VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

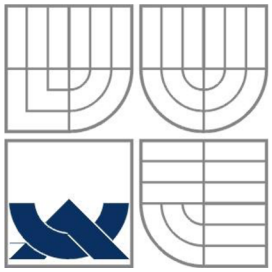
AGILNÍ MODELOVÁNÍ PŘI VÝVOJI SOFTWARE

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

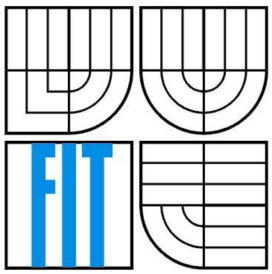
AUTOR PRÁCE
AUTHOR

Bc. MAREK RUPRECHT

BRNO 2011



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

AGILNÍ MODELOVÁNÍ PŘI VÝVOJI SOFTWARE

AGILE MODELLING IN SOFTWARE DEVELOPMENT

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. MAREK RUPRECHT
AUTHOR

VEDOUCÍ PRÁCE — Doc. Ing. JAROSLAV ZENDULKA CSc.
SUPERVISOR

BRNO 2011

Abstrakt

Diplomová práce je věnována procesu vývoje softwarových produktů. A to od jejich počátečního návrhu, přes vlastní realizaci až po předání finálního produktu zákazníkovi. Uvádí čtenáře do problematiky softwarového inženýrství s tím, že se pak detailněji věnuje jedné jeho části - moderním modelům životních cyklů software. Z uvedených modelů byl pro své předností a vhodnost realizace vybrán agilní přístup, jenž je zastoupen agilním vývojem řízeným modelem. Ten byl prověřen nejenom z teoretického, ale i z praktického hlediska a to příkladovou studií. K modelování software jsem použil notaci jazyka UML, jehož stručná charakteristika je součástí této diplomové práce.

Abstract

The thesis is focused on software development process and its products from initial designs through the way of implementation until final delivery to customer. The thesis brings up some basic facts about software engineering with further detailed description of one of its parts, the modern models of software life cycles with focus on the agile life cycle because of its significant benefits and effective implementation. This model is represented by Agile Model Driven Development which has been submitted not only theoretically but in practice. Finally, there is also a short description of Unified Modeling Language which is used as a modeling language.

Klíčová slova

Agilní vývoj řízený modelem, vývoj software, systémové inženýrství, UML, iterace, analýza systému, informační systém pro základní a střední školy.

Keywords

Agile Model Driven Development, software development, system engineering, Unified Modeling Language, iteration, system analysis, information system for basic schools.

Citace

Ruprecht Marek: Agilní modelování při vývoji software, diplomová práce, Brno, FIT VUT v Brně, 2011

Agilní modelování při vývoji software

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením doc. Ing. Jaroslava Zendulky CSc.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Marek Ruprecht
21. května 2011

Poděkování

V této části bych rád poděkoval panu doc. Ing. Jaroslavovi Zendulkovi, CSc. za odborné vedení, výstižné připomínky, užitečné rady, shovívavost, nezbytné konzultace a v neposlední řadě za vstřícnost a podporu při tvorbě zadání.

Obsah

1 Úvod

Díky studiu na Vysokém učení technickém v Brně jsem získal povědomí o tom, že v počítačovém světě existuje technický obor známý jako softwarové inženýrství a měl hrubou představu, co vše tento pojem představuje. Ve firmách, kde jsem sbíral své první pracovní zkušenosti, jsem se s tímto pojmem setkával spíše sporadicky a kdykoliv padlo s vedením firmy slovo na toto téma, tak jsem slýchával obdobné reakce: „My jsme malá firma, my něco takového nepotřebujeme.“, „To jsou zbytečné náklady navíc.“ či „Máme již své know-how a očividně funguje. Tak proč jej měnit?“. Mé pocity tak byly poněkud rozporuplné. Na jednu stranu jsem věděl, že velké projekty a renomované firmy se bez tohoto přístupu neobejdou, nejenom z důvodu podstatného snížení nákladů (časové, finanční, …), ale na stranu druhou mě zaráželo, jsem s ním na vlastní kůži v žádné firmě nesetkal. Možná je to specifikum českého trhu, že vše nové přijímá se skepsí a velmi pomalu, ostatně jako třeba testování software. Vše se ale změnilo v den, kdy jsem poprvé spatřil model systému vytvořený v jazyce UML. Ta preciznost a promyšlenost, se kterou byl model zaznamenán, mě naprosto uchvátila a přesvědčila o využitelnosti tohoto přístupu i v rámci menších firem.

Dané problematice jsem se tedy začal věnovat důkladněji. I přesto, že se v oblasti softwarové inženýrství pohybuji nějaký ten pátek, tak se stále nemohu zbavit pocitu, že jsem v jeho poznávání stále na začátku. Možná má na tom podíl i fakt, že tato disciplína je opravdu rozsáhlá.

Když přede mnou minulý rok vyvstala otázka tématu diplomové práce, tak jsem měl jasno v tom, že jsem chtěl, aby bylo nějakým způsobem provázáno s analýzou. Díky vstřícnosti a laskavosti pana doc. Ing. Jaroslava Zendulky, CSc, kterému tímto ještě jednou děkuji, mi to bylo umožněno a v diplomové práci se tak mohu zabývat agilním modelování při vývoji software. Tento přístup je relativně nový a z mého profesního hlediska zajímavý právě tím, co nového s sebou přináší. Jako cíl diplomové práce jsem si stanovil, že se s touto oblastí důkladně seznámím a ověřím ji v praxi. Díky svým stávajícím zkušenostem tak budu schopný nalézt její slabá místa, případně vyzdvihnout její přínosy. Dříve než tak učiním, tak si nejprve připravím základy, na kterých budu stavět, a uvedu čtenáře do kontextu problematiky související se zadáním.

Druhá kapitola tak slouží jako úvod do softwarového inženýrství, technického oboru, pod který tato práce spadá a je jeho součástí. Vysvětluje důvody jeho vzniku, objasňuje jeho náplň a zmiňuje něco málo z historie.

Moderní modely životních cyklů jsou nedílnou součástí vývoje nejen softwarových projektů, ale i jiných odvětví lidské činnosti. Patří jim třetí kapitola, ve které jsou uvedeni nejznámější představitelé této disciplíny spolu s bližším vysvětlením jejich jednotlivých fází. U každého modelu je uvedena jeho stručná charakteristika spolu s výhodami a nevýhodami, které s sebou nese.

Unifikovaný modelovací jazyk nebo-li UML je silný nástroj, který má před sebou slibnou budoucnost. Ve čtvrté kapitole se čtenář dozví, co vedlo k jeho vzniku, jaká je jeho koncepce a jaké možnosti nabízí.

Stěžejní pro tuto práci je kapitola pátá, která pojednává o agilním modelování. V ní se čtenář seznámí s jeho základními hodnotami, principy a technikami. Nechybí definice agilního modelu a podstata agilního vývoje řízeného modelem (Agile Model Driven Devlopment - dále jen AMDD).

Z roviny teoretické do roviny praktické se dostane čtenář v kapitole šesté, jejímž obsahem je demonstrace nabytých znalostí o agilním modelování na příkladové studii. Tou je vývin informačního systému pro střední a základní školy, který v nich najde uplatnění a zefektivní práci všech zúčastněných.

Závěru je věnována sedmá kapitola, která shrnuje přínosy diplomové práce a celkové shrnutí.

Použitá literatura spolu s použitým zdroji se nachází v kapitole poslední, a to osmé.

2 Softwarové inženýrství

Téma diplomové práce úzce souvisí se softwarovým inženýrstvím, jehož je součástí, a proto si neodpustím pár řádků k této vědní disciplíně. A co, že to vlastně to softwarové inženýrství je?

Univerzálně platná definice tohoto pojmu bohužel neexistuje a ani ji nelze jednoduše vytvořit, neboť každý autor se snaží zdůraznit určitý aspekt tohoto vědního oboru. Nejlépe ten, kterému se sám věnuje a který tím pádem považuje za nejdůležitější. Podle pánů Alaina Abrana a Jamese W. Moora [1] "Je softwarové inženýrství aplikace systematických, disciplinovaných a kvantifikovatelných přístupů s cílem vytvořit, spravovat a udržovat software s tím, že zároveň tyto přístupy studuje a inovuje."

[2] "Pojem softwarového inženýrství se začal příležitostně používat na přelomu 50. a 60. let minulého století." Do povědomí širšího okruhu veřejnosti se dostal až na základě konference NATO uspořádané v roce 1968, která měla za úkol reagovat na tehdejší softwarovou krizi. Ta se vyznačovala neúnosným prodlužováním a s tím souvisejícím prodražováním projektů, nízkou kvalitou programů, nesnadností či dokonce nemožností údržby stávajících softwarových řešení, neefektivitou vývoje, nejistotou výsledku (pro zadavatele zakázky to byl přímo alarmující fakt) a řadou dalších. Příčin softwarové krize bylo hned několik:

- **Špatná komunikace**

  Ať už v rámci kolektivu osob podílejících se na implementaci softwaru nebo mezi zákazníkem a vývojáři.
- **Nesprávné pochopení toho, co vlastně zákazník očekává**

  S výsledným softwarem byl plně spokojen vedoucí vývoje, který si pochvaloval použité technologie a kvalitu softwaru, zatímco zákazník dostal k dispozici těžkopádné řešení, které sice obsahovalo vše, co očekával, ale v rámci jeho organizace byl program v podstatě nepoužitelný.
- **Špatné plánování**

  Nebyly stanoveny žádné milníky, ani hlídány náklady a rozsah prací.
- **Nízká produktivita práce**

  Nikdo nekontroloval, za jak dlouho dobu stihne programátor danou práci vykonat a zda se náhodou nevěnuje jiným činnostem.
- **Neznalost základních pravidel vývoje softwaru**
- **Podcenění hrozeb a rizik**

  Možné hrozby byly sledovány minimálně a místo jejich předcházení se následně řešily jejich nákladné (nejen finančně, ale i časově) následky.
- **Nezvládnuté technologie**

  Nová technologie neznamenala nutně vyřešení neduhů její předchůdkyně. Ba naopak, občas přinesla více škody než užitku.

A právě všem výše uvedeným aspektům, a nejenom jim, se snaží předejít softwarové inženýrství.

Oprostím-li se na chvilku od definic, tak se dá softwarové inženýrství charakterizovat jako inženýrský obor, který v sobě zahrnuje řadu různých disciplín: od informatiky přes matematiku až po management. Poskytuje metodické postupy, nástroje a doporučení pro tvorbu softwarových systémů, které jsou natolik rozsáhlé či složité, že musí být vyvíjeny v týmech.

3 Moderní modely životního cyklu

Životním cyklem softwaru rozumíme proces jeho vývoje od myšlenky na jeho vznik až po samotné uvedení na trh s tím, že každý životní cyklus softwaru je reprezentován použitým modelem. Každý takový životní cyklus můžeme rozdělit do několika dílčích částí/fází, ve kterých definujeme jednotlivé aktivity, milníky a výstupy, které musí být splněny, abychom mohli pokročit dále. A právě v počtu těchto fází a jemnosti dělení, se většina autorů rozchází.

Někteří autoři upřednostňují jemnější dělení, jiní svazují životní cyklus softwaru s metodikou vývoje. V tuto chvíli tak neexistuje jednotný přístup chápání a definice životního cyklu softwaru. I přes tento nejednotný přístup však existuje pětice základních fází životních cyklů, ve kterých se většina autorů shoduje. Liší se pak "pouze" ve významu jednotlivých fází a jejich provázanosti.

I přesto, že modely životního cyklu softwaru definují jednotlivé fáze a jejich interakce, tak nesou pouze informace o tom, co se má dělat, ale ne jak. Konkrétní implementace modelu životního cyklu je plně v rukou organizace/vývojového týmu a je svým způsobem unikátní pro každou organizaci s tím, že organizace se nemusí striktně držet pouze jednoho modelu, ale pro každý projekt vždy vybere ten nejvhodnější a případně jej lehce zmodifikuje. Při vytváření specifik životního cyklu softwaru je nutno brát zřetel především na firemní kulturu dané organizace a specifika konkrétního projektu s tím, že jeho sestavování by se měli účastnit lidé s bohatými zkušenostmi znalí řešení projektů podobného významu.

3.1 Základní fáze modelu životního cyklu

3.1.1 Analýza požadavků

Oprávněně bývá též označována jako jedna z nejtěžších fází s klíčovou důležitostí. Jejím cílem je analýza a specifikace požadavků zadavatele, tedy zákazníka, pro kterého je software vyvíjen a takzvaných stakeholderů, jejichž požadavky jsou velmi rozmanité a mnohdy i protichůdné. Analytik je musí umět skloubit tak, aby výsledné řešení pokrývalo potřeby všech zúčastněných.

[9] "Stakeholdery rozumíme firemní veřejnost, která je působením firmy ovlivňována a která svým chováním zároveň působení firmy ovlivňuje. Patří sem zaměstnanci, akcionáři, investoři, ale i zákazníci, dodavatelé a místní komunity."

Analýza požadavků se sestává ze tří typů aktivit: sběr požadavků (identifikace stakeholderů a komunikace s nimi v přirozeném jazyce, nejlépe ve formě mítinků), vlastní analýza požadavků (identifikace jednotlivých požadavků, určení priorit, studium proveditelnosti) a jejich zaznamenání (nejlépe do dokumentu v běžně dostupném formátu.

Jedná se o dlouhý a mnohdy i svízelný proces, pro který musí mít analytik především nadání a mnohdy i psychologické dovednosti. Výstupem je kompletní seznam všech požadavků kladených na software a to takový, že každý požadavek v něm je proveditelný, testovatelný, měřitelný, dostatečně detailně definovaný, aby nedošlo k jeho nesprávnému pochopení, a musí se vztahovat ke konkrétnímu byznys požadavku.

Rozlišujeme několik základních typů požadavků a to:

- **Funkční požadavky**
  Funkčními požadavky rozumíme funkce, které musí umět software vykonávat. Jedná se o stěžejní bod specifikace.
- **Výkonnostní požadavky**
  Software musí být přístupný velkému množství přihlášených uživatelů a zároveň zpracovávat a vykonávat více požadavků najednou.
- **Požadavky týkající se grafického provedení**
  Software musí být uživatelsky přívětivý a graficky povedený.
- **Odvozené požadavky**
  Požadavky vyplývající z předchozích.

Tato fáze modelu životního cyklu by měla být uzavřena až tehdy, když jsou si zákazník a analytik téměř jisti, že na žádný zákazníkův požadavek nezapomněli a žádné dva požadavky nejdou proti sobě. Objevení nového či nekonzistence stávajících může v pozdějších fázích přinést značné komplikace - počínaje prodražením a zpožděním projektu, až po nemožnost zakomponování nového požadavku do stávajícího řešení, které může mít za následek neúspěšné ukončení projektu.

3.1.2 Návrh systému

Návrh systému je popis struktury softwarového systému, dat (která budou systém tvořit nebo jím budou využívána), rozhranní komponent (tvořící výsledný software), návrh databázového schématu a použitých algoritmů. Bere v úvahu platformu, na níž bude systém implementován, s cílem najít co nejlepší nástroje a přístupy pro dané řešení. Je úzce propojen s analýzou požadavků, ze které čerpá, a na rozdíl od ní nepoužívá pouze přirozený jazyk, ale i jazyk uměle vytvořený pro konkrétní skupiny lidí, například UML.

3.1.3 Implementace

Náplní implementace je především programování. Programátor však stojí před nelehkým úkolem. Dvě předchozí fáze mu jasně vytyčily, co se od softwaru očekává a jak by měl fungovat, ale neříkají mu, jak to má být přepsáno do programovacího jazyk. To už je na jeho úsudku a znalostech.

Z časového hlediska se jedná o nejnáročnější fázi modelu životního cyklu softwaru, ve které jsou navíc i nejčastěji nacházeny chyby. Proto jsou nedílnými součástmi implementace testování a ladění.

Cílem testování softwaru je zajištění a ověření požadované kvality. Ať už se jedná o procházení zdrojových kódů zkušeným programátorem (kontrola čistoty kódu a použitých algoritmů/přístupů) či revize chování a práce se softwarem (white-box a black-box testování).

Laděním softwaru rozumíme aktivity mající za úkol odstranění nalezených chyb, případně aktivity sloužící ke zlepšení výkonnosti systému.

3.1.4 Integrace a nasazení

[2] "Fáze integrace a nasazení znamená sestavení systému z již implementovaných a otestovaných komponent a následné nasazení u zadavatele k využívání." Ověřuje se především správná komunikace mezi jednotlivými komponentami (jejich provázanost), výsledná výkonnost softwaru a jeho bezpečnost.

3.1.5 Provoz a údržba

V této fázi je již software rutinně používán s tím, že dochází už pouze k minimálním úpravám (odstranění chyb, přidání nové/úprava stávající funkčnosti).

3.2 Vodopád

Jedná se o historicky první model životního cyklu softwaru, který byl hojně používán v počátcích softwarového inženýrství, s cílem demonstrovat sílu tohoto oboru a vnést do vývoje systémů jednotný řád, umožnit řešení komplexnějších problémů díky hierarchické dekompozici a snížit množství chyb precizní kontrolou všech výstupů jednotlivých etap. Byl úspěšně použit v řadě velkých projektů, které byly napsány v jazyce Cobol, avšak v dnešní době objektově orientovaného přístupu stojí v ústraní a slouží spíše jako učebnicový příklad.

Význačným rysem tohoto modelu je, že jednotlivé etapy životního cyklu se neprotínají, ale navazují na sebe (dokončená etapa je vstupem etapy následující) s tím, že jsou prováděny dle přesného plánu realizace - analýza požadavků, návrh systému, implementace, integrace a nasazení, provoz a údržba - a zpětně se k nim nevrací.

Mezi výhody tohoto přístup patří na první pohled viditelný pokrok a zavedení pevné struktury řešení projektu. Nevýhodou je nemožnost návratu do předchozí fáze a její úprava.

3.3 Spirála

Patří mezi zástupce iterativních neboli životních cyklů s přírůstky. Podstatou spirály je opakování vývojových kroků/fází životního cyklu tak, že v každém dalším cyklu je rozšířena funkčnost/odstraněna chyba/zefektivněna již hotová část systému. Opakování jednotlivých iterací probíhá tak dlouho, dokud není zákazník spokojen s celkovým výsledkem.

Výhodou spirály je, že zákazník vidí na konci každé iterace hmatatelné výsledky a aktivně se podílí na projektu. Na druhou stranu není možné přesně naplánovat termíny, ceny a prostředky.

3.4 RUP

Pod hlavičkou **R**ational **U**nified **P**rocess nalezneme více než model životního cyklu. Jedná se o metodiku vývoje, která vychází z osvědčených praktik a postupů při vývoji softwaru. Kromě modelu životního cyklu obsahuje rovněž i vývojové prostředí.

Specifikem RUPu je jeho praktická využitelnost pro všechny typy projektů nehledě na jejich velikost s tím, že díky velké univerzálnosti RUPu je vhodné přizpůsobit jeho metodiku specifickým potřebám konkrétní firmy.

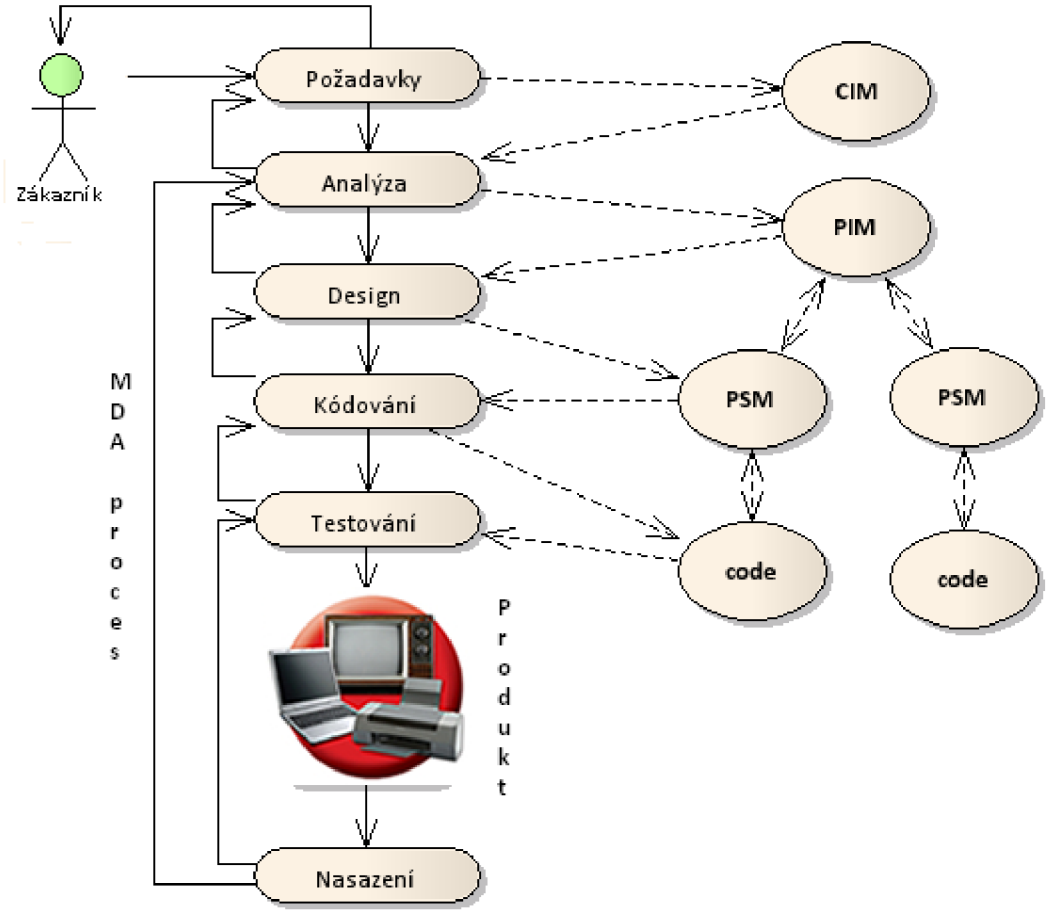
RUP má poněkud nejasnou strukturu. Z velké části odpovídá generickému modelu iterativního životního cyklu s tím rozdílem, že testování je zde pojato jako samostatný celek. To znamená, že není prováděno v každé fázi životního cyklu, ale až v závěru každé iterace a to komplexně.

Detailní a promyšlené zpracování představuje výhodu tohoto přístupu. V jeho neprospěch pak hovoří vysoká cena, jedná se o komerční řešení, a robustnost.

3.5 Architektura řízená modelem

Z anglického **M**odel **D**riven **A**rchitekture. Stěžejní myšlenkou tohoto rámce životního cyklu je vytvoření takového nástroje, který by dokázal na základě specifikace, formální analýzy, vygenerovat zdrojový kód softwaru. Za vznikem stojí konzorcium **O**bject **M**anagment **G**roup, což je organizace, která si klade za cíl standardizaci v oblasti objektově orientovaného přístupu a jí schválené standardy jsou zpravidla akceptovány i jako standardy průmyslové.

Využívá podpory ze strany **U**nified **M**odeling **L**anguage, kterému je věnována následující kapitola, a VHDL. MDA standardizuje a dále definuje způsob transformace jednoho modelu na druhý. [2]: "Předpokládá vývoj software jako posloupnost, nejlépe automatizovaných, transformací z formální specifikace přes etapy návrhu až po proveditelný kód." Každá transformace by navíc měla zajišťovat, případně umožnit verifikaci, že vstup je adekvátní výstupu, viz **Obrázek 1 - Životní cyklus MDA.**



Obrázek 1 - Životní cyklus MDA

MDA rozlišuje následující čtyři modely aplikací:

- **CIM (Computation Independent Model)**

  Model nezávislý na počítačovém zpracování v podstatě představuje vlastní činnosti, business aktivity, které vykonávají pracovníci konkrétní firmy. Je snadno čitelný i pro laika.

- **PIM (Platform Independent Model)**

  Model nezávislý na platformě reprezentuje koncepci řešení dané problémové oblasti na základě požadavků a umožňuje vyřešit určitou oblast na obecné úrovni, a teprve pak zvažovat konkrétní technologii pro vlastní realizaci.

- **PSM (Platform Specific Model)**

  Model řešení na dané platformě, například C++, je podkladem pro vlastní implementaci. Je vytvářen návrháři, ideálně automatizovaně, a má stejnou strukturu jako kód aplikace.

- **Code**

  Zdrojový kód neboli konkrétní realizace na dané platformě.

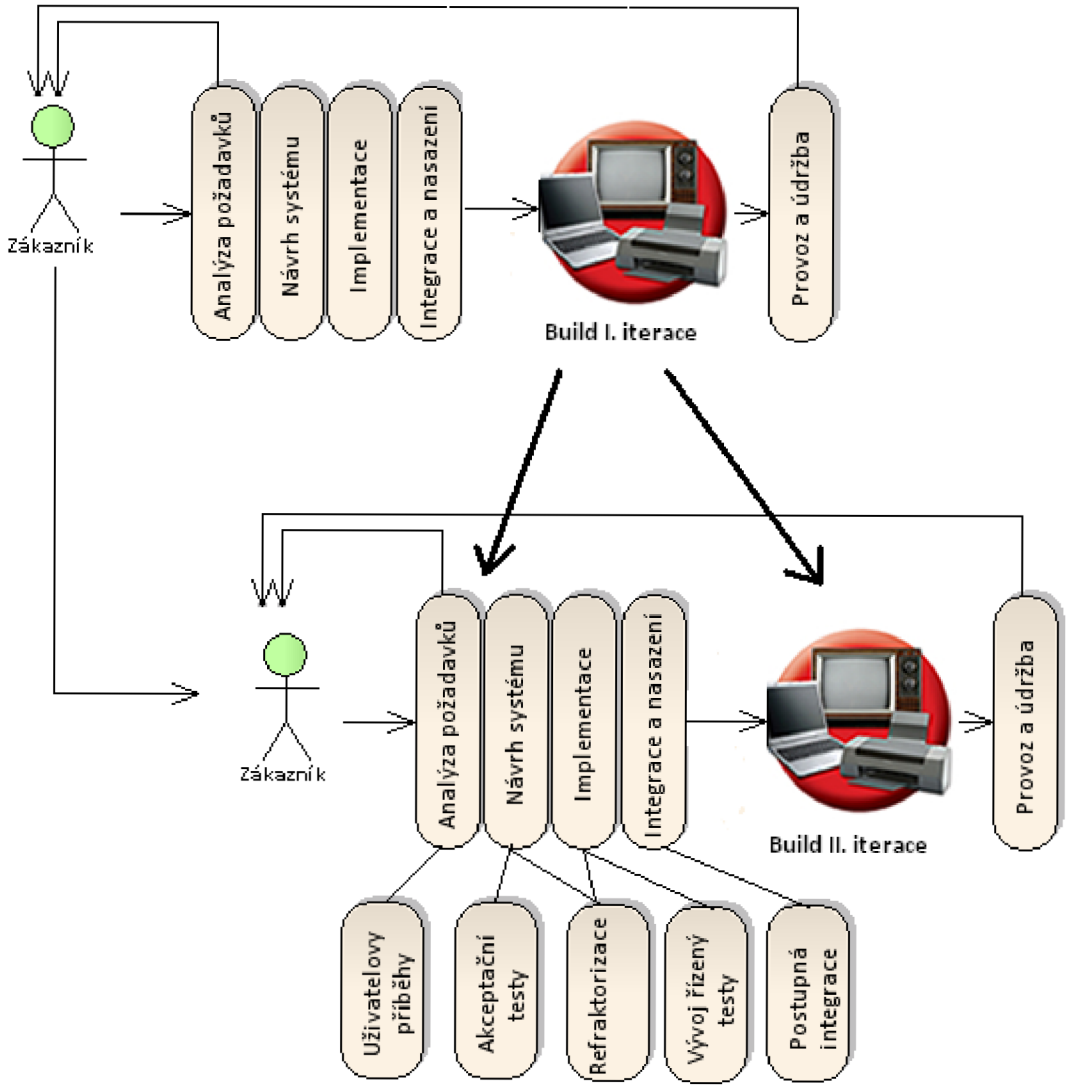
Výhody

- Přinese velké zjednodušení a úsporu prostředků.

Nevýhody

- Jsme teprve na počátku.
- Nutnost CASE nástrojů.

3.6 Agilní vývoj software



Obrázek 2 - Životní cyklus agilního vývoje

Agilní metodiky vývoje se začaly objevovat na konci devadesátých let minulého století a jejich společné rysy byly v roce 2001 organizací Agile Alliance formulovány v takzvaném Manifestu agilního vývoje softwaru. V něm je kladen důraz na tyto aspekty:

- **Průběžná komunikace mezi vývojovým týmem a zákazníkem.**
- **Důraz na kvalitní kód a funkce, které mají přímou obchodní hodnotu pro zákazníka.**
- **Týmová spolupráce včetně samoorganizace týmů.**
- **Průběžné, a pokud možno i co nejčastější, předávání hotové práce.**
- **Kvalita výsledného softwaru má přednost před zbytečným papírováním a termíny.**

Agilní vývoj upřednostňuje jednotlivce a týmy před procesy, nástroji a dokumentací. O to je kreativnější a pružnější než předchozí metody. Během vývoje softwaru je aktivně zapojen i zákazník, který získává zhruba každé dva týdny nový build, který hodnotí. A právě díky této provázanosti se zákazníkem si může agilní vývoj dovolit nevěnovat tvorbě dokumentace takovou důležitost a ušetřený čas věnovat vývoji softwaru.

Nejznámějšími představiteli metodik agilního vývoje software jsou Extrémní programování (Extreme Programming), Vývoj řízený testem (Test Driven Development), Vývoj řízený rysy (Feature Driven Development), Scrum, DSDM a Crystal Clear.

Jak je z obrázku **Obrázek 2 - Životní cyklus agilního vývoje** zřejmé, jedná se o dalšího zástupce z rodiny iterativních životních cyklů, i přesto, že používá jiné termíny. Klasická analýza požadavků je zde nahrazena uživatelovými příběhy (user stories), v nichž zadavatel líčí, co od softwaru očekává. Mohou být popsány krátkými větami či například pomocí případů užití. Následně proběhne ocenění, kolik budou jednotlivé požadavky stát a jak dlouho potrvá jejich vývoj. Na základě priorit pak zadavatel vytvoří harmonogram iterací, který říká, jaká funkčnost bude v které iteraci obsažena. Tradiční návrh systému je zde nahrazen kombinací akceptačních testů (validační testy sloužící ke kontrole, zda zákazník obdržel takové řešení, které očekával) a refraktorizací (vylepšení struktury systému). Akceptační testy jsou vytvořeny ještě před vlastní implementací testované funkčnosti. Tuto skutečnost označujeme jako vývoj řízený testem (test-driven development) nebo též programování s úmyslem (intentional programming).

Buildy jsou zpravidla vytvářeny každé dva týdny s tím, že po šesti iteracích přichází důkladnější revize odvedené práce.

Výhody

- Aktivní zapojení zadavatele sebou nese výhodu v tom, že vyvíjený software reflektuje jeho požadavky.
- Krátké iterace umožňují průběžnou kontrolu.

Nevýhody

- Jsme teprve na počátku, nevíme, jak se osvědčí.
- Netradiční přístup může mnohé zadavatele odradit. Například programování v páru může mnohým zákazníkům připadat finančně náročnější než běžnější přístupy.

4 UML

Unified **M**odeling **L**anguage, neboli unifikovaný modelovací jazyk, je univerzálním modelovacím jazykem pro vizuální modelování systémů. Slouží k jejich specifikaci, vizualizaci, modifikaci, konstrukci i dokumentaci. I přesto, že vychází z podstaty objektově orientovaných systémů, tak není zaměřen pouze na ně, ale má mnohem širší uplatnění. Jednou z jeho hlavních myšlenek je spojit nejlepší existující postupy modelovacích technik a softwarového inženýrství. Koncept UML je ve své podstatě adaptivní. Vždy, když se objeví nějaká kvalitní teze, kterou by mohlo UML přijmout za svou, tak se tak stane. Naopak od neosvědčených postupů se dokáže velmi rychle oprostit. Je navržen tak, aby jej mohly implementovat všechny nástroje CASE (computer-aided software engineering), bez kterých se moderní, většinou velmi rozsáhlé, softwarové systémy neobejdou, ale zároveň zůstává snadno srozumitelným pro lidi. Lze jej použít v libovolné fázi kteréhokoliv modelu životního cyklu softwaru nehledě na použitou implementační technologii. Tento fakt z něj dělá mocný a univerzální nástroj.

UML nenabízí žádný druh metodiky modelování, pouze poskytuje vizuální syntaxi, kterou lze využít při vytváření modelů. Není tak vázán na žádnou specifickou metodiku či životní cyklus. Existuje však metoda, kterou jazyk UML upřednostňuje a pro kterou je nejlépe přizpůsoben, a to metodika Unified Process, zmíněná v kapitole **3.4 RUP**.

4.1 Historie

Do roku 1994 existovalo několik soupeřících jazyků pro vizuální modelování a k nim několik metodik. Svět objektově orientovaných metod se tak zmítal v chaosu, protože žádný z přístupů nezískal tak velkou uživatelskou základnu, která by mu umožnila masivnější prosazení na trhu a diktování směru vývoje. Svůj podíl na tom měl i fakt, že každá metodika s sebou sice přinesla něco nového, obvykle v ruku v ruce s novou notací, ale rozmanité inovace jen výjimečně znamenaly novou kvalitu či přístup.

V této době si více než polovinu tehdejšího trhu uzmuly Boochova a Rumbaughova objektová modelovací technika (Object Modeling Technique) a na poli metodik vedla s převahou Jacobsonova Objectory. A právě z výše uvedených metod a objektově orientovaného softwarového inženýrství (OOSE) UML čerpá a sjednocuje je v jednu koncepci. Základy UML položili právě pánové Booch, Rumbaugh a Jacobson pod hlavičkou společnosti Rational Corporation. V roce 1997 sdružení Object Management Group (OMG) jazyk UML přijalo jako první otevřený průmyslový standard objektově orientovaného jazyka pro vizuální modelování.

OMG přijalo v roce 2005 standard UML 2.0, ve kterém byla provedena důsledná revize předchozích verzí a jejich nedostatky odstraněny. Navíc s sebou přináší verze 2.0 rozšíření o nové

prvky vizuální syntaxe s tím, že základní pravidla z předchozích verzí jsou víceméně zachována. V současné době je na trhu verze UML 2.2.

4.2 Proč unifikovaný?

Slovník spisovné češtiny uvádí jako význam slova unifikace sjednocení. A právě sjednocování je jednou z hlavních charakteristik UML, mimo jiné v těchto doménách:

- **Vývojové cykly**
  UML dokáže pomocí svých modelů pokrýt celý vývojový cyklus. Analýzou požadavků počínaje a kontrolou průběhu implementace konče.
- **Aplikační domény**
  Jazyk UML byl vytvořen pro modelování čehokoliv. Od byznys procesů po real-time systémy.
- **Implementační jazyky a platformy**
  Pro UML nehraje roli, v jakém programovacím jazyce bude modelovaný software implementován. Dokáže pokrýt potřeby čistě objektově orientovaných programovacích jazyků (C#), hybridních objektově orientovaných programovacích jazyků (C++), ale i jazyků neobjektových (C).
- **Vývojové procesy**
  I přesto, že UML upřednostňuje metodiku UP, tak podporuje mnoho dalších osnov procesu tvorby softwaru.

4.3 Modelování v UML

UML chápe modelovaný systém jako kolekci spolupracujících objektů, soudružné seskupení dat a funkcí, na které nahlíží ze dvou pohledů s tím, že jeden pohled bez druhého není úplný.

- **Statický pohled – struktura objektů**
  Definuje struktury jednotlivých objektů spolu s jejich atributy, operacemi a vazbami na ostatní - v jakém jsou k nim vztahu. Bývá popsán diagramem tříd a objektovým diagramem.
- **Dynamický pohled – chování objektů**
  Popisuje životní cykly objektů a způsob jejich vzájemné spolupráce a chování. Jedná se především o sekvenční diagram, diagram aktivit a stavový diagram.

4.4 Struktura jazyka UML

Zajímavostí jazyka UML je fakt, že on sám byl modelován a navržen v UML, což znamená, že je sám navrhovaným a sestavovaným systémem.

4.4.1 Stavební bloky jazyka UML

Předměty

Elementární prvky modelu, které též nazýváme věcmi či abstrakcí. Rozlišujeme čtyři druhy předmětů a to:

- **Předměty popisující strukturu**
  Podstatná jména zastupující třídy, rozhraní, komponenty, uzly, …
- **Předměty popisující chování**
  Slovesa reprezentující interakce, stavy, …
- **Předměty popisující seskupení**
  Balíčky sloužící k seskupování sémanticky souvisejících prvků do vyšších celků.
- **Poznámky**

Relace

Definuje vazbu mezi dvěma předměty a zachycuje tak sémantický/významový vztah mezi nimi. Vztahuje se na strukturní abstrakce a seskupování. Základními typy relací jsou:

- **Asociace (association)**
  Popis množiny spojení mezi objekty/třídami.
- **Závislost (dependency)**
  Změna u jednoho elementu (řídícího) ovlivňuje význam druhého (závislého).
- **Zobecnění (generalization)**
  Jeden element je specializací jiného elementu a lze jej nahradit obecnějším (univerzálnějším) elementem.
- **Realizace (realization)**
  Relace mezi klasifikátory, kdy jeden klasifikátor určuje dohodu, jejíž uskutečnění zaručuje druhý klasifikátor.
- **Agregace (aggregation)**
  Cílový prvek je součástí zdrojového prvku a může existovat i bez něj.
- **Kompozice (composition)**
  Je silnější formou agregace (má více omezení) a cílový prvek zaniká spolu se zdrojovým prvkem (nemůže existovat sám o sobě).
- **Ochranná nádoba (containment)**
  Zdrojový prvek obsahuje cílový prvek.

Diagramy

[3] "Model je repozitářem všech předmětů a relací vytvořených k tomu, aby popisovaly požadované chování systému, který se snažíme navrhnout." A právě diagramy slouží jako pohledy/náhledy

na model. Odstranění předmětu či relace z diagramu neznamená jeho odstranění z modelu! V modelu i nadále zůstává (a to i v tom případě, že není uveden v žádném diagramu), a proto na něj musí být brán zřetel/ohledy.

Modely tak mohou být:

- **Proškrtané**

  Předměty či relace jsou sice uvedeny v modelu, ale v konkrétním diagramu jsou skryty - pro jednodušší pohled.

- **Neúplné**

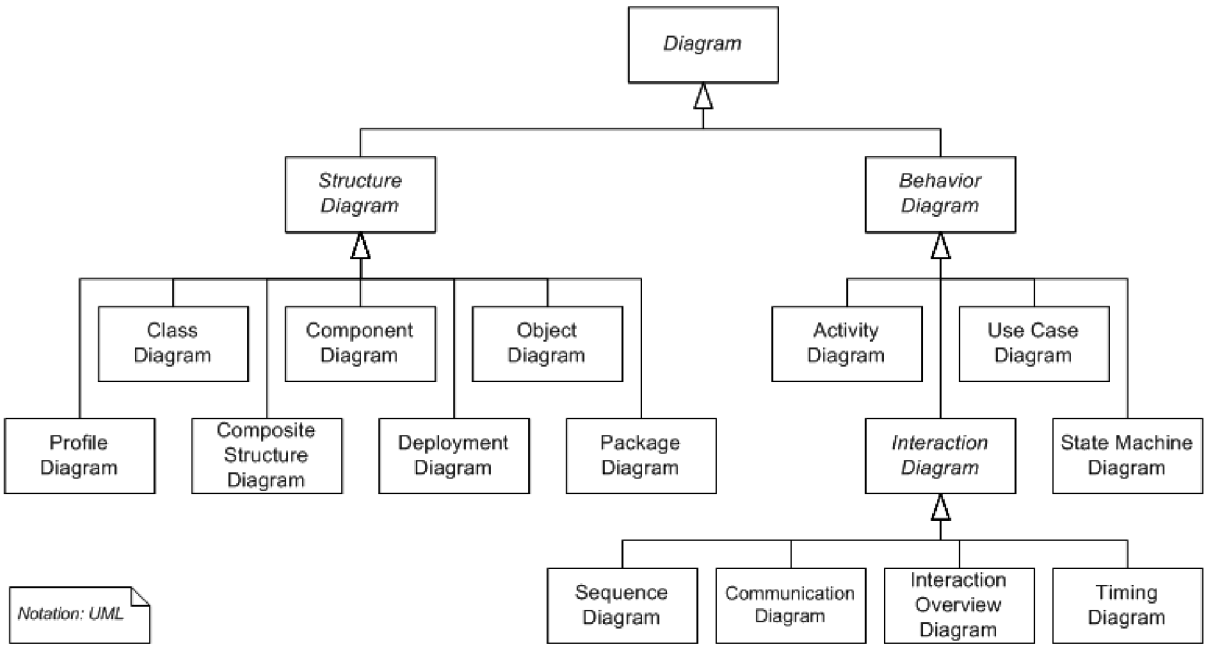
  Některá důležitá fakta nejsou v modelu uvedena vůbec (chybí v něm).

- **Nekonzistentní**

  Některé údaje/fakta si v modelu protiřečí.

I přesto, že UML poskytuje velkou dávku flexibility, tak cílem analytika je, aby model byl konzistentní a natolik úplný, aby umožnil hladkou tvorbu modelovaného softwaru/systému.

Diagramy nejsou prostým pohledem na model, ale též prvořadým a základním mechanismem pro zadávání nových informací do existujícího/stávajícího modelu. UML verze 2.2 rozlišuje 14 druhů diagramů a to, viz **Obrázek 3 - UML diagramy (převzato z [4]:**



Obrázek 3 - UML diagramy (převzato z [4])

Diagramy struktury systému popisují předměty a strukturální relace mezi nimi, zatímco diagramy chování definují, jak na sebe jednotlivé předměty působí. Při vytváření diagramů nejsme limitováni pevným pořadím jejich skladby. Většinou pracujeme se všemi najednou, díky interakci mezi nimi. Jako výchozí se však doporučuje zvolit diagram případu užití.

Diagram případu užití (Use Case Diagram)

Diagram případů užití je diagram, jenž popisuje sekvenci akcí, které mají měřitelnou hodnotu

pro aktéra, který je spustil. Poskytuje široký pohled na všechny, případně část, funkčních požadavků a je vhodný i pro stanovení hranic iterace.

Sestává se ze čtyř komponent, viz **Obrázek 4 - Diagram případů užití**:

- **Hranice systému**

  Ohraničení případů užití sloužící k vyznačení území nebo hranic modelovaného systému.

- **Aktéři**

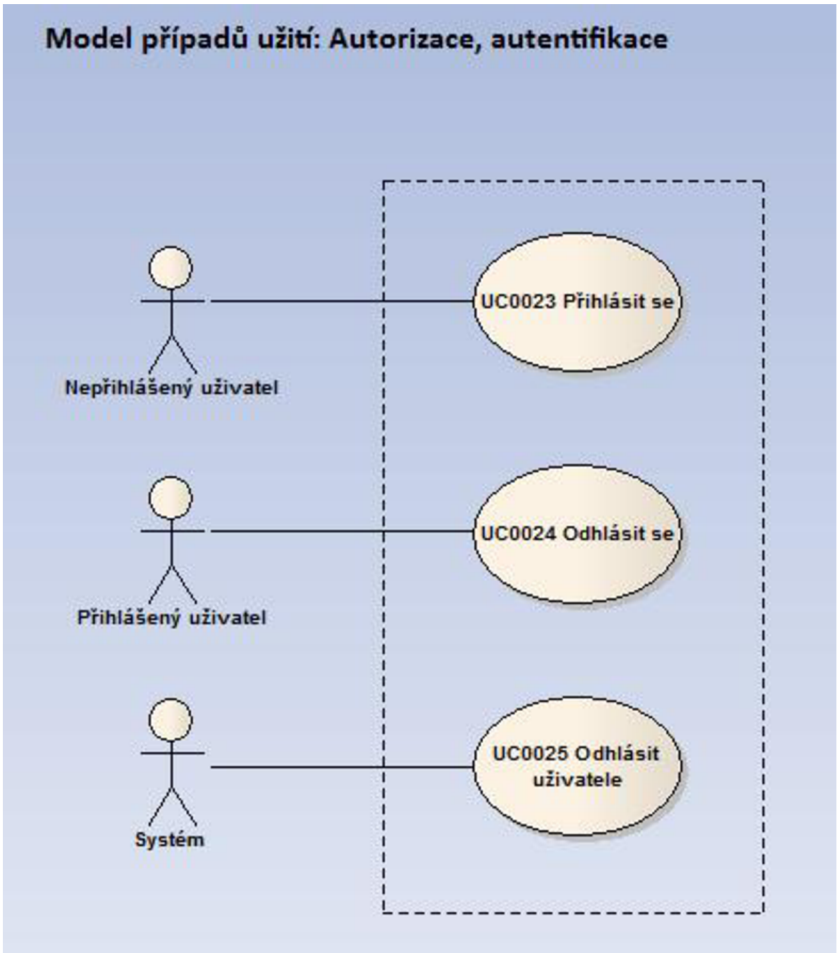
  Aktérem je osoba, organizace či externí systém, který má přidělenu nejméně jednu roli v rámci vyvíjeného systému. Většinou je zakreslen jako postava, případně jako objekt (tato varianta se volí pro „neživé" aktéry typu systém).

- **Případy užití**

  Činnosti, sekvence akcí, které mohou jednotliví aktéři v systému vykonávat a které jim přináší přidanou hodnotu.

- **Relace**

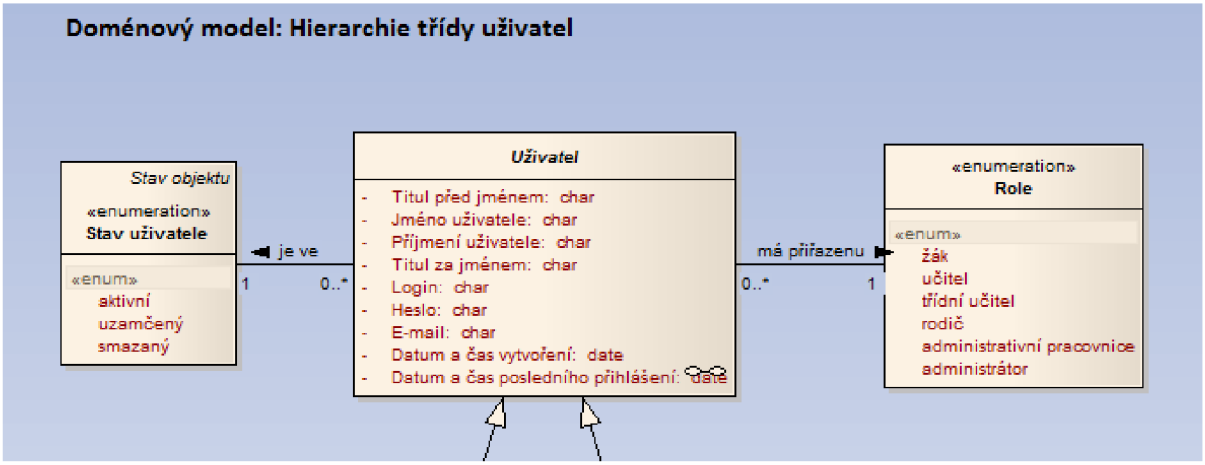
  Existuje celá řada vztahů, které se mohou objevit v diagramu případů užití. A to asociace mezi aktérem a případem užití, asociace mezi dvěma případy užití, generalizace mezi dvěma aktéry či mezi dvěma případy užití.



Obrázek 4 - Diagram případů užití

Diagram tříd (Class diagram)

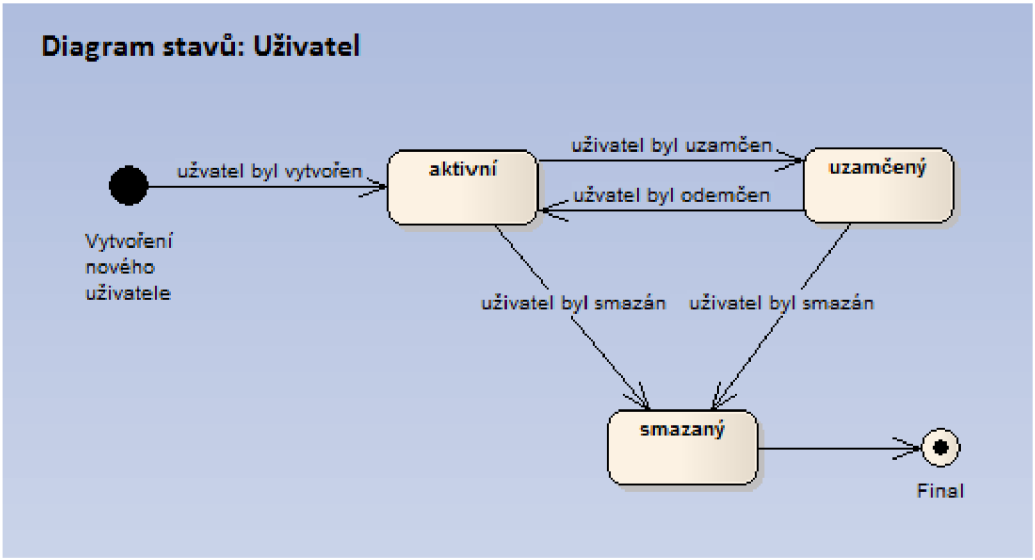
Diagram tříd, **Obrázek 5 - Diagram tříd**, je pilířem objektově orientovaných analýz a návrhů systému. Zachycuje systémové třídy, jejich atributy a operace, a vazby mezi sebou navzájem (dědičnost, agregace a asociace). Má široké uplatnění, od konceptuálního/doménového modelování až po detailní návrhy.



Obrázek 5 - Diagram tříd

Diagram stavů (State Machine Diagram)

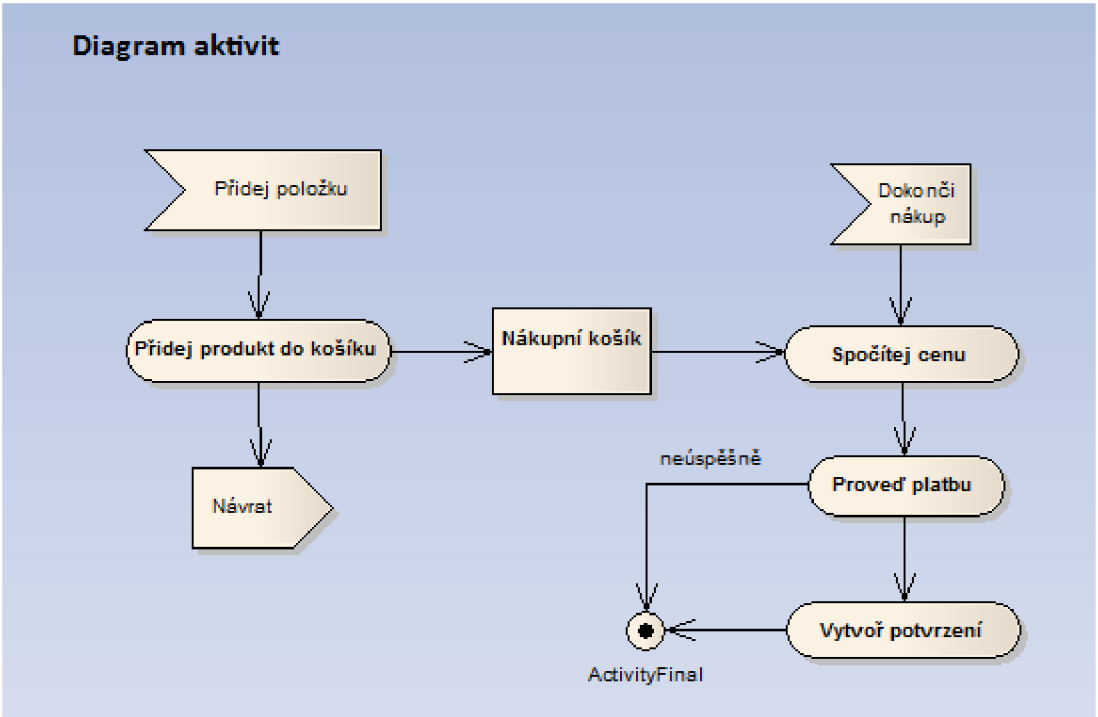
Některé objekty jsou natolik komplikované, že je nezbytné zachytit stavy, ve kterých se mohou nacházet. UML diagram stavů, **Obrázek 6 - Diagram stavů,** zachycuje různé stavy objektu a přechod mezi nimi. Existují dva speciální stavy objektu, počáteční a konečný stav. V počátečním stavu se nachází objekt v moment, kdy je vytvořen. Oproti tomu konečný stav je takový stav, ze kterého nevede žádný přechod a je tak nemožné se z něj dostat.



Obrázek 6 - Diagram stavů

Diagram aktivit (Activity Diagram)

Diagramy aktivit, jak již z názvu vyplývá, zachycují aktivitu a používají se pro modelování byznys procesů a jejich workflov. Jeho základními stavebními bloky jsou:



Obrázek 7 - Diagram aktivit

- **Aktivita**

  Základní element, nerozlišuje se, zda jsou elektronického či fyzického rázu.

- **Akce**

  Akce je menší jednotkou než aktivita a většinou je její součástí. Jedna aktivita může zastřešovat/obsahovat více akcí.

- **Hrana**

  Šipka v diagramu určující posloupnost aktivit.

- **Rozvětvení**

  Označuje počátek paralelních aktivit. Element s jedním vstupem a vícero výstupy.

- **Spojení**

  Ukončuje paralelní běh. Element s vícero vstupy a jedním výstupem. Zajímavostí je, že všechny aktivity směrující do spojení musí být splněny/naplněny, jinak není možné pokračovat.

- **Podmínka**

  Text definující podmínku, která musí být splněna, nabýt hodnoty „pravda“. Jinak dojde k přerušení toku aktivit.

- **Rozhodnutí**

„Diamant“ s jedním vstupem a mnoha výstupy. Vystupující hrany jsou většinou opatřeny podmínkou, která musí být splněna. Text podmínky není povinný, pokud je zřejmé, o jakou situaci se jedná.

- **Sloučení**

  „Diamant“ s několika vstupy a jedním výstupem. Není tak striktní jako spojení, to znamená, že ne přímo všechny vstupní aktivity musí být splněny, aby běh pokračoval dále.

- **Vnořená aktivita**

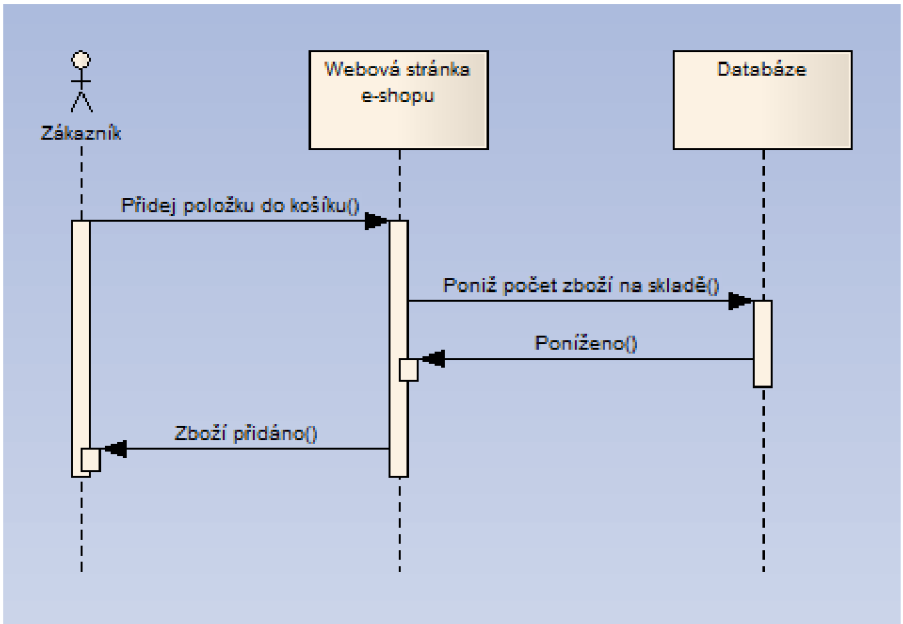
  Volání jiných procesů, tyto aktivity mohou být zastoupeny dalším diagramem aktivit.

- **Počáteční a koncový uzel**

  Viz diagram stavů v předchozím odstavci.

Sekvenční diagram (Sequence diagram)

Sekvenční diagramy jsou nejpopulárnějším nástrojem UML pro dynamické modelování, které je zaměřeno na identifikaci chování uvnitř systému - zachycuje logiku systému. Říká se, že spolu s diagramem tříd a případů užití tvoří základ moderního modelování.



Obrázek 8 - Sekvenční diagram

Sekvenční diagram nachází své uplatnění především v těchto oblastech:

- **Uživatelské scénáře**

  Nebo-li posloupnost případů užití. Více informací o uživatelských je v kapitole **5.3.2Specifické iterace AMDD.**

- **Logika metod**

  Sekvenční diagram lze použít ke komplexnímu prověření operací, procedur a funkcí systému. Ukazuje jejich provázanost a návaznost.

- **Logika služeb**

Službou rozumíme metodu s velkou mírou abstrakce, která může být vyvolána širokým spektrem klientů. Zastupuje jako webové služby, tak i byznys procesy uvnitř systému.

4.4.2 Obecné mechanismy jazyka UML

UML obsahuje čtyři společné mechanismy popisující čtyři různé strategie cesty k modelování objektů, jež jsou opakovaně používány v různých kontextech v celém jazyce UML.

Specifikace

Každý model v UML má alespoň dva rozměry – grafický a textový. Grafický rozměr zprostředkovává vizualizaci modelu prostřednictvím diagramů a symbolů (ikon), zatímco textový obsahuje specifikaci různých prvků modelu. Specifikace nám tedy umožňuje zachytit sémantiku modelů, respektive nám slovně objasňuje význam jednotlivých prvků.

[3] "Množina specifikací je jádrem modelu a utváří sémantický podklad, který udržuje celý model pohromadě a dává mu smysl. Různé diagramy jsou různými pohledy nebo obrazovými projekcemi tohoto podkladu."

Zadávání, prohlížení a úpravy specifikace jednotlivých prvků modelu provádíme pomocí CASE nástroje s tím, že specifikace je nezbytou součástí modelu.

Podrobnosti neboli ornamenty

Slouží pro zdůraznění či zvýraznění určitých důležitých detailů. Každý prvek modelu je vyjádřen velmi jednoduchým symbolem, který lze později, nejlépe průběžně, obohatit řadou ornamentů a vést tak u něj podrobnější informace. S ornamenty bychom však měli nakládat opatrně a používat je pouze v opodstatněných případech a s rozumem, abychom srozumitelnost modelu spíše nezhoršili. Někdy méně znamená více.

Podskupiny

Popisují různé vidění světa a to následovně.

- **Klasifikátor a instance**

  Klasifikátor je abstraktním vyjádřením předmětu/objektu, zatímco instance je jeho konkrétní zastoupení/provedení. Příkladem z reálného světa jsou například dům (množina domů) versus dům, ve kterém bydlím (instance).

  Instance i klasifikátory jsou v jazyce UML zastoupeny stejným symbolem s tím rozdílem, že názvy instancí jsou podtrženy. V současné verzi UML je zastoupeno třicet tři různých klasifikátorů (aktér, třída, komponenta, rozhraní, uzel,…).

- **Rozhranní a implementace**

  Rozhranní nese informaci o tom, co předmět vykonává, oproti implementaci, jež naopak definuje, jak to vykonává. Jako příklad mohu uvést tlačítka pro zapnutí a nastavení klimatizace (rozhranní)

a vlastní fungování klimatizace (implementace), kdy nás „nezajímá“ jak to, že je nakonec v místnosti požadovaná teplota. Rozhranní nás tedy vlastně oprošťuje od toho, jak daný předmět funguje.

Mechanismy rozšiřitelnosti

I přesto, že se jazyk UML snaží být naprosto univerzálním modelovacím jazykem, který by pokryl potřeby všech svých uživatelů, tak neobsahuje přímou podporu pro vyjádření sémantiky některých netriviálních faktů. Z tohoto důvodu jsou v něm obsaženy tři jednoduché mechanismy, které překlenují propast mezi všeobecně známými standardními elementy jazyka UML a snahou přiměřeným způsobem zohlednit ve vytvářeném modelu charakteristické a specifické aspekty problémové domény.

- **Omezení**

  Pro zápis omezení byl definován jazyk OCL (Object Constraint Language), který se však v praxi příliš neujal. Proto je za omezující podmínku standardně považován textový řetězec uzavřený do složených závorek. Omezení se vztahuje k určitému prvku modelu a musí být v daném kontextu vždy pravdivé. Definují tak prvkům v modelu nová pravidla, která musí být splněna, například {stav_uctu >= odecitana_hodnota}.

- **Stereotypy**

  Vycházejí z existujících elementů jazyka UML a mění/upravují jejich význam tak, že se přidá dvojice lomených závorek s názvem stereotypu k původnímu elementu <<Název stereotypu>>. Zužují význam původního elementu nebo kompletně mění jeho význam v diagramech. Existují i stereotypy měnící radikálně sémantiku elementu. Stereotypy se změněným grafickým symbolem můžeme nelézt třeba v diagramu nasazení, kdy fyzické uzly systému reprezentují symboly počítačů, laptopů a tiskáren, namísto neforemných “kvádrů“. Každý stereotyp je nutno důsledně dokumentovat. Záleží na samotném uživateli, zda dá přednost poznámce v diagramu nebo ke každému projektu založí samostatný dokument s vysvětlivkami pro používané stereotypy.

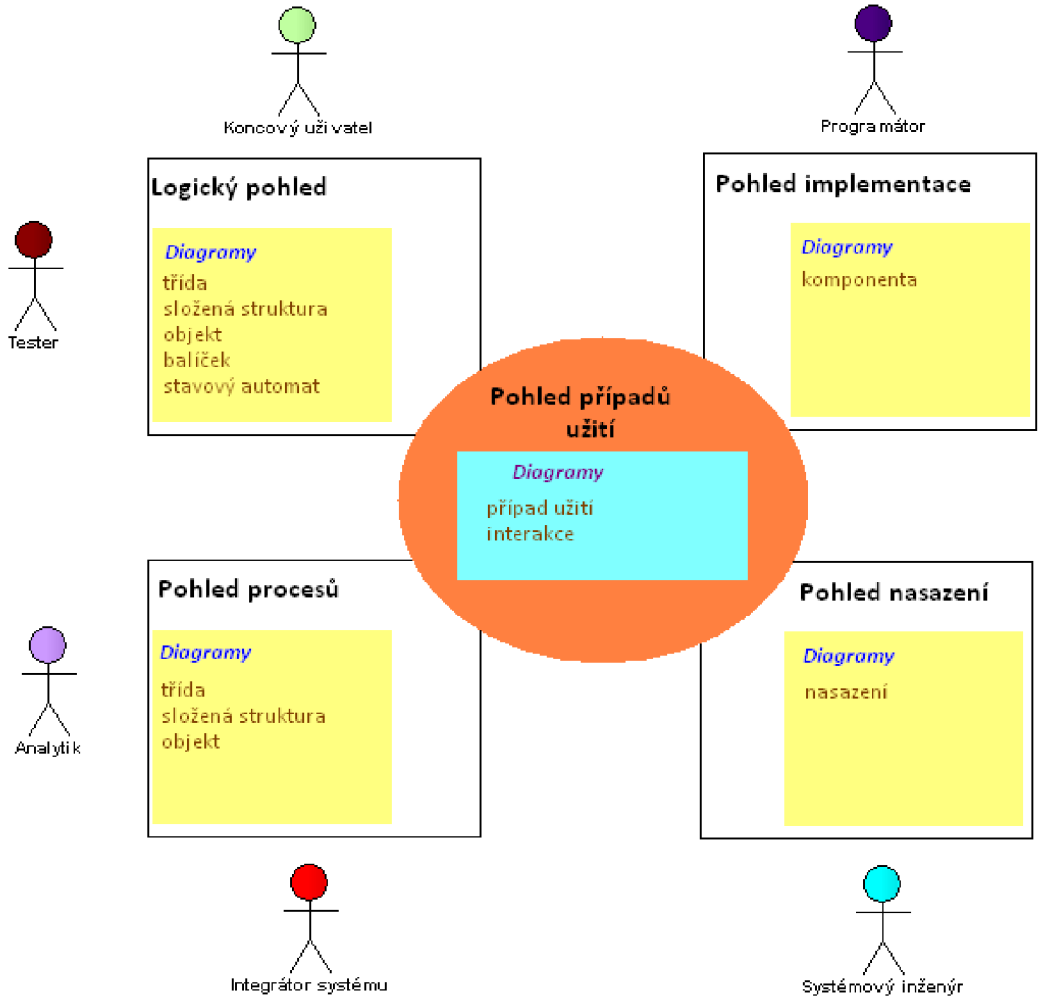
- **Označené hodnoty**

  Umožňují rozšíření prvků modelu o podstatné informace o elementu a také mohou zavádět nové vlastnosti u stereotypu. Shodně s omezeními se zapisují do složených závorek s tím rozdílem, že jejich syntaxe vyžaduje navíc použití atributu s přidruženou hodnotou {atribut=hodnota}. K elementu lze přiřadit více označených hodnot najednou, je však zapotřebí je oddělit čárkou. Jako příklad použiji informaci o verzi třídy {version = 1.1}.

4.4.3 Architektura

Architekturou systému rozumíme podle(3) "organizační strukturu systému, včetně jeho rozkladu na součásti, jeho propojitelnosti, interakce, mechanismů a směrných zásad, které pronikají do návrhu systému." Je to vlastně nejvyšší úroveň koncepce systému v jeho vlastním prostředí.

Na architekturu lze nahlížet mnoha způsoby, ale asi nejčastějším je pohled 4+1 pana Kruchta.



Obrázek 9 - Pohled 4+1

Pohled případu užití

Zbývající pohledy jsou odvozeny právě z něj. Zachycuje základní požadavky kladené na daný software ve formě množiny případů užití.

Logický pohled

Týká se slovníku domény řešeného problému, který prezentuje jako množinu tříd a objektů, a funkcí. Zabývá se zobrazením způsobu, jakým objekty a třídy tvořící základ systému implementují jeho chování.

Pohled procesů

Obdoba logického pohledu, která je zaměřena na procesy. Společně definují chování systému. Náplní pohledu procesů je výkon, škálovatelnost a propustnost softwaru. Prezentuje spustitelná vlákna a procesy jako aktivní třídy.

Pohled implementace

Prezentuje systémová seskupení, konfigurace a vedení pomocí souborů a komponent. Ty utvářejí výsledný kód systému. Znázorňuje závislosti mezi jednotlivými komponentami (součástmi) a modeluje konfiguraci množin z nich vytvořených. Zároveň slouží k definici verze systému.

Pohled nasazení

Zachycuje topologii systému, distribuci, doručení a vlastní instalaci. Nalezneme v něm fyzické nasazení komponent na množině fyzických výpočetních uzlů (počítačů a periferních zařízení).

Model 4+1 jde ruku v ruce s metodikou Unified Process s tím, že jazyk UML jí poskytuje opravdu silné zázemí.

5 Agilní modelování a agilní modelování řízené modelem

Pan Scott W. Ambler charakterizoval agilní modelování (dále jen AM) „jako metodiku založenou na praktičnosti, která slouží k efektivnímu modelování a dokumentaci softwarově založených systémů.“ (zdroj 25) a jak již název napovídá, má velmi blízko k agilnímu životnímu cyklu, který je detailněji popsán v kapitole **3.6 Agilní vývoj software.**

Jinými slovy lze AM popsat jako kolekci hodnot, principů a praktik, které mohou být použity při vývoji softwarových projektů s cílem tento proces v maximální možné míře zefektivnit a zjednodušit. Agilní přístup má velmi blízko k extrémnímu programování (dále jen EP), jehož zásadami se ve velké míře inspiruje, a metodice RUP (Rational Uniffied Process).

5.1 Stavební kameny/podstata AM

5.1.1 Hodnoty

AM převzalo a upravilo základní hodnoty EP k obrazu svému tak, aby se s nimi mohlo plně ztotožnit. Během vývoje software je nezbytné pamatovat na:

- **Komunikace**
  Efektivní komunikace se všemi účastníky projektu, stakeholdery a vývojáři, sehrává klíčovou roli při jeho úspěšné realizaci. Pokud nemají účastníci k dispozici aktuální informace, případně některé důležité chybí, může dojít k vývoji špatným směrem či jeho úplnému pozastavení.
- **Jednoduchost**
  Snahou je vyvíjet co nejjednodušší možné řešení, které bude plně pokrývat všechny zákazníkovy požadavky. Není důvod k implementaci složitých robustních systémů, pokud existuje adekvátní jednoduché a elegantní řešení.
- **Zpětná vazba**
  Kdykoliv některého z členů týmu napadne jakákoliv myšlenka, postřeh či náznak možného řešení, tak by je měl bez prodlení zaznamenat a předložit ostatním k vyjádření. Není důležité, jestli ideu zaznamená nejlepším možným způsobem (slovně, obrázkem či diagramem) podstatné je, že i ostatní dostanou příležitost se s ní seznámit a identifikovat, zda je to cesta správným směrem či nikoliv. Včasná a častá zpětná vazba umožní ušetřit prostředky věnované slepému vývoji.
- **Odvaha**

Důležitá a velká rozhodnutí se neobejdou bez odvahy stejně tak jako situace, kdy je zapotřebí prosadit a razantně změnit směr vývoje, který se vydal nesprávnou cestou. Mnohdy je výhodnější zahodit část stávajícího řešení než později napravovat škody, které způsobilo.

- **Pokora**

  Nikdo není dokonalý a přesně tuto skutečnost by měl mít dobrý vývojář na paměti. Každý z účastníků projektu má svou vlastní parketu, na kterou je specializován a které velmi dobře rozumí - zákazník byznysu, vývojář vývoji. I přesto by však jednotliví účastníci měli být připraveni naslouchat a přijímat názory/řešení ostatních. Názor každého člena týmu má stejnou váhu jako názory ostatních.

5.1.2 Principy

AM se opírá o dvě základní množiny principů užitých v průběhu vývoje a modelování. A to principy stěžejní a podpůrné.

Stěžejní principy

Stěžejními principy rozumíme ty principy, jež je nutné bezpodmínečně dodržet, abychom mohli prohlásit, že modelujeme v duchu agilního vývoje řízeného modelem (Agile Model Driven Development - dále jen AMDD).

- **Modelování s účelem**

  Podstatou tohoto principu je zamyšlení se, zda-li má smysl danou skutečnost do modelu/zdrojového kódu/dokumentu zanést či nikoliv. Je důležitá? Má své opodstatnění v tuto chvíli nebo ji stačí zaznamenat až v pozdější fázi? Velmi častou chybou je, že v drtivé většině případů chceme, aby naše výstupy byly zaznamenány s velkou precizností, mírou detailu, a mnohdy si tak ani neuvědomíme, zda má zanášená informace vůbec nějaké opodstatnění nebo zda ji uvádíme, jen pro naše klidnější svědomí či úplnost. AMDD dbá na to, aby každá uchovávaná informace měla své opodstatnění a smysl, jinak je zbytečná.

  Jedním z účelů vytváření dokumentů je, aby v nich čtenáři nalezli to, co hledají. Při jejich psaní je tedy nutné pamatovat na to, kdo je bude číst, pro koho jsou určeny, a přizpůsobit tomu formu. Pro zákazníka, který se orientuje v oboru, lze použít odbornější terminologii, zatímco u laika je to nepředstavitelné.

- **Maximální návratnost investic všech účastníků projektu**

  a jejich účelné využití. Účastníci projektu investují nemalé množství prostředků (čas, peníze, zkušenosti) proto, aby společnými silami vytvořili software splňující všechny potřeby zadavatele. AMDD si toho je vědomo a vyžaduje, aby byly všechny „investice“ využity pokud možno, co nejefektivněji a nedocházelo tak k jejich plýtváni.

- **Jednoduchá údržba**

a rychlá reakce za změnové požadavky. I přes to, že jsou dodrženy veškeré zásady AMDD, tak může nastat situace, kdy již vytvořené dokumenty vyžadují menší či větší úpravy. AMDD to nepovažuje za něco negativního, ba naopak s touto situací počítá. Z tohoto důvodu by měly být dokumenty psány tak, aby byla jejich případná modifikace co nejjednodušší. Vyvarování se redundance dat je základem.

- **Více typů diagramů/dokumentů**

  Množina užitých diagramů/modelů vždy záleží na povaze vyvíjeného softwaru. V některých případech si vývojový tým vystačí pouze s modelem a jedním dokumentem, jindy je nezbytné připojit grafické návrhy, seznam rizik, otevřené otázky, … Existuje celá řada použitelných dokumentů/modelů. Vždy je na místě důkladně zvážit, který z nich se pro zachycení dané informace hodí nejvíce a zabránit redundanci ukládaných informací v rámci různých modelů.

- **Rychlá zpětná vazba**

  Již byla zmíněna v kapitole **5.1.1 Hodnoty** a opětovná zmínka o ní poukazuje na její důležitost. Čas mezi vykonanou akcí a její zpětnou vazbou je kritický. Pokud je v plánu nějaká zásadní změna, která se dotkne více lidí, včetně zákazníka, je naprosto nezbytné od nich získat zpětnou vazbu ještě před její realizací. Nedojde tak k ohrožení/vzdálení se od vize projektu.

- **Osvojení si jednoduchosti**

  V jednoduchosti je síla a tvoří jeden ze základních pilířů agilního přístupu, jak je uvedeno v kapitole **5.1.1 Hodnoty.** Nejjednodušší řešení bývá tím nejlepším. Software, ani byznys procesy v něm uvedené, nejsou zatíženy žádnými složitostmi/výjimkami. Jsou popsány tak, aby byly obecně platné a nebyla nutná jejich další úprava.

- **Chopení se změn**

  Nebát se změny a chopit se její realizace, i tak se dá charakterizovat tento princip. Tak jako se mění účastníci projektu (někteří přicházejí, jiní odcházejí), tak se mění i požadavky kladené na systém. Podstatné je udržovat všechny podklady ve stavu odpovídajícímu skutečnosti, nejlépe s minimálním úsilím.

- **Modelování s přírůstkem**

  Agilní přístup uznává iterativní životní cyklus modelu, tedy modelování s přírůstky. Vychází z faktu, že málokdy se podaří před vlastním rozběhnutím projektu nasbírat přesné a neměnné požadavky kladené na vyvíjený systém, tak jak to požaduje životní cyklus vodopád. Modelování s přírůstkem klade důraz především na ty požadavky, které jsou důležité a nezbytné v dané iteraci a ostatním nevěnuje pozornost - maximálně k nim přihlíží, aby nenastal konflikt. Není tedy podstatné zachytit celý obsáhlý systém již od začátku, ale nejprve vytvořit ten nejzákladnější pohled, který bude v následujících iteracích postupně obohacován o větší míru detailu či novou funkcionalitu.

- **Odvádění kvalitní práce**

Nikdo neocení lajdáckou práci. Pokud má některý z účastníků projektu výhrady k jeho kvalitě, není nic jednoduššího než si jeho námitky vyslechnout a zvážit jejich adekvátnost. Pokud se členům vývojového týmu na vývoji software něco nelíbí, ať už se jedná o jeho složitost, těžkou orientaci v něm či s některými věcmi nesouhlasí, tak se to velmi často podepíše na kvalitě práce, kterou odvádějí. Vývojový tým by se měl s projektem ztotožnit.

- **Fungující software plnící svůj účel**
  je primárním cílem projektu. Software by měl být vysoce kvalitní, plně fungující a reflektovat všechny požadavky zadavatele, které plní co nejefektivněji v rámci možností. Nesmí to být na úkor jeho jiných vlastností, například jednoduchosti. Tvorba doplňující dokumentace, modelů a manažerských produktů není hlavním cílem projektu nýbrž jeho součástí.
- **Rozšiřitelný software**
  Jedním z častých požadavků zadavatelů je, aby byl systém v budoucnu lehce rozšiřitelný. I přes maximálně vyvinuté úsilí se stává, že vyvinutý software obsahuje skryté chyby či postrádá nějakou funkcionalitu. Je tedy zřejmé, že i po jeho předání zákazníkovi v něm budou nadále probíhat úpravy. Měl by být proto napsán tak, aby orientace v jeho zdrojovém kódu byla jednoduchá a budoucí vývojový tým, tak mohl nastudování jeho struktury a funkcionality věnovat minimum času. Údržba, oprava neduhů či práce na následníkovi pak bude podstatně jednodušší a efektivnější.

Podpůrné principy

Jejich užití není nezbytné, ale doporučuje se.

- **Otevřená a upřímná komunikace**
  Otevřená a upřímná komunikace je zárukou správného směru vyvíjeného software a vysoké míry informovanosti. Lidé by měli mít prostor vyjádřit své názory, nebát se oznámit zpoždění vydání a podobně.

5.1.3 Praktiky

Kromě stěžejních a podpůrných praktik existuje i řada dobrých postřehů, které stojí za zvážení i přesto, že nejsou součástí AMDD.

Stěžejní praktiky

Stěžejní praktiky se vyznačují tím, že je musíme striktně dodržet, abychom mohli mluvit o modelování v duchu AMDD.

- **Aktivní zapojení stakeholderů**
  Aktivní zapojení stakeholderů je nesmírně důležité. Stakeholdeři poskytují adekvátní informace o tom, jak by měl vyvíjený systém fungovat a co se od něj očekává. Navíc, právě oni jej budou využívat nejvíce, se jejich aktivním zapojením předejde komplikacím, které by mohly nastat

v momentu předání software zákazníkovi, když by jej stakeholdeři označili za nepoužitelný/nevyhovující.

- **Modelování ve spolupráci s ostatními**

  Při modelování větších systémů je práce v týmu nevyhnutelná, a proto je nutné zavést efektivní a koordinovanou spolupráci, která nejen že znatelně usnadní práci, ale především zabrání různým neduhům (přepsání zaktualizovaného dokumentu spolupracovníka svou starší verzí), spojených s prací v týmu.

- **Užití nejvhodnějších řešení**

  Každý z artefaktů jazyka UML má své specifické uplatnění, pro které byl vyvinut a kde je nepřekonatelný. Například, diagram aktivit v UML je vhodný pro popis byznys procesů, zatímco pro po popis statických struktur databáze je naprosto nepoužitelný. Stejně tak platí, že obrázek je lepší než tisíc slov a model čitelnější než zdrojový kód. Proto je velmi důležité znát přednosti i slabosti každého nástroje/artefaktu a zvážit, kdy se jej hodí použít, případně kdy sáhnout po jiné variantě.

- **Souběžná práce na různých částech modelu/dokumentech**

  AMDD doporučuje přepínat kontext mezi jednotlivými projektovými dokumenty/diagramy a udržovat tak jejich provázanost. Například při modelování diagramu případů použití může nastat situace, že se stane vytrženým z kontextu. Analytik se mu věnuje tak důkladně, například v těle některého scénáře uvede atributy, které je ve formuláři nutné vždy vyplnit, že na ostatní diagramy „zapomene“ a v diagramu tříd pak tyto informace u dané třídy chybí.

  Během vývoje je nezbytné přecházet mezi dokumenty/modely nejenom z důvodu jejich aktualizace, ale především i z důvodu uvědomění si souvislostí/provázanosti jednotlivých aspektů.

- **Ověření modelu kódem**

  Soubor diagramů je abstraktním vyjádřením zákazníkem požadovaného softwarového řešení. Někdy může být teoreticky velmi dobře zvládnutý, ale prakticky téměř neproveditelný. Aby se tomu předešlo, je dobré napsat část zdrojového a testovacího kódu, který prověří správnost například navrženého sekvenčního diagramu popisujícího logickou strukturu byznys pravidel.

  Modelování, psaní kódu a jeho následné otestování to je jeden ze základních pilířů AMDD.

- **Obsah důležitější než forma**

  Každý artefakt může být reprezentován mnoha způsoby. Případy užití mohou být zaznamenány na listech sešitu, nakreslen na tabuli a následně vyfocen či rovnou zanesen do některého z CASE nástrojů. Záleží pouze na autorovi, pro který způsob se v danou chvíli rozhodne. Důležité ale je, aby to v danou chvíli byl ten nejvhodnější způsob.

  Při prezentaci nástinu obsahu iterace bohatě postačí hrubý náčrt jejího řešení na tabuli, klidně i s nepřesnostmi v UML notacích, zatímco během vývoje už je to nedostačující a je nezbytné mít

tyto informace lépe zapracované v širším kontextu a v elektronické podobě, například v CASE nástroji. Důležité je vždy mít k dispozici všechny potřebné podklady nehledě na jejich provedení.

- **Vývoj v přírustcích - inkrementacích**

  Tento princip byl zmíněn již v předchozí kapitole **5.1.2 Principy**, kde na něj bylo nahlíženo z pohledu požadavků. Z hlediska agilních praktik je brán z jiného pohledu a to pohledu dělení složitějších problémů na menší dílčí části, které jsou řešeny postupně, většinou v iteracích trvajících od několika týdnů do měsíce. Tato praktika umožňuje seznámit zákazníka s průběhem prací na projektu a postup demonstrovat ukázkou fungujícího software, který představuje část z vyvíjeného řešení.

- **Informace na jednom místě**

  Pozor na duplicitu, ta je vždy zárukou nadbytečné práce. Daná informace by měla být uložena pouze na jednom místě (v jednom souboru či diagramu). Před jejím zanesením je vždy dobré zvážit, zda má smysl ji vůbec ukládat či nikoliv.

- **Kolektivní vlastnictví**

  Každý může pracovat na libovolné části projektu (diagramy, zdrojový kód, dokumentace) a dle svého uvážení ji upravovat - samozřejmě v souladu s ostatními.

- **Co nejmenší obsah modelu**

  Cílem není sesbírání a zanesení co největšího množství informací a podkladů. Důležité je vybírat pouze ty, které mají své opodstatnění a odůvodnění. Stakeholdeři „rádi“ zahlcují informacemi a je potřeba je v tomto směru usměrnit.

- **Co nejjednodušší popis**

  Účelem není co nejefektivnější a vytříbený popis skutečnosti. Důraz je kladen především na to, aby vše bylo popsáno nejjednodušším možným způsobem s tím, že nejsou opomenuty klíčové vlastnosti systému.

- **Veřejně přístupné úložiště**

  Všichni členové týmu, společně se zákazníkem, případně stakeholdery, by měli mít přístup do datového úložiště, kde jsou vystaveny všechny důležité dokumenty a zdrojové kódy. Je na zvážení každého vedoucího týmu, zda se rozhodne zákazníkovi zpřístupnit pouze některé soubory či celou složku.

  V současnosti existuje celá řada softwarových řešení, která se touto tématikou zaobírají, například Turtoise SVN, a umožňují i nastavení přístupových práv pro jednotlivé uživatele.

Podpůrné praktiky

- **Dodržení modelovacích standardů**

  Standardy pro modelování softwarových projektů, které byly „vytvořeny“ a následně schváleny, mají svá opodstatnění. Rozumný vývojář by je proto měl přijmout za své a časem se s nimi sžít.

- **Aplikování analytických/programovacích vzorů dle svého přesvědčení**

Analytik/programátor, který nechce ustrnout v osobním rozvoji, se neustále učí novým věcem, jež pak následně vhodně aplikuje ve svých modelech/kódech. Ať už se jedná o architektonické, grafické či analytické vzory. Nicméně by se měl vyvarovat použití takových vzorů, u kterých cítí, že jsou pro danou situaci nevhodné, případně že jdou proti jeho přesvědčení. Užití vzorů by nemělo komplikovat model.

- **Zrušení dočasných dokumentů**

  Rozsáhlé množství dokumentů, které jsou v průběhu prací na projektu vytvořeny, jsou pouze dočasné dokumenty (designové skici, potenciální návrh architektury), které splnily své účely v minulosti, ale nyní již nemají žádnou hodnotu.

  Dokumenty, které již splnily svůj účel, je vhodné nezahazovat, ale uchovávat je v archivu. Ať už jen tak pro úplnost nebo pro účely nahlédnutí (proč byla databáze navržena právě tímto způsobem?). Tyto dokumenty by ale měly být uloženy na jiném místě než ty aktuální/živé, aby zbytečně nepoutaly pozornost.

- **Formulace „dohodnutých" modelů**

  V případě, že vyvíjené softwarové řešení bude spolupracovat, využívat funkcionalitu, se systémem třetí strany (účetní softwary, systémy obstarávající skladové hospodářství) pak se nelze vyvarovat použití takzvaných „dohodnutých" modelů.

  Tyto modely slouží pro definování struktury a formátu dat, které si systémy mezi sebou budou posílat přes dohodnuté komunikačními kanály. Formulace těchto požadavků musí být jednoznačná a srozumitelná pro všechny zúčastněné strany, které svým schválením stvrdí, že jsou schopny dodržení výše uvedeného zajistit a připraví na to i své systémy.

- **Aktualizace modelu pouze tehdy, je-li to zapotřebí**

  Model by měl být zaktualizován pouze v tom případě, že by jeho stávající podoba přinesla větší komplikace než námaha vynaložená na jeho aktualizaci. To znamená, že modely nemusí být za každou cenu nejaktuálnější, ale informace v nich obsažené by neměly být v rozporu s realitou.

Zajímavé postřehy

Následující praktiky doplňují praktiky AM, ale nejsou nutně jeho součástí.

- **Refaktorizace**

  Původně se tento proces týkal psaní zdrojového kódu, nyní má však mnohem širší využití, například i v modelování. Refaktorizace je disciplinovaný proces provádění změn v softwarovém systému takovým způsobem, že nedochází ke změně vnějšího chování systému, ale „pouze" je zpřehledněna a vylepšena jeho vnitřní struktura. To má za následek lepší čitelnost, větší efektivnost a hlavně menší chybovost.

5.2 Agilní model

Dodržením výše uvedených hodnoty, principů a praktik agilního modelování je docíleno agilního modelu, který se vyznačuje tím, že:

- **Plní svůj účel**

  Některé modely jsou vytvářeny především pro vyšší management (zachycení byznys procesu), jiné pak pro vývojáře (definice jednotlivých tříd v Javě,…). Tak či tak, i přesto, že je každý model specifický, tak vždy primárně plní svůj účel, pro který byl vytvořen.

- **Je lehce pochopitelný**

  Agilní modely jsou srozumitelné a lehce pochopitelné pro, nejlépe všechny, čtenáře. V případě, že je daný dokument určen například pro manažery, tak nic nebrání tomu použít výrazy a slovní obraty, které jsou jim blízké, aby do něj snáze pronikli.

  Je důležité mít na paměti, že použité notace a výrazy ovlivňují pochopení modelu. Model, kterému skvěle rozumí autor, a ostatní v něm tápají, nemá žádnou přidanou hodnotu. Snadná pochopitelnost je stejně důležitá jako vyvarování se křížení čar.

- **Je dostatečně přesný**

  Model by měl být tak přesný, jak je potřeba. To znamená, že v něm nemusí být zachyceny ty informace, které jsou nepodstatné a stejně by je nikdo nevyužil. Stanovení dostatečné přesnosti záleží na dvou faktorech. Na čtenářích modelu a na přínose, který se od něj očekává (načrtnutí správné implementace či její detailní popis?).

- **Je dostatečně konzistentní**

  Konzistence je velmi důležitým aspektem. Měla by však být vždy přiměřená. Přehnaná konzistence s sebou přináší velké náklady, které spolkne její údržba. Oproti tomu její ignorace může přinést velké nesrovnalosti, které se ve výsledku můžou pěkně prodražit. Je proto důležité hlídat větší a středně velké změny v systému a upravit podle nich oblasti, kterých se dotknou.

- **Má pozitivní přidanou hodnotu**

  Nebo-li nestojí víc, než přinese.

- **Je jednoduchý tak jak to jenom jde**

5.3 Agilní vývoj řízený modelem

Jak již název napovídá, AMDD je agilní verzí Model Driven Development (dále jen MDD). MDD se vyznačuje mimo jiné tím, že modely vyvíjeného software jsou vytvořeny a schváleny ještě před tím, než začne jeho implementace. Z tohoto důvodu jsou pak modely velmi robustní (detailně zpracované), snažící se pokrýt řešenou oblast co nejobsáhleji. Pro potřeby tohoto přístupu je typické, že využívá životní cyklus vodopád, který plně reflektuje jeho potřeby. Zadání projektu je tak známo

před jeho vlastní implementací, kdy je i zafixováno, a zákazník tak má pouze minimální možnost změnit jeho znění ve fázi implementace.

AMDD si oproti tomu zakládá na iterativním přístupu, kdy zákazník získává prostor pro pozměnění/doplnění zadání projektu. Model vytvořený podle pravidel AMDD není, a ani v poslední iteraci nebude, tak robustní jako ten, který vznikl dle norem MDD. Podstatou AMDD je, aby v modelu byly uvedeny pouze ty informace, které jsou nezbytné pro jeho implementaci. Ani AMDD není všelék na všechno a vyplatí se především při vývoji středních a velkých projektů, kterého se účastní větší počet lidí. Pro malé projekty s malým vývojovým týmem je lepší použít „klasický" MDD přístup.

5.3.1 Průběh iterace v duchu AMDD

Obecné informace o iterativním životním cyklu jsou uvedeny v kapitole **3.3 Spirála**, a proto jim v tuto chvíli nebudu věnovat bližší pozornost.

Každá iterace se sestává z několika fází s tím, že všechny jsou stejně podstatné, a proto by žádná z nich by neměla být přehlížena či vynechána.

Plánování

Na začátku iterace je nutné co nejpřesněji stanovit její hranice (oblast, kterou bude pokrývat) a určit scénáře a požadavky, které řeší. Každá iterace by měla být stejně náročná, stát stejné množství zdrojů (času, lidí, prostředků), jako ostatní. Pokud je počet požadavků či jejich náročnost nad rámec řešené iterace, tak se vyberou ty s největší prioritou a zbylé jsou automaticky přesunuty do iterace následující.

Modelování a vývoj

Jakmile jsou odsouhlaseny hranice iterace společně s jejím obsahem, tak se analytik pustí do jejich nastudování. Na jejich základě si udělá hrubou představu o tom, co daná iterace obnáší a namyslí si její řešení. Následně uspořádá schůzku s ostatními členy týmu, kde veřejně odprezentuje základní obrysy iterace spolu s nástinem řešení. Členové týmu se pak formou brainstormingu k navrženému řešení vyjadřují a směřují jej do stavu, který bude pro všechny akceptovatelný. Slovo každého člena týmu má stejnou váhu, a je na analytikovi, aby diskuzi moderoval a společnými silami se pokusil najít optimální řešení vyhovující, pokud možno všem. Respektive, žádný z členů týmu by neměl řešení označit za neproveditelné či špatné. V takovém případě by bylo neakceptovatelné. Nalezené řešení se zaznamená a jako výstup bohatě poslouží poznámky a hrubé náčrty (nemusí nutně splňovat notaci použitého jazyka), na flipchartu či listech papíru, kterým všichni zúčastnění rozumí a mají tak povědomí o tom, co je jejich cílem.

Každý ze zúčastněných má v tuto chvíli všechny potřebné informace pro počáteční práci na iteraci a může k ní přistoupit.

- **architekt**

  Navrhne/upraví databázi pro účely iterace. Stanoví způsob komunikace s okolními systémy/moduly.
- **analytik**

  Zanáší do modelu odsouhlasený návrh řešení. Začíná diagramy s vysokou úrovní abstrakce, které postupně zjemňuje a blíže specifikuje - model by měl plně reflektovat potřeby vývojářů. To znamená, že v počátku iterace poskytuje hrubý/ucelený pohled na řešení a až ke konci nabízí potřebou úroveň detailu. Agilní model neobsahuje takovou míru detailu jako model v duchu MDD.
- **vývojář**

  Zahájí práce na implementaci s tím, že každou část, kterou naprogramuje, vzápětí prověří testy (nejlépe testovacím kódem, který sám napsal před vlastní implementací dané části). Během vývoje nahlíží do analýzy, aby se ujistil, že postupuje správně a nacházel v ní nově upřesněné skutečnosti.
- **tester**

  Testuje výsledek předchozí iterace a ve „volném“ čase, kdy čeká na opravu nalezených bugů, připravuje test casy pro aktuální iteraci. Proces testování bude blíže popsán v následující kapitole.
- **projektový manažer**

  Koordinuje a kontroluje práci členů týmu a zajišťuje potřebné prostředky, zdroje, pro jejich práci. Zodpovídá za dodržení projektového plánu, který vypracoval.

Během prací na iteraci je nutné mít na paměti:

- **úzkou spolupráci se stakeholdery**

  Pracemi na iteraci komunikace se stakeholdery nekončí. Kdykoliv je identifikována nějaká nesrovnalost či některé podklady chybí, pak je nutno kontaktovat patřičného stakeholdera, který sníží riziko špatné implementace či nesprávného pochopení zadání a dodá chybějící požadavky.
- **implementaci funkcí na základě jejich priorit**

  AMDD umožňuje flexibilní změnu požadavků, včetně jejich priorit, i během vývoje. Výhodou tohoto přístupu je, že vyvíjený systém plně reflektuje očekávání zadavatele, ale na druhou stranu zatěžuje vývojový tým, především pak analytika, případným přeplánováním a změnou priorit. Změny se však projeví nejdříve v následující iteraci, právě probíhající iterace proběhne beze změn.
- **analýzu a projektování**

  Analýza a projektování je nedílnou součástí agilního modelování jako celku, a proto je těmto dvěma disciplínám věnován prostor i v rámci každé iterace. Pokud je v průběhu iterace zjištěno,

že některý z požadavků nemá své opodstatnění a nepředstavuje žádný přínos, tak jej lze odebrat ze seznamu požadavků a vůbec neimplementovat.

- **zajištění kvality**
  Kvalita je zajištěna souladem několika faktorů: refaktorizací, není povinná, dodržením základních programátorských a analytických konvencí. Projekt by měl vykazovat známky kvality na všech jeho frontách a výstupem by měl být plně fungující software.

Testování

Především pak technika návrhu řízeného testy (test driven design - dále jen TDD), která spočívá v tom, že nejprve je definováno, jaké testy musí vyvinutý software, případně jeho právě vyvíjená část, splnit a pak teprve přichází na řadu vlastní psaní kódu, které je musí bezpodmínečně splnit. Testy by měly být přiměřené, ne přehnaně přísné, ani příliš shovívavé.

Kromě výše uvedených testů je více než vhodné podrobit právě vydaný build integračním testům s cílem prověřit jeho integraci s již hotovým celkem.

Vydání

Uvolnění nejnovější verze systému, nově vyvinutá část byla úspěšně integrována, k ostrému nasazení. Počáteční provoz je bedlivě sledován, aby se včas odhalily případné nesrovnalosti v jeho chování.

Může se stát, že ve vydaném releasu nejsou obsaženy všechny požadavky, které (by) měl obsahovat. I s tím AMDD počítá a tuto situaci řeší tak, že takovýmto požadavkům přiřadí nejvyšší prioritu a zařadí je do další iterace.

5.3.2 Specifické iterace AMDD

Drtivá většina iterací má stejný průběh i cíl, implementaci určité části systému. Existují však i iterace netypické, které se od těch ostatních rapidně liší. Jedná se iteraci mínus prvou, nultou a poslední.

Iterace -1

Nebo-li předprojektové plánování. Stručně řečeno, cílem této iterace je udělat si hrubý obrázek o tom, co nového projekt přinese, jak bude proveden a zda se vůbec vyplatí.

- **Definice obchodních (business) příležitostí**
  Ve spolupráci se zadavatelem jsou zváženy možné obchodní příležitosti a přínosy, které by mohl nový projekt čistě hypoteticky nabídnout. Přinese nová funkcionalita lepší prezentaci společnosti? Dojde k nárůstu nových zákazníků? Bude práce stávajících zaměstnanců podstatně efektivnější?
- **Identifikace řešení realizace**
  Bude lepší vylepšit stávající řešení nebo začít na zelené louce? Bude přizvána ke spolupráci třetí strana? Bude výsledný systém sestaven z menších navzájem propojených balíčků nebo se bude jednat o robustní řešení?

- **Zhodnotit životaschopnost projektu**

  Smyslem tohoto bodu je analýza návratnosti a životaschopnosti projektu. Zaměření se na ekonomické aspekty, technickou proveditelnost, provozuschopnost a vytvoření hrubého obrazu o možnostech projektu. I přesto, že má agilní modelování větší úspěšnost oproti „klasickému“, tak i zde je stále okolo 30% projektů, které skončí nezdarem a jsou předurčeny k zániku.

Iterace 0

V případě, že předchozí iterace neodsoudila projekt k zániku, nastává čas pro iteraci 0, jejímž cílem je identifikace rozsahu vyvíjeného systému spolu s architekturou, která mu bude odpovídat. Většinou netrvá déle než dva týdny a svou délkou se tak vymyká oproti ostatním.

Alfou a omegou této iterace je sběr požadavků, které stakeholdři na vyvíjené softwarové řešení kladou. Na základě nich získáte první ucelenou představu o tom, co je podstatou vyvíjeného systému, jak by měl pracovat a jaké budou vaše další kroky.

V této iteraci není cílem obsáhlá detailní specifikace, která s sebou nese celou řadu negativ, ale pokud možno co nejširší poznání modelovaného systému a zvolení vhodné strategie pro jeho řešení. Této části by nemělo být věnováno přehnaně velké úsilí, aby nedošlo k takzvanému přemodelování, kdy je do modelu zaneseno více požadavků, než je v danou chvíli nezbytné, a dojde tak k nárůstu jeho složitosti. Na druhou stranu může dojít i k přehlédnutí některých velmi důležitých aspektů, jejichž pozdější identifikace může mít za následek předělání části systému, která znamená navýšení prostředků a mnohdy nemalé prodražení.

- **Od uživatelských scénářů k požadavkům**

  Cesta ke konkrétnímu požadavku začíná sběrem takzvaných uživatelských scénářů. Uživatelský scénář popisuje konkrétní požadavek v širším kontextu, většinou ve spojitosti s jeho důvodem, prostředím a kvantifikátorem. Například: „Každý učitel bude mít přiřazeny předměty, které učí, abychom věděli, co je jeho specializací. Mnohdy to totiž nevíme.“ Z uživatelských scénářů lze jednoduše vyčíst, co zákazník požaduje a co jej k tomu vede. S nadsázkou se dá říci, že jsou obdobou neformalizovaných případů užití.

  Počáteční sběr požadavků se sestává z těchto fází:

  Získ uživatelských scénářů

  Uživatelské scénáře jsou získávány od stakeholderů, kteří jsou těmi nejpovolanějšími, co se definice požadavků na vyvíjený software týče. Vždyť právě oni ví, co jim na současném řešení vadí, kde jsou jeho slabá místa, případně kterou funkcionalitu postrádá.

  Nejjednodušší cestou, jak získat uživatelské scénáře, je uspořádat interview, v případě potřeby i několik, kde budou stakeholdeři vyzváni, aby formou volné diskuze řekli vše,

co je napadne. Nemusí to být nutně uživatelské scénáře ve formátu: „Jako <role> chci <přísudek>, abych <popis přínosu>“, ale i to co je momentálně trápí, co by uvítali.

Získané informace není v tuto chvíli nutné nějak strukturovat, podstatné je zachytit vše, co během diskuze zazní a dá se použít pro potřeby projektu. Nyní je na čase uživatelské scénáře, nejenom ty co přímo zazněly na interview, ale i ty co byly analytikem vyvozeny z informací, které během nich zazněly, převést na formální požadavky. Algoritmus převodu je následující:

- Očištění uživatelských scénářů (dále jen US) od jakýchkoliv kvantifikátorů, které přinášejí nepřesnosti a většinou i výjimky.
- Převedení US na jednoduché věty. Pokud se některý US sestává ze souvětí, tak je to horký kandidát na takzvaný epos (eposem se rozumí několik jednoduchých US spojených v jeden obsáhlejší). Každý epos je nutno důkladně prozkoumat a rozdělit na dílčí US.
- Spolu související US, pokrývají stejnou oblast, je vhodné udržovat pohromadě. O takovýchto celcích pak hovoříme jako o námětech.
- Protiřečící si a sporné UC nepřidávat k ostatním, ale evidovat je bokem. Jakmile jsou všechny UC zpracovány, tak uspořádejte ještě jednu schůzku se stakeholdery, kde s nimi všechny sporné UC proberete a společnými silami naleznete uspokojivé řešení.

Identifikace požadavků

Bližší informace o požadavcích a jejich typech jsou uvedeny v kapitole **3.1.1Analýza požadavků.** Každému uživatelskému scénáři nyní přiřaďte jedinečné číselné označení, pod kterým bude nadále vystupovat, a určete, zda splňuje kritéria funkčního či nefunkčního požadavku. V tuto chvíli se z US staly regulérní požadavky.

Každý požadavek může být rozšířen o doplňující atributy, jakými jsou: priorita, délka trvání, zodpovědná osoba, byl schválen?, … AMDD v tomto směru nemá žádná omezující pravidla.

Správa požadavků

Správa požadavků, přidání nového a odebrání či úprava stávajícího požadavku, není vázána pouze na iteraci 0 a ve své podstatě probíhá neustále. Správná práce s požadavky nese nemalý podíl na tom, zda projekt skončí úspěchem či neúspěchem, a proto není radno ji podceňovat. Agilní modelování si zakládá na řazení požadavků do takzvaného zásobníku, kdy jsou požadavky zapracovávány na základě jejich priorit s tím, že ty s největší prioritou jsou nahoře a ty nejméně důležité vespod. U těch s větší prioritou není na škodu je zapracovat do většího detailu a věnovat jim větší pozornost než ostatním. Složitější požadavky je lepší rozdělit na menší, jednodušší.

- **Počáteční návrh architektury**

  Počáteční návrh architektury slouží k eliminaci nepříjemností, které by nás v pozdějších iteracích mohly překvapit (technické komplikace, nevhodnost řešení) a přináší řadu výhod, jakými jsou zvýšená produktivita práce, redukce času potřebného na vývoj, snížení nákladů.

Cílem je snaha identifikovat takovou architekturu systému, která se pro dané řešení hodí a má tak velkou šanci se osvědčit a zajistit, že systém bude řádně fungovat.

V pozdějších iteracích se bude seznam požadavků rozrůstat o nové přírůstky a velmi často bude docházet k upřesnění těch stávajících. To je jeden ze základních rozdílů oproti MDD, kdy by již v této iteraci byly vyžadován konečný naprosto neměnný seznam požadavků spolu s popisem a vzhledem systému.

Poslední iterace

Vydání releasu v rámci poslední plánované iterace a uvedení softwaru do vlastního provozu nutně neznamená ukončení prací na projektu.

- **Závěrečné otestování**

  Testování je součástí každé iterace. V rámci finální iterace však přichází na scénu specifický typ testování a to takzvané průkazné testování (confirmatory testing), které mimo jiné prokáže, jak vyvinutý systém funguje jako celek a jako odezvu nabídne z pohledu koncových uživatelů. Průkazné testování je kombinací vývojářského (testování oproti modelu) a agilního akceptačního testování (musí být splněny všechny požadavky zákazníka, které označil za klíčové pro akceptaci díla). V mnoha případech je průkazné testování ekvivalentem testování proti specifikaci, které slouží k potvrzení, že vyvinutý software plně pracuje v zájmu a přání zákazníka.

  I přesto, že to není AMDD přímo vyžadováno, není na škodu podrobit hotový software testy, které zajistí a provede nezávislý tým profesionálních testerů, ať už vlastní či cizí. Takovéto testy je dobré provést vždy na konci a v průběhu vývoje po určitém počtu iterací, například pěti. Jedná se o takzvané investigativní testování prováděné školenými profesionály, kteří jsou zběhlí ve vyhledávání defektů či nesplněných požadavků ze strany zákazníka.

- **Zapracování nalezených defektů**

  Defekty a nesrovnalosti nalezené v předchozím bodě lze opravit právě v tuto chvíli. Není nezbytné opravit naprosto všechny bugy, ale ty závažné, ohrožující funkčnost systému, určitě.

- **Dokončení veškeré dokumentace – systémové, uživatelské**

  Některé dokumenty již byly dokončeny v rámci iterace, které se týkaly, jiné na své dokončení teprve čekají. Každý systém by měl být zdokumentován. Dokumentaci zákazník většinou věnuje stejnou důležitost jako zapracování všech jeho požadavků. Záleží pouze na něm, jakou prioritu dokumentaci přiřadí, jaký očekává rozsah, typ (odbornou, lehce pochopitelnou), a kolik peněz je ochoten do ní investovat.

- **Vyškolení stakeholderů**

  Proškolení koncových uživatelů, zaměstnanců zadavatele, zaměstnanců podpory tak, aby uměli efektivně pracovat se systémem, je „slavnostním“ zakončením projektu.

5.3.3 Spolupráce se zákazníkem po předání softwaru

V některých případech nekončí závazky dodavatele vůči zákazníkovi dodáním software, nýbrž až po jeho odstavení. I na takové případy AMDD pamatuje a staví se k nim následovně.

Provoz, případně i rozvoj, systému

Cílem této fáze je udržovat systém tak, aby plnil svou funkci a byl užitečný. Tento proces je pro každou organizaci jiný, ale základní cíle zůstávají stejné: zajistit chod systému tak, aby reflektoval aktuální požadavky na něj kladené (nový release, pozor na zpětnou kompatibilitu dat), a pravidelně školit uživatele, aby jej uměli efektivně využívat. Tato fáze končí v moment, kdy dodavatel řešení přestává zajišťovat podporu systému, protože se zákazník rozhodl jej vyřadit z provozu.

Vyřazení systému

Důvodů k odpojení systému může být hned několik.

- **systém bude kompletně nahrazen jiným**
  Firma se například rozhodne využít osvědčené softwarové řešení, které již bylo prověřeno řadou uživatelů, a které nabízí renomovaná společnost jako svůj produkt, jenž dokáže přizpůsobit svým zákazníkům na míru. Výhodou tohoto přístupu je stále aktuální software od odborníků, kteří se v dané byznys doméně dobře orientují.
- **systém již není zapotřebí k podpoře současného byznys modelu**
  Zadavatel upustil od byznys domény, kterou se doposud zabýval a vydal se úplně jiným směrem. Stávající softwarové řešení tak již pro něj není potřebné.
- **systém je redundantní**
  Organizace vyvíjela zároveň dva systémy s tím, že časem došlo k překrytí jejich funkcí, a jeden z nich tak musí být odstaven z provozu.
- **systém je zastaralý**
  A již neplní svou funkčnost. Jeho aktualizace by byla finančně či technicky příliš náročná a tak jej organizace raději nahradí úplně novým řešením.

S problematikou odpojení systému z ostrého provozu či jeho úplné fyzické likvidace se v současné době potýká řada společností a není to vůbec jednoduchá záležitost. Cílem je vypořádat se s touto situací s minimálním dopadem na chod společnosti.

6 Vlastní implementace v duchu AMDD

Nyní, když mám za sebou důkladné seznámení se s teorií, mohu přejít k praktickému řešení diplomové práce. Jako příkladovou studii pro demonstraci AMDD jsem zvolil vývoj informačního systému pro základní a střední školy. Důvodů, proč jsem se rozhodl pro toto téma, bylo hned několik. Tím pro mě asi nejdůležitějším byl fakt, že po právě takovém informačním systému se poptával můj kamarád, který učí na základní škole. Naskytla se mi tak jedinečná příležitost prověřit agilní přístup na projektu z reálného prostředí ve spolupráci se skutečnými stakeholdery. Ti byli zastoupeni mým kamarádem, kolektivem jeho spolupracovníků (v roli učitelů a administrativních pracovníků) rodiči, našimi společnými přáteli (v roli (bývalých) studentů a (budoucích) rodičů). Další velkou výhodou je i skutečné využití modelovaného informačního systému, který tak najde uplatnění a neskončí pouze uložen v archivu (do budoucna, mimo časový rámec diplomové práce, plánuji jeho další rozvoj tak, aby plně reflektoval potřeby škol a byl reálně nasazen). V neposlední řadě je pak toto téma blízké i čtenáři této práce a nečiní mu tak problémy se s ním ztotožnit.

6.1 Iterace -1

Jak jsem již uvedl výše v textu, jedná se o takzvané předprojektové plánování, jehož cílem je zhodnotit přínosy realizace projektu a porovnat je s náklady, které s ním budou spjaty. Jinými slovy se dá říci, že v rámci této iterace se rozhoduje o tom, jestli se vyplatí projekt uskutečnit či nikoliv.

Za tímto účel jsem se svým kamarádem uskutečnil dvě sezení, kde mě nejprve seznámil se svou vizí projektu a poté jsme společnými silami identifikovali, jaké přínosy by jemu a škole mohl projekt přinést. Tyto byznys příležitosti, spolu s nástinem technického řešení a zhodnocení životaschopnosti projektu, jsem zaznamenal do dokumentu **Iterace_-1.docx**. Největší váhu v tomto dokumentu má část pojednávající o životaschopnosti projektu. Zbylá část je víceméně orientační sloužící pouze jako hrubý nástin, který se může v rámci realizace několikrát změnit.

6.2 Iterace 0

Tato iterace je především o sběru uživatelských scénářů, jejich správě a následném rozdělení, vytvoření hierarchie.

Za účelem zisku uživatelských scénářů jsem uspořádal řadu interview, na která jsem si zval stakeholdery dle toho, jakou roli v budoucím systému budou zastávat - na jednom interview se mi tak sešli žáci, na jiném učitelé, … Jejich očekávání, uživatelské scénáře, jsem si poctivě zapisoval. Po každém takovémto sezení jsem zaznamenané scénáře procházel a přepisoval do dokumentu **Iterace_0.docx**, kde jsem se je snažil seskupit do tématických celků a utvořit z nich hierarchickou

strukturu. V moment kdy se stakeholdeři začali víceméně opakovat (nejenom v rámci skupiny, ale mnohdy i napříč skupinami, což jsem uvítal, protože jsem tak nabil pocitu, že jsem nic důležitého neopomněl) jsem tato sezení ukončil. Jedním z principů agilního přístupu je i ten, který říká, že v dané situaci se má jít do takové míry detailu, jak je v dané chvíli zapotřebí. Proto jsou výstupem této iterace ne příliš detailní scénáře, které se snaží zachytit především podstatu vyvíjeného systému a nároky, na něj kladené.

Analýza softwaru vždy primárně vychází z požadavků, které jsou na něj kladeny. V duchu agilního přístupu jsem proto začal jednotlivé uživatelské scénáře převádět na regulérní požadavky tak, jak je uvedeno v kapitole **5.3.2 Specifické iterace AMDD**. Prvním krokem bylo jejich očištění od jakýchkoliv kvantifikátorů. Následovalo jejich „rozbití" na jednoduché věty. Všechny dvojice protichůdných uživatelských scénářů jsem si značil bokem a dodatečně poprosil stakeholdery o to, aby se k nim vyjádřili a určily ty, které jsou opodstatněné. Některé případy užití (dále jen UC) bylo zapotřebí ještě upřesnit.

Příklad:

Krok	Uživatelský scénář
1.	Každá třída má svého třídního a zastupujícího učitele spolu s kmenovou učebnou.
2.	Třída má svého třídního a zastupujícího učitele spolu s kmenovou učebnou.
3.	Třída má svého třídního učitele. Třída má svého zastupujícího učitele. Třída má kmenovou učebnu.

Tabulka 1 - Rozpad uživatelských scénářů

Jak je patrné z uvedeného příkladu, viz **Tabulka 1 - Rozpad uživatelských scénářů**, díky tomuto procesu převodu uživatelských scénářů na atomické scénáře došlo k rapidnímu navýšení jejich počtu. Získané scénáře jsem rozdělil na dvě základní skupiny a to funkční (popisují budoucí funkcionalitu systému) a nefunkční (na jaké platformě bude řešení vystaveno, množství záznamů). Následně jsem každému scénáři přiřadil unikátní číslo, pod kterým bude nadále vystupovat, a váhu. Tím jsem získal prvotní seznam požadavků.

Součástí této iterace je i návrh architektury. Já se rozhodl pro rozdělení systému na moduly/okruhy, které se budou zabývat konkrétní oblastí a budou navzájem propojeny tak, aby mohly spolupracovat. Na základě řady diskuzí a po dlouhém zvažování jsem dospěl k závěru, že za databázový systém zvolím MySQL a informační systém pak vznikne kombinací značkovacího jazyka HTML se skriptovacími jazyky PHP a JavaScript.

Pro MySQL hovořila řada faktů. Velkou rozhodovací váhu měla skutečnost, že se jedná o volně dostupné řešení, které nepředstavuje pro školu žádnou finanční zátěž. S volbou PHP to již tak jednoduché nebylo. Původně jsem zvažoval možnost využití Javy či .NETu, jejichž podstatou je objektový přístup, který jde ruku v ruce s UML. Nicméně po té, co jsem si v těchto prostředích

vyzkoušel vytvořit pár menších „programů“, jsem nabyl pocitu, že v rámci diplomové práce nemá cenu riskovat, že v pro mě neznámém prostředí narazím na nějaký problém, který rapidně zbrzdí můj vývoj. Proto jsem raději zvolil osvědčené, i když ne tolik „módní“ PHP.

V tuto chvíli jsem již věděl, že k prezentaci jazyka UML využiji Enterprise Architect, a proto jsem se zaměřil na jeho podporu MySQL a PHP. Byl jsem mile překvapen, jaké možnosti v tomto směru nabízí. Od fyzického vygenerování databáze na základě jejího modelu, včetně všech omezení a atributů, po propojení zdrojového kódu s objektem a jeho chováním v modelu. V tomto ohledu dokonce nabízí lepší podporu pro PHP, oproti Javě a MySQL, a to především z historického hlediska.

Posledním krokem v rámci této iterace bylo opětovné zhodnocení životaschopnosti projektu včetně důsledné analýzy rizik. V ideálním případě bych ke spolupráci na tomto bodu přizval zbylé členy vývojového týmu (odborníky na jednotlivé aspekty), avšak v rámci řešení diplomové práce jsem si musel vystačit sám. Usoudil jsem, že investovat do projektu se vyplatí (kromě jeho využití v rámci diplomové práce vidím potenciál v jeho reálném nasazení a možnému komerčnímu uplatnění) a mohu tak s klidným svědomím přejít do jeho další fáze.

V návaznosti na skutečnost uvedenou v předchozím odstavci, bych rád uvedl, že kdykoliv v následujícím textu narazíte na termín „vývojový tým“, tak v rámci tohoto dokumentu je zastoupen pouze jednou osobou (a to autorem diplomové práce), zatímco v ideálním případě by se sestával z týmu programátorů, architektů, analytiků, projektových manažerů, …

6.3 Iterace 1

Předchozí dvě iterace jsou specifické a od ostatních se odlišují nejenom svým obsahem, ale i formou. Jak jsem již zmínil výše, jejich podstatou je zhodnocení úspěšnosti projektu, případně příprava podkladů pro jeho úspěšné spuštění. Následující iterace jsou pak zaměřeny především na praktickou část (modelování a vývoj systému). A iterace s číslem jedna je první z nich.

V této chvíli přichází na scénu všichni účastníci vývojového týmu, kteří dříve na samotném řešení přímo neparticipovali, sloužili především jako konzultanti, ale na základě jejichž doporučení byla dělána klíčová rozhodnutí. Iteraci jsem si rozdělil do tří fází, jejichž rozdělení a užití bude totožné pro všechny následující iterace.

6.3.1 I. fáze

Tato fáze se zabývá stanovením hranic iterace spolu s hrubým nástinem jejího řešení.

Hranice iterace

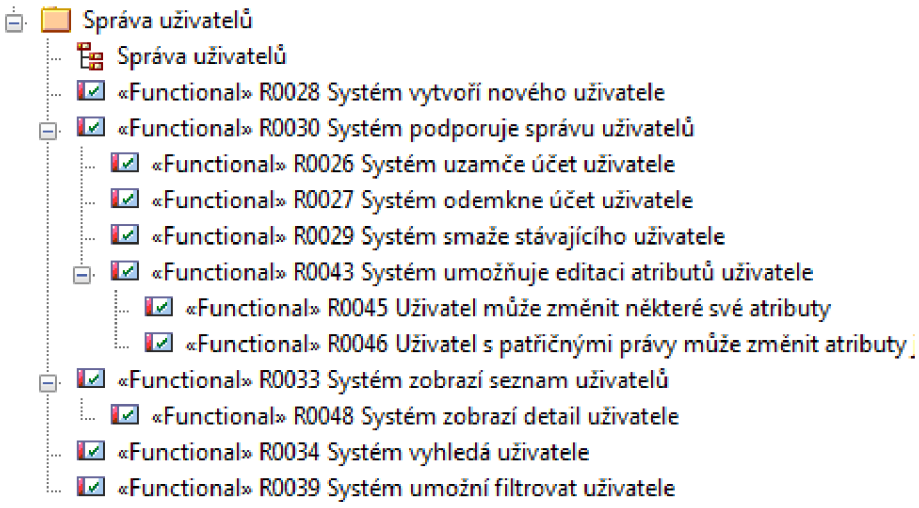
Vymezují oblast, logický celek, kterou se daná iterace zabývá. V předchozí iteraci jsem sesbírané uživatelské scénáře, nyní již požadavky, seskupil do logických celků pokrývajících určitou funkcionalitu (správa učitelů, správa předmětů, správa uživatelů, …).

Po jejich důkladném nastudování jsem definoval hranice této iterace jako proces autentizace a autorizace. Jinými slovy jejím cílem je pokrýt oblast přihlášení/odhlášení uživatele do systému a nastavení jeho přístupových práv. S touto problematikou je provázána správa uživatelů, která je nedílnou součástí této iterace. Měl jsem na paměti, že realizace iterace by neměla přesáhnout délku 20 člověkodní a všechny iterace by měly mít přibližně stejnou délku.

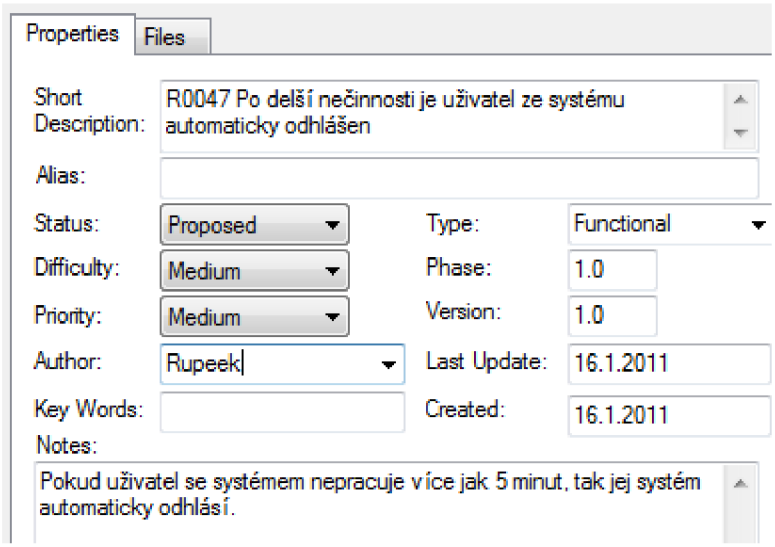
Požadavky

Jak jsem zmínil již v kapitole **6.2 Iterace 0,** nalezené požadavky se vyznačovaly velkou mírou abstrakce (nešly příliš do detailu), což v dané chvíli nevadilo, ba naopak. Nyní však nastal čas pro zpřesnění právě těch požadavků, které pokrývají oblast této iterace. Za tímto účelem jsem uspořádal interview, na které jsem sezval všechny stakeholdery, nehledě na jejich roli, jenž se na projektu podíleli. Všichni účastníci byli dopředu seznámení s probíraným tématem a mohli se tak náležitě dopředu připravit. Samotné setkání pak probíhalo formou brainstormingu, kdy jsme společnými silami zpřesňovali a doplňovali požadavky vázající se k této iteraci. Ty požadavky, jejichž cílem je především zpříjemnit práci se systémem a nemají přímo vliv na jeho funkcionalitu, jsem označil jako nepovinné (nice to have).

Rozšiřující seznam požadavků jsem dále zpracoval. Nejprve jsem jej rozdělil do tematických celků a porovnal se stávajícím seznam získaným v průběhu 0 či -1 iterace. Jednotlivé požadavky jsem pak postupně zanášel do katalogu požadavků, který jsem se rozhodl vést/spravovat v CASE nástroji Enterprise Architect.



Obrázek 10 - Hierarchie požadavků



Obrázek 11 - Detail požadavku

	UC0026 Vytvořit nového uživatele	UC0027 Uzamčít uživatele	UC0028 Odemknout uživatele	UC0029 Smazat/zneplatnit uživatele	UC0030 Zobrazit detail uživatele	UC0031 Editovat uživatele	UC0032 Zobrazit seznam uživatelů	UC0033 Vyhledat uživatele	UC0034 Filtrovat uživatele	UC0035 Editovat mé údaje	UC0036 Změnit heslo
R0026 Systém uzamče účet uživatele		X									
R0027 Systém odemkne účet uživatele			X								
R0028 Systém vytvoří nového uživatele	X										
R0029 Systém smaže stávajícího uživatele				X							
R0030 Systém podporuje správu uživatelů	X						X				
R0033 Systém zobrazí seznam uživatelů							X				
R0034 Systém vyhledá uživatele								X			
R0039 Systém umožní filtrovat uživatele									X		
R0043 Systém umožňuje editaci atributů uživatele											
R0045 Uživatel může změnit některé své atributy										X	X
R0046 Uživatel s patřičnými právy může změnit atri...						X					
R0048 Systém zobrazí detail uživatele					X						

Obrázek 12 - Matice požadavků

Tento přístup jde ruku v ruce s vizí agilní metodiky. Z vlastní zkušenosti vím, že v tomto prostředí lze s požadavky nakládat/operovat mnohem pohodlněji a efektivněji než v textovém či tabulkovém souboru. V dřívějších iteracích postačovala jejich evidence v nedigitální podobě (mé zápisky zaznamenané na listech papíru), nyní však vyvstala potřeba jejich digitalizace. Ať už z důvodu jejich správy (prostředí EA nabízí možnost jejich hierarchického řazení (utváření stromů); obohacení

o atributy jakými jsou například priorita, obtížnost, fáze, autor, bližší popis… (viz **Chyba! Nenalezen zdroj odkazů.** a **Chyba! Nenalezen zdroj odkazů.**)) či potřeby distribuce mezi stakeholdery a ostatními členy týmu. Dříve sloužily nasbírané požadavky především pro účely analytika, nyní je na čase je sdílet s ostatními.

Nespornou výhodou EA je fakt, že v něm zavedené požadavky lze jednoduše propojit s modelovanými diagramy (konkrétní případy užití, třídy, …). Zákazník tak může pomocí náhledu velmi jednoduše zjistit, jestli byl daný požadavek zapracován, případně jak, viz **Obrázek 12 - Matice požadavků.** V této fázi jsem dočasně přiřadil všem požadavkům stejnou výchozí prioritu a stav „schválen".

Stakeholdeři během sezení zmínili i pár požadavků, které nebyly přímo součástí řešené iterace. Pro jistotu jsem je raději zaznamenal do svého pracovního seznamu, abychom se k nim v budoucnu mohli případně vrátit a nezapomněli na ně.

Otázky

Při zanášení požadavků do modelu jsem měl čas se nad nimi zamyslet a uvědomit si je v širším kontextu. Mnohdy jsem tak narazil na otevřené otázky, ke kterým bylo zapotřebí získat stanovisko zákazníka. Seznam otevřených otázek jsem udržoval v dokumentu, kam jsem pravidelně exportoval zanesené požadavky. Tento dokument, jsem pak každý den (pokud jsem evidoval nějaké otázky) zasílal zákazníkovi e-mailem k vyjádření s tím, že jeho odpovědi jsem okamžitě zapracovával.

Rizika

Jakmile byly všechny otázky zodpovězeny a všechny požadavky zaneseny, tak jsem se pustil do identifikace rizik s nimi souvisejících. U rizik jsem se pokusil nalézt způsob jak jim zabránit, případně jak minimalizovat jejich škody. Některá rizika se mi podařilo ve spolupráci se zákazníkem eliminovat a to tak, že jsme patřičně upravili požadavek, který je způsoboval. Ponechali jsme pouze ta, u kterých zákazník určil, že nejsou kritická. Více informací v přiloženém souboru Iterace_1.docx.

Nástin řešení

Nyní, když byly identifikovány a zaneseny všechny požadavky, tak jsem se mohl pustit do jejich nastudování a přemýšlet nad návrhem jejich řešení. Poté, až bych měl jasnou představu, jak budu dále postupovat, bych uspořádal schůzku, kick off meeting, se zbylými členy týmu tak, jak je uvedeno v kapitole **5.3.1 Průběh iterace v duchu AMDD**.V rámci diplomové práce jsem si však musel vystačit sám.

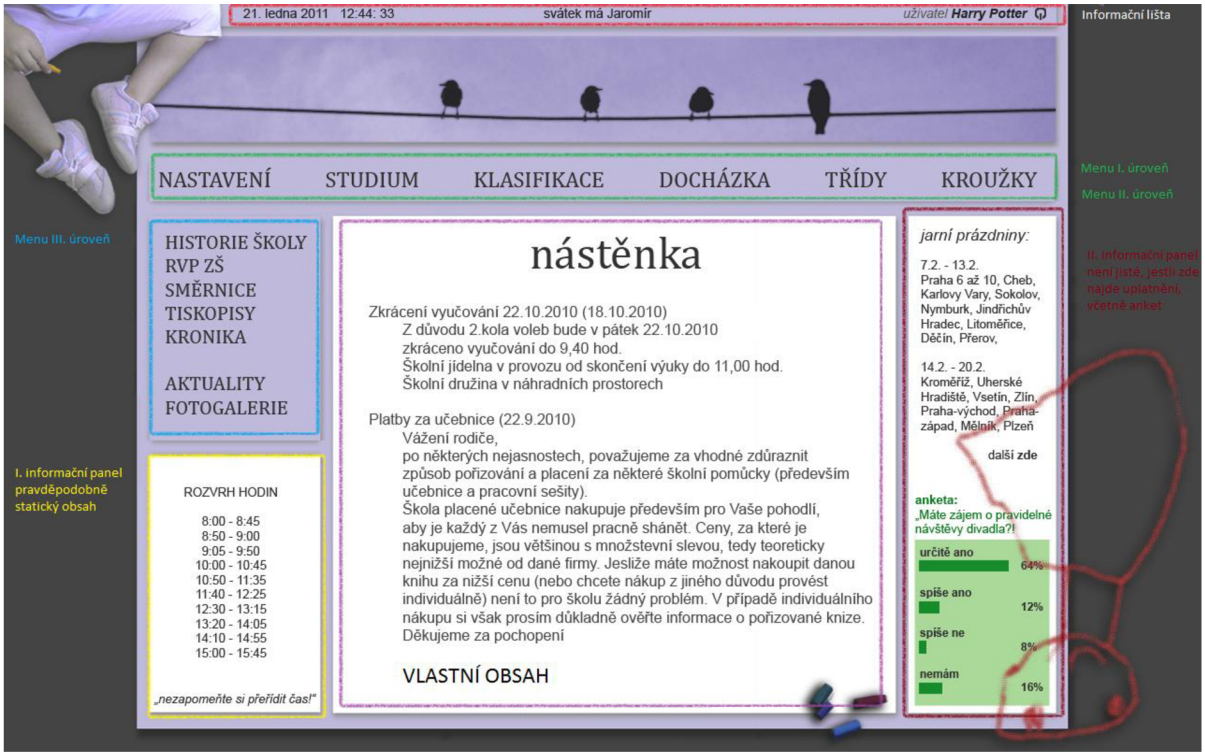
Díky uživatelským scénářům, které jsem sesbíral v iteraci 0, jsem měl hrubé povědomí o tom, co zákazník od systému očekává a jak by měl fungovat. Rozhodl jsem se, že ještě dříve než se pustím do implementace první iterace, tak se hlouběji zamyslím nad systémem jako celkem. Přišlo mi to příhodné, nejen z toho důvodu, že v danou chvíli jsem měl k dispozici dostatek informací, ale především proto, že první iterace je svým způsobem klíčová. Jsou v ní dělány první kroky,

které určují následující směr vývoje, a o mnohém rozhoduje. Proto je lepší mít vše důkladně namyšlené, i za cenu toho, že dojde k prodloužení délky této iterace a tím pádem i navýšení prostředků. V mém případě to činilo tři člověkodny (man-days) a myslím si, že to byly dobře investované prostředky. Měl jsem možnost se zamyslet nad budoucím vývojem systému a identifikovat potenciální problémy: u každé třídy bude nezbytné evidovat číselník roku, ve kterém platila; jak to bude s importem žáků do tříd na začátku každého roku (existuje nějaké obecné pravidlo, že studenti většinou postoupí do vyšší třídy nebo je nutné naimplementovat načítání z xls souboru?). Vždy je výhodnější slepým vývojovým cestám předcházet, než jimi způsobené škody napravovat.

Grafický návrh

V tuto chvíli jsem již věděl, co od této iterace očekávám. Dříve než jsem se mohl pustit do vlastního modelování/programování, tak jsem musel vyřešit ještě jednou oblast, která je pro iteraci 1 typická a činí ji tak výjimečnou oproti ostatním. Touto oblastí je grafický návrh.

O grafické podobě software by měl zákazník uvažovat v závěru iterace 0, kdy je již zřejmé, zda bude plánovaný systém vyvíjen či nikoliv. Grafika může být dodána samotným zákazníkem (grafický návrh nechal vyhotovit třetí stranou) nebo vývojovým týmem, kterému zadal kompletní zakázku. V mém případě ponechal zákazník vše na mně (vývojovém týmu). Jelikož nemám umělecké nadání, tak jsem přizval na pomoc svou sestřenici, Martinu Novákovou. Té jsem nastínil svou představu vzhledu a „obsahu“ (menu I. a II. úrovně, prostor pro panel se vzkazy, zobrazení přihlášeného uživatele, atd.).



Obrázek 13 - Grafický návrh systému

Dodaný grafický návrh jsem doplnil o popisky vysvětlující funkcionalitu jeho jednotlivých komponent. Do vyšší míry detailu jsem se nepouštěl, neboť to agilní přístup nevyžaduje.

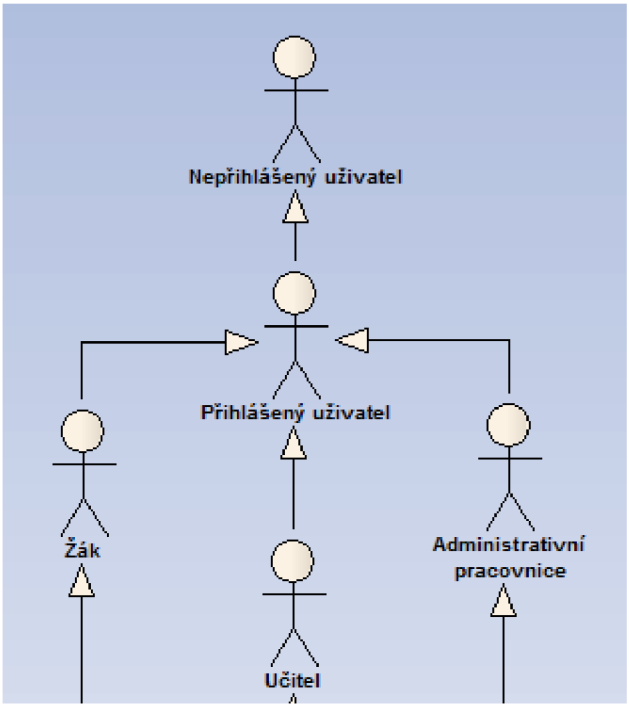
6.3.2 II. fáze

Jakmile byly splněny všechny náležitosti I. fáze, tak jsem mohl přistoupit k vlastnímu vývoji systému. V ideálním případě by programování probíhalo souběžně s modelováním. Já jsem se rozhodl, zcela logicky, že začnu modelováním.

Modelování

Základem každého systému jsou uživatelé, kteří jej využívají. Proto jsem jako první zanesl do modelu diagram popisující stromovou strukturu všech identifikovaných aktérů systému. Vycházel jsem, stejně jako v ostatních případech, ze skic, které jsem dal dohromady se zbylými členy týmu v první fázi této iterace.

Mezi stěžejní praktiky AMDD patří užití nejvhodnějšího řešení, jak je uvedeno v kapitole **5.1.3 Praktiky**. V tomto případě se jako nejvhodnější jevil diagram případů použití, který obsahuje všechny potřebné náležitosti, aktéry a asociace (v tomto případě generalizace) mezi nimi, pro zachycení hierarchie.



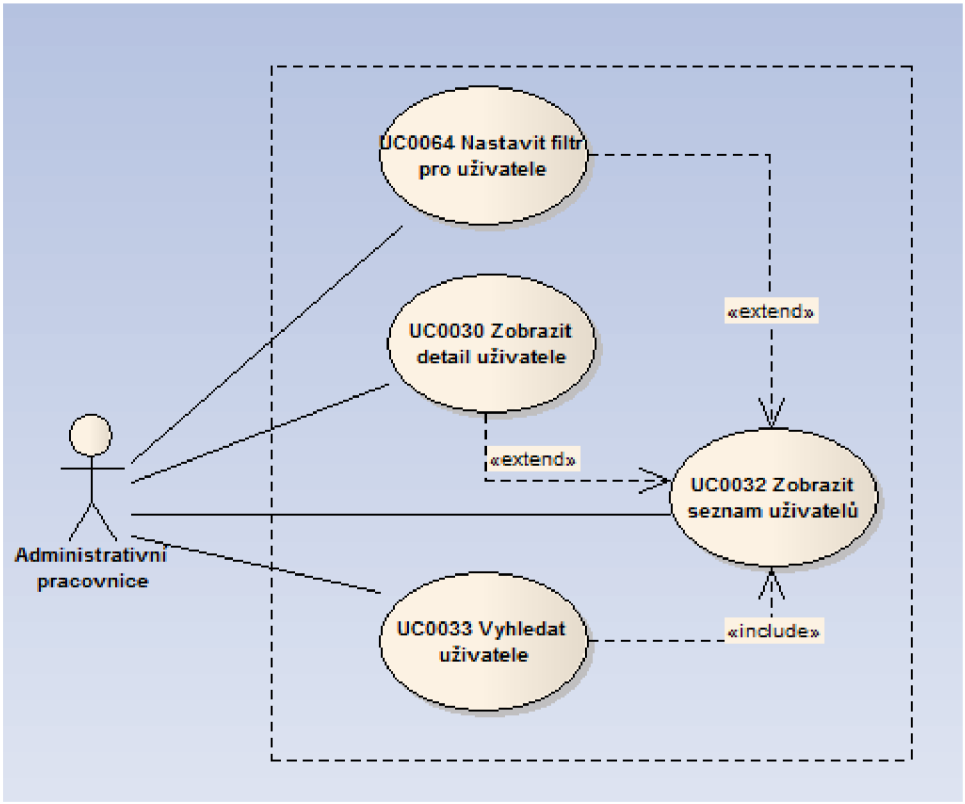
Obrázek 14 - Diagram případů užití

U jednotlivých aktérů jsem stručně a výstižně popsal, co jejich role obnáší, jaká mají práva. V každé iteraci jsem pak popis jednotlivých rolí obohacoval o nové funkcionality, které daná iterace přinesla a pro které měli daní aktéři potřebná oprávnění, viz **Tabulka 2 - Popis rolí v průběhu iterací.**

Role	Iterace 1	Iterace 2
Administrativní pracovnice	- spravuje uživatele	- spravuje uživatele - spravuje předměty - spravuje třídy

Tabulka 2 - Popis rolí v průběhu iterací

Pro činnosti, které mohou aktéři se systémem vykonávat, se nejlépe hodí výše zmíněný diagram případů užití. Do něj jsem tedy postupně začal zavádět jednotlivé činnosti, které mohou uživatelé spouštět. První přišel na řadu diagram popisující správu, vytvoření/editace/smazání/uzamčení, uživatelů. V rámci celého modelu jsem se snažil vyvarovat se zbytečných detailů a udržet si vysokou granularitu, a ani případy užití nebyly výjimkou. U nich jsem se vyvaroval toho, abych ty obsáhlejší rozdělil na jednodušší podpřípady. Například „UC033 Vyhledat uživatele“ bych mohl pomocí asociace „include“ rozdělit na dílčí případy užití jako jsou: „Zadej jméno uživatele“, „Zadej jeho login“ a podobně, viz **Obrázek 15 - Diagram Operace nad uživateli,** což je nežádoucí.



Obrázek 15 - Diagram Operace nad uživateli

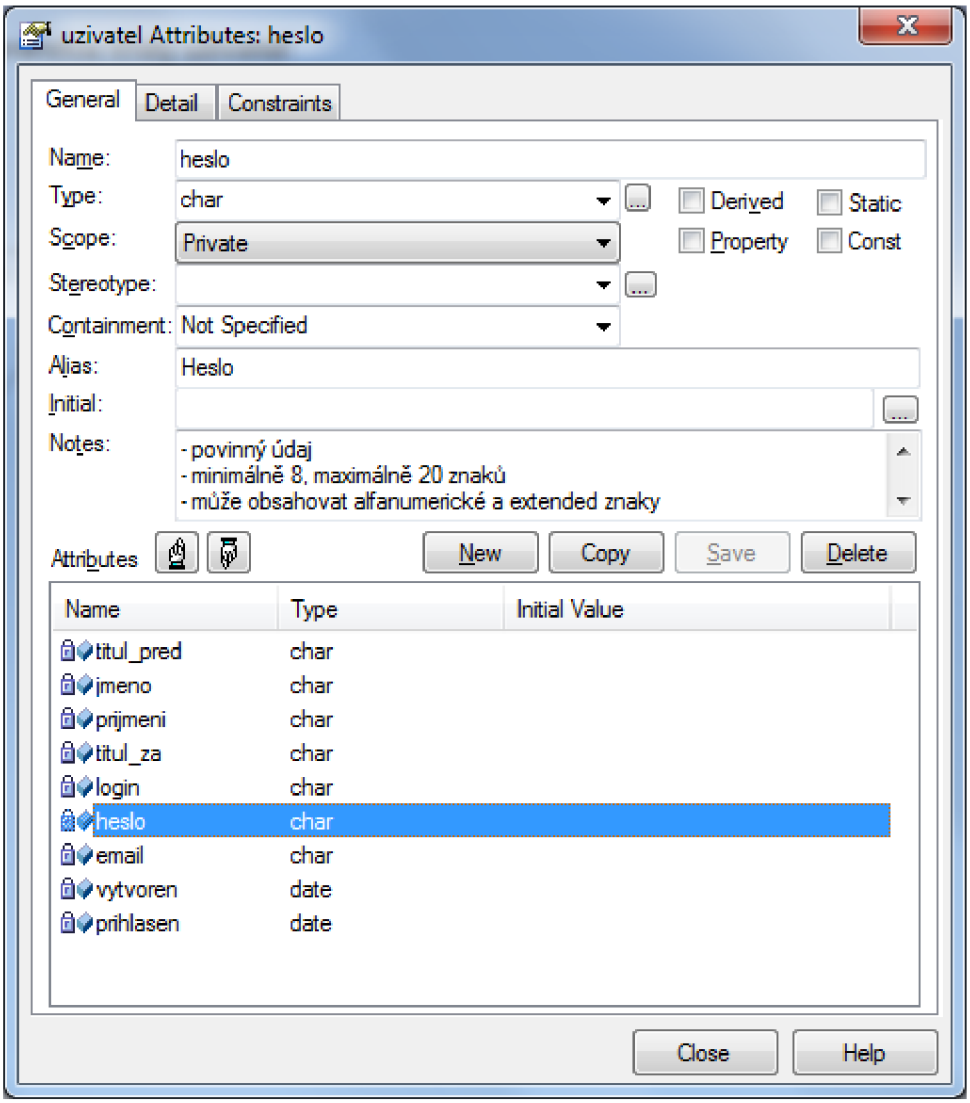
Další zásadou AMDD je jednoduchost. Proto jsem se snažil vyvarovat zanesení jakékoliv informace, která by nenalezla své uplatnění. U případů užití jsem definoval jejich stručný popis (Případ užití sloužící k vyhledání uživatele dle zadaných kritérií), vstupní a výstupní podmínky, stručný scénář spolu s jeho alternativní podobou a požadavky zákazníka, které daný UC naplňuje. Propojení případu užití či jiného elementu s požadavkem je velmi důležité. Tato vazba se promítne do výše uvedené

matice - **Matice** požadavků, kde lze velmi jednoduše sledovat, zda a kde byl konkrétní požadavek naplněn. Tuto funkcionalitu jsem uvítal především ve chvílích, kdy jsem zákazníkovi prezentoval výstup z iterace.

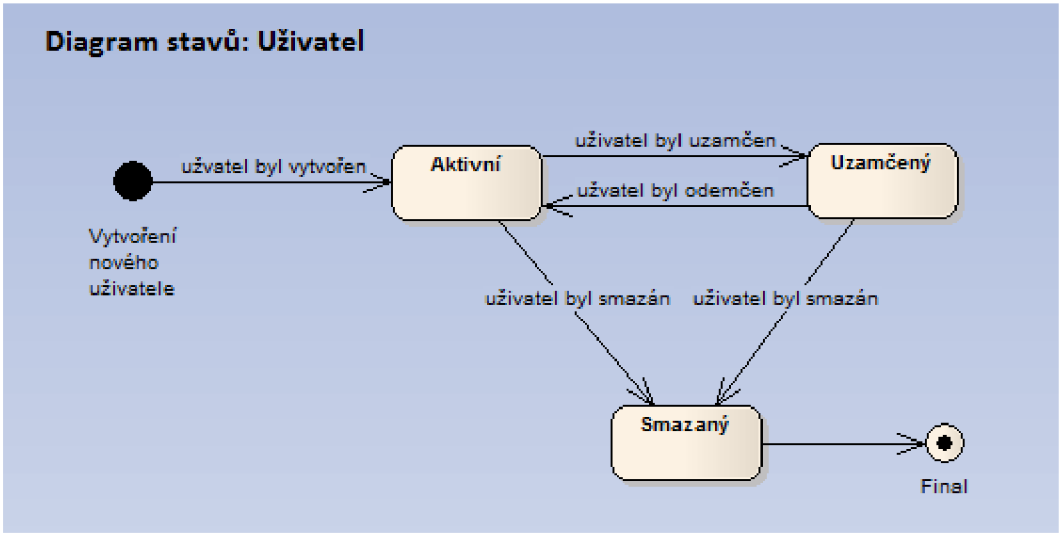
Druhým diagramem, kterému jsem se věnoval, byla „Správa uživatelů“. Zde jsem si při zpracování „UC026“ uvědomil, že by bylo vhodné se souběžně s ním věnovat i třídě „Uživatel“, která s tímto případem užití úzce souvisí.

Během prací na diagramu „Správa uživatelů“ jsem při zpracování „UC026 Vytvořit nového uživatele“, došel k závěru, že by bylo vhodné souběžně s ním zavést do modelu i jemu odpovídající třídu. Během jejího modelování jsem řadu jejích aspektů promítl nejen do výše uvedeného případu užití, ale i do ostatních souvisejících diagramů tak, aby zůstala zachována konzistence modelu. U třídy jsem mimo jiné definoval její atributy (v úzké spolupráci se zákazníkem), u kterých jsem se snažil být co nejpřesnější/nejvýstižnější (atributy představují jedny z mála věcí, které musí být precizně zaznamenány i v rámci AMDD, protože svým způsobem slouží k dokumentaci návrhu systému), jak je naznačeno na obrázku **Chyba! Nenalezen zdroj odkazů..** V ostatních částech modelu jsem se snažil vyvarovat uvedení konkrétního názvu atributu, pokud to nebylo vyloženě nezbytné, abych neztěžoval proces udržení aktuálního modelu. Kdybych toto pravidlo nedodržoval, pak bych musel každou případnou změnu atributu (změna jeho názvu, rozdělení na dva) zanést na všechna místa, kde byl zmíněn a to představuje značnou náročnost na zdroje. Proto jsem raději používal formulace typu: je možné vyhledávat dle všech systémových i nesystémových atributů uživatele. Nesystémovými atributy uživatele rozumím ty atributy, které může uživatel s patřičným oprávněním změnit (jméno, příjmení). Naopak systémové atributy jsou atributy, které zadává automaticky systém a nelze je ručně změnit (datum vytvoření uživatele, stav).

Obdobným způsobem jsem pak postupoval při modelování zbylých případů užití a diagramů. V moment kdy jsem usoudil, že práce na výše uvedených typech diagramů jsou u konce, tak jsem přemýšlel nad zavedením diagramu stavů pro třídu uživatel. Někdo by mohl namítat, že třístavový automat není nutno modelovat. Já se k tomuto kroku rozhodl poté, co jsem zjistil, že některým stakeholderům činí potíže si stavy „Uživatele“ představit. A protože mi jeho zavedení nejen že ulehčí práci při vysvětlování (obrázek je mnohdy lepší než tisíc slov), tak mi zároveň poslouží k dokumentaci řešení.



Obrázek 16 - Atributy třídy Uživatel



Obrázek 17 - Stavový diagram Uživatele

V tuto chvíli jsem měl v modelu zavedeny všechny typy diagramů, které našly v rámci iterace 1 své uplatnění. Pro diagram aktivit jsem postrádal patřičný byznys proces, pro diagram komunikace pouze jednu třídu, atd.

Než jsem však přešel k vlastnímu programování, tak jsem se rozhodl pro krok, který AMDD přímo nevyžaduje, ale podporuje. Enterprise Architect umí velmi jednoduchým způsobem převést navržené třídy na tabulky a já se rozhodl toho využít. Vytvořený diagram, návrh databáze, jsem doplnil o informace, které nebyly v třídách uvedeny a já je postrádal (primární a cizí klíče, omezení, počáteční hodnoty). Nad tímto souborem tabulek jsem spustil automatický proces, který vygeneroval jejich DDL popis v MySQL.

Programování

Dříve než jsem se pustil do psaní zdrojového kódu, tak jsem uplatnil výše vygenerované skripty a spustil je v prostředí MySQL. Vytvořené tabulky jsem pro jistotu prošel a zkontroloval s tím, že ty, které budou sloužit jako číselníky, jsem rovnou naplnil příslušnými hodnotami. Tento krok by v agilním přístupu, který je vhodný především pro větší týmy, chyběl. Architekt by totiž navrhl strukturu databáze na základě kick off meetingu se všemi členy týmu.

Začal jsem tím, že jsem vzal dodaný grafický návrh a ten jsem v programu Adobe Photoshop rozdělil na menší části, které jsem pak za pomoci kaskádových stylů upravil a dal dohromady tak, že jsem získal výchozí „šablonu“. Velkou důležitost jsem kladl ve vyvíjeném software na bezpečnost. Proto jsem v rámci svých znalostí a možností udělal maximum, abych jí dostál a šablonu o tyto prvky doplnil. Implementaci zabezpečeného přístupu jsem zajistil použitím PHP sessions. Při přihlášení uživatele dojde k porovnání hashe zadaného hesla s kódem uloženým v databázi. Nikdy tak nedochází k přenosu nezakódovaného hesla po síti. Po ověření uživatele skriptem login.php je vytvořena session, která nese informaci o přihlášeném uživateli a v databázi je vytvořen záznam o IP adrese, ze které se uživatel přihlásil. V případě, že během uživatelovy práce se systémem dojde ke změně jeho IP adresy, tak je uživatel vyzván k opětovnému zadání svého hesla. Platnost session je nastavena na 3600 sekund. Po uplynutí této doby a uživatelovy nečinnosti dojde k automatickému odhlášení uživatele.

Co se týče programování vlastní funkcionality systému, tak jsem začal přihlašovací stránkou a po ní se věnoval výchozí stránce systému, v níž jsem se zaměřil především na oblast správy uživatelů. Při psaní kódu jsem se měl na paměti metaforu dvou klobouků pana Kenta Becka, která říká, že v moment, kdy přidávám do systému novou funkcionalitu či upravuji stávající, bych se měl vyvarovat refaktorizaci a naopak. Více informací o daném tématu lze získat zde [10].

Nedílnou součástí agilního přístupu je průběžné testování. V prostředí HTML/PHP bohužel nelze dodržet test driven design. Já se však pokusil o jeho simulaci. Pokaždé, než jsem napsal konkrétní proceduru, tak jsem se zamyslel, co od ní očekávám (omezení, vstupy, výstupy) a tyto informace si bokem poznačil. Když jsem proceduru naprogramoval, tak jsem jí postupně na vstup

zadával hodnoty, které měly prověřit její funkcionalitu, rychlost či reakci na daný vstup, a výstupy porovnával s těmi očekávanými. Teprve, když jsem byl 100% spokojen, jsem přistoupil k další.

Při standardním řešení by vývojář postupoval tak, že by nejprve implementoval základní funkcionalitu dané iterace, kterou si odnesl kick off meetingu s tím, že později by již nahlížel do modelu, kde by našel zpřesnění jednotlivých případů užití a jejich souvislosti. Já měl v tomto případě výhodu v tom, že jsem byl jak analytikem, tak vývojářem a měl jsem tedy přehled o tom, jak při programování postupovat. Do modelu jsem tak nahlížel pouze minimálně. Když jsem nabyl pocitu, že mé programování je v rámci této iterace u konce, tak jsem ještě provedl hrubé otestování/proklikání systému jako celku.

Testování

V předchozím odstavci jsem zmínil, že systém byl průběžně testován již za jeho implementace. Nyní přišlo na řadu jeho důkladné otestování jako celku, po kterém by následovalo otestování integrace nově vyvinuté části do stávajícího řešení.

Standardně by testy prováděl tým profesionálních testerů, kteří nebyli zapojeni do vývoje a prováděli by tak objektivní testy. Přece jenom člověk, který se aktivně podílí na implementaci, bude lehce zaslepen (ať už tím, že něco přehlédne nebo tím, že jeho práce je „bezchybná“). I tentokrát jsem si musel vystačit sám a provedl zevrubné otestování systému. Byl jsem mile překvapen, že jsem na žádné chyby nenarazil. Ale to přikládám za důsledek toho, že tato iterace nebyla technicky náročná a mému zevrubnému testování již během vývoje.

V rámci této iterace nebylo zapotřebí provádět integrační testy, jednalo se o první release, a tak jsem je s klidným svědomím vynechal.

Release

Release jsem provedl tak, že jsem pouze zreplikoval prostředí testovacího prostředí na ostrý server. Replikace mimo jiné obnášela vytvoření kopie databáze, zkopírování zdrojových kódů a nastavení přístupu. Překlopený systém pro jistotu překontroloval.

6.4 Iterace 2

Iterace s číslem dvě, stejně jako iterace následující, byla obdobou předchozí s pouze nepatrnými rozdíly. A právě odlišnostem chci věnovat tuto kapitolu.

6.4.1 I. fáze

Hranice iterace

Tato část iterace se od té předchozí neliší postupem ale obsahem. Hranice této iterace pokrývají oblast práce se třídami (jejich naplnění žáky a třídními učiteli) a předměty (jejich definice, přiřazení učitele,

který je vyučuje). V podstatě se tedy jedná o správu tříd a předmětů, kterou bylo nutné doplnit o správu žáků a učitelů, kteří jsou nedílnou součástí výše uvedeného.

Obsah iterace jsem se snažil ve spolupráci se zákazníkem stanovit tak, aby svou délkou a náročností nepřesahoval předchozí iteraci.

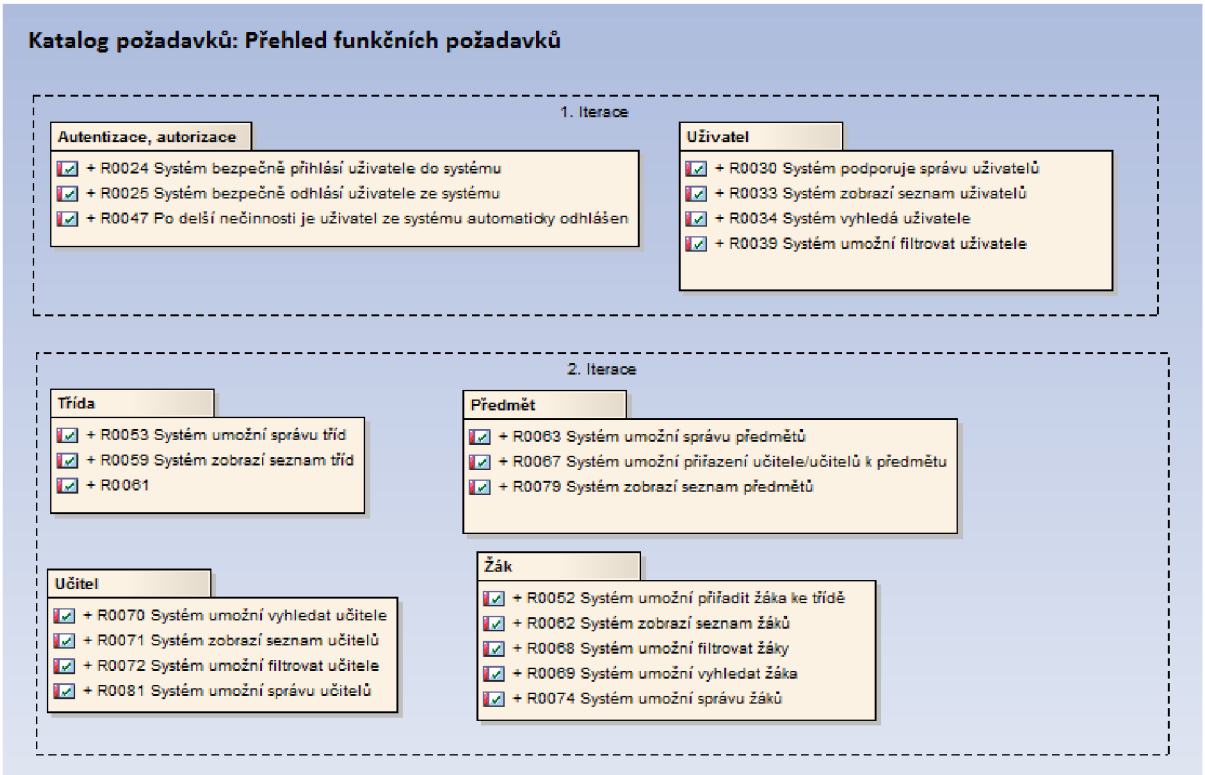
Grafický návrh

Protože byl vývojový tým s grafickým návrhem seznámen již dříve, tak nebylo zapotřebí se mu blíže věnovat. Bližší informace o umístění formulářů a tlačítek (včetně jejich stylů), týkajících se této iterace, budou součástí kick off meetingu vývojového týmu.

6.4.2 II. fáze

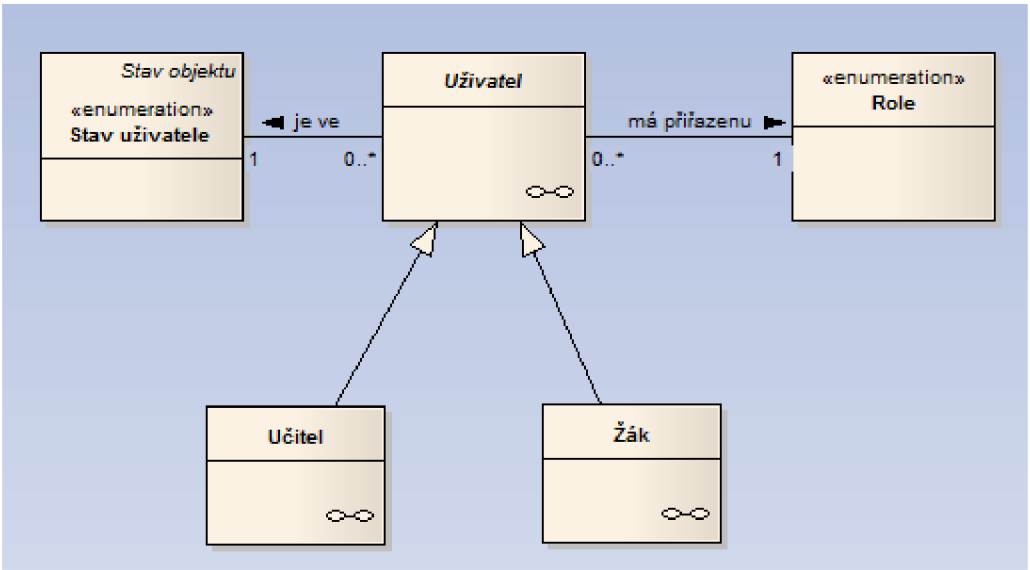
Modelování

Ani v této části se mnoho nezměnilo. Ještě před tím, než jsem se pustil do modelování, tak jsem využil možnosti, kterou nabízí EA. V globálním nastavení lze každému nově vytvořenému objektu přiradit číslo iterace, do které spadá. Tím pak zjednoduším čitelnost modelu, pokud by čtenář hledal změny týkající se pouze určité iterace. Požadavky jsem pro větší přehlednost rozlišil ještě graficky, viz **Chyba! Nenalezen zdroj odkazů.** .



Obrázek 18 - Katalog požadavků

Při modelování jsem využíval stejné postupy a prostředky jako v předchozí iteraci. Začal jsem modelováním případů užití pro žáka a učitele, sloužících pro jejich správu, které pak budu potřebovat při definici tříd a předmětů. V tomto případě jsem si pomohl generalizací. Učitel a žák jsou totiž speciálními případy uživatele (uživatel v konkrétní roli), viz **Obrázek 19 - Generalizace žáka a učitele.**



Obrázek 19 - Generalizace žáka a učitele

Stejný přístup, generalizaci, jsem použil k již nadefinovaným případům správy uživatele. Rozhodl jsem se, že absolutní generalizaci, například v „UC0064 Nastavit filtr pro uživatele“, využiji pouze v těch případech, kdy mi to prokazatelně ulehčí práci. Vedlo mě k tomu hned několik důvodů. Absolutní generalizace je pro zákazníka mnohdy těžce pochopitelná (jak jsem zjistil v průběhu iterace, když jsem mu v jejím průběhu ukazoval, jak budu postupovat). A protože jsem plánoval, že uživatelské scénáře se stanou součástí dokumentace, tak jsem se snažil o jejich co nejjednodušší čitelnost. Dalším důvodem byl fakt, že některé z nich byly tak specifické, že ji uplatnit nešlo. Například je vyvolával jiný aktér.

V scénáři případu užití jsem používal následující notaci uvedenou v knize [3].

- zděděno beze změny - 3. (3.) Zákazník zadá požadované hodnoty.
- zděděno s přečíslováním - 3.1. (2.4.) Zákazník stiskne tlačítko „Cancel“
- zděděno a překryto - 3. (03.) Zákazník stiskne tlačítko „Zveřejnit“
- zděděno, překryto a přečíslováno - 3.1 (03.5) Zákazník se odhlásí ze systému
- přidána - 6.3 Zákazník zadá nepovinné hodnoty

Obdobným způsobem jsem pak přistupoval k předmětům a třídám, kdy jsem souběžně modeloval jejich případy užití a odpovídající třídy.

Programování a testování

Programování a testování bylo tentokrát poněkud složitější. Musel jsem zohlednit již implementované části a přizpůsobit se jim. Navíc, jak se řešení stávalo rozsáhlejším a složitějším, tak jsem mu musel přizpůsobit i testovaní. Nyní již nestačilo otestovat chování objektu pouze na jedné stránce, ale bylo nutné prověřit jeho funkcionalitu ve „všech" částech systému, kde se vyskytoval.

Ani tentokrát jsem neprovedl integrační testy. A to z toho důvodu, že novou funkcionalitu jsem přidával přímo do stávajícího řešení.

6.5 Dodržel jsem zásady agilního přístupu?

Během celého vývoje software jsem měl na paměti základní hodnoty, principy a praktiky agilního přístupu. V průběhu prací na projektu jsem aktivně komunikoval se zákazníkem, přes jeho kontaktní osobu (SPOC) ať už to bylo ve fázi příprav, analýzy či programování. I přes to, že jsem se snažil v analýze obsáhnout všechny nezbytné informace pro danou iteraci, tak se mi několikrát stalo, že jsem byl nucen požádat zákazníka o dodatečné informace i v průběhu programování. Jednou mi chybějící informace zamezila pokračovat v programování. „Získaný" čas jsem investoval do revize kódu a testování. Při komunikaci jsem bral zákazníka jako rovnocenného partnera a jeho názor měl pro mě patřičnou váhu. I přesto jsem se nebál mu v rámci možností oponovat a upozornit jej tak, na možná rizika jeho rozhodnutí. On mi na oplátku poskytoval zpětnou vazbu, co se mých rozhodnutí týče. Hodnoty agilního přístupu (komunikace, jednoduchost, odvaha, pokora a zpětná vazba) jsem tak do jedné respektoval.

Nejenom při tvorbě modelu jsem dbal na snadnou čitelnost zanesených informací, která jde ruku v ruce s maximální možnou jednoduchostí. Kdykoliv jsem ukládal nějakou informaci, ať už do dokumentů, modelu či kódu, tak jsem zvážil její nezbytnost/přidanou hodnotu. A právě tento přístup zamezil tomu, aby získané podklady nabyly robustních rozměrů. Díky jejich přiměřenému množství jsem pak měl i méně práce s jejich údržbou, doplněním či změnou. V průběhu modelování i vývoje jsem se několikrát dostal do situace, kdy jsem musel zanesené informace pozměnit. Činil jsem tak pouze vždy, když to bylo nezbytné a byla k tomu příhodná doba. Například na začátku druhé iterace jsem počítal s tím, že číselník stavů bude využívat pouze třída uživatel. V jejím průběhu jsem však situaci přehodnotil a zjistil, že tento číselník bude sloužit více objektům (třída, předmět) a jeho využití je nevyhnutelné. Agilní přístup standardně říká, že za běhu iterace již požadavky měnit nelze, a „chybu" je tak možné napravit až na začátku následující iterace, kdy se změnovému požadavku přiřadí nejvyšší priorita. Já usoudil, že v tomto případě se jedná o klíčovou funkcionalitu a svou chybu jsem napravil ještě za běhu iterace, tak jak to agilní přístup ve výjimečných případech umožňuje. Ostatní nesrovnalosti jsem již řešil standardní cestou.

UML nabízí celou řadu diagramů a je pouze na analytikovi, pro které z nich najde uplatnění ve svém modelu. Já využil diagram případů užití, diagram tříd, návrh databáze a stavový diagram s tím, že mezi jednotlivými diagramy jsem v průběhu modelování přecházel, abych zachoval jejich provázanost. Všechny diagramy pak společně tvořily celek, který byl propracován do takové míry detailu, jak bylo v danou chvíli zapotřebí a žádná informace v něm uvedená nebyla redundantní. Diagramy tak plnily svůj účel na 100%.

Při psaní kódu jsem dbal na jeho snadnou čitelnost, maximální funkčnost a jednoduchou rozšiřitelnost s tím, že refaktorizaci jsem bral jako přirozenou součást implementace.

I přes to, že agilní modelování řízené modelem přesnou formulaci scénářů případů užití nevyžaduje, já se pro tento krok rozhodl. Z vlastní zkušenosti totiž vím, že scénář případu užití je pro zákazníka lehce čitelný a on si na jeho základě dokáže snáze představit fungování systému. Ze scénářů případů užití jsem vycházel i při tvorbě testovacích scénářů (test casů), takže se dá říci, že jsem je maximálně využil.

Během celého procesu vývoje software jsem efektivně hospodařil s dostupnými prostředky, především pak s časem. Pokud například nastala situace, že z důvodu chybějících informací ze strany zákazníka jsem nemohl pokračovat v modelování, tak jsem využil získaný čas k otestování již naimplementované části systému. Jedním z mých cílů bylo odvádět kvalitní práci tak, abych na ní mohl být pyšný nejenom před zákazníkem, ale i před sebou samým.

Informace vztažené k projektu jsem udržoval na jednom místě, v mém případě ftp úložišti, kam měli povolen přístup všichni účastníci projektu (stakeholdeři, zákazník i vývojový tým). Podklady k právě probíhající iteraci byly umístěny odděleně od archivu, do kterého byly pravidelně přesouvány neaktuální dokumenty, například dokumenty z předchozí iterace.

Jak je z výše uvedených skutečností patrné, pilíře agilního vývoje řízeného modelem (hodnoty, principy, praktiky) jsem měl stále na paměti a dodržoval je.

7 Závěr

7.1 Zhodnocení práce

Cílem této diplomové práce bylo seznámit se s tématikou agilního vývoje řízeného modelem, proniknout do ní a ukázat její praktické využití v praxi.

Než jsem tak učinil, tak jsem uvedl posluchače do problematiky softwarového inženýrství, aby pochopil smysl a uplatnění této disciplíny, orientoval se v modelech životního cyklu software (obzvláště pak v jeho dvou relativně nových zástupcích a to architektuře řízené modelem a agilním vývoji) a měl představu o unifikovaném modelovacím jazyce UML, jehož osvojení si je klíčové především pro pochopení modelu vyvíjeného systému, který je součástí příkladové studie.

Stěžejní pro tuto diplomovou práci je oblast agilního vývoje software, kterou jsem zevrubně prostudoval a základnímu pohledu na ni věnoval jednu celou kapitolu. Neopomněl jsem uvést základní hodnoty, principy a praktiky agilního přístupu tvořící jeho pilíř, spolu s netypickými iteracemi.

Získané znalosti jsem podrobil zatěžkávací zkoušce ve formě příkladové studie, která je věnována procesu vývoje informačního systému od zákazníkovy první zmínky o něm až po jeho odevzdání (po několika iteracích) zákazníkovi. Informační systém je určen pro základní a střední školy, a plně reflektuje jejich potřeby. V prvních dvou řádných iteracích (řádnou iterací rozumím iterací, kdy došlo k implementaci dané funkcionality), na kterých jsem demonstroval získané znalosti a dovednosti, jsem se zaměřil na oblast autorizace a autentifikace, která s sebou nese i oblast správy uživatelů; dále pak na správu žáků, učitelů, tříd a předmětů.

7.2 Zhodnocení agilního přístupu

Agilní přístup zastoupený agilním vývojem řízeným modelem se z mého pohledu osvědčil. V předchozím zaměstnání, kde jsem pracoval na pozici junior analytika, jsem měl tu čest se lépe seznámit s jiným modelem životního cyklu a to vodopádem. Ten v praxi vykazoval všechny neduhy, které jsou mu právem vytýkány – nemožnost měnit zákazníkovy požadavky za běhu (což mělo na následek, že jsme zákazníkovi nakonec předložili systém, který naplňoval jeho potřeby ze 70% a následně se jeho funkcionalita vylepšovala pomocí víceprací ve formě požadavků na změnu (change requestů)), příliš robustní model systému, kterému byla dána až přehnaná důležitost a váha (občas bylo možné provést některé úkony jednodušeji a elegantněji), a tak dále.

Myslím si, že právě díky této předchozí zkušenosti dokážu ocenit agilní přístup, který mi plně vyhovoval. Ať jsem se nacházel v počáteční fázi, kdy jsme se zákazníkem definovali jeho představu o systému, nebo během vlastní implementace.

Výhody agilního přístupu jsem pak nejvíce ocenil v průběhu iterace. Líbí se mi, že během počátečního kick off mítinku získají všichni účastníci vývojového týmu ucelenou představu o náplni iterace a mohou začít bezprostředně pracovat na oblastech, za které jsou odpovědní. Přináší to nejenom úsporu prostředků, především pak času, ale dává to prostor všem členům týmu, kteří tak mohou prokázat své dovednosti a přijít s vlastním návrhem řešení. U vodopádu spočívá návrh především na analytikovi, který ostatním účastníkům „nadiktuje" výslednou podobu řešení a oni tak mají pouze minimální šanci jej pozměnit. Dále se mi líbila efektivní práce s požadavky, která umožní skoro okamžitě reflektovat požadované změny, nejpozději pak v další iteraci. Velkým přínosem je i efektivní hospodaření s prostředky. Není nezbytně nutné mít naprosto „úžasnou" dokumentaci či model. Vynaložené úsilí je pěkně rozprostřeno mezi všechny části projektu tak, jak je to pro jeho potřeby nejvhodnější. Agilní vývoj řízený modelem přikládá velkou váhu testování, kdy je programátor odpovědný za část kódu, který implementoval a svým způsobem tak ručí za kvalitu. V jiných přístupech jsem se spíše setkával s tím pohledem, že na hledání chyb jsou určeni testeři a programátor tak není povinen ani zběžně zkontrolovat svou práci. Ve vyjmenování kladů bych mohl pokračovat i nadále, protože se netajím tím, že mě agilní vývoj řízený modelem oslovil. Za nevýhodu považuji fakt, že není možné s větší přesností určit, jak dlouho budou práce na systému pokračovat (zákazník může rozsah systému v důsledku jeho nových potřeb několikanásobně rozšířit nebo i zmenšit) a pro zaměstnavatele vývojového týmu je tak nesnadné naplánovat firemní zdroje do budoucna. Na druhou stranu se v ideálním případě může jednat o dlouhodobou spolupráci, která zajistí firmě příjem na dlouhý čas, protože požadavky kladené na systém se budou v průběhu času jistě měnit.

7.3 Možná rozšíření v budoucnosti

Nespornou výhodou agilního vývoje řízeného modelem je i problematika, které je věnována tato kapitola. Samotný iterativní přístup značí, že s možnými rozšířeními se počítá a jejich implementace tak nebude problematická. Co se týče možných rozšíření systému, tak po stránce bezpečnostní bych doporučoval SSL komunikaci s tím, že po stránce funkční je zde prostor pro: automatické sestavování rozvrhu, doplnění systému o možnost objednávání obědů a správu elektronického účtu, propojení systému se systémem spisové služby, možnost tisku vysvědčení a podobně. Nepatrnou, ale pro uživatele nejvíce viditelnou změnou, je změna grafického provedení systému.

8 Literatura

[1] SWEBOK executive editors, Alain Abran, James W. Moore ; editors, Pierre Bourque, Robert Dupuis. (2004). Pierre Bourque and Robert Dupuis. ed. *Guide to the Software Engineering Body of Knowledge - 2004 Version*. IEEE Computer Society. p. 1–1. ISBN 0-7695-2330-7. http://www.swebok.org.

[2] ZENDULKA, Jaroslav, BARTÍK, Vladimír, KVĚTOŇOVÁ, Šárka. Analýza a návrh informačních systémů. *Studijní opora* [online]. 2006 [cit. 2010-01-01].

[3] ARLOW, Jim, NEUSTADT, Ila. *UML 2 a unifikovaný proces vývoje aplikací : Průvodce analýzou a návrhem objektově orientovaného softwaru*. [s.l.] : [s.n.], 2007. 567 s

[4] PATTON, Ron. *Testování softwaru*. I. [s.l.] : [s.n.], 2002. 313 s. ISBN 80-7226-636-5.

[5] AMBLER, Scott W.. *Agile Modeling (AM)* [online]. 2001-2009 [cit. 2010-01-01]. Dostupný z WWW: <http://www.agilemodeling.com>.

[6] STEIN, René. Návrh aplikací v jazyce UML. *Interval* [online]. 2004 [cit. 2010-01-01]. Dostupný z WWW: <http://www.interval.cz>

[7] ŠMÍD, Vladimír. *Životní cyklus informačního systému* [online]. neuvedeno [cit. 2010-01-01]. Dostupný z WWW: <http://www.fi.muni.cz/~smid/mis-zivcyk.htm>.

[8] KADLEC, Václav. *Agilní programování : Metodiky efektivního vývoje softwaru*. I. [s.l.] : [s.n.], 2004. 278 s. ISBN 80-251-0342-0.

[9] POSTLER, Milan. *MANDK* [online]. 2007 [cit. 2011-05-23]. Marketing, udržitelný rozvoj a společenská odpovědnost firem. Dostupné z WWW: <http://www.mandk.cz/rservice.php?akce=tisk&cisloclanku=2007080006>.

[10] FOWLER, Martin. *Refaktoring : Zlepšení stávajícího kódu*. I. [s.l.] : [s.n.], 2003. 381 s.