kdy je to vhodné a kdy ne a ve které fázi životního cyklu projektu podle OpenUp se podobné artefakty v sekvenčních diagramech mohou začít vyskytovat?

Korespondenční úkol:

Nakreslete jednoduchý diagram případů užití, diagram tříd, sekvenční diagram a diagram komponent pro problematiku „Objednáváni pizzy online“.

Shrnutí obsahu kapitoly

V této kapitole byly přestaveny pokročilejší postupy a konstrukce při reprezentaci modelovaného informačního systému pomocí UML diagramů.

9 Literatura

[AgM] Manifest agilního vývoje. Citováno: září 2007. Dostupné na: [http://www.agilemanifesto.org/].

[Ag06] Agitar: *Why unit testing*. Citováno: leden 2008. Dostupné na: [http://www.agitar.com/solutions/why_unit_testing.html].

[Am06] Ambler, S.: *The Full Life-Cycle Object-Oriented Testing (FLOOT) Method*. Citováno: leden 2008. Dostupné na: [http://www.ambysoft.com/essays/floot.html].

[Am02] Ambler, S.: *Agile Modeling: Effective Practices for eXtreme Programming and the Unified Process*. John Wiley & Sons. 2002. ISBN#: 0471202827.

[Arl03] Arlow, J., Neustadt, I.: *UML a unifikovaný proces vývoje aplikací*. Computer press. Brno. 2003. ISBN 80-7226-947-X.

[Co05] Cockburn, A.: *Use Cases. Jak efektivně modelovat aplikace*. Computer Press. Brno. 2005. ISBN 80-251-0721-3.

[Do97] Dohnal, J., Pour, J.: *Architektury informačních systémů v průmyslových a obchodních podnicích*. Ekopress. 1997.

[Fo10] Fowler, M.: *Domain-Specific Languages*. Addison-Wesley Professional 2010. ISBN 978-0-321-7194-3.

[Ga03] Galin, D.: *Software Quality Assurance: From Theory to Implementation*. Addison-Wesley 2003. ISBN 0201709457.

[Hu07] Hunt, A., Thomas, D.: *Programátor pragmatik*. Computer Press. Brno. 2007. ISBN 978-80-251-1660-9.

[ISO] ISO 12207 Standard. Dostupné na: [http://www.12207.com].

[ISO1] ISO 9000 Standardy. Dostupné na: [http://www.iso.org].

[IRB1] Wahli, U., Brown, J., Teinonen, M., Trulsson, L.: *Software Configuration Management: Clear Case for IBM Rational ClearCase and ClearQuest UCM*. IBM Redbook. IBM Corp. 2004. Dostupné na: [http://www.redbooks.ibm.com/redbooks/pdfs/sg246399.pdf].

[Jazz] Jazz homepage. Dostupné na: [http://jazz.net].

[Kli04] Klimeš, C., Procházka, J.: *Projektování informačních systémů 1*. Učební text pro distanční studium. Ostravská univerzita. Ostrava. 2004.

[Kn04] Kniberg, H.: *Scrum and XP from the Trenches. How do we Scrum*. Crisp. 2006. Dostupné na: [www.crisp.se/henrik.kniberg/ScrumAndXpFromTheTrenches.pdf].

[Kr03a] Kroll, P., Kruchten, P.: *The Rational Unified Process. Made Easy*. Addison-Wesley. 2003. ISBN 0321166094.

[Kr03b] Kroll, P., Kruchten, P.: *The Rational Unified Process. An Introduction*. Addison-Wesley. 3. vydání. 2003. ISBN 0321197704.

[Kr05] Kroll, P., Royce, W.: *Key principles for business-driven development*. Dostupné na Rational Edge: [http://www.ibm.com/developerworks/rational/library/oct05/kroll/index.html?S_TACT=105AGX15&S_CMP=EDU].

[Kr06] Kroll, P., MacIsaac, B.: *Agility and Discipline Made Easy: Practices from OpenUP and RUP*. Addison-Wesley. 1.vydání. 2006. ISBN 0-321-32130-8.

[Luk04] Lukasík, P., Procházka, J., Vaněk, V.: *Procesní řízení*. Elektronický učební text. Přf. Ostravská univerzita. 2004.

[Mi09] Miller, R.: *Practical UML: A Hands-on Introduction for Developers*. Dostupné na [http://edn.embarcadero.com/article/31863] (20.2.2009).

[OMG] Object Management Group homepage: [http://www.omg.org].

[OUP] OpenUP homepage: [http://www.eclipse.org/epf/general/getting_started.php].

[Pro06] Procházka, J.: *Procesní řízení realizace projektů*. Elektronický učební text. Systém dalšího vzdělávání pracovníků výzkumu a vývoje. Ostravská univerzita. 2006.

[RE] Rational Edge homepage (RUP články): [http://www-128.ibm.com/developerworks/views/rational/libraryview.jsp?topic_by=rup%20(rational%20unified%20process)].

[SEI2] Clements, P. et al.: *Documenting Software Architectures: Views and Beyond* (The SEI Series in Software Engineering). Addison-Wesley. 2002. ISBN 0201703726.

[SEI3] Barbacci, M., R. et al.: *Quality Attribute Workshop (QAWs)*. 3rd edition. *Technical Report* CMU/SEI-2003-TR-016. Dostupné na: [http://www.sei.cmu.edu/pub/documents/03.reports/pdf/03tr016.pdf].

[SEI4] SEI CMMI homepage: [www.sei.cmu.edu/cmmi/].

[Sta02] Výzkum Standish Group. Dostupné na: [http://ciclamino.dibe.unige.it/xp2002/talksinfo/johnson.pdf].

[Sut07] Sutherland, J.: *Scrum and CMMI level 5. The Magic Potion for Code Wariors*. In Proceedings of the AGILE 2007, pp. 272-278. Dostupné na: [jeffsutherland.com/scrum/Sutherland-ScrumCMMI6pages.pdf]

[Von02]Vondrák, I.: *Úvod do softwarového inženýrství*. Elektronický učební text. Verze 1.1. VŠB-TU. Ostrava. 2002.

Vysvětlivky symbolů

	**Průvodce studiem** – vstup autora do textu, specifický způsob, kterým se studentem komunikuje, povzbuzuje jej, doplňuje text o další informace
	**Příklad** – objasnění nebo konkretizování problematiky na příkladu ze života, z praxe, ze společenské reality, apod.
	**Pojmy k zapamatování**.
	**Shrnutí** – shrnutí předcházející látky, shrnutí kapitoly.
	**Literatura** – použitá ve studijním materiálu, pro doplnění a rozšíření poznatků.
	**Kontrolní otázky a úkoly** – prověřují, do jaké míry studující text a problematiku pochopil, zapamatoval si podstatné a důležité informace a zda je dokáže aplikovat při řešení problémů.
	**Úkoly k textu** – je potřeba je splnit neprodleně, neboť pomáhají dobrému zvládnutí následující látky.
	**Korespondenční úkoly** – při jejich plnění postupuje studující podle pokynů s notnou dávkou vlastní iniciativy. Úkoly se průběžně evidují a hodnotí v průběhu celého modulu.
	**Úkoly k zamyšlení**.
	**Část pro zájemce** – přináší látku a úkoly rozšiřující úroveň základního modulu. Pasáže a úkoly jsou dobrovolné.
	**Testy a otázky** – ke kterým řešení, odpovědi a výsledky studující najdou v rámci studijní opory.
	**Řešení a odpovědi** – vážou se na konkrétní úkoly, zadání a testy.