VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

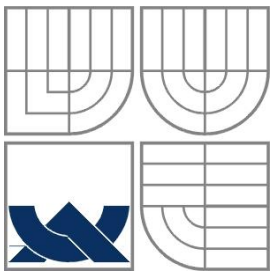
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

RATIONAL UNIFIED PROCESS JAKO METODIKA VÝVOJE SOFTWARU

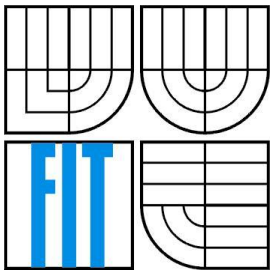
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. VLADIMÍR RYTÍŘ
AUTHOR

BRNO 2007



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

RATIONAL UNIFIED PROCESS JAKO METODIKA VÝVOJE SOFTWARU

RATIONAL UNIFIED PROCESS AS METHODOLOGY OF SOFTWARE DEVELOPMENT

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. VLADIMÍR RYTÍŘ

VEDOUCÍ PRÁCE
SUPERVISOR

RNDr. JITKA KRESLÍKOVÁ, CSc.

BRNO 2007

Abstrakt

Cílem této práce je seznámit se s procesem vývoje softwaru dle vybraných metodik s hlavním zaměřením na metodiku Rational Unified Process od firmy IBM.

Fázi zahájení a fázi rozpracování této metody také aplikuji na praktickém příkladě.

Klíčová slova

Informační systém, Unified Process, Rational Unified Process, fáze zahájení, fáze rozpracování, životní cyklus, iterativní vývoj, Systém pro podporu řízení vztahu se zákazníky, agilní vývoj softwaru, Extrémní programování, Scrum.

Abstract

Goal of my work is to introduce software development process methods specialized to Rational Unified Process method from IBM.

I apply inception and elaboration phases of RUP on practical example.

Keywords

Information system, Unified Process, Rational Unified Process, Inception phase, Elaboration phase, lifecycle, RUP, iterative development, Customer Relationship Management, Agile software development, Extreme Programming, Scrum.

Citace

Rytíř Vladimír: Rational Unified Process jako metodika vývoje softwaru, Brno, 2008, diplomová práce, FIT VUT v Brně.

Rational Unified Process jako metodika vývoje softwaru

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením RNDr. Jitky Kreslíkové, CSc.
Další informace mi poskytl konzultant, pan Vítek Urban, z firmy Unicorn a.s.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Vladimír Rytíř
5. 5. 2008

Poděkování

Děkuji panu Vítku Urbanovi za poskytnuté konzultace.

Obsah

Úvod

V dnešní technické době jsou na vývoj informačních systémů kladeny stále větší nároky. Vysoké požadavky mají zákazníci na rychlost vývoje systému, výše jeho kvalit, množství funkcí a nízkých pořizovacích a provozních nákladů.

Vývoj rozsáhlejšího systému není jednoduchá záležitost. Obsahuje množství rizik, na kterých se může systém prodražit, nebo úplně zastavit. Proto je důležité dodržovat stanovené praktiky a metody, které vývojový tým provedou celým procesem vývoje od sepsání počátečních požadavků zákazníka až po nasazení systému do praxe. Jedná se o soubor dohodnutých principů, pravidel a postupů, které ovlivňují efektivnost a organizaci práce. Výsledkem je úspěšné vyvinutí systému.

Vlastní tvorba a sjednocení metod pro vývoj systémů prošla v historii spoustou změn. Dnes je nejrozsáhlejší metoda od firmy IBM nazvaná Rational Unified Process (RUP) a volně dostupný standard Unified Process. Objevují se ale již i nové inovativní metodiky vývoje, tzv. agilní metodiky.

Zadání pro tuto práci vydala firma Unicorn a.s., která je největším českým softwarehouse poskytující komplexní služby a řešení v oblasti vývoje a implementace podnikových aplikací a informačních systémů.

V první kapitole se seznámíme s životním cyklem vyvíjených informačních systémů a uvedeme si několik dřívějších metodik vývoje, které jsou z historického hlediska nejdůležitější.

Další dvě kapitoly se věnují samotné metodice Rational Unified Process od firmy IBM. Nejdříve si řekneme něco o praktikách, kterými se tato metodika řídí a popíšeme si základní stavební prvky metodiky a pohledy na architekturu. Poté budeme analyzovat vývoj softwaru podle této metodiky a stručně si projdeme jednotlivé fáze vývoje a základní disciplíny.

Ve čtvrté kapitole se seznámíme s obecným procesem vývoje software a porovnáme obecnou metodiku s metodikou RUP a srovnáme hlavní výhody a nevýhody obou metodik.

Pátá kapitola popíše další dnes používané metodiky vývoje informačních systémů a jejich hlavní přednosti a nedostatky opět porovná s metodikou RUP. Tato kapitola je také poslední teoretickou přípravou před praktickým použitím metodiky Rational Unified Process.

Poslední kapitola nás provede prvními dvěma fázemi metodiky RUP na praktickém příkladě. V této kapitole se tak podrobněji setkáme s několika důležitými činnostmi a artefakty samotného vývoje informačního systému. Vyvíjený systém bude webový systém pro správu zákazníků, který tvoříme výhradně pro tuto diplomovou práci a pro lepší znázornění a pochopení metodiky samotné. První čtyři kapitoly byly vytvořeny v rámci semestrálního projektu, u kterého jsem získal potřebné teoretické znalosti pro tvorbu praktického příkladu. Pro tuto práci jsem ale některé kapitoly upravil a rozšířil o další sekce.

Protože se překlad pojmů může v různých literaturách lišit, uvádím u některých nadpisů i originální anglické názvy.

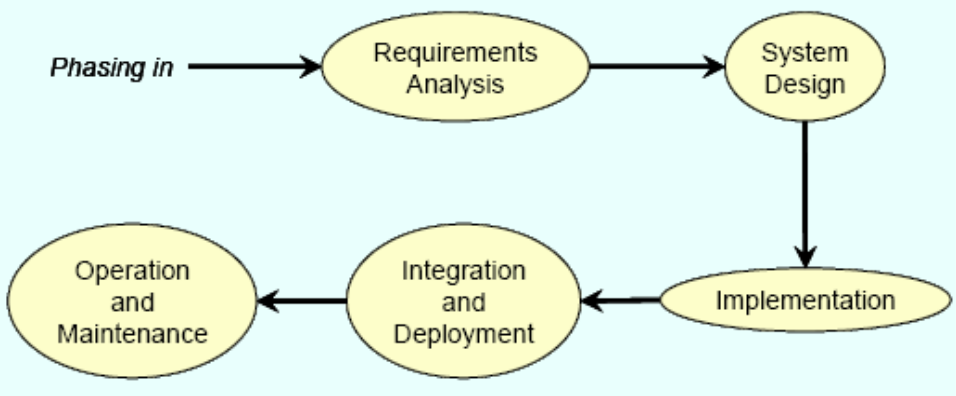
1 Životní cyklus software

Životním cyklem softwaru rozumíme změny, kterými software prochází v jeho „životě“. Tyto změny se značí jako různé fáze životního cyklu mezi „narozením“ a „úmrtím“ softwarového produktu.

Uvažujme čtyři základní fáze vývoje systému a jednu fázi činnosti systému po nasazení do provozu:

- analýza požadavků - analyzujeme požadavky na systém,
- návrh systému - navrhujeme samotný systém dle požadavků. Tato fáze také obsahuje popis struktury softwaru, data a rozhraní komponent, které mají být v další fázi implementovány.
- implementace – programování a testování navrhovaných komponent,
- integrace – nasazení systému na pracoviště, rozmístění jeho jednotlivých částí, vstupní testy a zaučení uživatelů,
- činnost a údržba – samotný provoz a údržba systému v provozu. V této části se systém stává více přizpůsobitelným a dolaďují se nedostatky a chyby, které nebyly objeveny při vývoji softwaru. Tato část již do samotného životního cyklu vývoje softwaru nepatří.

Návaznost jednotlivých fází můžeme vidět na obrázku 1.



Obrázek 1: Životní cyklus vývoje softwaru [1]

Životní cyklus prošel mnoha fázemi vývoje a vzniklo tak několik modelů pro vývoj softwaru. V dalších podkapitolách některé metody blíže popíšu.

1.1 Vodopádový životní cyklus a jeho modifikace

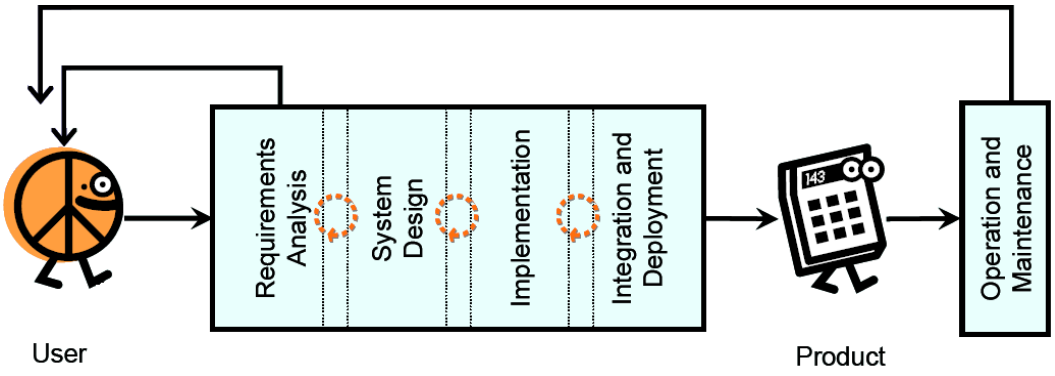
Vodopádovým životním cyklem rozumíme sekvenční návaznost jednotlivých fází vývoje tak, jak jsem je uvedl v kapitole 1. Jedná se o historicky nejstarší model, kde se na další fázi přešlo až po skončení fáze předchozí. Tento cyklus už z letmého pohledu vykazuje několik hlavních nedostatků.

Před začátkem analýzy je nutné získat od zadavatele veškeré požadavky a materiály, které se sepíší do formy specifikace požadavků na systém a z toho se pak vychází. Zadavatel je dále z celého procesu vývoje softwaru vynechán a výsledek uvidí až po skončení vývoje. Protože vývoj

rozsáhlejšího softwaru může trvat i několik měsíců a požadavky se stále vyvíjí, tento model na ně dostatečně nereaguje a změny jsou pak velmi drahé a složité.

Hlavním nedostatkem je tedy zpětná vazba hotového softwaru na požadavky zákazníka. Z toho důvodu byl základní vodopádový model o tuto zpětnou vazbu rozšířen (viz obrázek 2). Vodopádový model se zpětnou vazbou ale již neříká jak pokračovat, pokud zpětná vazba zjistí, že výsledný software neodpovídá požadavkům zadavatele.

V další modifikaci vodopádového modelu oproti předchozímu bylo přidáno překrývání jednotlivých fází vývoje (viz obrázek 2). Díky tomuto přístupu je možné odhalit např. navrhované nedostatky v rané fázi implementace a tyto nedostatky v návrhu odstranit.



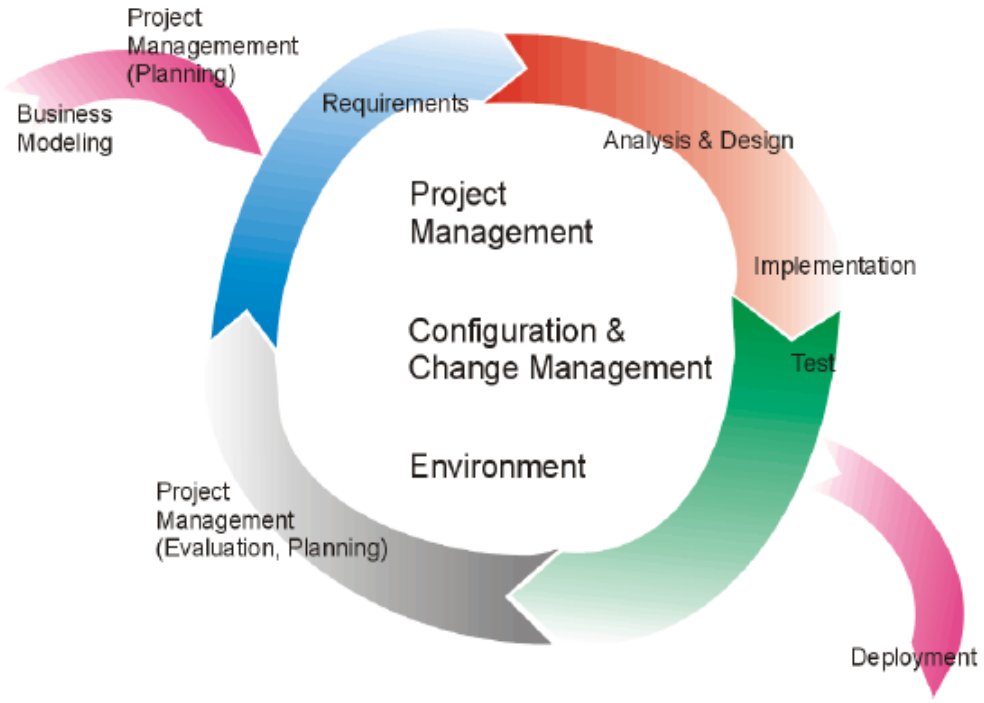
Obrázek 2: Vodopádový model se zpětnou vazbou a přesahy [1]

1.2 Iterativní životní cyklus s přírůstky

V praxi se ukázalo, že menší systémy se navrhují mnohem jednodušeji, než systémy většího rozsahu. Proto je dobré u rozsáhlých systémů nevyvíjet celé softwary najednou, ale rozbít software na menší subsystémy a postupně k základu přidávat další funkce.

Každá vyvíjená část projde celým vodopádovým modelem od sepsání požadavků až po vydání produktu (viz obrázek 3). Tyto jednotlivé části se nazývají iterace a jejich výsledné produkty se nazývají konstrukce (buildy).

Před prvním vývojem sytému je nutné určit, jaké funkce se budou v jakých iteracích navrhovat a implementovat. Každá další iterace staví na iteracích předchozích. Doplňuje se plánovaná funkčnost a opravují se nedostatky předchozího buildu. Build je plně funkční produkt obsahující i vlastní dokumentaci, který po poslední iteraci odpovídá výslednému požadovanému softwaru. Nedostatkem tohoto modelu je nutnost naplánování iterací předem a tím tedy i určení počtu iterací, po kterých se dosáhne výsledného softwaru.



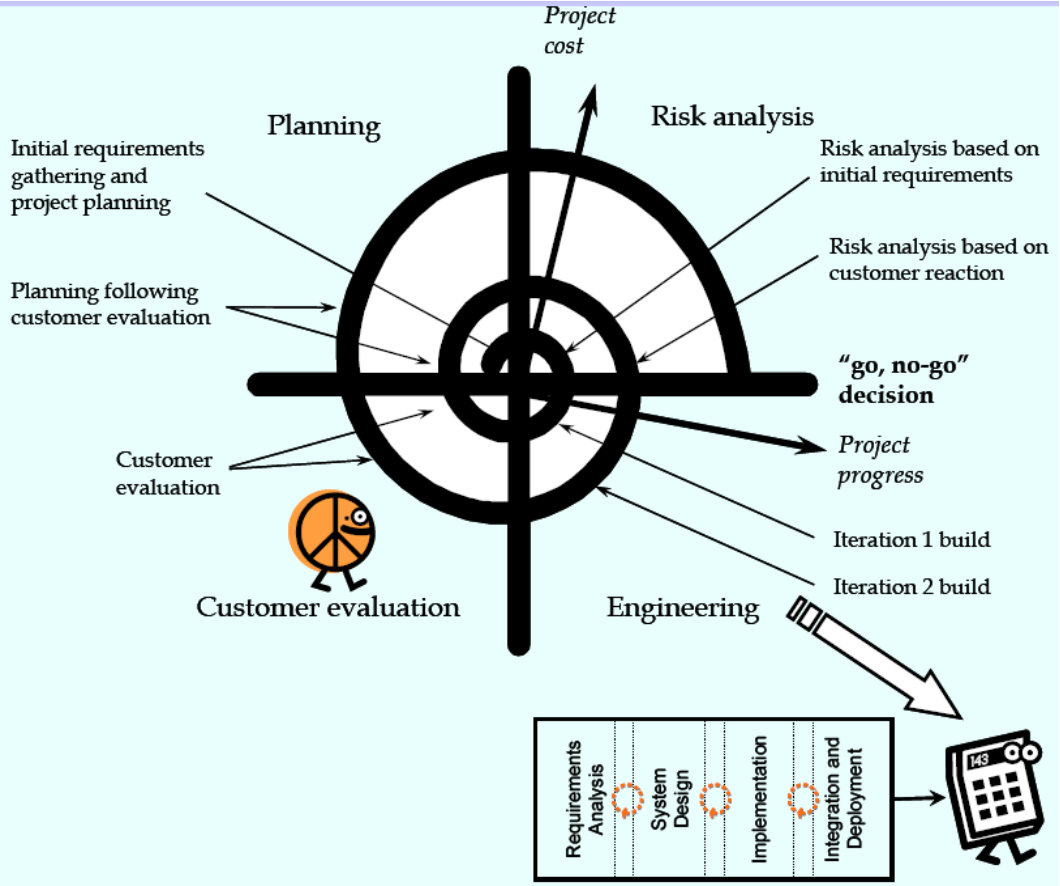
Obrázek 3: Iterativní a inkrementální model [2]

1.3 Spirálový model

Spirálový model je ve skutečnosti rámcem nebo metamodelem, který může obsahovat jiné modely životního cyklu. Model se prezentuje v podobě spirály, která prochází čtyřmi kvadranty – plánování, analýza rizik, inženýrství a hodnocení zákazníkem [1].

První fází je plánování, kde se plánují požadavky na systém prováděné v dané iteraci. Ve druhé fázi se analyzují rizika požadavků a stanoví se cena projektu. Zde se zadavatel rozhoduje, zda se má v projektu pokračovat nebo zda jsou rizika moc vysoká. Pokud ano, přejde se do třetí fáze, která odpovídá vodopádovému modelu s přesahy a výsledek se předá zadavateli projektu. Ten ve čtvrté fázi projekt ohodnotí, doplní požadavky a předá je dál do první fáze. Takto projekt pokračuje stále dokola, až do doby, kdy se zadavatel projektu rozhodne, že další uskutečnění požadavků je již nezbytné nebo by přesáhlo cenu a životní cyklus skončí.

Obrázek 4 přehledně zobrazuje celý životní cyklus projektu, který se zhotoví ve dvou fázích a před vývojem třetí se ukončí.



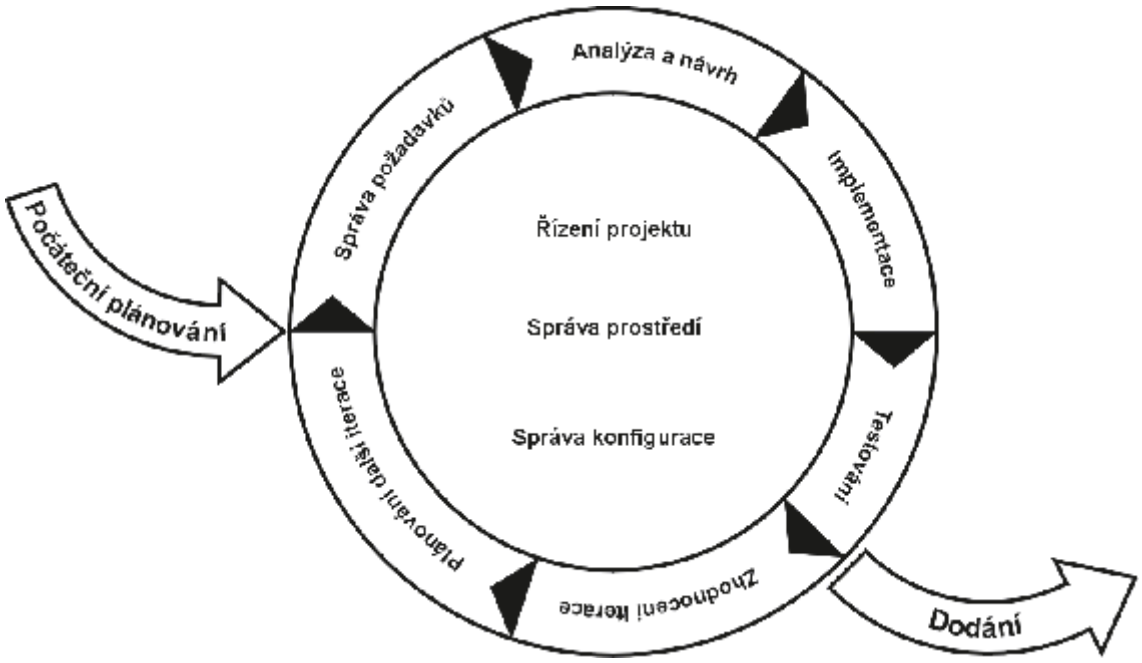
Obrázek 4: Spirálový model [1]

Výhodou tohoto modelu je analýza rizik, která v předchozích modelech chyběla a díky které nemusíte předem přesně stanovit počet iterací.

2 IBM Rational Unified Process

Tato metodika je komerční produkt firmy Rational, kterou později koupila firma IBM. Autoři produktu jsou zároveň autoři jazyka UML, kteří se snažili vytvořit jednotnou metodiku pro vývoj softwaru. Protože se jedná o komerční produkt, nebavíme se zde pouze o popisu jakési metody, ale již přímo o platformě, která obsahuje různé šablony dokumentů apod.

Proces se skládá z klasických fází a navíc je přidána fáze testování (viz obrázek 5).



Obrázek 5: Rational Unified Process [6]

Jak jsem popsal v kapitole 1.2. Iterativní životní cyklus s přírůstky, jedno opakování těchto fází se nazývá iterace a každá iterace tvoří základní linii. Ty jsou na sebe vrstveny tak dlouho, dokud není dosažena konečná podoba vytvářeného systému. Rozdíl mezi těmito liniemi je označován za přírůstek. Životní cyklus vývoje softwaru podle metodiky RUP je tedy iterativní a přírůstkový. Iterace ale nemusí nutně obsahovat všechny fáze vývoje softwaru. Obsah každé iterace je závislý na tom, v jakém stadiu životního cyklu (v jaké fázi) se daný projekt nachází.

Iterativní vývoj softwaru patří mezi šest nejlepších praktik vývoje informačních systémů.

2.1 Nejlepší praktiky vývoje

Metodika RUP vznikla ze znalostí jiných předchozích metod používaných v praxi. Snažila se spojit dobré vlastnosti a vyvarovat se těch špatných. Protože se ale problémům nikdy nevyhneme, stanovili tvůrci RUP šest základních praktik vývoje systému (viz obrázek 6), které nám pomohou problémy eliminovat na minimum. Jednotlivé praktiky si nyní projdeme.

Obrázek 6: Šest nejlepších praktik [6]

2.1.1 Iterativní vývoj

V úvodu kapitoly 2 jsem již uvedl, jak takový iterativní vývoj vypadá. Hlavní výhody nejlépe shrnul Aldorf F. v [5] :

- Možnost objektivního posouzení stavu projektu.
- Rovnoměrnější pracovní vytížení vývojového týmu. Přímo souvisí s předchozím bodem. Projekt je koncipován tak, aby co nejdříve přinášel konkrétní výsledky. Při rozdělení na iterace je možné snáze sledovat průběh projektu a dodržování stanovených termínů pro jednotlivé iterace.
- Možnost testování mezi-verzí.
- Spolupráce s uživateli v průběhu celého projektu.
- Včasné rozpoznání nesrovnalostí mezi požadavky, návrhem a implementací. Tato výhoda přímo vyplývá z předchozích dvou bodů. Díky tomu, že uživatelé mají možnost kontrolovat a hodnotit dílčí části systému, značně se omezuje riziko vysokých nákladů způsobených úpravami produktu v pozdní fázi vývoje.
- Snazší zapracování změn požadavků.

2.1.2 Průběžné ověřování kvality

Ověřování kvality se neprovádí pouze na konci každé iterace, ale během celého procesu vývoje, protože jakékoliv dodatečné opravy mohou dokončení projektu prodloužit a prodražit. Testy se proto také týkají všech účastníků vývojového týmu. Během vývoje stále přibývají nové funkce a narůstají

nároky na testy, proto vývojáři používají testy automatické (skripty), které jim pomáhají se s kvantem testů vypořádat.

Zajištění kvality tedy není považováno za něco, co stojí mimo hlavní linii vývoje produktu a není to záležitost zvláštní aktivity realizované speciální skupinou.

2.1.3 Vizuální modelování

Jedná se o proces, který ukazuje, jak namodelovat strukturu a chování architektur a komponent. To nám také umožňuje skrýt detaily a zdrojový kód, protože jednotlivé elementy systému modelujeme graficky. Vizuální abstrakce hlavně pomáhá vyjadřovat různé pohledy na systém. Ukazuje, jak jednotlivé části systému zapadají do celku, pomáhá udržovat konzistenci mezi implementací a návrhem, stanoví jednoznačná pravidla pro komunikaci atd.

Standardem pro vizuální modelování se stal Unified Modeling Language (UML), vytvořen zaměstnanci firmy Rational Software.

2.1.4 Řízení změn

Schopnost řídit změny udává určitý druh jistoty, že každá změna je akceptována. Mít možnost tyto změny vysledovat a vrátit zpět je základ pro tvorbu softwaru, ve kterém se změnám nevyhneme. Proces popisuje jakým způsobem kontrolovat a monitorovat změny při zachování úspěšného iterativního procesu. Dále hlídá a zaznamenává veškeré změny všech pracovníků a to nejen v kódu, ale i v dokumentech a artefaktech používaných při vývoji softwaru.

2.1.5 Komponentová architektura

Komponentu můžeme definovat jako netriviální softwarový balíček, modul nebo subsystém. Pokud řešený problém rozložíme na jednotlivé komponenty a pevně stanovíme jejich rozhraní, může každou komponentu vyvíjet jiný tým, jiná firma nebo se může komponenta zakoupit.

Komponentová architektura také velmi přispívá k identifikaci rizik, jejich předcházení a odstranění již ve fázi návrhu.

RUP podporuje standardní komponentové architektury COM (Component Object Model) a CORBA (Common Object Request Broker Architecture). COM je programovací technika, která určuje základní vlastnosti objektů a pravidla pro práci s nimi. Prostředí CORBA tvoří architekturu pro podporu tvorby distribuovaných objektově orientovaných aplikací. Zde nezáleží, v jakém jazyce je objekt implementován a na jakém stroji, a pomocí jakého komunikačního protokolu, je objekt zpřístupněn.

2.1.6 Správa požadavků

U vodopádového modelu jsem poznamenal, že zadavatel musí předem znát všechny požadavky na systém. V praxi toto ale není možné, protože požadavky se mohou nejen za dobu vývoje projektu změnit, ale mohou se objevit nové a naopak zase jiné se stanou nepotřebnými.

Proto je důležité zpracovávat a dokumentovat veškeré požadavky zadavatele v průběhu celého vývoje softwaru a aktivně na ně reagovat.

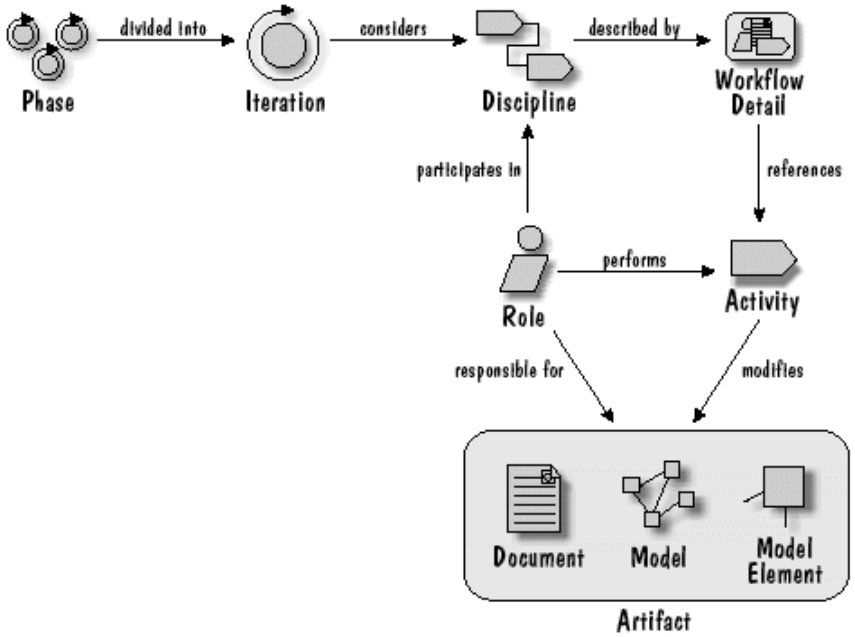
2.2 Sedm nedopatření RUP

Vyvinout software není jednoduchá záležitost. I když budeme dodržovat postupy a principy RUP a pomocí něho plánovat a provádět projekty, stále se můžeme chytit do pastí, které mohou způsobit nedokončení projektu. Zde uvádím sedm základních hříchů nebo pastí, které se mohou během vývoje projektu objevit. Jedná se o rizika, kterým bychom měli předejít:

- Neustálé plánování
- Velmi podrobný popis
- Přeskakování problému analýzy
- Nedodržení konce termínů iterací
- Započetí konstrukční fáze před dosažením milníku fáze projektování
- Testování pouze na konci projektu
- Selhání přechodu produktu do fáze údržby

2.3 Základní elementy RUP

V dokumentaci u každé disciplíny najde jedna osoba nebo vývojový tým několik rolí („kdo“), kterými se mohou stát v průběhu projektu. Každá role obsahuje několik aktivit („jak“), které jsou potřeba provést a každá aktivita má detailní popis provádění krok za krokem. Posloupnost souvisejících aktivit je označována jako pracovní postup, tok pracovních činností nebo dokumentů („kdy“). Prováděním aktivit roste počet artefaktů („co“), které aktivity produkují. Jako artefakty označujeme hmatatelné mezivýsledky projektu - dokumenty, modely, části kódů apod. Osoba, která je v pozici dané role za tyto artefakty odpovídá. Viz obrázek 7.



Obrázek 7: Základní elementy a jejich vztahy [4]

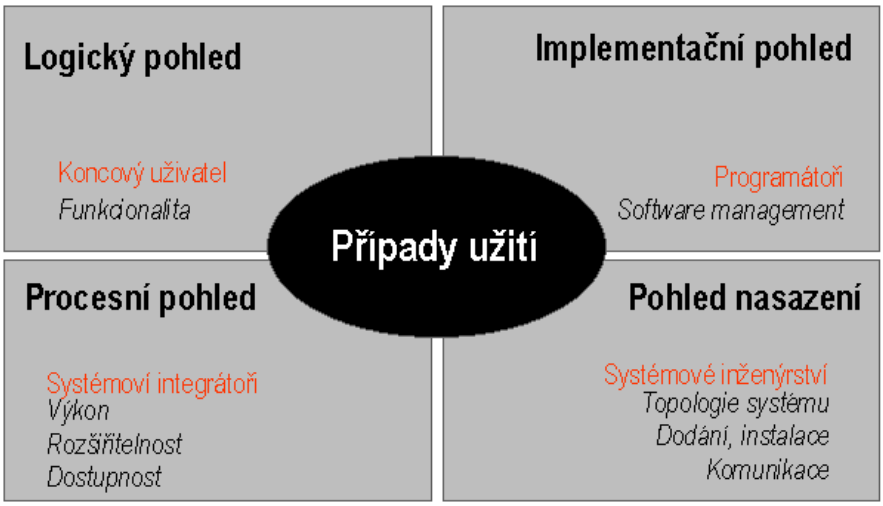
Podrobnějšímu vysvětlení základních elementů se věnuje celá kapitola 3.

2.4 Architektura systému

Příručka *The Unified Modeling Language Reference* definuje architekturu systému jako „organizační strukturu systému, včetně jeho rozkladu na součásti, jeho propojitelnosti, interakce, mechanismů a směrných základ, která proniká do návrhu systému“ [3].

Modelovací jazyk UML definuje několik základních pohledů na systém. Popis veškerých pohledů a modelů ale není cílem této práce.

V RUP je použita architektura 4+1 (viz obr. 8), která definuje pět základních pohledů na systém. Tuto architekturu není možné vytvořit během několika málo kroků, ale bude vznikat postupně v průběhu vývoje systému a zachytí dostatečný počet informací, aby bylo možné systém vytvořit.



Obrázek 8: Architektura 4+1 [5]

Jednotlivé pohledy řeší různé aspekty fungování systému, nejsou však na sobě závislé a do určité míry se překrývají [5].

- Logický pohled *(logical view)* – objektový model návrhu pohlíží na funkce sytému.
- Procesní pohled *(process view)* – pohled na souběžnost a synchronizaci jednotlivých aspektů návrhu.
- Implementační pohled *(development view)* – zachycuje statickou organizaci softwaru v prostředí vývoje.
- Pohled nasazení *(physical view)* – zachycuje mapování softwarových částí na hardwarové stroje.
- Pohled na případy užití *(use-case view)* – tyto modely zachycují především tzv. scénář systému

2.4.1 Logický pohled

Logický pohled je abstraktní model návrhu, který znázorňuje logickou strukturu systému z hlediska požadované funkcionality. Popisuje systém rozložený na množinu abstraktních částí, jako jsou objekty a třídy. Ty vysvětlují principy abstrakce, dědičnosti a zapouzdření částí systému. Je vhodné jednotlivé části zapouzdřovat do balíčků, které pak mají pevně definované rozhraní pro propojení s dalšími částmi systému.

Pro návrh systému z logického pohledu se používá objektově orientovaný styl, protože tak můžeme pomocí samostatných a spojených objektových modelů udržet jednoduchost v celém

systému a vyhneme se tak návrhu specializovaných tříd a mechanizmů pro jednotlivé procesory. Důraz je kladen na zobrazení způsobu, jakým objekty a třídy implementují chování systému.

Hlavními diagramy pro znázornění logického modelu jsou:

- diagramy tříd,
- stavové diagramy,
- objektové diagramy.

2.4.2 Procesní pohled

Procesní pohled zachycuje nefunkční požadavky systému, jako je výkon a dostupnost. Řeší konkurenci a paralelismus procesů, integritu systému a také jak logický pohled zapadá do architektury systému (na jakém vláknu je řízen jaký objekt apod.).

Architektura procesů může být popsána na několika úrovních, kde se každá úroveň týká jiné záležitosti (např. na nejvyšší úrovni bude znázorněno pouze propojení sítí).

Procesní pohled znázorňují procesní diagramy přidané do specifikace UML již v době vzniku tohoto jazyka:

- externí model, vytvořený pomocí diagramu use-case a
- interní model vytvořený pomocí diagramu tříd.

2.4.3 Implementační pohled

Implementační pohled zobrazuje strukturu fyzických souborů a jejich vzájemné propojení. Soubory mohou být zdrojové kódy, knihovny, spustitelné soubory ale i datové soubory apod. Ty jsou rozděleny do částí sytému, tzv. modulů a jsou vzájemně propojeny pomocí jejich rozhraní. Většinou jsou jednotlivé moduly rozděleny do vrstev, které mají hierarchickou strukturu.

Tento implementační pohled znázorňují:

- diagramy komponent.

2.4.4 Pohled nasazení

Pohled nasazení se zaměřuje spíše na nefunkční požadavky systému, jako jsou dostupnost, spolehlivost, výkonnost a rozšiřitelnost. Software může být rozložen na síti několika počítačů nebo výpočetních uzlů (periferní zařízení, síťové prvky, apod.). Různé části systému mohou být mapovány na různé uzly, které mohou být navíc nakonfigurovány pro různé použití (testování, vývoj) nebo určené pro různé uživatele. Mapování systémových částí musí být velmi flexibilní a musí mít minimální dopad na samotný zdrojový kód.

Pohled nasazení přehledně znázorňují:

- diagramy nasazení.

2.4.5 Případy užití

Všechny čtyři předchozí pohledy spojuje množina případů použití, která zachycuje nejdůležitější požadavky funkčnosti kladené na vyvíjený systém. Jedná se o základní pohled z hlediska koncového uživatele, podle kterého se ostatní pohledy modelují.

Případy užití modelují:

- diagramy případů užití modelující statické struktury a
- diagramy interakce.

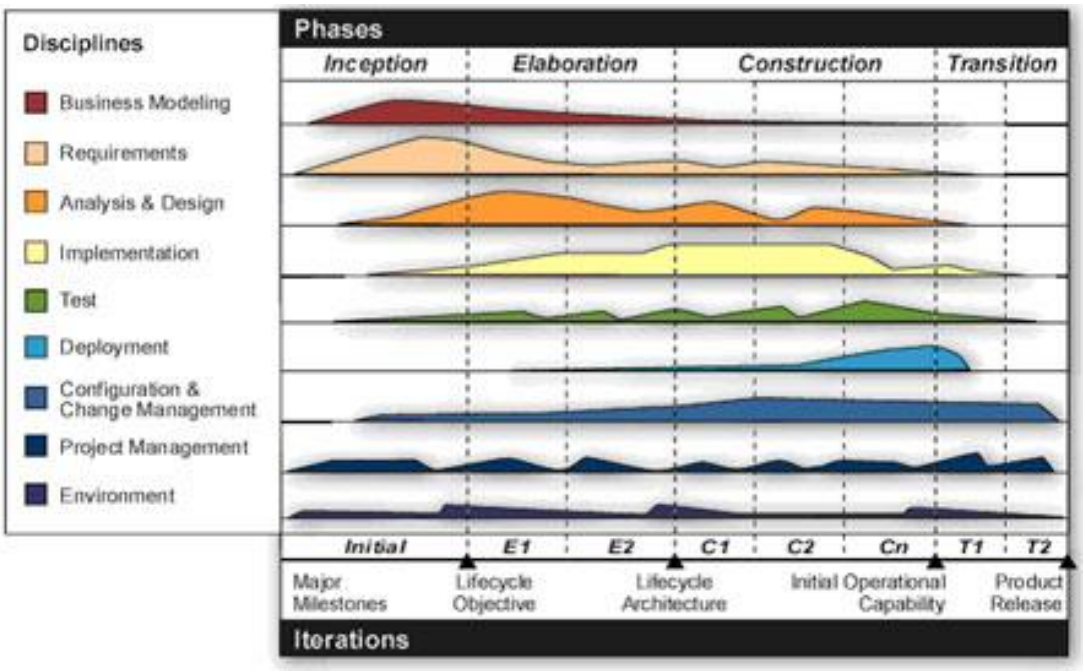
3 Analýza procesu vývoje software podle RUP

3.1 Fáze a milníky

Životní cyklus projektu je rozdělen na čtyři fáze - zahájení, rozpracování, konstrukce a zavedení. Každá fáze obsahuje několik iterací a končí definovanými milníky projektu.

Mezi velikostí daného projektu a počtem iterací v jednotlivých fázích platí přímá úměra. Správným řešením ale je, že jedna iterace by neměla přesáhnout dobu trvání delší než tři měsíce.

Milníky projektu určují, jakému stádiu se software přibližuje a jakými stádii již vyvíjený software prošel. Hlavní milníky odpovídají hlavním fázím projektu.



Obrázek 9: Čtyři fáze metodiky RUP s disciplínami, iteracemi a vyznačenými milníky [5]

Na obrázku 9 je zobrazen dvojdimenzionální pohled na vývoj systému, kde je počet iterací do jednotlivých fází určen pro středně velký projekt, který budu v rámci diplomové práce zpracovávat. Disciplíny (na obrázku 6 vlevo) jsou statické aspekty podílející se na vývoji softwaru. Disciplíny budou podrobněji probrány v kapitole 3.2 Disciplíny.

Další podkapitoly nás seznámí s každou fází vývoje zvlášť. Kromě popisu u každé fáze uvádím tři základní body. Dokumenty, které fáze potřebuje pro vlastní proces (vstupy, většinou se jedná o výstupy předchozích fází), dokumenty, které jsou výsledkem dané fáze (výstupy) a milník fáze, který musí být dosažen.

3.1.1 Fáze zahájení *(inception phase)*

Cílem této fáze je shrnout požadavky zadavatele a sestavit vizi, tedy požadavky na systém, které zadavatel odsouhlasí. Podle vize se určí rozsah projektu. Sestavuje se zde také jednoduchý plán

projektu, který obsahuje důležité milníky, a který teprve bude v dalších fázích rozšířen a upřesněn. Dále se vytváří dokument odhadující rizika projektu a musí být sepsány veškeré případy použití systému, z nichž se v této fázi namodelují pouze ty nejdůležitější.

Před touto fází není možné stanovit cenu a dobu trvání vývoje softwaru. V praxi to většinou vypadá tak, že si zadavatel nechá vypracovat veškeré dokumenty, které tato fáze obsahuje a podle nich se teprve rozhodne, zda chce v projektu pokračovat, změnit požadavky nebo projekt nerealizovat.

Vstupy fáze zahájení:

- Požadavky na sytém
- Případný stávající systém

Výstupy fáze zahájení:

- Vize systému a rozsah projektu
- Případy použití
- Plán projektu s nadefinovanými milníky
- Obchodní případy a přibližná cena projektu
- Glosář (slovník)
- Odhad rizik
- Počáteční sestavení architektury

Milníkem fáze zahájení je předmět životního cyklu a rozsah systému. Pro dosažení milníku je nutné dodat všechny dokumenty uvedené ve výstupu a tím prokázat, že je projekt realizovatelný v dané kvalitě, rozsahu, termínu a stanovené ceně.

3.1.2 Fáze rozpracování *(elaboration phase)*

Hlavním cílem této fáze je vytvořit spustitelný software, který ukáže, že navržený architektonický základ je stabilní a plně funkční. Nejedná se o pouhý prototyp, který bude v budoucnu zahozen, ale o základní stavební prvek vyvíjeného softwaru, který bude rozšiřován a upravován až do finální podoby na konci fáze zavedení. Pro vytvoření architektury je důležité znát celý rozsah systému, jeho funkční a nefunkční požadavky a požadavky na výkon.

Kromě navržení, implementování a testování architektury také fáze obsahuje eliminaci nejvyšších rizik, vytvoření většiny případů použití, rozšíření a upřesnění požadavků, rizik a milníků a tvorbu plánu konstrukční fáze. Musí být přesně stanovena míra rozsahu a hloubky sytému.

Tato fáze je považována za nejkritičtější část vyvíjeného systému. I při dalších změnách a nových požadavcích fáze projektování zajišťuje velmi stabilní a neměnnou architekturu. Na konci této fáze je možnost již velmi přesně odhadnout cenu a časový plán dokončení vývoje projektu.

Vstupy:

- Výstupy fáze zahájení

Výstupy:

- Popis a spustitelný základ architektury
- Značná část modelů případů použití
- Model návrhu
- Datový model
- Veškeré požadavky na systém

- Upravené obchodní případy
- Upravený odhad rizik
- Upravený a již přesně definovaný plán projektu

Druhý milník projektu označuje za hotovou nejen architekturu systému, ale také odstranění hlavních rizik a detailního doplánování celého rozsahu projektu. Rizika mohou být odstraněna použitím alternativních postupů nebo vyřešením technologických problémů.

3.1.3 Fáze konstrukce *(construction phase)*

V této fázi vývojové týmy dokončí zbylé modely analýzy a návrhu a dílčí části produktu se implementují a integrují do systému. Vzhledem k popsání veškerých modelů, struktur a komponent je možné přiřadit fázi konstrukce několika vývojovým týmům a jednoduše je poté dosadit dohromady. Jako základní stavební kámen vývojáři použijí systém naimplementovaný ve fázi rozpracování.

Důležité je zachování integrity architektury systému. V praxi se často stává, že termín tlačí vývojáře a ti si práci urychlují narušením původní vize. Tím snižují výslednou kvalitu systému a dodatečné úpravy a změny mohou vývoj software prodražit.

Další nedílnou součástí této fáze je integrační testování a vytváření automatických testů.

Vstupy:

- Výstupy fáze rozpracování

Výstupy:

- Výsledné systémy pro dané platformy
- Uživatelský manuál
- Sada testů
- Popis dané verze software

Milníkem této fáze je nainstalování softwaru do praxe a provedení testů. Zde se rozhoduje, zda software, prostředí a uživatelé jsou provozuschopní. Výsledný produkt nazýváme beta-verze. Pokud systém nebude připraven k provozu, musí se naplánovat odstranění rizik nasazení, podle plánu rizika odstranit a opět zkontrolovat provozuschopnost celého systému.

3.1.4 Fáze zavedení *(transitiv phase)*

V poslední fázi vývoje systému se software po otestování předává do provozu a instaluje se na všechna připravená pracoviště. Vývojáři reagují na podněty uživatelů a dolaďují software spíše z uživatelského pohledu než z pohledu funkčního. Jak je jednou systém předán koncovému uživateli je nutné pro opravy chyb a ladění vydávat opravné balíčky a revize.

Započetím této fáze znamená, že určitá podskupina hlavních požadavků dosáhla akceptované výše kvality a že uživatelská dokumentace je připravena ke snadnému pochopení sytému koncovými uživateli.

Tato fáze kromě beta-testování, přestavby databází a souběžného provozu se starým systémem také obsahuje vyškolení uživatelů a školitelů a zajišťuje podporu uživatelům při jejich počátečním používání.

Fáze zavedení může nabývat rozsahu od jednoduché fáze instalace nové verze s jednou iterací až po integraci složitého sytému v několika krocích a na několika pracovištích zároveň.

Vstupy:

- Výstupy fáze konstrukce

Výstupy:

- Softwarový produkt
- Uživatelské příručky
- Plán uživatelské podpory

Posledním milníkem vývoje softwaru je sada uživatelských a přejímacích testů prováděných v pracovním prostředí. Zde se rozhodne, zda bylo dosaženo veškerých požadavků.

3.2 Disciplíny

V metodě RUP rozlišujeme disciplíny na dvě skupiny. První skupina obsahuje disciplíny, které zajišťují vlastní práci na projektu. Druhá skupina pak slouží k řízení a konfiguraci skupiny první. Disciplínám v této skupině se říká podpůrné disciplíny.

Disciplíny jsou v RUP vyjádřeny jako pracovní procesy *(Workflows)*. Protože každá disciplína obsahuje větší množství prováděných činností, jsou disciplíny ještě děleny na tzv. podrobnosti pracovních procesů *(Workflow Details)* a ty teprve obsahují činnosti, které pracovníci provádějí.

Jaké disciplíny se jakou měrou se podílejí na vývoji softwaru v jednotlivých fázích, nám přehledně zobrazuje obrázek 9 v kapitole 3.1.

3.2.1 Hlavní disciplíny

Tvorba podnikového modelu *(business modeling)*

Problém, který nastává v každém podniku, je problém komunikace mezi softwarovými návrháři a pracovníky zodpovědnými za obchodní modelování. Pro zvýšení porozumění mezi těmito skupinami RUP na úrovni řízení projektu obě činnosti propojil a zavedl výkladové slovníky, aby komunikaci zjednodušil.

Tuto disciplínu je možné u některých projektů úplně vypustit.

Modelovacím prostředkem je jazyk UML a modelují se dva základní pohledy:

- podnikový model případů užití a
- podnikový objektový model.

Jak budou obě skupiny komunikovat, je důležité stanovit na začátku vývoje softwaru. Tato disciplína je tedy obsažena z větší části ve fázi zahájení projektu a dopracována ve fázích rozpracování a konstrukce.

Správa požadavků *(requirements)*

Základní požadavky na systém zadavatel dodá před samotným vývojem systému. V rámci celého vývoje se ale požadavky mění, přizpůsobují a objevují se další. Z toho důvodu je nutné samotné požadavky zaznamenávat, evidovat jejich změny a revize.

Splněním požadavků je podmíněn úspěch projektu, proto se musíme na požadavky dívat jako na důležitou část vývoje systému a musíme požadavky striktně dodržovat.

Tato disciplína je nejvíce obsažena ve fázi zahájení, ale doplňována je ve fázích rozpracování a konstrukce.

Analýza a návrh (analysis & design)

Zatímco u správy požadavků je kladen důraz na to, co bude systém umět, tato disciplína modeluje, jak bude systém jednotlivé činnosti provádět, jaká bude jeho architektura a jak bude vypadat datové úložiště.

Nejdříve se provádějí modely analýzy. Později se zpřesňují na modely návrhu, ze kterých se vychází v implementaci.

Na tuto disciplínu je kladen vysoký nárok ve fázi rozpracování a na začátku fáze konstrukce.

Implementace (implementation)

Implementaci lze chápat jako převod modelu návrhu do spustitelného programového kódu. Je důležité dodržet navrženou integraci a jednotlivé komponenty v průběhu implementace testovat.

Implementace je již obsažena ve fázi rozpracování, kde se implementuje architektura sytému. Na tu se ve fázi konstrukce nabalují další komponenty, modely a části systému. Software se dále upravuje podle přání zadavatele až do poslední fáze vývoje.

Metodika RUP obsahuje podrobný popis pro znovupoužití komponent a pro implementaci komponent nových takovým způsobem, aby byly jednoduše znovupoužitelné.

Testování *(test)*

RUP obsahuje testování jako samostatnou disciplínu. Tím je zřetelně kladen důraz na testování implementovaných komponent a jejich integraci do systémů a subsystémů. Testování rozdělujeme na tři základní skupiny – testy kvality, funkcionality a výkonu softwaru.

Protože požadavky na testování narůstají, není možné vše testovat ručně. Implementují se tedy automatické testy, které výsledky porovnávají s referenčními a informují vývojáře o správné implementaci či integraci částí sytému.

Testování ale netestuje pouze funkcionalitu sytému a nehledá pouze chyby v softwaru, ale provádí i validaci, tedy proces, při kterém kontroluje, zda implementovaná funkce odpovídá požadavkům na systém.

Testování se provádí ve fázi rozpracování, kde se testuje architektura systému, dále probíhá po celou dobu fáze konstrukce a v rámci uživatelských testů ji obsahuje i fáze zavedení softwaru.

Nasazení (deployment)

Tato disciplína probíhá v poslední fázi vývoje sytému podle předem již naplánovaných činností. Protože je ale důležité mít zpětnou vazbu od zadavatele v průběhu celého vývoje softwaru, je tato disciplína částečně obsažena ve všech fázích vývoje.

3.2.2 Podpůrné disciplíny

Řízení změn a konfigurace (configuration & chase management)

Řízení změn a konfigurace popisuje, jakým způsobem udržet a řídit velké množství artefaktů od velkého počtu osob, které na vývoji softwaru pracují.

Hlavní činností této disciplíny je také kontrola závěrů a vyhodnocení artefaktů ve třech různých oblastech:

- Současné aktualizace – přepsání artefaktu novější verzí zruší změny, které provedl jiný pracovník.

- Limitované ohlášení – změny, jako opravy problémů, zanesené do artefaktu se nemusejí dostat ke všem vývojářům.
- Více verzí – Větší projekty jsou vydávány postupně v několika verzích. Běžně se stává, že zatímco jedna verze je používaná v praxi, druhá se testuje a třetí je teprve ve fázi vývoje. Pokud se zjistí chyba v některé z verzí, je nutné tuto chybu opravit ve všech verzích. Kdyby tyto opravy nebyly řízeny procesem, mohl by se do oprav vnést zmatek a opravy tím prodražit.

Tato disciplína tedy zaznamenává veškeré změny v každém artefaktu a nabízí možnosti řešení od ohlášení chyby po její opravení.

Řízení projektu (project management)

Řízení projektu můžeme označit za umění. Umění docílit hladkého průběhu vývoje softwaru a úspěšného dokončení projektu, umění dodržet termíny, vyhnout se rizikům a ustát a problémy, které se objeví během vývoje a kterým nelze předejít.

Recept, který by dovedl projekt k úspěšnému konci, neexistuje. RUP ale obsahuje přístupy k řízení projektu, které šance úspěchu zvyšují. Jedná se o rady, návody, monitorování a samotné frameworky pro řízení rizik a celých projektů.

Správa prostředí *(environment)*

Správa prostředí řeší rozdíly mezi prostředím, kde je software vyvíjen a prostředím, do kterého bude software nasazen. Podporuje procesy a nástroje pro vývojáře a zaměřuje se na tvorbu průvodců nasazení systému. Veškeré tyto artefakty jsou v RUP zaneseny do balíku nazvaném Develompent Kit.

3.3 Role

Přeloženo z [7]: „Pojem role je abstraktní definice pro množinu zodpovědností za prováděné aktivity a vytvořené artefakty.“

Role jsou rozděleny do množin podle oblastí vývoje a podle pracovního umístění.

- Role analytiků jsou primárně určeny ke zkoumání požadavků.
- Role vývojářů jsou určeny k návrhu a implementaci.
- Manažerské role jsou určeny k řízení a konfiguraci procesu vývoje.
- Role testerů jsou určeny pro vývoj a provádění různých jedinečných testů.
- Výrobní a podpůrné role jsou potřebné pro další materiály a produkty použité při vývoji. Například operátoři, výtvarníci apod.
- Pomocné role je poslední skupina rolí, jejíž pracovníci nespadají do žádné jiné množiny rolí. Jednou z hlavních představitelů této množiny je role nazvaná „Any Role“. Činnosti definované touto rolí může provádět jakýkoliv pracovník.

3.4 Artefakty

V kapitole 2.3 jsem uvedl, že artefakty označujeme jako hmatatelné mezivýsledky projektu - dokumenty, modely, části kódů apod.

Všechny tyto artefakty vyvíjené metodikou RUP je možné zpracovávat formálně nebo neformálně. Hlavičky artefaktů obsahují název práce, název vyvíjeného artefaktu a historii verzí.

Formální nebo neformální zpracování závisí na důležitosti dokumentu a faktu, kdo všechno bude artefakt posuzovat. Například zákazník. Formálně zhotovené artefakty dodržují jednotný standardní formát. Historie verzí je zde uvedena, protože artefakt se nemusí pouze vytvořit, ale také modifikovat během vývoje softwaru.

Další postup závisí již na aktuálně vyvíjeném artefaktu. Formálně zhotovované artefakty ale obsahují ještě sekci úvod, ve kterém je stručně popsán jeho záměr a rozsah vzhledem k vyvíjenému produktu. Hlavičku dokumentů společně s úvodní sekcí zobrazuje příklad 1.

Rational Unified Process jako metodika vývoje softwaru

Diplomová práce

Specifikace požadavků

(Software Requirements Specifications)

Revize *(Revision)*

Datum	Verze	Popis	Autor
03/04/2008	1.0	-	Vladimír Rytíř
11/04/2008	2.0	Nový funkční požadavek	Vladimír Rytíř

Úvod *(Introduction)*

Záměrem tohoto artefaktu je shromáždit, zanalyzovat a definovat klíčové potřeby a vlastnosti systému. Vize je zaměřena na ty vlastnosti systému, které jsou vyžadovány zainteresovanými osobami a koncovými uživateli a dále na skutečnost, proč tyto potřeby existují

Záměr (Purpose)

Tento dokument popisuje vnější chování systému POS, nefunkční požadavky, omezení systému a další faktory potřebné ke specifikování veškerých požadavků na systém.

Rozsah *(Scope)*

Tento artefakty popisuje požadavky kladené na celý systém a na všechny jeho komponenty.

Příklad 1: Hlavička artefaktů a úvodní sekce

Je důležité zmínit, že není pravidlem, aby výstupem či vstupem každé činnosti byl artefakt. Vstupních a výstupních artefaktů může být více nebo naopak pouze jejich jednotlivé části.

4 Unified Process

Unified Process (dále jen UP) je průmyslovým standardem procesu tvorby softwarového vybavení a stejně jako RUP definuje při vývoji systému otázky kdo, co, kdy a jak.

4.1 Historie

Ivar Jacobson z firmy Rational (metodika RUP) publikoval v roce 1999 knihu Unified Software Development Process, v níž důkladně popisuje metodiku UP, která se tak stala otevřeným standardem vývoje softwaru. Vzhledem k autorovi je zřejmé, že má mnoho vlastností společných s komerční metodikou RUP, který UP v některých oblastech rozšiřuje a doplňuje.

4.2 Základní axiomy

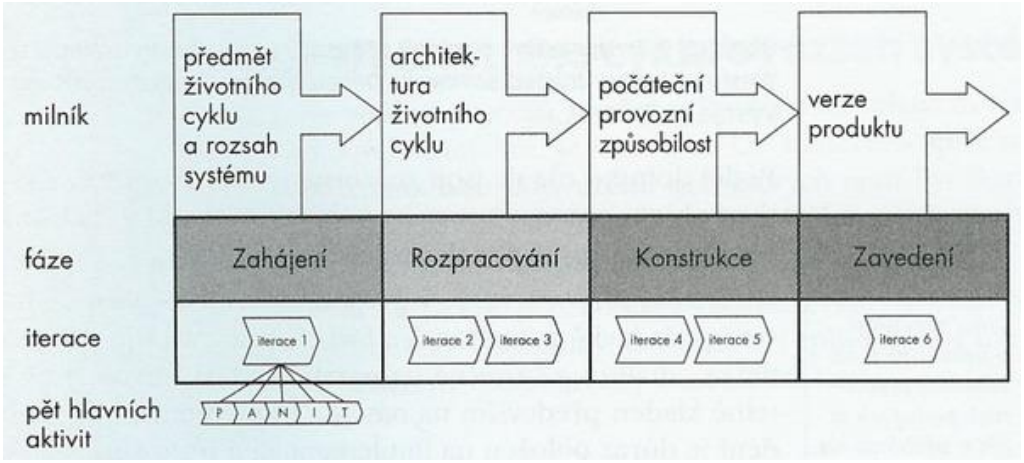
Metodika UP obsahuje tři zásady, kterými je dobré se při návrhu a vývoji softwaru řídit:

- řízení případem užití a rizikem,
- soustředění se na architekturu a
- iterace a přírůstky.

4.3 Fáze a milníky

Fáze a milníky mají u metodiky UP stejný význam, jako u metodiky RUP (viz obrázek 10). Jejich vysvětlení můžeme nalézt v kapitole 3.1 Fáze a milníky.

Vysvětlení důležitosti pěti hlavních aktivit v jednotlivých fázích metodiky nalezneme v kapitole 4.4 Aktivity.



Obrázek 10: Fáze a milníky UP [3]

4.4 Aktivity

V této kapitole si uvedeme a vysvětlíme pět hlavních aktivit, které procházejí přes všechny fáze vývoje systému. Při bližším zkoumání zjistíme, že jsou velmi podobné určitým fázím vývoje u metodiky RUP. Některé aktivity jsou ovšem v určitých fázích nepoužitelné nebo volitelné.

4.4.1 Požadavky

Požadavky na systém zachycují definici toho, co by měl systém dělat, co by měl umět, jak by měl vypadat, a stanovují omezení, za kterých musí systém pracovat. Kromě toho se musí vyřešit protichůdné požadavky a stanovit priority. Počáteční požadavky většinou dodává zadavatel projektu a společně s vývojáři dále spolupracuje na specifikaci jejich rozšíření a doplnění. V první fázi vývoje, tedy ve fázi zahájení, se sepíší všechny funkční a nefunkční požadavky na systém. Ve fázi rozpracování se tyto požadavky dále upřesní a ve fázi konstrukce vyvíjíme snahu o doplnění požadavků do konečné podoby.

4.4.2 Analýza

Analýza požadavků je vlastně modelové zobrazení, přesné vysvětlení a strukturování požadavků na systém. Toto zobrazení nazýváme analytický model. Analýza začíná ve fázi zahájení, pokračuje zpřesňováním ve fázi rozpracování a končí ve fázi konstrukce.

4.4.3 Návrh

Návrh systému spočívá v realizaci požadavků do architektury systému. Zatímco analytický model nám říká, „co“ je potřeba udělat, tato aktivita nám poví, „jak“ to máme udělat. Ve fázi zahájení je možné navrhnout model, kterým můžeme simulovat činnosti systému a tím potvrdit správnost rozhodnutí (např. při výběru technologií). Model se nazývá prototypem a po simulaci bude zahozen. Ve fázi rozpracování navrhujeme architekturu sytému a ve fázi konstrukce je model systému dokončen. Pokud se při testování systému vyskytnou chyby, které je nutné odstranit, muže být model ve fázi zavedení upraven.

4.4.4 Implementace

Implementace obsahuje hlavní programátorskou činnost, při které se píše kód dle předchozích modelů a návrhů. Fáze zahájení obsahuje tvorbu prototypu, pokud byl v předchozí aktivitě navržen. Ve fázi rozpracování je vytvořena spustitelná kostra softwaru a základní prvky architektury. Ve fázi konstrukce tvoříme samotný systém a ve fázi zavedení se opravují chyby a softwarový produkt je přizpůsoben pro pracovní prostředí.

4.4.5 Testování

Aktivita testování ověřuje funkčnost implementace a dodržení požadavků systému. Protože v první fázi není co testovat (prototyp se zahazuje), začíná testování až ve fázi rozpracování, kde máme již hotovou architekturu. Ve fázi konstrukce se testují jednotlivé mezivýsledky a ve fázi zavedení se provádějí hlavně přijímací testy pro pracovní prostředí.

4.5 UP versus RUP

Metodiky RUP a UP více věcí spojuje, než dělí. Hlavní rozdíly spočívají spíše v úplnosti a v jednotlivých detailech než v otázkách sémantiky a ideového obsahu.

Jak UP, tak RUP modelují aspekty kdo, kdy, co a jak procesu tvorby softwarového vybavení – činí tak ale nepatrně odlišnými způsoby [3].

Rozdíl obou metodik nalezneme i ve vyjádření základních aspektů, jedná se ale spíše o terminologii, protože sémantika zůstává stejná. Například role, kterou jsme popisovali u metodiky RUP v kapitole 2.3 je v metodice UP nazývána dělník *(worker)*, disciplíny v RUP (viz kapitola 3.2) odpovídají aktivitám v UP (viz kapitola 4.4) apod.

Pokud se podíváme blíže na disciplíny obou metod, tak zjistíme, že v UP nejsou u jednotlivých iterací zahrnuty pracovní postupy například pro plánování, odhad, specifikaci projektu a další, jak je tomu u metodiky RUP. Tam také můžeme některé disciplíny vynechat, ale v UP je klíčovým východiskem skutečnost, že iterace obsahuje všechny prvky softwarového projektu.

Můžeme také pozorovat vylepšení RUP oproti UP. Zatímco v UP jsou „Analýza“ a „Návrh“ systému rozděleny na dvě různé aktivity a existuje pouhé doporučení, aby obě dvě zpracovával jeden vývojový tým, RUP tyto disciplíny již přímo spojil do jedné pod název „Analýza a návrh systému“.

Metodika UP také místo sestavování týmu pro určité aktivity sestavuje tým pro tvorbu výsledků a milníků. Zaměřuje se tedy více na „cíl“ než na „úlohu“.

5 Další uznávané metodiky

V předchozích kapitolách bylo uvedeno několik metodik a postupů vývoje informačních systémů. Tyto metodiky mají více či méně společných rysů: komunikace se zákazníkem, zpracování požadavků, analýza a návrh systému, implementace a dodání. Takovýmto metodikám budeme dále říkat „tradiční metodiky“ vývoje softwaru. Existují ale i jiné metodiky vývoje, a to metodiky agilní.

Vznik agilních metodik si vynutila samotná poptávka na trhu. U některých projektů je důležité dlouze specifikovat a navrhovat všechny odvětví a úskalí vyvíjené aplikace, u jiných je zase důležité vyvinout systém v co možná nejkratším termínu. Vznikl tedy důležitý požadavek, aby analýza systému byla rychlá, ale zároveň si zachovala požadovanou kvalitu.

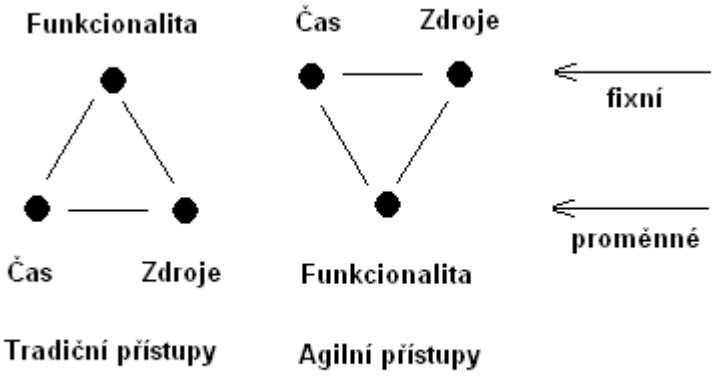
Dle [8] existuje několik vhodných metodik:

1. Většina „výrobců“ klasických metodik přidává do nových verzí nějaký dodatek, kapitolu, „best practice“ pro vývoj internetových projektů a pro podporu malých týmů.
2. Vznikají zcela nové metodiky, které se snaží komplexně řešit dnešní požadavky na rychlý a flexibilní vývoj aplikací. Souhrnně jsou tyto metodologie nazývány „agile methodologies“ (agilní, hbité metodiky) a asi nejznámějším představitelem je Extrémní programování. Kromě něj však nalézáme i další zajímavé projekty.

Společné rysy těchto řešení jsou opět pěkně sepsány v [8]:

„Důraz na sepětí se zákazníkem či zadavatelem, důkladná komunikace, zpětná vazba, kladení zásadní důležitosti na fázi implementace, snaha redukovat dobu potřebnou pro analýzu a návrh a integrovat tyto fáze s implementací, pravidelné vyvíjení prototypů, odvaha zapracovávat změny, snaha o jednoduchost a především důsledné testování.“

Rozdíly mezi tradičními a agilními metodikami nejlépe shrnuje obrázek 11.



Obrázek 11: Rozdíl mezi tradičními a agilními metodikami [8]

Na tomto obrázku vidíme, že pro tradiční přístupy vývoje softwaru je důležitá funkcionalita produktu. Zdroje a čas se tedy mohou měnit během projektu. Naopak u agilních přístupů jsou právě zdroje a čas pevně definované a funkcionalita nemusí být vždy přesná a kompletní.

5.1 Extrémní programování

Extrémní programování je pravděpodobně nejznámějším a nejrozšířenějším zástupcem agilních metodik. Jedná se o přirozený intuitivní přístup k vývoji systému. Extrémní slovo v názvu označuje skutečnost, že všechny známé postupy a myšlenky dotahuje do extrémů. Autorem metodiky je američan Kent Beck, který je známým expertem na vývoj softwaru.

Základním filozofickým východiskem extrémního programování je přesvědčení, že jediným exaktním, jednoznačným, změřitelným, ověřitelným a nezpochybnitelným zdrojem informací je zdrojový kód [8].

V předchozí kapitole jsem definoval rozdíl mezi tradičním a agilním přístupem. Oba přístupy pracují se třemi základními proměnnými, akorát každý k nim přistupuje z jiného pohledu (viz obrázek 11). V extrémním programování ale hraje důležitou roli, a podle autora možná i nejdůležitější, další proměnná - šíře zadání. Šíře zadání označuje podmnožinu všech funkcí požadovaných zákazníkem za klíčové. Vývoj ostatních funkcí bude záležet na času a zdrojích. Zákazník tuto skutečnost musí pochopit a akceptovat.

5.1.1 Základní hodnoty

Extrémní programování se řídí pěti základními hodnotami:

Komunikace

Je důležité, aby všichni členové vývojového týmu, ve kterém se objevuje i zákazník nebo uživatel, mezi sebou otevřeně komunikovali. Bez komunikace může dojít k opomenutí předání důležitých informací, ke špatnému pochopení zadání, ke špatnému návrhu a tedy i ke špatnému výsledku.

Jednoduchost

XP se snaží o co nejjednodušší návrh a co nejjednodušší implementaci požadovaných funkčností. Pro XP metodiku je lepší navrhnout jednoduchý systém a později ho přepracovat, než rovnou vyvinout složitější funkce, které zůstanou p dlouhou dobu nevyužity.

Vývojový tým se tak musí naučit myslet pouze na současnost a na daný problém. Předpokládat nějakou budoucnost je podle XP nesmyslné, protože ta se stejně dříve či později změní.

Zpětná vazba

Zpětná vazba probíhá v XP v několika časových rovinách. Programátoři píší testy a testují aplikační logiku v řádu dní nebo hodin a zároveň i zákazníci píší testy funkcionality systémů nebo jeho částí, kde získáme zpětnou vazba v řádech týdnů až měsíců.

Čím častěji kontrolujeme plnění požadavků, čím častěji komunikujeme se zákazníky a čím častěji provádíme testy, tím získáme více zpětné vazby a zvýšíme pravděpodobnost úspěchu projektu.

Odvaha

Protože před vlastní implementací neprobíhá detailní navrhování architektury systému (viz kapitola 6.1.2), můžeme narazit na slepou uličku vývoje, například nemožnost integrace dalšího modulu do systému. Nutnou skutečností je zde oprava předchozích částí vývoje i za cenu vyhození několikadenní práce, což vyžaduje notnou dávku odvahy.

Respekt

Posledním důležitým aspektem XP je respektování ostatních členů vývojového týmu. Každý člen se musí starat o ostatní kolegy, zajímat se o jejich dosavadní práci a pomáhat jim při řešení problémů.

5.1.2 Základní postupy

Kent Beck, autor metodiky XP, definoval 12 základních postupů, které by měly vést ke kvalitnímu produktu. Pokud někdo při vývoji použije základní hodnoty XP a nějaké z níže popsaných základních postupů, nejedná se o metodiku XP. Autor klade důraz na dodržování všech uvedených činností zároveň. V této kapitole si všechny postupy pouze stručně vysvětlíme.

Plánovací hra

Dle XP je vývoj softwaru dialog mezi klíčovými požadavky a těmi ostatními (šíře zadání). Pro plánování projektu se tedy nepoužívají běžné specifikace požadavků a dokumenty vize projektu, ale zvláštní, tzv. plánovací hra. Této hry se účastní celý vývojový tým, který definují šíři zadání a rozhodují o prioritách, verzích a důležitých datech. Z plánovací hry se potom u rozsáhlejších projektů mohou vytvořit detailnější pohledy na jednotlivé verze produktu.

Malé verze

Nové verze je nutno uvolňovat v co možná nejkratších intervalech, které dávají určitý přínos pro zákazníka.

Metafora

XP klade důraz na co nejjednodušší vysvětlování funkčnosti systému. Proto se nepoužívají přesné definice, ale metafory. Například výpočet ceny je tabulky. Ve skutečnosti není tabulka, jsou to určité vzorce a postupy, ale ve výsledku vypadá jako tabulka.

Jednoduchý návrh

Navrhujeme pouze pro danou část vývoje. Nové prvky se budou navrhovat až v okamžiku jejich implementace.

Testování

Viz zpětná vazba v sekci 5.1.1.

Refaktorizace

Důležité je oddělit implementaci nových funkcí od refaktorizace stávajícího kódu. Pokud bude integrace nové funkce jednodušší po změně nějaké metody, je nutné nejdříve nefaktorizovat. Je možné tedy refaktorizovat před změnou programu (přidání nové funkce) nebo zjednodušit kód po změně.

Párové programování

Vychází z představy dvou lidí na jeden počítač. Zatímco jeden programátor píše novou funkci, druhý přemýšlí nad celkovou strategií vývoje, a jak nová funkce do této strategie zapadne.

Společné vlastnictví

Všichni vývojáři vlastní kompletní zdrojový kód a všichni dohromady jsou za něho zodpovědní.

Nepřetržitá integrace

Integrace systému musí probíhat nepřetržitě, minimálně denně. K tomu je vhodné vyčlenit jeden počítač pouze pro integraci a sestavování celého projektu.

Čtyřicetihodinový pracovní týden

Neúměrné přetěžování vývojářů (lidských zdrojů) vede k větší nepozornosti a větší chybovosti psaného kódu. Přetěžování lidských zdrojů je podle XP nesprávné řízení projektu.

Zákazník na pracovišti

Pro lepší komunikaci je nutné mít zákazníka fyzicky přítomného na pracovišti.

Standardy pro psaní zdrojového textu

Jak už jsme si vysvětlili v úvodu, zdrojový kód je jediným důležitým stanoviskem vývoje systému. Protože jsou do něho zaneseny veškeré požadavky a specifikace, je nutné, aby byl kód dobře strukturovaný a srozumitelný.

5.1.3 XP versus RUP

XP je podobně jako RUP velmi pružná metodika. Dokonce je možné XP metodiku implementovat do jiné metodiky tím, že ji neaplikujeme na celý vyvíjený produkt, ale pouze na určitý problém.

XP má s metodikou RUP hodně společného a zároveň i hodně rozdílného. Stále mluvíme o inkrementálním a iterativním vývoji a definujeme požadavky na systém, avšak u každé metodiky probíhá definice jiným způsobem. Zatímco RUP je velmi formální a lpí na tvorbě spousty artefaktů, XP klade důraz pouze na zdrojový kód.

XP se proti RUP hodí spíše na menší projekty a pro menší vývojové týmy. Nejvýhodnější je pravděpodobně pro vývoj webových aplikací, kde je důležitější doba vývoje před implementací veškerých požadavků. Naopak bychom ji velmi složitě používali pro rozsáhlé například bankovní aplikace, kde může vývoj trvat i několik let. Na tomto místě si snáze představíme RUP.

Obě metodiky mají i jednu společnou nevýhodu. Prvotní aplikace metodiky na projekt. Zatímco XP je relativně jednoduchá metoda, obtíže si na ní vývojáři zvykají. Rup je zase nutné předem dlouze nastudovat, ale po zavedení dodržuje stále tradiční postupy známé z předchozích jednodušších metodik.

U RUP je nutné vidět do budoucna, předpokládat, co by mohl zákazník ještě potřebovat a zabudovat pro takové potřeby a další rozšíření funkce. XP oproti tomu nehledí do budoucna ale je velmi tolerantní ke změnám a tedy jednoduše budoucí změny akceptuje.

Závěrem je nutné také poukázat na mentalitu jednotlivých kontinentů a států. Zatímco je v USA běžné několikrát přepisovat a měnit postupy během projektu, hlavně že je dosaženo cíle, v Evropě je tento postup označován samotnými vývojáři za chybný. Proto bude aplikace metodiky XP, tak jak je definována, jednodušší v USA, tedy v zemi jejího vzniku, než u nás v České republice.

5.2 Scrum

Další z agilních metodik, která není již tak extrémní, jako metodika předchozí, a používá základy jak agilních, tak objektově orientovaných metodik.

Celý vývojový proces si lze představit jako „dotlačení se do cíle“, čemuž odpovídá právě název této metodiky. Autoři se také, podobně jako autor XP, snažili vyvinout flexibilní metodiku reagující na měnící se požadavky.

5.2.1 Klíčové pojmy

Základní části metodiky si blíže popíšeme v další kapitole 5.2.2 Postupy. Zde si pouze stanovíme základní pojmy (hodnoty), obdobně jako u metodiky XP.

Flexibilní předměty dodání

Předměty dodání jsou dokumenty a jiné dodávky (artefakty), které jsou vybírány podle průběhu projektu. Obsahy nejsou striktně definovány. Vše záleží na aktuálních potřebách a požadavcích zákazníků.

Flexibilní harmonogram

Oproti metodice XP nedodržuje Scrum fixní dobu vývoje. Předem je nutné zákazníkům vysvětlit, že se doba vývoje může vychýlit jedním či druhým směrem.

Malé týmy

Tento pojem neznamená, že tuto metodiku nemohou požívat větší vývojové týmy, ale že by se vývojáři měli rozdělit do několika týmů o třech až šesti členech. Jednotlivé týmy pak mohou být zodpovědné za vrstvy systému nebo za skupinu funkcí.

Časté revize

V rámci vývoje se týmy scházejí každý den na krátké maximálně půlhodinové schůzky *(Scrum meetings)*, jejichž obsahem jsou přehledy vykonaných prací od předchozí schůzky, nová rizika, další naplánování práce, problémy apod.

Spolupráce

Podobně jako v metodice XP je důležitá komunikace nejen uvnitř celého týmu, ale i s okolím (zákazníci, uživatelé). Scrum fyzickou přítomnost zákazníka na pracovišti pouze doporučuje.

5.2.2 Postupy

Vývoj je rozdělen do tří základních fází – fáze předehry *(Pregame)*, fáze hry *(Game)* a fáze dohry *(Postgame)*.

První fáze obsahuje plánování rozsahu, harmonogramu, zdrojů a podobných náležitostí vývoje. Důležitým výstupem je seznam úkolů, které je důležité definovat - tzv. blacklog. Seznam úkolů je seřazen podle priorit, které stanoví zákazník. V rámci první fáze se také tvoří architektura v závislosti na stanovených požadavcích.

Druhá fáze je provádění činností v jednotlivých iteracích, které se zde nazývají sprinty. Každý sprint trvá přibližně 30 dnů a je zakončen schůzkou. Činnosti pro jednotlivé sprinty vybírá manažer

projektu z blacklogu. Na závěrečné schůzce jsou zúčastnění vývojáři i zákazníci. Zákazníkům se zde předvede prototyp projektu a definují se nové požadavky do blacklogu.

Důležitou součástí každého sprintu jsou každodenní setkání celého týmu – scrum meetingy (viz sekce časté revize v kapitole 5.2.1).

Poslední třetí fází je uzavření. Tato fáze nastává, pokud vedení týmu (nebo zákazník) určí, že další sprint již není potřeba. V této fázi se provádí kompletní integrace systému, testování a tvorba dokumentace.

5.2.3 Scrum versus RUP

Scrum se má k metodice RUP obecně blíže, než předchozí metodika XP. JE to dáno tím, že hlavní důraz se neklade pouze na zdrojový kód, ale vyvíjejí se i formální dokumenty. Tyto dokumenty ale nemají striktně stanovený obsah, jako artefakty v RUP.

Další společnou vlastností je odpovědnost za jednotlivé výstupy činností. U metodiky Scrum je každý vývojář odpovědný za své třídy a své objekty. Problém by mohl nastat, pokud by daný pracovník opustil vývojový tým.

Scrum se také přibližuje k RUP důrazem, který klade na řízení projektu.

I přes některé společné vlastnosti se ale stále jedná o agilní metodiku, která neklade veliký důraz na detailní návrh před implementací. Proto je tato metoda také vhodná spíše pro menší projekty.

5.3 Agilní metodiky versus RUP

Pokud porovnáme obecné přístupy k vývoji produktů, je zřejmé, že se jedná o různé druhy metodik a tedy jsou určeny pro různé aplikace.

Zatímco RUP je určen pro rozsáhlé projekty, které vyžadují detailní zpracování a návrh požadavků a detailní návrh architektury, agilní metodiky jsou určeny spíše pro menší a často se měnící software, např. webové aplikace, kde je obsah všech vysněných funkcí sytému až na druhém místě.

To ale neznamená, že RUP není možné použít pro takovéto druhy projektů. RUP je obecná metodika, kterou lze upravit na míru velkého množství projektů. K úpravě pro webové aplikace pomáhá dodatek této rozsáhlé metodiky nazvaný „RUP for Small Project“ a dodržování nejlepších praktik vývoje (viz kapitola 2.1), kde je nastíněna např. i neustálá komunikace se zákazníkem (uživatelem) systému.

U RUP je nutné pro zákazníky vytvářet velké množství artefaktů reflektující aktuální vývoj, které si mohou projít a zkontrolovat. Naproti tomu u agilních metodik je kladen velký důraz na stálé komunikaci se zákazníkem a vztah je založen na důvěře.

Hlavní rozdíly obou přístupů přehledně shrnuje tabulka 2.

	Požadavky	Obsah	Použití
Tradiční metodiky (UP, RUP)	Většina definována předem	Činnosti, dokumenty, procesy, znovu použitelnost	Rozsáhlé projekty a velké vývojové týmy
Agilní metodiky (XP,)	Jen hrubý nástin	Principy, praktiky, pravidla	Menší vývojové týmy, časově definované projekty (tim-to-market)

Tabulka 2: Porovnání tradičních a agilních metodik

6 Profesionální organizační systém

6.1 Zadání

Pro praktické znázornění metodiky RUP jsem vyvinul webový informační systém pro podporu řízení vztahu se zákazníky *(Customer Relationship Management, CRM)* nazvaný Profesionální informační systém (dále jen POS). CRM systém jsem zvolil ze dvou důvodů. Prvním důvodem je, že se bez takového systému, nebo alespoň jeho základní podoby, v dnešní době žádná úspěšná společnost, která komunikuje se zákazníky, neobejde. Dalším je univerzálnost tohoto systému. V základu budou CRM systémy velmi podobné, protože každý udržuje seznam zákazníků a umožňuje k nim dopisovat různé informace bez ohledu na to, v jakém oboru společnost podniká.

Prvotní neformální zadání sytému, který zákazník požaduje, by mohlo vypadat následovně:

„Úkolem je vytvořit systém pro uchovávání zákazníků firmy. Takového zákazníka bude moct pracovník firmy do systému uložit, měnit jeho údaje a případně ho i ze systému odstranit. U zákazníka bude moct vyplnit i základní údaje pro komunikaci.

Pracovníci si poté mohou se zákazníkem domlouvat různé schůzky.

Kontakt se zákazníkem je ale důvěrný, takže domluvené schůzky bude moct vidět pouze ten pracovník firmy, který se zákazníkem může komunikovat.

Přiřazení k zákazníkovi provádí stanovený interní pracovník firmy.

Pracovníci firmy musejí mít přístup k systému umožněn i z domova.“

V dalších podkapitolách budeme postupovat již přesně podle metodologie RUP. U každého artefaktu si nejprve vysvětlíme, k čemu slouží, a potom si neformálně uvedeme příklady. Formální vyjádření dokumentů můžeme nalézt v přílohách této práce (viz příloha 3).

6.2 Fáze zahájení *(Inception phase)*

6.2.1 Vývojová studie *(Development Case)*

První činností fáze zahájení (viz kapitola 3.1.1) je vytvoření artefaktu Vývojová studie, který popisuje veškeré vývojové procesy a definuje vytvářené artefakty pro celý projekt.

Pro nadpis sekce se použije název disciplíny z RUP. Sekce se dále skládá z několika částí.

První částí je popis změn pracovního procesu (workflow) vzhledem ke standardnímu procesu. Další částí je tabulka se seznamem vyvíjených artefaktů, u kterých se definuje použití pro každou fázi vývoje, jaký způsobem se artefakt hodnotí a nakonec použité nástroje pro tvorbu. Pod seznamem artefaktů se mohou uvádět poznámky k dané disciplíně. Většinou se v poznámkách uvádějí další artefakty, které se pro daný vývoj produktu nepoužijí spolu s vysvětlením, proč nejsou důležité. Poslední částí je výpis dalších problémů a úskalí při použití dané disciplíny.

Součástí Vývojové studie by měl být i plán vývoje zobrazen většinou Ganttovým diagramem. Plán je sestaven pro každou fázi zvlášť a přehledně zobrazuje počáteční naplánování první iterace dané fáze. Z Ganttova diagramu lze vyčíst, nejen jak budou jednotlivé činnosti prováděné za sebou, ale i jejich paralelní zpracování. V našem praktickém příkladě byl tento diagram vynechán, protože

vývojový tým je složen z jednoho pracovníka, tudíž paralelní zpracování zde nemůžeme využít. To ale neznamená, že artefakty nemohou být vyvíjeny najednou.

Ganttův diagram nám plně nahrazuje artefakt Plán vývoje *(Software Development Plan)*.

Specifikace požadavků *(Requirements)*

Pracovní proces (*Workflow*)
Beze změn.
Artefakty (*Artifacts*)

Artefakty (*Artifacts*)	**Použití**		**Zhodnocení**	**Použité nástroje**
	Fáze zahájení (*Inception*)	**Fáze projektování (*Elaboration*)**		
Vize (*Vision*)	Musí	Musí	Formální	Microsoft® Word®
Storyboard	Může	Může	Neformální	Microsoft® PowerPoint®
Specifikace softwarových vlastností (*Software Requirements Specifications*)	Musí	Musí	Formální	Microsoft® Word®
Doplňková specifikace (*Supplementary Specifications*)	Nebude	Nebude	Formální	Microsoft® Word®

Poznámky

- Veškeré požadavky na systém jsou uchovávány v dokumentu Vize a v use-case modelu a v doplňkové specifikaci dokumentu Specifikace softwarových vlastností. Z toho důvodu již nejsou požadavky uloženy v repositáři artefaktu Požadované atributy (*Requirements Attributes*).
- Vzhledem k rozsahu projektu a neexistence fyzického zadavatele je vynechán artefakt Požadavky zúčastněných osob (*Stakeholder Requests*).

Ostatní problémy
Žádné.

Příklad 2: Artefakt Vývojová studie

V této části artefaktu vidíme, že se jedná o disciplínu Specifikace požadavků, u které pracovní proces proběhne tak, jak jej definuje RUP. Ze seznamu vyvíjených artefaktů v této disciplíně je zde

pro ukázku uveden artefakt Vize, který musí být vyvinut nebo k dispozici v obou fázích vývoje. Zhodnocení artefaktu bude formální, protože není tvořen pouze pro interní použití, ale bude se předkládat zákazníkovi. Pro vývoj bude použit nástroj Microsoft Word. Naopak artefakt Doplňková specifikace vyráběn nebude, protože je již obsažen v jedné z částí artefaktu Specifikace softwarových vlastností, jak je popsáno v poznámce.

Vytvoření artefaktu Vývojová studie musí být v seznamu také uvedeno. Kompletní artefakt je formálně zpracován v příloze 4.

6.2.2 Vize *(Vision)*

Tento artefakt je důležitým artefaktem pro shromáždění, zanalyzování a definování klíčových potřeb a vlastností systému. Vize je zaměřena na ty vlastnosti systému, které jsou vyžadovány zainteresovanými osobami a koncovými uživateli a dále na skutečnost, proč tyto potřeby existují. Záměrem je tedy definování klíčových požadavků na potřeby koncových zákazníků.

Artefakt včetně počátečního úvodu (viz kapitola 3.4) popisuje 10 důležitých částí.

Uvedení

Definujeme, jaké obchodní výhody a příležitosti přinese používání systému.

Dále vymezíme problém, tedy definujeme, proč je vývoj systému nutný a jaká úskalí provázejí firmu bez existence sytému.

Nakonec stanovíme, pro jaké společnosti je daný systém vyráběn. Zda se jedná o velké tuzemské společnosti nebo jestli je systém připraven i pro mezinárodní trh a podobně.

Zainteresované osoby a popis uživatelů

Jak z názvu vyplívá, tato část definuje všechny zainteresované osoby. Těmi nemusí být nutně pouze uživatelé systému. Jedná se o všechny, kteří mají k vyvíjenému systému určitý vztah. Nezáleží tedy pouze na lidech, kteří budou systém ve výsledku používat. Zákazníkem může být například i úplně cizí firma. Důležitou osobou je také ten, kdo vývoj systému zaplatí.

Do této sekce patří také definice potřeb uživatelů. Není zde potřeba stanovit veškeré jejich potřeby, ale definovat klíčové požadavky.

Přehled základních informací o produktu

Tato část popisuje základní architekturu systému a technologie, které budou potřebné pro provoz systému. Dále popisuje veškeré softwarové a hardwarové předpoklady, na kterých bude systém závislý. Důležité je zde vyzdvihnout nutné licenční požadavky.

Vlastnosti systému

Tato kapitola definuje vlastnosti systému, jeho schopnosti a uživatelské možnosti.

Omezení

Zde se popisují další závislosti kladené na systém.

Definování kvality

Kvalitu zde definujeme v závislosti na výkonu, dostupnosti, použitelnosti a udržovatelnosti systému.

Precedence a priority

V této části se určí prioritní funkčnost systému. Pokud je například systém rozdělen na několik na sobě nezávislých modulů, stanovíme zde pořadí vývoje jednotlivých modulů a určíme, v jaké verze budou dané moduly obsahovat.

Další požadavky

Mezi další požadavky patří požadavky na pracovní prostředí, výkon, systémové požadavky a aplikační standardy.

Požadavky na dokumentaci

Seznam příruček a dokumentací, které budou dodány spolu s programem.

6.2.3 Seznam rizik *(Risk List)*

Cílem artefaktu seznamu rizik je definovat rizika, která by mohla ohrozit projekt a navrhnout možná řešení. RUP se tímto nesnaží rizika vývoje obejít, ale naopak jít jim naproti a co nejdříve se s nimi vypořádat. Rizika definovaná u produktu POS a jejich možná řešení zobrazuje tabulka 1.

Technická rizika	**Možné řešení**
Větší počet uživatelů pracující se systém ve stejném čase by mohlo vést k rapidnímu snížení výkonu systému.	V první iteraci fáze implementace vytvoříme prototyp a otestujeme zatížení systému.
Nekompatibilita klientské části systému se staršími webovými prohlížeči.	V každé iteraci od počátku fáze implementace otestujeme systém na všech nejčastěji používaných webových prohlížečích a to i ve třech starších verzích.
Vývojový tým má malé zkušeností s metodikou Rational Unified Process a s objektově orientovanými technikami. To by mohlo vést k nízké kvalitě produktu.	Stanovíme konzultanta, který bude vývojovému týmu nápomocen.

Příklad 3: Artefakt Seznam rizik

6.2.4 Plán vývoje *(Software Development Plan)*

Plán vývoje (v literatuře uváděn také jako produktový plán) je rozsáhlý dokument, který obsahuje všechny důležité informace vyžadované k řízení projektu. Je v něm uvedena například organizace vývojové firmy, role, které zastávají jednotlivý vývojáři a samozřejmě detailnější rozpracování plánu vývoje pro jednotlivé fáze RUP a jejich iterace. Samozřejmě nemůže být kompletní plán stanoven a počátku vývoje, proto je tento artefakt modifikován během celého vývojového procesu.

Pro náš vyvíjený produkt postačí neformální verze tohoto dokumentu. Dokument zde slouží pouze pro naplánování jednotlivých prováděných akcí a ke stanovení celkové doby vývoje.

Role a zodpovědnosti

Jméno	Role
Vladimír Rytíř	Project Manager System Analyst Software Architect Designer Integrator Test Manager, Designer, Analyst Tester

Fáze zahájení

Datum dokončení: 7. 4. 2008

Project Management
Workflow Detail: Plan the Project
Role: Project Manager

Aktivita	Vyvíjený artefakt	Datum dokončení
Define Project Organization and Staffing	Software Development Plan	18. 3. 2008
Define Monitoring and Control Processes	Software Development Plan	21. 3. 2008
Plan Phases and Iterations	Software Development Plan	26. 3. 2008
Compile Software Development Plan	Software Development Plan	28. 3. 2008

Fáze projektování

Datum dokončení: 9. 5. 2008

1. ITERACE
Requirements
Workflow Detail: Understand Stakeholder Needs
Role: System Analyst

Aktivita	Vyvíjený artefakt	Datum dokončení
Manage Dependencies	Vision (refined)	8. 4. 2008

Příklad 4: Artefakt Plán vývoje

6.2.5 Glosář *(Glossary)*

Glosář neboli slovník slouží pro lepší komunikaci mezi zákazníkem a vývojářem. Pokud by slovník nevznikl, mohli by obě strany sice používat stejné názvy, ale každý by si pod daným jménem mohl představit něco jiného.

V našem případě Glosář obsahuje pracovní definice Profesionálního organizačního systému, tedy všechny výrazy, které mají specifický význam.

Jak takový seznam výrazů vypadá, přehledně zobrazuje příklad 5.

Seznam zákazníků
Seznam všech zákazníků zanesených v systému.

Informace o zákazníkovi
Kontaktní informace zákazníka.

Detailní informace
Informace o zákazníkovi, schůzky a poznámky.

Schůzka
Schůzka se zákazníkem domluvená mezi pověřeným pracovníkem a zákazníkem. Schůzky mohou být naplánované (budoucí) i proběhnuté (minulé).

Poznámka
Poznámka ke schůzce.

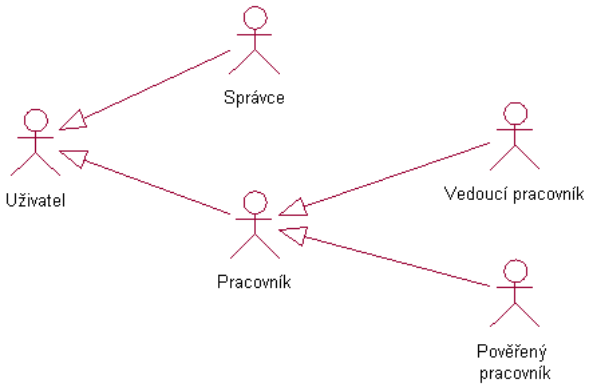
Příklad 5: Artefakt Glosář

Glosář je také nutno rozšiřovat během celého vývoje produktu.

6.2.6 Specifikace požadavků *(Software Requirements Specifications)*

Výstupem činnosti Nalezení aktérů a use-casů *(Find Actors and Use Cases)* je use-case model. Artefakt má tedy podobu diagramů. Později, ve fázi projektování, se z těchto diagramů vychází a převádějí se do výsledné podoby. Diagramy se sdružují v artefaktu Specifikace požadavků.

Příklad 6 zobrazuje nalezené aktéry systému. Use-casy systému budou zobrazeny v příkladu 10 v následující iteraci vývoje.



Příklad 6: Diagram aktérů artefaktu Specifikace požadavků

Protože RUP používá definici požadavků zvanou FURPS+, musíme i zde uvést veškeré požadavky na systém, ne jen funkční. Tyto požadavky se zapisují v sekci doplňková specifikace *(Supplementary Requirements)*. Doplňková specifikace (viz příklad 7) je rozdělena do podkategorií podle typů požadavků, které obsahují vždy název a popis každé požadované funkcionality.

Funkční požadavky (Functionality)

Funkční požadavky jsou definovány v předchozí kapitole pomocí use-case diagramů.

Použitelnost (Usability)

- **Jednoduché instinktivní ovládání**
  Systém nevyžaduje školení uživatelů.

Spolehlivost (Reliability)

- **Dostupnost**
  Systém bude dostupný 24 hodin denně, 7 dní v týdnu.

Výkonnost (Performance)

- **Latence**
  Veškeré úpravy se musí v systému promítnout okamžitě.

Podporovatelnost (Supportability)

- **Uživatelský software**
  Uživatel při práci se systémem bude používat standardní webový prohlížeč. Pro práci se systémem není potřeba instalovat žádný podpůrný program.

Příklad 7: Doplňková specifikace artefaktu Specifikace požadavků

6.2.7 Hodnotící zpráva *(Review Record)*

Jedná se o zprávu, která je vyvíjena po určité aktivitě nebo po několika aktivitách. Reviduje vytvořené artefakty, popisuje, do jaké fáze jednotlivé artefakty dorazily, čeho všeho dosáhly a pokud narazí na problémy, sepíše je a případně i doporučí řešení.

Hodnotící zpráva tedy může nabývat různých typů, podle toho, pro jaké účely je vytvářena. U méně rozsáhlých projektů postačí, pokud bude vytvořena pro kontrolu dosažení milníků fází vývoje. Pro každý milník se tvoří jedna zpráva, která zhodnocuje, zda je potřeba projekt pozastavit, zrušit, přidat další iteraci nebo stanoví fázi za dokončenou a pokračuje se fází následující.

Milník pro fázi zahájení se jmenuje Lifecycle Objectives Milestone (LCO). Jakých stavů je pro tento milník nutné dosáhnout v jednotlivých artefaktech zobrazuje příklad 8.

Typ zprávy

Hodnocení milníku fáze zahájení (Lifecycle Objectives Milestone Review).

Zhodnocení artefaktů

Základní artefakty	**Stav milníku**
Vize	Klíčové vlastnosti projektu a hlavní omezení dokumentována.
Seznam rizik	Počáteční projektová rizika identifikována.
Plán vývoje	Důležité cíle plánu dokončeny a zhodnoceny.
Vývojový proces	Přizpůsobení Rational Unified Procesu definováno a zhodnoceno.
Glosář	Glosář obsahuje veškeré důležité pojmy; glosář zhodnocený.
Specifikace požadavků (Use-Casy)	Důležití aktéři a důležité případy použití byly identifikovány.

Příklad 8: Artefakt Hodnotící zpráva - LCO

Protože hodnotitelé nemají žádné požadavky na změny artefaktů a veškeré požadavky milníku jsou splněny, může se vývoj informačního systému POS přesunout do další fáze, fáze projektování.

6.3 Fáze projektování *(Elaboration phase)*

Během této fáze se požadavky na systém zpodrobňují a zbytečné se odstraňují.
Abychom v našem příkladě dosáhli efektu potřeby upravit artefakty vyvinuté ve fázi zahájení, stanovili jsme si nový požadavek od zákazníka, který zní:

„Rychlé vyhledání zákazníků podle zadaných kritérií“.

Tento požadavek je nutné zanést do již vytvořených artefaktů.

Protože by fáze mohla trvat dlouhou dobu, rozhodl jsem se ji rozdělit na dvě iterace. První iterace provede revizi stávajících dokumentů use-case pohled na architekturu v artefaktu Architektura software *(Software architecture document)*, vytvoření modelu návrhu a analýzu tříd.

Ve druhé iteraci návrhové modely a analýzu tříd zakomponujeme do artefaktu Architektura software a vyvineme a otestujeme fungující prototyp. Prototyp dokáže, že navržená architektura je implementovatelná a bude obsahovat minimálně 10% navržených use-case případů.

6.3.1 Vize (upraveno)

Vzniká nová verze Vize 2.0, ve které je zakomponován nový klíčový požadavek: Vyhledávání zákazníků. Je nutné projít celý dokument vize a pozměnit všechny části tak, aby byly v souladu i s novým požadavkem. V našem případě se nejedná o požadavek, který by zasahoval do architektury produktu nebo do změn obchodního zaměření. Vymezení problémů a produktů tedy zůstává stejné, jako v předchozí verzi dokumentu.

Přehled jednotlivých zainteresovaných osob je ale nutné pozměnit. Vzhledem k činnosti funkčního požadavku je změněn popis pouze pracovníka. Požadavek je také vložen mezi klíčové potřeby uživatelů.

Požadavek je nutné zanést také do požadavků na systém. Ostatní sekce dokumentu ale nijak neovlivní.

Změněné sekce jsou zobrazeny v příkladu 9.

Zainteresované osoby

Přehled uživatelů

Pracovník má možnost prohlížet zkrácené informace o všech zákaznících zanesených v systému. Může do systému nové zákazníky vkládat a vyhledávat již uložené.

Klíčové požadavky / Potřeby uživatelů

Jedním z hlavních požadavků je dostupnost a snadná použitelnost systému. Ten bude proto postaven na webových technologiích, aby byl přístupný z jakéhokoliv počítače připojeného k Internetu. Nutnou potřebou je pak ochrana údajů do systému vložených. Do systému proto nebude připuštěn uživatel, který nemá přístupové údaje.

Další důležitý požadavek je rychlé vyhledání zákazníků a schůzek podle zadaných kritérií.

Vlastnosti systému

Vyhledávání zákazníků

Všichni uživatelé systému mají k dispozici jednoduchý a rychlý vyhledávač zákazníků, který zákazníky vyhledá dle zadaných kritérií.

Příklad 9: Změněné sekce artefaktu Vize

Protože jsme změnili dokument vize, ze kterého vychází dokument Softwarové požadavky, musíme změny reflektovat i do něho.

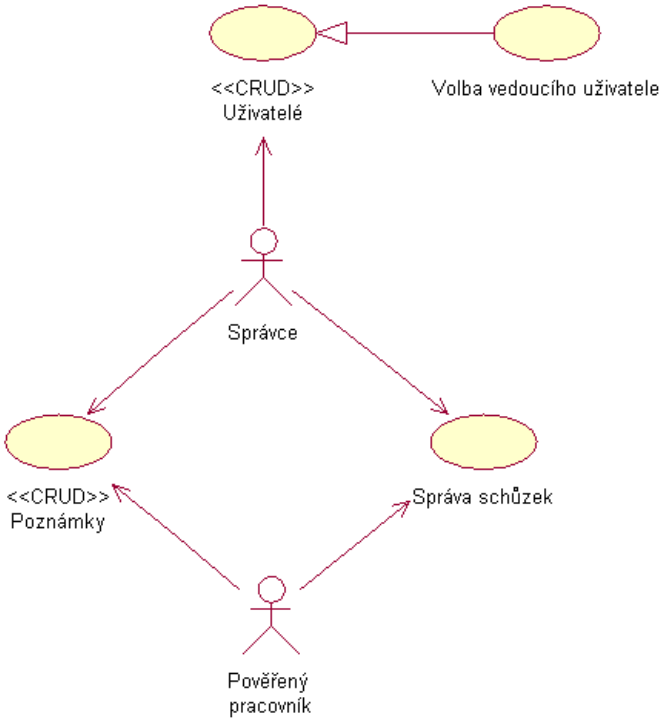
Glosář zůstává v našem případě beze změn.

6.3.2 Softwarové požadavky (upraveno)

Nový požadavek se v tomto artefaktu promítnul pouze na jediném místě, a to v celkovém pohledu na důležité use-casy, které zobrazuje příklad 10. Kromě nich je v příkladu uveden celkový pohled na schůzky, poznámky a uživatele.

Protože tři ze čtyř hlavních entit (uživatelé, poznámky a schůzky) můžeme namodelovat stejným způsobem, vytvořil jsem nový stereotyp CRUD, který jsem rozepsal ve specifikaci. Případ pro přidání uživatele lze také nalézt v příkladu 10.

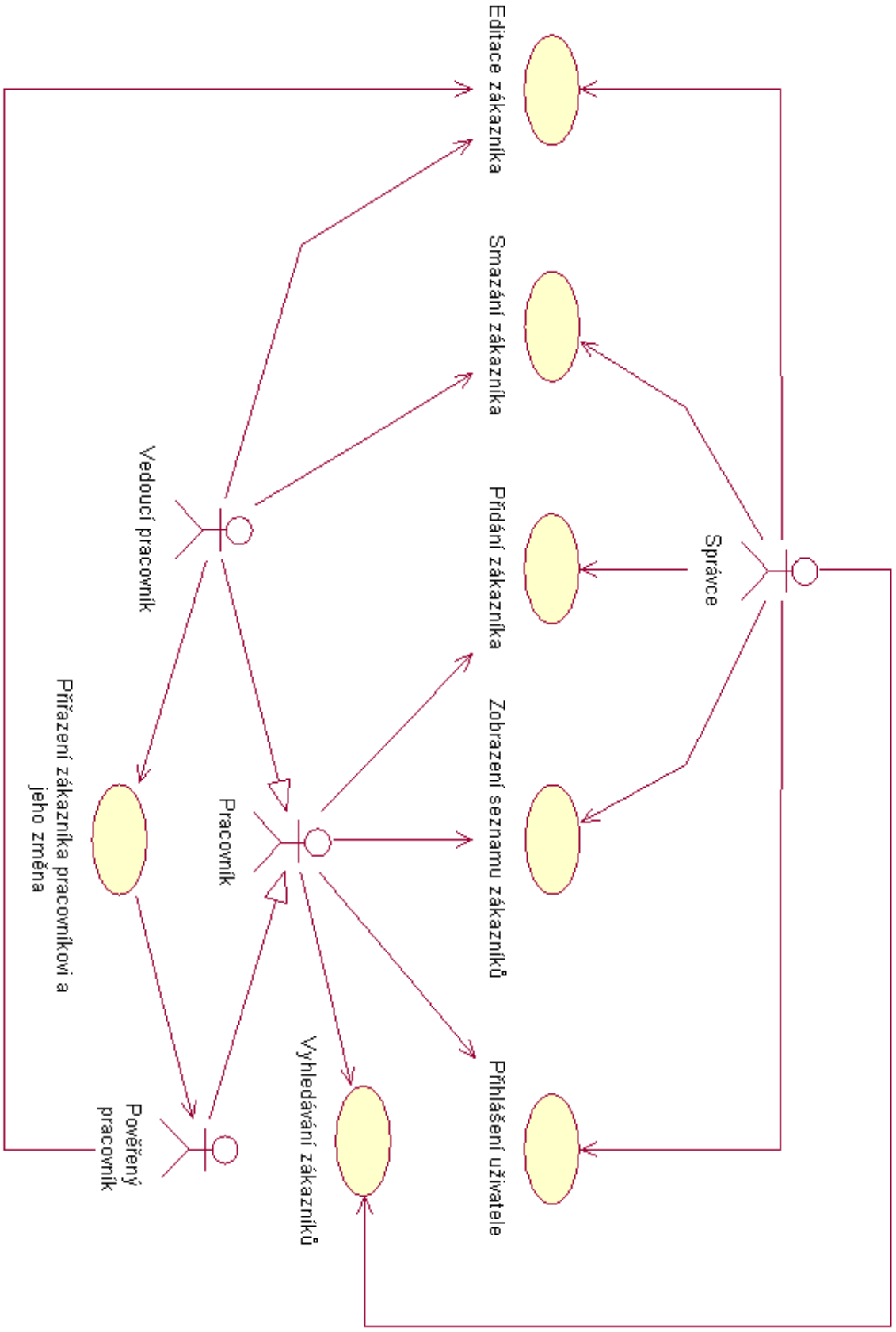
Celkový pohled na schůzky, poznámky a uživatele



Specifikace use-case – Uživatelé <<CRUD>>

Případ použití: Přidání uživatele
ID: 3
Stručný popis: Systém přidá uživatele do systému
Primární aktéři: Správce
Sekundární aktéři: Žádný
Předpoklady: Primární aktér musí být v systému zaevidován a přihlášen.
Hlavní tok: 1. Případ použití se spustí, když primární aktér vybere „přidat uživatele“. 1. Dokud nejsou vyplněny všechny povinné údaje 1.1. Systém požaduje, aby primární aktér zadal povinné údaje – Jméno, příjmení, přihlašovací jméno a heslo. 1.2. Systém ověří, zda jsou zadané všechny povinné údaje. 2. Systém uloží uživatele.
Stav po ukončení: Uživatel byl přidán do systému.
Alternativní toky: 1. Nevyplněné veškeré povinné položky

Celkový pohled na důležité use-casy



Příklad 10: Artefakt Softwarové požadavky - Use-Casy a stereotyp CRUD

6.3.3 Architektura softwaru *(Software architecture document)*

V tomto artefaktu navrhneme architekturu, na které budeme daný projekt stavět.

Architektura se skládá z několika pohledů (viz kapitola 2.4 Architektura systému), ze kterých budeme modelovat pouze některé.

Poslední sekce tohoto dokumentu tvoří zhodnocení výkonnosti a kvality vybrané architektury.

Pohled případu použití

Pro tento pohled vybereme pouze ty případy použití, které jsou architektonicky zajímavé (viz příklad 11). Pokud bychom chtěli systém časem rozšiřovat (což je pravděpodobné), postavíme rozšíření na základních kamenech, které jsou zde popsány.

Základní kameny, tedy architektonicky důležité use-casy, zobrazuje první use-case diagram v příkladu 11. Jedná se o všechny CRUD případy, tedy jeho rozpady na zobrazení, vytvoření, úpravu a smazání daných elementů.

Pro všechny architektonicky důležité use-casy bychom měli vytvořit také diagramy aktivity nebo diagramy komunikací, aby všichni vývojáři implementovali tyto use-casy stejným způsobem podle jednoho vzoru. V našem případě toto zapotřebí není. Pro ukázku jsem ale vytvořil diagram aktivit, který znázorňuje tok řízení mezi jednotlivými aktivitami pro práci se zákazníky. Diagram je uložen v příloze 3 a není obsažen v žádném vyvíjeném artefaktu.

Logický pohled

V tomto pohledu jsem vytvořil konceptuální diagram, ve kterém navrhuji hlavní entity a jejich rozhraní. Konceptuální diagram zobrazuje příklad 13.

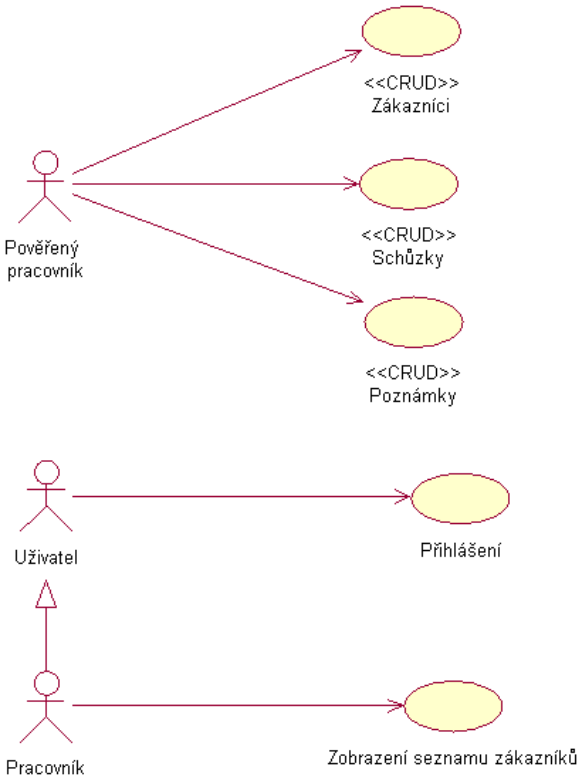
Obecně konceptuální diagram spadá pod artefakt Modely návrhu *(Design Model)*, který ale není třeba vytvářet jako samostatný artefakt. Modely návrhu obsahují ty navrhované modely elementů, které jsou pro danou architekturu důležité. Vysvětluje důležité třídy a zobrazuje jejich organizaci v balíčcích a ve vrstvách.

Datový pohled

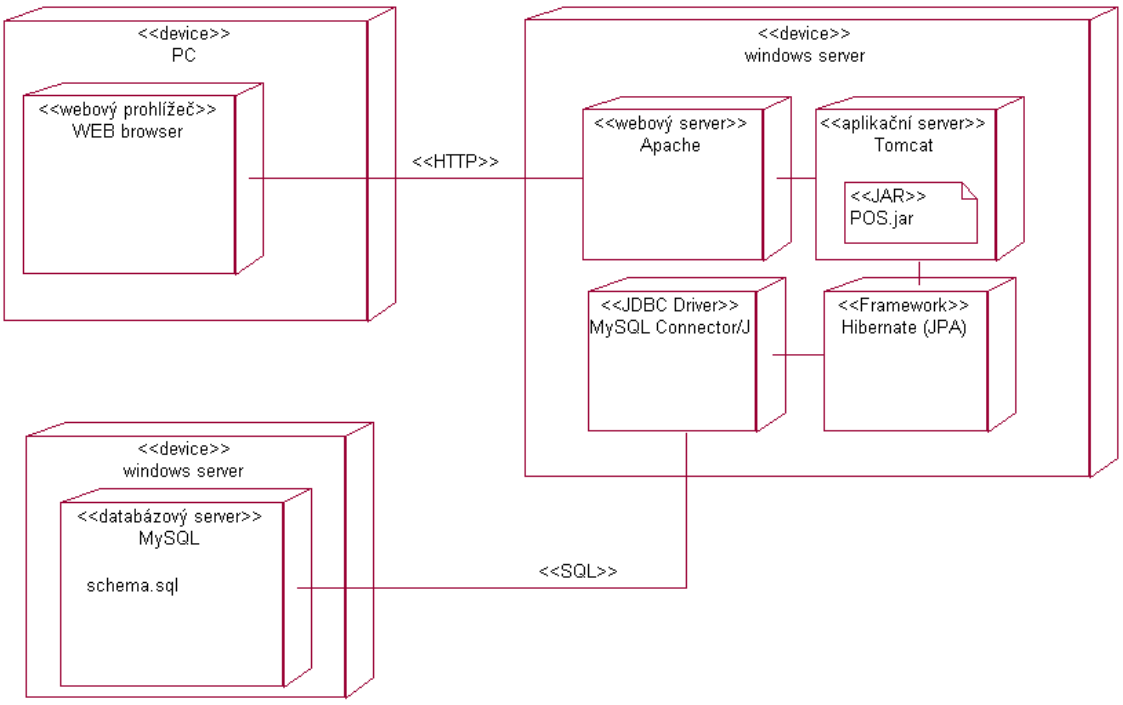
Pro náš příklad postačí pouze pohled na úrovni databázových tabulek a jejich vlastností.

Pohled nasazení

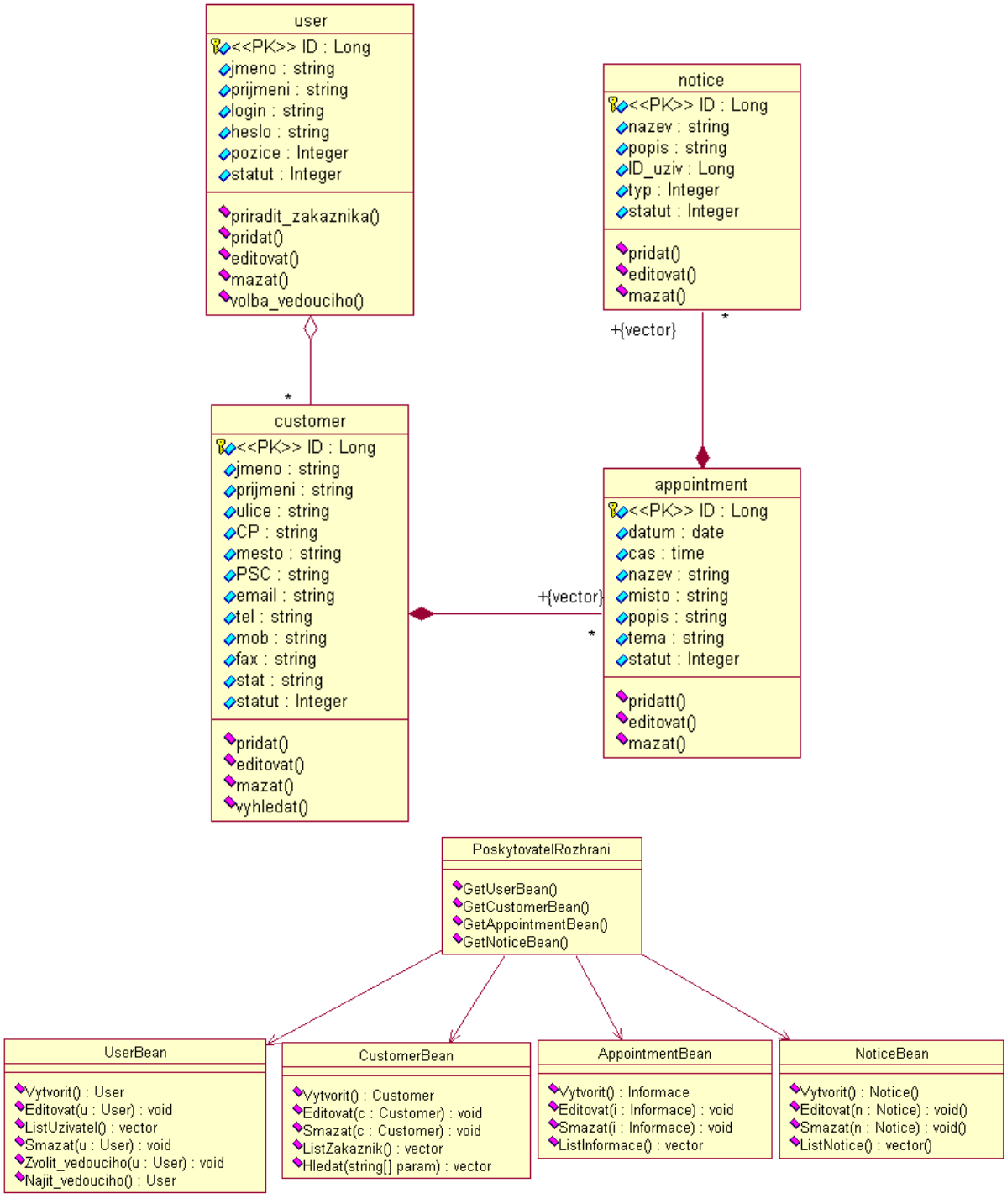
Tímto pohledem popisujeme fyzické komponenty, protokoly, kterými mezi sebou komunikují, a instance komponent nebo objektů, které jsou na nich puštěny (viz příklad 12).



Příklad 11: Artefakt Architektura softwaru - Use-case pohledy



Příklad 12: Artefakt Architektura softwaru - pohled nasazení



Příklad 13: Artefakt Architektura softwaru - Logický pohled

6.3.4 Prototyp Profesionálního organizačního systému

Ve fázi projektování tvoříme první prototyp vyvíjeného systému. Tímto prototypem ověřujeme, že vybraná architektura je implementovatelná tak, jak jsme ji navrhli. Kromě toho obsahuje minimálně 10% případů použití. Další implementace v následujících fázích staví na této kostře systému.

V našem případě jsem použil programovací jazyk Java pro webové technologie (J2EE). Pro generování webových stránek používám nástroj Java Server Pages (JSP) a pro řízení průchodu Java ServerFaces (JSF).

U JSP stránek používám také standardizovanou knihovnu značek JSP Standard Tag Library (JSTL).

Modelem MVC odděluji funkcionalitu aplikace od Modelu a funkcionalitu uživatelského rozhraní do View vrstvy a Controlleru.

Modelová vrstva je tvořena pomocí java beanů (JavaBeans) nesoucí data (viz příklad 14).

```
@Entity
@Table(name = "customer", schema = "pos")
public class Customer
{
// ----------------------    variables ---------------------------------
      @Id
      private long id;
      private String jmeno;
      private String prijmeni;

      @ManyToOne
      private User uzivatel;

      @OneToMany(cascade=ALL, mappedBy = "owner")
      private Collection<Appointment> appointments;

// ----------------------    constructor -------------------------------
      public Customer()
      {
            appointments = new Vector<Appointment>();
      }


// ----------------------    get and set methods -----------------------
      public Collection<Appointment> getAppointments()
      {
            return appointments;
      }
      public void setAppointments(Collection<Appointment> appointments)
      {
            this.appointments = appointments;
      }

      public long getId()
      {
            return id;
      }
      public void setId(long id)
      {
            this.id = id;
      }
```

Příklad 14: Zdrojový kód Customer.java – Modelová vrstva – Java beans

Controller je tvořen pomocí tzv. Backing beanů, které obsahují komponenty zobrazované ve view vrstvě. Backing beans jsou obyčejné Java beans, které řídí běh aplikace a obstarávají přístup k modelu (viz příklad 15).

```
public class CustomerBean
{

// ----------------------    variables ---------------------------------

      private Customer customer;
      private Appointment appointment;
      private Notice notice;
      private HtmlDataTable listTable;

// ----------------------    constructor -------------------------------

      public CustomerBean()
      {
            customer = new Customer();
            appointment = new Appointment();
            notice = new Notice();
      }

// ----------------------    get and set methods ------------------------

      public Customer getCustomer()
      {
            return customer;
      }

      public void setCustomer(Customer customer)
      {
            this.customer = customer;
      }
```

Příklad 15: Zdrojový kód CustomerBean.java – Vrstva Controller – Backing beans

View vrstva je tvořena pomocí JSP a User Interface komponent, které implementuje právě JSF a které jsou pomocí Expression Language vázány na Model a Controller (viz příklad 16).

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">

<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>

<html>
<f:view>
<head>
      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
      <title>Profesionální organizační systém</title>
</head>
<body>
      <h:commandLink  action="#{customerBean.actionNewCustomer}">
            <h:outputText value="Přidat zákazníka" />
```

```
        </h:commandLink>
        <br />

        <h:dataTable value="#{customerBean.customers}" var="customer"
binding="#{customerBean.listTable}" cellspacing="2px" >

            <h:column>
                <f:facet name="header">
                    <h:outputText  value="Jméno" />
                </f:facet>
                <h:outputText value="#{customer.jmeno}" />
            </h:column>
```

Příklad 16: Zdrojový kód welcome.jsp – Vrstva View – JSP a JSF

Framework Hibernate poskytuje mapování objektově orientovaných dat na relační databázi a JDBC ovladač umožňuje komunikaci s MySQL databází.

V příloze 4 můžeme nalézt kompletní zdrojové soubory spolu s java archivem (JAR souborem) a dokumentaci vygenerovanou nástrojem Javadoc, který je obsažena ve vývojovém rozhraní Eclipse. Dále příloha obsahuje soubor schema.sql, který obsahuje SQL příkazy pro vytvoření relační databáze, kterou naplní ukázkovými daty.

6.3.5 Hodnotící zpráva *(Review Record)*

Jedná se o podobný artefakt se stejnou strukturou, jako uvedený v kapitole 6.2.7. Typem tohoto artefaktu je ale milník fáze projektování Lifecycle Architecture Milestone (LCA). Jakých stavů je pro tento milník nutné dosáhnout v jednotlivých artefaktech zobrazuje příklad 17.

Typ zprávy

Hodnocení milníku fáze projektování (Lifecycle Architecture Milestone Review).

Zhodnocení artefaktů

Základní artefakty	Stav milníku
Prototyp	Vytvořen funkční spustitelný prototyp, který zhodnocuje kritické sekce architektury.
Vize	Aktualizována, doplněna o nové informace, které byly zjištěny během fáze. Porozumění důležitým případům použití.
Seznam rizik	Seznam rizik zhodnocen.
Architektura softwaru	Vytvořena. Obsahuje detailní popis architektonicky důležitých částí a identifikaci klíčových mechanizmů. Dále obsahuje návrh jednotlivých částí softwaru (Logický pohled) a definuje Pohled nasazení.
Plán vývoje	Důležité cíle plánu dokončeny a zhodnoceny.
Specifikace požadavků	Specifikace doplněna o nové funkční a nefunkční požadavky a zhodnocena.

Příklad 17: Artefakt Hodnotící zpráva - LCA

Protože hodnotitelé nemají žádné požadavky na změny artefaktů a veškeré požadavky milníku jsou splněny, může se vývoj informačního systému POS přesunout do další fáze, fáze implementace.

Závěr

Protože samotnou tvorbu systémů již pomalu přebírají automatizované nástroje, stanovené postupy a používají se již naprogramované komponenty, je stále více nároků kladeno na přesnou analýzu požadavků a na návrh systému.

V této oblasti je ještě hodně věcí k vylepšování. Nejlepší cestou se ubírají metodiky založené na modelovacím jazyce UML, který je dnes považován za standard návrhu systému, a který lze použít při vývoji systémů o různých rozsazích a s různými zaměřeními.

Nejznámější metodikou je pak Rational Unified Process (RUP), a její volně dostupný standard Unified Process (UP). Od dob zveřejnění UP se ale metodika RUP již natolik změnila a doplnila, že dnes můžeme mluvit přímo o platformě, která obsahuje veškeré důležité postupy, příklady a nástroje nutné pro tvorbu softwarového produktu. Její rozsáhlost a univerzálnost je ale zároveň i její špatnou vlastností, protože je nutné nastudovat spoustu materiálů a neustále kontrolovat správné provádění jednotlivých činností.

Metodika RUP se tak hodí spíše pro větší společnosti zabývající se tvorbou rozsáhlejších systémů, zatímco s metodikou UP si vystačí menší společnosti, pro které vývoj rozsáhlých systémů není hlavní náplní podnikání.

Nově se rozšiřují i další, tzv. agilní metodiky. Tyto metodiky vznikly kvůli poptávce trhu, který vyžaduje vývoj systémů stále v kratším čase. Agilní metodiky jsou ale vzhledem k velmi zkrácenému návrhu použitelné spíše pro menší informační systémy, například pro webové aplikace. Neznámějším zástupcem je Extrémní programování (XP).

Pokud srovnáme RUP a XP, zjistíme, že tyto metodiky stojí na opačných koncích. RUP přesně definuje postupy, role, činnosti a obsahy artefaktů, zatímco XP nutí zákazníka pracovat ve vývojovém týmu, programovat v páru a úplně odbourává vývoj jiných dokumentů, kromě zdrojového kódu. Na druhou stranu jsou obě metodiky velmi pružné a modifikovatelné, takže je lze teoreticky zaměnit. I když tato představa zůstane opravdu pouze u teorie.

V dnešní době již nezáleží pouze na správném objektovém návrhu a přesném dodržování struktury zdrojových kódů. Důležitý je hlavně výběr metodiky pro vývoj systémů. Protože pokud máte skvělý tým profesionálů a budete je nutit používat metodiku, která jim není příjemná, a navíc vyberete metodiku, která se k danému projektu vůbec nehodí, můžete projekt již před začátkem vývoje odepsat. Pokud si ale nějakou metodiku vyberete, je nutné striktně se držet stanovených postupů a požadavků.

Osobním přínosem této práce je shrnutí důležitých částí metodiky RUP a tím vytvořen návod pro vývoj menších až středně rozsáhlých projektů. Použitím zde popsaných postupů a vytvořených artefaktů se výrazně snižuje riziko předčasného zastavení vývoje softwaru. Ve smyslu návodu pro vývoj menších projektů byla práce použita jako zdroj informací pro tvorbu odborného příspěvku Rational Unified Process and Small Projects pro konferenci a soutěž Student EEICT 2008.

Další rozvoj této práce by mohl vést několika různými způsoby Jedním je pokračování vývoje systému POS implementační fází a fází dodání. Další postup by mohl vést k popisu jiných zajímavých, třeba i méně používaných metodik. Velice zajímavým rozšířením by potom bylo sledovat vývoj zde navrhnutého systému pomocí jiných metodik, například pomocí agilní metodiky XP.

Konzultant Vítek Urban z firmy Unicorn a.s. tuto práci zhodnotil kladně ze dvou pohledů. Prvním je praktický příklad využití metodiky RUP pro menší projekty: „Je zde vidět, že i pro takovéto projekty lze RUP využít. Přičemž existuje dost oponentů, kteří tvrdí, že toto nelze a že se

RUP stává v menších projektech neúměrně velkou zátěží.“ Druhým pohledem je využití Javy a Hibernate jako business frameworku ve tří-vrstvé architektuře.

Velmi mě potěšila žádost o svolení využít tuto práci jako dobrý praktický příklad do školení RUP či Hibernate, která firma Unicorn a.s. poskytuje.

Literatura

[1] doc. Ing. Zendulka J., aj.: *Analýza a návrh informačních systémů* [studijní opora], verze 31. 10. 2006, VUT v Brně

[2] Bergström, S., Råberg, L.: *Adopting The Rational Unified Process*: *Succes with the RUP*, II. vydání Boston, Addison-Wesley, 2004, ISBN 0-321-20294-5

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

[4] Eeles, P., Houston, K., Kozaczynski, W., *Building J2EE™ Applications with the Rational Unified Process*, I. vydání Boston, Addison-Wesley, 2002, ISBN10 0-201-79166-8

[5] Aldorf, F.: *Metodika RUP*, diplomová práce, KIT VŠE v Praze, [citace listopad 2007], Dostupný z WWW: <http://objekty.vse.cz/>

[6] *Wikipedie: Otevřená encyklopedie: Iterative and incremental development* [online]. c2008 [citováno 13. 11. 2008].
Dostupný z WWW: <http://en.wikipedia.org/wiki/Iterative_and_incremental_development>

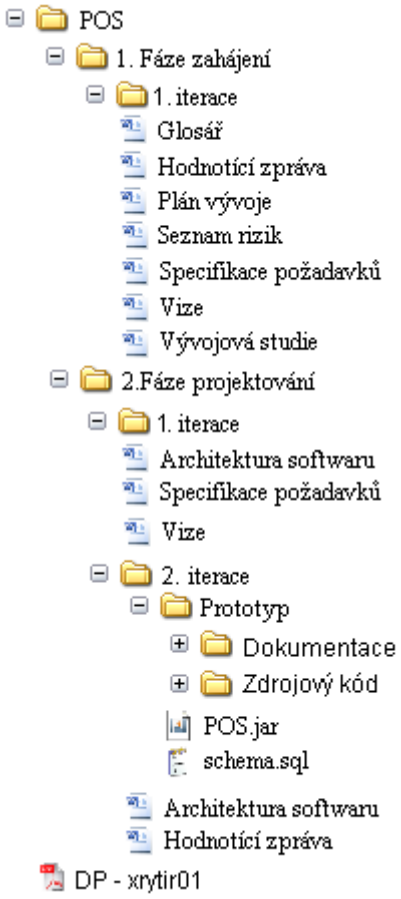
[7] Metodika firmy IBM: *Rational Unified Process*, verze 2003.06.12.01[firemní dokumentace], 2003

[8] Kadlec, V.: *Agilní programování*, I. vydání Brno, Computer Press®, 2004, ISBN 80-251-0342-0

[9] *Wikipedie: Otevřená encyklopedie: Framework* [online]. c2008 [citováno 2. 05. 2008]. Dostupný z WWW: <http://cs.wikipedia.org/w/index.php?title=Framework&oldid=2124221>

Seznam příloh

Příloha 1. Názvosloví
Příloha 2. Seznam příkladů
Příloha 3. Diagram aktivit
Příloha 4. CD (struktura CD je zobrazena na obrázku 12)



Obrázek 12: Struktura CD

Příloha 1. Názvosloví

Agilní programování – inovativní přístup vývoje informačních systémů.

Blacklog – seznam požadavků

Extrémní programování *(Extreme Programming, XP)* – agilní metodika určená pro menší vývojové týmy.

Framework – softwarová struktura, která slouží jako podpora při programování a vývoji a organizaci jiných softwarových projektů. Může obsahovat podpůrné programy, knihovnu API, návrhové vzory nebo doporučené postupy při vývoji [9].

Ganttův diagram – slouží k zobrazení časové náročnosti a posloupnosti jednotlivých částí projektu.

Java Enterprise Edition, Java EE, J2EE - platforma určená pro vývoj přenositelných, robustních a bezpečných serverových aplikací v jazyce Java. Rozšiřuje standardní Java SE (Standard Edition).

Java Persistence Api, JPA – Java EE standard pro zajištění perzistence dat prostřednictvím objektově relačního mapování (ORM). JPA standard implementuje framework Hibernate.

Java ServerFaces, JSF – nadstavba nad Java Servlety, nyní také standard v J2EE.

Java Servlet – nástroj, který obsluhuje http požadavky.

JavaBeans – každá třída v Javě, která splňuje určitá pravidla (prázdný konstruktor, Get-Set metody pro všechny proměnné apod.)

Model – abstraktní pojem reality.

Model – View – Controller, MVC – architektura programování, která aplikaci dělí na tři vrstvy: Model, View a Controller.

Model vývoje software – abstraktní prezentace vývoje software.

Rational Unified Process, RUP – komerční metodika vývoje software s mnoha podpůrnými nástroji, programy a propracovaným pracovním prostředím.

Refaktorizace – změna zdrojového kódu bez změn funkčnosti software.

Revize – kontrola nebo změna dokumentu.

Scrum – agilní metodika podobná metodice XP

Unified Process, UP – unifikovaná metodika vývoje softwaru [3].

Unifikovaný modelovací jazyk *(Unified Modeling Language, UML)* – univerzální jazyk pro vizuální modelování systémů [3].

Životní cyklus vývoje software – časové období softwaru mezi první a poslední vývojovou fází.

Příloha 2. Seznam příkladů

Příloha 3. Diagram aktivit