

MASARYKOVA UNIVERZITA V BRNĚ
FAKULTA INFORMATIKY



Agilní metodiky vývoje software

DIPLOMOVÁ PRÁCE

Bc. Tomáš Hajdin

Brno, květen 2005

Prohlášení

Prohlašuji, že tato práce je mým původním autorským dílem, které jsem vypracoval samostatně. Všechny zdroje, prameny a literaturu, které jsem při vypracování používal nebo z nich čerpal, v práci řádně cituji s uvedením úplného odkazu na příslušný zdroj.

V Brně dne 18. 5. 2005

Poděkování

Rád bych zde poděkoval Ing. Václavu Kadlecovi za uvedení do problematiky agilního programování, Mgr. Barboře Zimmerové za její cenné rady a zkušenosti při realizaci této práce a svým rodičům za podporu během celého mého studia.

Shrnutí

Agilní metodiky vývoje software jsou v poslední době rychle se rozvíjejícím odvětvím softwarového inženýrství, kterému mnozí odborníci předpovídají slibnou budoucnost. Agilní pojetí vývoje se snaží o rychlé dodání kvalitního produktu, ale ponechává velký prostor pro flexibilitu a přizpůsobivost vývojového procesu. Práce uvádí do problematiky agilního vývoje software a blíže se zaměřuje na metodiky FDD, SCRUM a TDD. Pro tyto metodiky je uveden hlubší popis, případové studie a jejich srovnání.

Klíčová slova

agilní metodiky, FDD, SCRUM, TDD, vývoj software

Abstract

Agile software development methodologies are recently the fast evolving part of software engineering which is prognosticated a hopeful future by lots of experts. Agile development approach aims to rapid delivery of a high-quality product but leaves room for flexibility and adaptability of a development process. This thesis introduces agile software development and focuses in FDD, SCRUM and TDD. For these methods is more detailed description, case studies and comparison included.

Keywords

agile methods, FDD, SCRUM, TDD, software development

Obsah

1	Úvod	1
1.1	Struktura písemné práce	2
2	Tradiční metodiky	3
2.1	Vodopádový model	3
2.2	Spirálový model	5
2.3	Rational Unified Process	7
3	Agilní metodiky	9
3.1	Co jsou to agilní metodiky	9
3.2	Manifest agilního programování	12
3.3	Stručný přehled nejběžnějších agilních metodik	13
3.4	Feature-Driven Development	16
3.4.1	Charakteristika	16
3.4.2	Základní pojmy	17
3.4.3	Role	18
3.4.4	Procesy	20
3.4.5	Praktiky	24
3.4.6	Závěr	27
3.5	SCRUM Development Process	27
3.5.1	Charakteristika	27
3.5.2	Základní pojmy	28
3.5.3	Role	30
3.5.4	Procesy	31
3.5.5	Praktiky	33
3.5.6	XP@Scrum	34
3.5.7	Závěr	35
3.6	Test-Driven Development	35
3.6.1	Charakteristika	36
3.6.2	Procesy	37
3.6.3	Praktiky	39
3.6.4	Závěr	40

4	Případové studie	41
4.1	Ilustrační projekt	41
4.2	Řešení projektu dle zvolených metodik	43
4.2.1	Řešení podle FDD	43
4.2.2	Řešení podle SCRUM	49
4.2.3	Řešení podle TDD	53
4.2.4	Srovnání jednotlivých řešení	57
5	Závěr	59
	Literatura	61
A	Prvotní specifikace	63
A.1	Systém správy externích lidských zdrojů (SEZ)	63
B	Změny ve specifikaci systému	68
B.1	Dodatek k prvotní specifikaci	68
B.2	ERD diagram	71
B.3	Use-case diagram	71

Seznam obrázků

2.1	Schéma životního cyklu typu Vodopád	4
2.2	Schéma životního cyklu typu Spirála	6
3.1	Rozdíl tradičního a agilního pojetí vývoje software	10
3.2	Spektrum plánovacích metod	11
3.3	Posloupnost vývojových fází v metodice FDD	21
3.4	Dynamické týmy v metodice FDD	24
3.5	Rozložení času do vývojových fází metodiky FDD	25
3.6	Ukázka Product Backlogu	29
3.7	Ukázka Sprint Backlogu	29
3.8	Schéma metodiky SCRUM	30
3.9	Proces metodiky SCRUM	31
3.10	Průběh jednoho sprintu metodiky SCRUM	33
3.11	Schéma vývoje podle metodiky TDD	37
3.12	Časový průběh vývoje podle metodiky TDD	38
4.1	Rozdělení rolí v týmu	44
4.2	Diagram tříd	45
4.3	Sekvenční diagram	49
4.4	Pokrytí fází projektu jednotlivými metodikami	58
B.1	ERD diagram	71
B.2	Use-case diagram	72

Kapitola 1

Úvod

Oblast vývoje software a obecně celé softwarové inženýrství jde stále kupředu mílovými kroky. Už od doby, kdy spatřily světlo světa první počítače, bylo nutné vyvíjet software, který by z jednoúčelového stroje udělal univerzálního pomocníka. Software je to, co dnes dělá počítač počítačem. Přitom právě kvalita software je mnohdy tím, co určuje výslednou cenu celého počítačového systému. Aby byl software kvalitní, je nutné použít kvalitní vývojový proces. V důsledku toho je kvalitě vývoje software věnována patřičná pozornost a jsou do ní investovány nemalé prostředky.

V posledních pár letech si řada odborníků v této oblasti přestala jen stěžovat na nevýhody stávajících metod vývoje software, ale začala podnikat kroky ke zlepšení, zrychlení a zkvalitnění vývojového procesu. Zpočátku to bylo převážně na úrovni firem, ve kterých daní lidé pracovali. Postupem času, když se jejich nové metody osvědčily v praxi, byly představeny veřejnosti. Postupně se začaly objevovat další a další přístupy a metodologie, které ač pocházely od různých tvůrců, měly překvapivě hodně společného. Využití inkrementálního přístupu, rychlý vývoj v různě dlouhých cyklech, časté dodávky betaverzí, těsný kontakt s klienty a uživateli, důležitost zpětné vazby, snaha omezit byrokracii v procesu vývoje a zvýšení efektivity a výsledné produktivity. To všechno jsou principy, které se s různě velkou intenzitou objevují snad ve všech těchto nových přístupech. Přístupech, které byly nazvány nejprve *lehké* (lightweight), později pak *agilní* (agile).

Cílem této práce je prostudovat principy a funkce těchto metodik, pochopit jejich účel a strategii a podat ucelený pohled na jejich použitelnost. Vzhledem k tomu, že agilních metodik je mnoho a rozsah práce mi nedovoluje věnovat se podrobně každé z nich, vybral jsem tři zástupce, které popíšu detailněji, ostatní pouze stručně představím. Danými zástupci budou Feature-Driven Development, SCRUM Development Process a Test-Driven Development. Aby ovšem práce nebyla jen čistě teoretická, nadefinuji ilustrační projekt vývoje software, na kterém ukážu, jak tyto tři metodiky k vývoji přistoupí. Na závěr se pokusím o jejich srovnání.

1.1 Struktura písemné práce

Práce je rozdělena do pěti kapitol. V první kapitole je rozebráno oficiální zadání a stanoveny cíle práce. Druhá kapitola se věnuje stručnému popisu tradičních metodik. Principy agilního vývoje a popisy jednotlivých metodik jsou náplní třetí kapitoly. Jsou zde také podrobně rozpracovány tři vybraní zástupci agilních metodik. Čtvrtá kapitola obsahuje případové studie, kde je nejprve zadefinován ilustrační projekt, na kterém je následně předveden postup řešení třemi metodikami teoreticky popsány v předcházející kapitole. Na závěr kapitoly je uvedeno srovnání jednotlivých přístupů. Pátá kapitola je pak věnována závěru, zhodnocení perspektiv agilního přístupu k vývoji software a možnostem dalšího výzkumu v této oblasti. K práci jsou připojeny dvě přílohy. Příloha A obsahuje dokument prvotní specifikace a příloha B pak jeho následnou úpravu. Oba dokumenty jsou součástí definice ilustračního projektu.

Kapitola 2

Tradiční metodiky

Pro lepší pochopení důvodů, které vedly ke vzniku agilních metodik, si nejprve představme tradiční metodiky životního cyklu software. Poskytne nám to lepší vhled do problematiky a pochopení důležitých historických souvislostí. Problémem tradičních metodik začala být jejich přílišná složitost a malá flexibilita, takže nestačily reflektovat požadavky nově přicházejících projektů. Tradiční metodiky přestaly být vhodné pro malé projekty a malé týmy kvůli přílišné „byrokratické zátěži“. Příliš často vyžadovaly striktní dokumentaci a nejrůznější revize, inspekce a schvalování. Ani postupné vylepšování těchto metodik nebylo schopno vyřešit zmíněné problémy. Bylo tedy nutné udělat v koncepci vývoje software radikální změnu. Tou změnou bylo právě agilní pojetí vývoje software, kterému se budu věnovat v kapitole 3.

Z tradičních metodik jsem pro ukázkou vybral tři zástupce, kteří znamenali milníky v historii softwarového inženýrství. Nyní tedy krátce představím vodopádový model, spirálový model a Rational Unified Process.

2.1 Vodopádový model

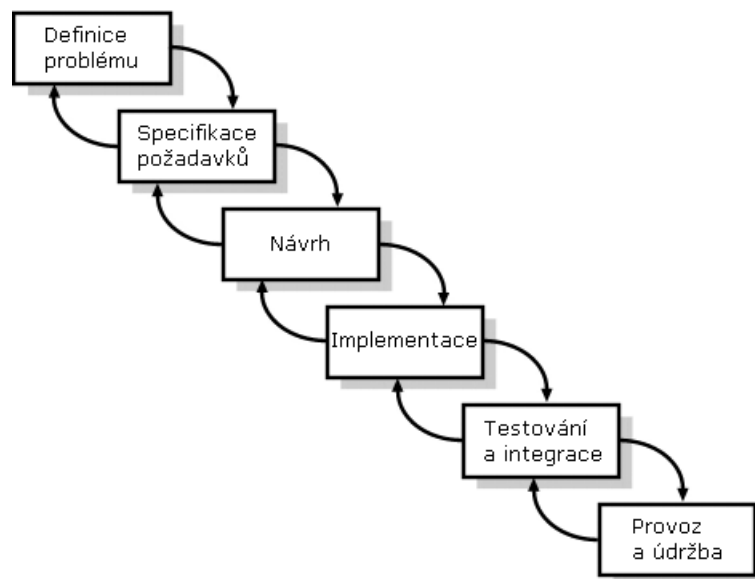
Vodopádový model je všeobecně považován za první ucelenou metodiku vývoje software. Jeho charakteristickým znakem jsou sekvenčně seřazené fáze bez iterací. Mezi jednotlivými fázemi je schvalovací proces, přes který musí všechny dokumenty projít, aby vývoj mohl pokročit do další fáze.

Základní fáze vodopádového modelu jsou následující:

1. Definice problému, poznání zákazníka a cílové oblasti
2. Analýza a specifikace požadavků
3. Návrh

4. Implementace
5. Testování a integrace
6. Provoz a údržba

Schéma posloupnosti jednotlivých fází je znázorněno na obrázku 2.1. Různých variant vodopádového modelu je nespočet, tato pochází z [18].



Obrázek 2.1: Schéma životního cyklu typu Vodopád

Mluvil jsem sice o tom, že vodopádový model je striktně sekvenční, ale z obrázku je patrné, že šipky nevedou jen jedním směrem. V modelu se můžeme pohybovat vždy o jednu fázi dopředu nebo zpět. Když například ve fázi implementace zjistíme, že jsme udělali chybu v návrhu, můžeme se vrátit o jednu fázi zpět, opravit návrh a vrátit se opět do fáze implementace. Při přechodu od návrhu k implementaci (byť je to už podruhé) je důležité, aby všechny dokumenty prošly schválením.

Teprve ve fázi údržby se může vývoj vrátit do kterékoliv z předchozích fází a umožnit tak provedení jakýchkoliv úprav. To je ale z dnešního pohledu nevhodné, protože z praxe víme, že změny v požadavcích nastávají zpravidla dříve než ve fázi údržby, navíc někdy není seznam požadavků dostupný v kompletní formě ani v době, kdy probíhá fáze jejich specifikace. Při vývoji podle tohoto modelu je komunikace se zákazníkem potřeba jen v úvodu projektu při specifikaci požadavků, a pak až v závěru při předání a ve fázi údržby. To je také jeden z rysů, od kterého se dnes snažíme upustit a zavést komunikaci se zákazníkem jako jednu ze stěžejních činností během celého vývojového procesu.

Kromě toho, že vodopádový model byl prvním modelem, který jednoznačně definoval posloupnost vývojových fází, je jeho nespornou výhodou jednoduchost. Model

je jednoduchý jak na pochopení a práci podle něj, tak také na řízení. Každá fáze je zakončena zpracováním předem daných dokumentů, přechody mezi fázemi zajišťuje schvalovací řízení. Lze tak dobře kontrolovat, v jakém stádiu se projekt právě nachází a jaká část specifikace požadavků je již splněna.

Seznam nevýhod je daleko větší. Je to dáno především stářím modelu a změnou postoje společnosti k vývoji software. Stejně jako byla jednoduchost vodopádového modelu výhodou, je zároveň i nevýhodou. Pro malé projekty je výhodné, že je model jednoduchý, ale pro větší už je to překážka, která brání pokrýt veškeré aspekty rozsáhlých projektů. Vývoj podle vodopádu není vůbec flexibilní, veškerý vývoj probíhá podle striktně naplánované posloupnosti fází. Pokud v průběhu implementace zjistíme nový požadavek, musíme znovu projít fázemi specifikace požadavků, analýzy a návrhu, včetně schválení všech dokumentů. To vývoj velice zdržuje v případě, kdy se dodatečné požadavky objevují postupně až v pozdních fázích projektu. Neposlední nevýhodou je to, že zákazník po celou dobu vývoje neví, co vlastně vývojový tým dělá, v jaké fázi je vývoj a kdy bude projekt dokončen. S oblibou je pro tento způsob dodání požíván pojem „velký třesk“. Přitom případů, kdy je zákazník po dodání v dohodnutém termínu zcela spokojen s výsledkem je velice málo. Smutné, leč pravdivé. Nejhorší je situace, kdy analytik při sbírání podkladů k vytvoření specifikace špatně pochopí zákaznickova slova a ten pak po několika měsících vývoje a nemalém množství vložených finančních prostředků zjistí, že nedostal to, co původně očekával.

Ve své době byl vodopádový model revoluční a vznikla podle něj spousta softwarových produktů. Dnes se ale spíše používá jako referenční model pro srovnávání s ostatními modely, nebo jako příklad toho, jak se při vývoji postupovat nemá. Pro současné projekty je lepší použít některou z metodik, která reflektuje soudobé trendy ve vývoji software.

Další informace o vodopádovém modelu lze nalézt v [16] a [26].

2.2 Spirálový model

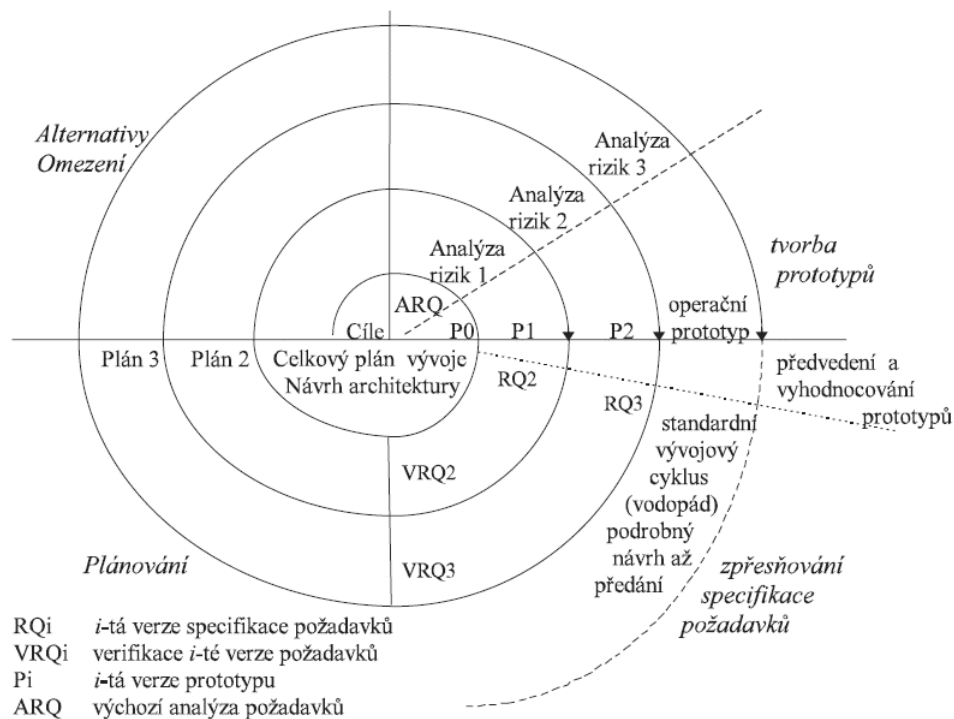
Vodopádový model popsáný v předchozí kapitole brzy po svém vzniku přestal být vyhovující. Proto se softwaroví inženýři snažili vymyslet nový přístup, který by odstranil nedostatky vodopádu. Stalo se tak v roce 1985, kdy Barry Boehm přišel na svět se svým spirálovým modelem. Jak v případě spirály, tak i v případě vodopádu mluvím stále o modelu, protože to nejsou metodiky vývoje software, ale modely životního cyklu softwarového produktu. Rozdíl je ten, že model životního cyklu popisuje fáze, kterými softwarový produkt prochází, kdežto metodika určuje, co se v které fázi má přesně dělat, jaké postupy se mají použít a k jakému výsledku se má nakonec dojít.

Spirálový model vychází z modelu vodopádového, ale přináší dvě zcela nové a zásadní vlastnosti – iterativní přístup a podrobnou analýzu rizik. Proto se také

někdy říká, že patří do skupiny přístupů řízených riziky (risk-driven approach). Iterativní přístup byl důsledkem toho, že u spousty projektů se nepodařilo stanovit přesnou a úplnou specifikaci požadavků, což činilo vodopádový model takřka nepoužitelným. Ukázalo se, že bude mnohem lepší stanovit na počátku jen rámec architektury a funkčnosti celého systému a ten postupně rozpracovávat do detailů. Cyklické opakování jednotlivých kroků vývoje, tzv. iterace, znamenalo ve své době přelom v chápání životního cyklu.

Kromě iterativního postupu je v této souvislosti druhým důležitým pojmem analýza rizik. Ta je prováděna v každém cyklu a určuje další směr vývoje projektu. *Riziko* je tedy klíčovým pojmem, pod kterým se rozumí jakákoliv událost nebo situace, která může ohrozit projekt. Při analýze je nutno každému riziku přiřadit jeho nebezpečnost a pravděpodobnost výskytu. Častá a podrobná analýza rizik má za úkol dostatečně dopředu odhalit nevhodná řešení nebo skryté problémy, které by mohly ohrozit průběh projektu.

Schéma spirálového životního cyklu je zakresleno na obrázku 2.2 pocházejícího z [12].



Obrázek 2.2: Schéma životního cyklu typu Spirála

Hlavní výhodou spirálového modelu je podle autora nezávislost na konkrétní metodice, která se při samotném vývoji použije. Dalšími dobrými vlastnostmi je komplexnost, díky které se tento model hodí i pro větší projekty. Model díky neu-

stálé analýze rizik výrazně snižuje možnost nevhodného řešení. Pokud by se chtěl vývoj ubírat špatným směrem, chyba se odhalí daleko dříve než tomu bylo u jeho předchůdce.

Jak jsem uvedl, model je komplexní, což nahrává větším projektům, ale u menších je to právě naopak. Po malé projekty se nehodí, protože je pro ně zbytečně příliš komplikovaný. Nevýhodou, kterou model zdědil od vodopádového modelu je to, že výsledný produkt je předán až po dokončení posledního cyklu. V každém cyklu je sice vytvořen prototyp, ale ten se může obecně týkat různě malých částí systému a nemusí být použitelný v ostrém provozu. Je zde tedy zavedena jakási zpětná vazba, ale problém „velkého třesku“ to neřeší. Nakonec musím zmínit i to, že model zcela spoléhá na metodiku, která se při vývoji použije a nijak podrobně nezpracovává jednotlivé činnosti, které by se měly v průběhu vývoje provádět.

V softwarovém inženýrství je spirála důležitým milníkem a v minulosti byla použita k řadě projektů. V současné době se už ale používá stále méně. Jednak proto, že jsou na světě daleko propracovanější metodiky, ale také proto, že pro současné typy aplikací je spirálový model těžkopádný a nepružný.

Pro další studium doporučuji zdroje [16], [27], [28] nebo [6].

2.3 Rational Unified Process

Rational Unified Process (dále jen RUP), jenž je komerčním produktem firmy Rational (která byla později koupena firmou IBM), je rozsáhlá a detailně propracovaná metodika vývoje software. Celá metodika je objektově orientovaná a patří do skupiny přístupů řízených případy užití (use-case-driven approach). Vývoj podle RUP probíhá v iterativních cyklech. První cyklus se nazývá *úvodní vývojový cyklus* a jeho výsledkem je funkční softwarový produkt implementující nejpodstatnější část funkcionality. Zde ovšem vývoj nekončí, ale pokračuje v různém množství *rozvíjejících cyklů*. Jejich přesný počet závisí na rozsahu projektu.

Každý cyklus se skládá ze 4 fází (v závorce je uvedeno typické rozdělení doby pro jednotlivé fáze úvodního vývojového cyklu):

1. Zahájení (10%)
2. Projektování (30%)
3. Realizace (50%)
4. Předání (10%)

RUP je dodávána ve formě internetových stránek, které slouží jako online instruktor, který vývojáře vede celým průběhem vývojového procesu, poskytuje instrukce,

radly, metodické pokyny a příklady ke konkrétní fázi vývoje. RUP je těsně provázán s jazykem UML. Všechny dokumenty, modely a případy užití jsou modelovány právě v UML.

Největší výhodou RUP je jeho robustnost, obecnost a přizpůsobivost pro celou řadu nejrozličnějších projektů. Jedna z úvodních fází při použití RUP je modifikace samotné metodiky na míru konkrétnímu projektu, který začínáme realizovat. Není problém upravit metodiku ani pro malý tým a jednodušší projekt. Součástí metodiky jsou nejrozličnější průvodci, šablony a podpůrné nástroje.

Díky své rozsáhlosti je RUP přece jen určen spíše větším týmům pro realizaci velkých projektů. Zvládnutí metodiky vyžaduje hlavně v úvodních fázích poměrně hluboké studium a dobře trénovaný vývojový tým. Sama firma Rational na rozvoji metodiky stále pracuje, stejně tak je dostupná řada dokumentů podpůrných nástrojů třetích stran, které se zabývají RUP.

Další podrobnosti k tomuto tématu jsou k nalezení v [13] nebo [29].

Kapitola 3

Agilní metodiky

Nyní se dostávám k jedné z nejdůležitějších kapitol této práce. Nejprve vysvětlím, co vlastně agilní programování je, zmíním se o *Manifestu agilního programování*, uvedu stručný přehled nejznámějších agilních metodik a nakonec vybrané tři z těchto metodik popíšu detailněji.

3.1 Co jsou to agilní metodiky

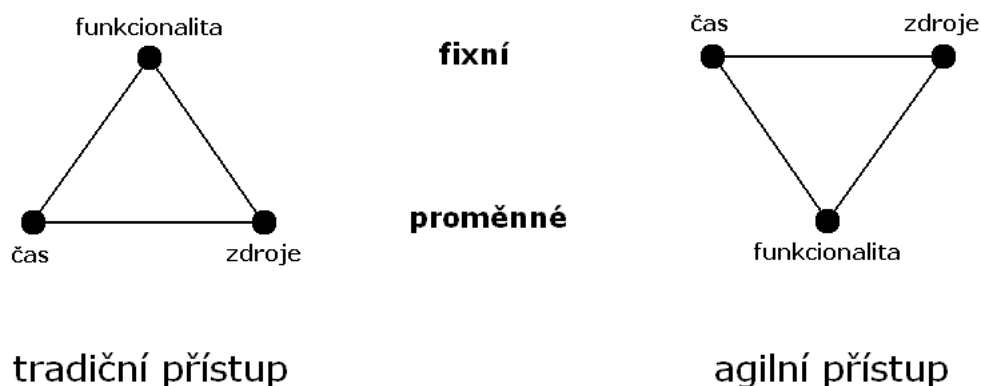
O tom, že technologický vývoj jde stále rychleji kupředu, není třeba pochybovat. O vývoji software to ovšem platí dvojnásob. Díky nepřetržitému zdokonalování technologií, vybavení a nástrojů, ale také díky rozšiřování dostupnosti IT snad ve všech oborech, jsou informační technologie stále více využívány, mnohdy se dokonce staly klíčovými pro dané odvětví. Firmy začaly IT technologie (zvláště pak ty webové) zahrnovat do svých obchodních plánů a strategií, počítají s Internetem jako se základním médiem nejen pro prezentaci a reklamu, ale i pro komunikaci a administrativu svých obchodních procesů. A poněvadž žádný podnik není stejný, každý má svá specifika, potřeby a možnosti, přichází na řadu vývoj software jakožto součást marketingové strategie firmy.

Softwarový trh se od dob svého vzniku podstatně změnil. Například vývoj webových aplikací se natolik zjednodušil, že napsat internetovou prezentaci zvládne průměrně inteligentní středoškolsky vzdělaný člověk. S touto silící konkurencí se postupně místo kvality klade větší důraz na vysokou rychlost a nízkou cenu vývoje. Představme si firmu, která chce svým zákazníkům nabídnout obchodování po Internetu a zaplatí si proto vývoj elektronického obchodu. Bohužel tuto firmu často nezajímá detailně rozpracovaná analýza, neprůstředný návrh a obsáhlá dokumentace, jediné co sleduje jsou její obchodní cíle, tedy zvýšení prodeje svých produktů díky rozšíření skupiny potencionálních zákazníků. K tomu potřebuje především fungující aplikaci, která jí přinese zisk. A navíc tuto aplikaci potřebuje

rychle, než se objeví konkurence s podobným produktem a stáhne si k sobě velkou část budoucích zákazníků.

Právě kvůli výše uvedeným požadavkům se po roce 2000 začaly dostávat na světlo metodiky, které umožnily vyvíjet software rychle a přitom byly schopny reagovat na průběžnou změnu zadání. Začalo se jim říkat agilní. Název vzešel z anglického *agile*, což znamená hbitý, čilý, bystrý nebo svižný.

Jak se tedy liší ono agilní pojetí vývoje software od toho klasického? Velice pěkně to vysvětluje obrázek 3.1, který je publikován v [7].



Obrázek 3.1: Rozdíl tradičního a agilního pojetí vývoje software

Zde jasně vidíme, v čem jsou oba přístupy odlišné. Klasické metodiky počítají s funkcionalitou jako s fixní veličinou, to znamená, že na počátku vývoje se stanoví pevná specifikace požadavků, která se musí dodržet. Naopak čas a zdroje jsou proměnné, a mění se podle toho, jak vývoj postupuje kupředu. Proto se u projektů vyvíjených podle tradičních metodik nezděravě setkáváme s překročením termínu dodání a zvýšenými náklady oproti původnímu plánu. U agilního přístupu je tomu právě naopak. Jako fixní jsou při startu projektu stanoveny zdroje, tzn. finance, lidské zdroje, vývojové prostředí a jiné, a funkcionalita je proměnná. V konečném důsledku to znamená, že zákazník může dostat produkt, který zatím implementuje jen část funkcionality, ovšem díky agilnímu pojetí vývoje se snadno upravuje a vylepšuje i za provozu. Většinou je pro zákazníka výhodnější, když dostane nedokonalou aplikaci v termínu a za dané peníze, než kdyby musel zaplatit další pokračování vývoje a ještě několik týdnů počkat na dokončení aplikace. Přitom ona nedokonalost spočívá v tom, že nejsou implementovány některé funkce, kterým byla stanovena nejnižší priorita. Tyto priority zákazník přiřazoval společně s vývojovým týmem, takže je to pro něj většinou akceptovatelné. V této době totiž stále ještě nemá žádný produkt, na který by mohl navázat své obchodní aktivity, naopak je to pro něj zatím stále jen investice, která se nemusí vyplatit. A takovou situaci jistě žádný klient rád nemá.

Z těchto poznatků vychází jedna z nejdůležitějších pouček týkajících se agilního

programování. Ta říká, že jedinou cestou, jak prověřit správnost navrženého systému, je co nejrychleji jej vyvinout, předložit zákazníkovi a podle zpětné vazby upravit.

Agilní programování sleduje následující principy, které jsou společné pro všechny metodiky:

- **Iterativní a inkrementální vývoj s krátkými iteracemi**

Vývoj probíhá v krátkých fázích, takže celková funkcionalita je dodávána po částech. Zákazník tak má možnost průběžně sledovat vývoj, může se k němu vyjádřit a oponovat změnám. Na konci má jistotu, že nedostane něco, co neočekával.

- **Komunikace mezi zákazníkem a vývojovým týmem**

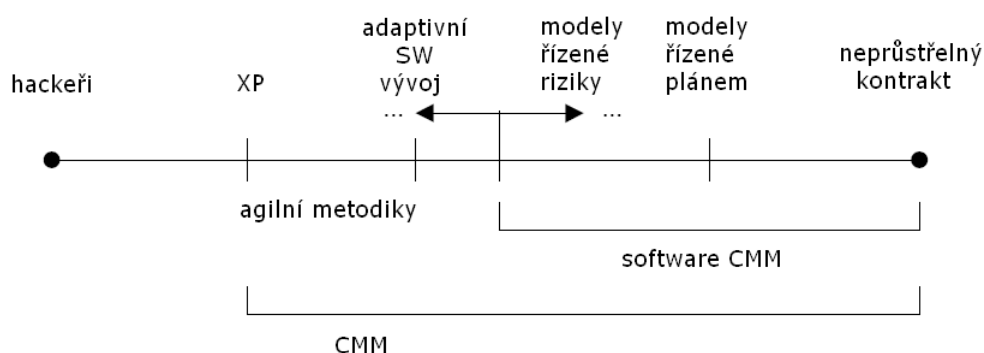
V ideálním případě je zákazník přímo součástí vývojového týmu, má možnost okamžitě vidět průběžné výsledky a reagovat na ně. Účastní se sestavování návrhu, spolurozhoduje o testech a poskytuje zpětnou vazbu pro vývojáře.

- **Průběžné automatizované testování**

Díky krátkým iteracím se aplikace mění velice rychle, proto je nutné pro zajištění co nejvyšší kvality ověřovat její funkčnost průběžně. Testy by měly být automatizované, předem sestavené a měly by být napsány ještě před samotnou implementací testované části. Při každé změně musí být aplikována kompletní sada testů, aby nebyla porušena integrace jednotlivých částí.

Tyto principy vycházejí z *Manifestu agilního programování*, kterému se budu věnovat v následující kapitole.

Barry Boehm, autor spirálového modelu životního cyklu, publikoval v [5] spektrum plánovacích metod, znázorněné na obrázku 3.2.



Obrázek 3.2: Spektrum plánovacích metod

Je vidět, že agilní metodiky leží v tomto spektru vlevo od středu, to znamená, že jsou více adaptabilní, flexibilní, řízené změnami a ne předem daným plánem.

Boehm dále při své analýze agilních metodik zkoumal jejich postavení vedle vývoje na bázi open-source a metodikami řízenými plánem (plan-driven methods). Výsledky srovnání v jednotlivých oblastech ukazuje tabulka 3.1.

oblast	Agilní metodiky	Open-source software	Metodiky řízené plánem
Vývojáři	Agilní, zkušený, spolupracující a soustředění na jednom místě	Geograficky oddělené, zkušené, spolupracující, agilní týmy	Orientovaní na plán, potřebné schopnosti, přístup k externím znalostem
Zákazníci	Nadšení, zkušený, soustředění, spolupracující, reprezentativní a zmocnění	Nadšení, schopní, spolupracující a zmocnění	Přístup ke schopným, spolupracujícím, reprezentativním a zmocněným zákazníkům
Požadavky	Převážně neurčité, rychle se měnící	Převážně neurčité, rychle se měnící, vlastněné společně, průběžně se vyvíjející, nikdy úplné	Znamé brzy, většinou dlouhodobě stálé
Architektura	Navržená pro aktuální požadavky	Otevřená a navržená pro aktuální požadavky	Navržená pro současné i předpokládané požadavky
RefaktORIZACE	Levná	Levná	Drahá
Velikost	Ménší týmy a produkty	Větší rozptýlené týmy a menší produkty	Větší týmy a produkty
Primární cíl	Rychlý výsledek	Řešení problému	Vysoká spolehlivost

Tabulka 3.1: Srovnání agilních a plánem řízených metodik

Hlavním aspektem agilních metodik jsou jednoduchost a rychlost. Při vývoji se tým orientuje jen a pouze na funkcionalitu potřebnou v tomto okamžiku. Rychle ji vyvine, předá zákazníkovi, sesbírá zpětnou vazbu od uživatelů a podle toho aplikaci upraví. Kdy je tedy daná vývojová metodika označována za agilní? Když je vývoj software inkrementální (dodáván rychle, po malých částech), kooperativní (vývojáři a zákazník aktivně spolupracují, dbá se na komunikaci), přímý (metodika jako taková je snadno pochopitelná a modifikovatelná, dobře dokumentovaná) a adaptivní (schopná pokrýt změny v průběhu vývoje).

3.2 Manifest agilního programování

I když většina technik byla svými autory používána už dříve, vznik agilního přístupu k vývoji software se datuje k únoru 2001. Tehdy se sešlo v americkém státě Utah sedmnáct odborníků z oblasti vývoje software a softwarového inženýrství. Za všechny můžu jmenovat ty nejznámější: Kent Beck, Mike Beedle, Alistair Cockburn, Ward Cunningham, Martin Fowler, Jim Highsmith, Ken Schwaber, Jeff Sutherland. Když po společné diskuzi zjistili, že se jejich snahy oprostí vývojový

proces od všech zbytečností, až nečekaně prolínají, sestavili *Manifest agilního programování* (Manifesto for Agile Software Development), pod který se všichni hned podepsali. Současně byla založena *Aliance pro agilní vývoj* (Agile Alliance), jíž se tito aktéři stali členy.

Manifest staví na dvou základních principech:

- Umožnit změnu je mnohem efektivnější, než se jí snažit zabránit.
- Je třeba být připraven reagovat na nepředvídatelné události, protože ty nepochybně nastanou.

Autoři na základě těchto tezí dávají přednost:

individualitám a interakci	před	procesy a nástroji
fungujícím software	před	obsáhlou dokumentací
spolupráci se zákazníkem	před	sjednáváním smluv
reakci na změnu	před	plněním plánu

A k těmto bodům dodávají, že sice je určitá hodnota i v položkách na pravé straně, ale oni více hodnotí položky vlevo.

Prakticky žádná agilní metodika není striktně daná kuchařka zaručených a ověřených postupů. Vždy je nechán prostor pro přizpůsobení metodiky konkrétnímu projektu. Jinak by daná metodika ani nemohla být považována za agilní, protože slovo „agilní“ má velice blízko k „flexibilní“, tedy přizpůsobivý. To je přesná aplikace jednoho z doporučení agilního vývoje. Raději než abychom se snažili vytvořit všeobjemnou univerzální metodiku, která bude vyhovovat každému projektu, zákazníkovi i vývojáři, vytvořme metodiku jednoduchou, lehkou a flexibilní, kterou pak budeme moci přizpůsobit danému konkrétnímu problému. To nás opět vede k jednomu ze základních tvrzení agilního programování, které říká „programujeme jen to, co je v této chvíli nezbytně nutné, nic navíc; do řešení by se mělo zahrnout jen to, co *prokazatelně* potřebuje *každý*, nikoliv to, co *možná* bude potřebovat *někdo*“.

3.3 Stručný přehled nejběžnějších agilních metodik

Počet metodik, které vycházejí z manifestu nebo se jím nechávají alespoň silně inspirovat, je čím dál vyšší. Stále vznikají nové metody a postupy, často jsou to úpravy některých dlouho používaných metodik, které si autor tak dlouho upravoval pro své potřeby, až z toho vynikla de facto metodika nová.

Nyní uvedu stručný přehled těch nejznámějších zástupců agilních metodik společně s jejich autory a krátkým popisem. Protože podrobný popis všech metodik není

cílem této práce, u každé z nich uvedu po stručné charakteristice odkazy na zdroje dalších informací. Některé metodiky budu uvádět jejich originálním anglickým názvem, protože mnohdy ani české pojmenování nemají.

1. **Extrémní programování** (Extreme Programming, XP): Kent Beck, Ward Cunningham, Ron Jeffries

Nejznámější zástupce agilních metodik. Mnoho lidí se mylně domnívá, že agilní metodika = extrémní programování. XP je sice nejrozšířenější a považováno za jakéhosi zástupce, ale pořád je to jen jedna z mnoha agilních metodik. Hodí se pro menší projekty a malé týmy, vyvíjející software podle zadání, které je nejasné nebo se rychle mění. Vychází s přesvědčení, že jediným exaktním, jednoznačným, změřitelným, ověřitelným a nezpochybnitelným zdrojem informací je zdrojový kód. Používá běžné principy a postupy, které dotahuje do extrémů. Například: jestliže se osvědčují revize kódu, budeme tedy neustále revidovat (myšlenka párového programování), pokud se vyplácí testování, budeme nepřetržitě testovat jak my, tak i zákazník (testování jednotek, funkcionality, akceptační testy), osvědčuje-li se návrh, stane se součástí naší každodenní činnosti (refaktorizace), atd.

- Beck, Kent: Extrémní programování, Grada, Praha 2002
- Beck, Kent: Extrême Programming Explained, Addison-Wesley 1999
- <http://www.xprogramming.com>

2. **Vývoj řízený vlastnostmi** (Feature-Driven Development, FDD): Jeff de Luca, Peter Coad

Zaměřuje se na vývoj po malých kouscích – vlastnostech, rysech, což jsou elementární funkcionality přinášející nějakou hodnotu uživateli. Vývoj probíhá v pěti fázích, první tři jsou sekvenční, poslední dvě pak iterativní. Iterace trvají zpravidla 2 týdny. Začíná se vytvořením modelu, ten se převede do seznamu vlastností, které se postupně implementují. Velice dobře se měří pokrok ve vývoji projektu, FDD umožňuje detailně plánovat a kontrolovat vývojový proces. Hlavní výhodou je zaměření na dodávání fungujících přírůstků každé dva týdny.

- Palmer, Stehen R., Felsing, John M.: A Practical Guide to Feature-Driven Development, Prentice Hall 2002
- <http://www.featuredrivendevelopment.com>
- <http://www.nebulon.com/fdd>

3. **SCRUM Development Process**: Ken Schwaber, Mike Beedle

Principem je opět iterativní vývoj definující 3-8 fází, tzv. sprintů, každý z nich trvá přibližně měsíc. Metodika nedefinuje žádné konkrétní procesy,

pouze zavádí pravidelné každodenní schůzky vývojového týmu, tzv. *scrum meetings*. Zde si jednotliví členové sdělují, které položky byly od minulé schůzky dokončeny, které nové úkoly nyní přijdou na řadu a jaké překážky stojí vývoji v cestě a musí se vyřešit. Každý sprint je zakončen předvedením funkční demo-aplikace zákazníkovi, který poskytne zpětnou vazbu.

- Schwaber, Ken, Beedle, Mike: Agile Software Development with SCRUM, Prentice Hall 2001
- <http://www.controlchaos.com>

4. **Adaptivní vývoj software** (Adaptive Software Development, ASD): Jim Highsmith

ASD definuje místo tradičních sekvenčních fází plánování – návrh – realizace iterativní fáze spekulace – spolupráce – učení. Hlavním přínosem této metodiky je, že odchylky od plánu nechápe jako chyby, ale jako příležitosti k učení.

- Highsmith, Jim: Agile Software Development Ecosystems, Addison-Wesley 2002
- <http://www.jimhighsmith.com/learn.html>

5. **Lean Development**: Mary Poppendieck, Tom Poppendieck

Není metodikou v pravém slova smyslu, ale spíše souhrnem deseti pravidel, kterých dodržování slibuje efektivnější a rychlejší vývojový proces. Příkladem pravidel může být: odstranit vše zbytečné, minimalizovat zásoby, zavést zpětnou vazbu, odstranit lokální optimalizaci, apod. Dále na základě pravidel definuje 7 principů spolu s nástroji, jak tyto principy realizovat v praxi.

- Poppendieck, Mary, Poppendieck, Tom: Lean Software Development, Addison-Wesley 2003
- <http://www.leanprogramming.com>

6. **Vývoj řízený testy** (Test-Driven Development, TDD): Kent Beck

Nezabývá se tvorbou specifikací, plánů a dokumentace, to si každý tým musí zvolit sám podle toho, jak mu to vyhovuje. TDD doporučuje přistoupit k testům jako k hlavní fázi celého vývojového procesu. Základním pravidlem je psát testy dříve než samotný kód a implementovat jen přesně takové množství kódu, které projde testem. Nic méně, ale také nic více.

- Beck, Kent: Test Driven Development: By Example, Addison-Wesley 2002
- <http://www.testdriven.com>

7. Crystal metodiky (Crystal family of methodologies): Alistair Cockburn

Již z původního anglického názvu je patrné, že se nejedná jen o jednu metodiku, nýbrž o celou rodinu metodik. Autor vychází z toho, že sebelepší metodika nemůže vyhovovat každému projektu a je lepší přizpůsobit metodiku na míru danému projektu. Dalo by se říct, že první fází vývoje je vlastně vytvoření metodiky. Kritérii pro výběr správné metodiky je například velikost projektu, velikost vývojového týmu nebo kritičnost projektu.

- Cockburn, Alistair: Crystal Clear, Addison-Wesley 2004
- <http://alistair.cockburn.us/crystal/crystal.html>

Z všech těchto zástupců jsem vybral 3, které popíšu podrobněji v následujících kapitolách. Konkrétně půjde o Feature-Driven Development, SCRUM a Test-Driven Development.

3.4 Feature-Driven Development

Feature-Driven Development (dále jen FDD) je jedním z hlavních představitelů agilního pojetí vývoje software. Základy FDD položil Peter Coad, který spojil iterativní přístup s praktikami, které se osvědčily jako efektivní v průmyslu. Později se k němu přidali Jeff de Luca a Stephen Palmer, kteří mu pomohli myšlenku FDD rozvést a zdokonalit. Snažili se, aby se vývoj software stal jednodušší, efektivnější a čitelnější. Všichni tři jsou v literatuře uváděni jako spoluautoři metodiky.

3.4.1 Charakteristika

Poprvé byla metodika FDD prezentována v práci [8]. Tato metodika se ve spektru životního cyklu software zaměřuje na fáze návrhu a implementace, nepokrývá tedy kompletně celý vývojový proces. Těží z neustálého monitoringu stavu projektu, častých dodávek fungující aplikace a dohlížením na kvalitu v průběhu vývojového procesu. FDD se skládá z pěti sekvenčních procesů, z nichž poslední dva jsou iterativně opakovány. Stephen Palmer ve své knize [15] dokonce říká, že FDD se hodí na rozdíl od většiny ostatních agilních metodik i k vývoji kritických aplikací.

FDD se snaží z vývojového procesu odstranit co nejvíce chaosu a nejistoty, umožnit lidem pracovat efektivně a maximalizovat produktivitu. Říká, že výsledný produkt je to jediné, k čemu by měla naše snaha směřovat. Zakládá svou myšlenku na tom, že vyvíjený systém se dá rozložit na množinu vlastností a ty pak postupně implementovat. Už to samo o sobě dává vědět, že dodání kvalitní aplikace je primární cíl. FDD se tedy zaměřuje na vlastnosti produktu, nicméně jeho podstatnou částí je i modelování. Na počátku vývoje se vytvoří celkový (globální) model systému,

který je značně abstraktní, oproštěn od všech detailů a maličností. Má za úkol poskytnout zevrubný pohled na systém a stanovit hlavní vývojovou linii, po které se bude tým ubírat. V iterativních fázích pak jde o vesměs jednoduchou opakovanou činnost návrhu a implementace jednotlivých vlastností. Tyto iterace jsou krátké, většinou trvají přibližně dva týdny, konkrétní doba je však závislá na vlastnostech vyvíjeného produktu. V jedné iteraci samozřejmě probíhá implementace několika vlastností paralelně. Nemalou devizou FDD je pravidelné a časté dodávání fungujících betaverzí, je možné zákazníkovi představit novou verzi dokonce i po každé iteraci. Má to tu výhodu, že zákazník má jednak jistotu, že vývoj jeho produktu jde kupředu, ale má také možnost vstoupit do vývojového procesu a provést patřičné změny. To dělá metodiku dostatečně flexibilní k požadavkům zadavatele, zejména když přicházejí postupně, souběžně s vývojem.

3.4.2 Základní pojmy

V úvodu jsem řekl, že základním pojmem, kolem kterého se celý FDD točí, je vlastnost. Vlastnost (feature) je z pohledu FDD definována jako malý kousek funkčnosti, který přináší užitnou hodnotu pro zákazníka. Důležitými charakteristikami vlastnosti jsou:

- **měřitelnost:** musíme umět rozhodnout, jestli implementovaná vlastnost je právě ta, kterou zákazník požadoval
- **srozumitelnost:** musíme být schopni vlastnosti porozumět, abychom byli schopni ji jednoduše popsat, musíme znát její účel a umět definovat její výsledek
- **realizovatelnost:** musíme si být jistí, že vlastnost bude možné implementovat v rámci jedné iterace (přibližně dva týdny); pokud je vlastnost příliš rozsáhlá, je nutné ji rozdělit do několika menších vlastností a ty pak implementovat zvlášť

FDD kvůli odstranění chaosu předepisuje formát zápisu vlastností. Definice vypadá takto:

<akce> <předmět> <podrobnosti>

Akce označuje činnost prováděnou v rámci dané vlastnosti, *předmět* je artefakt, na kterém je daná akce prováděna, a *podrobnosti* upřesňují vlastnost, přidávají další požadavky nebo ji naopak specializují na konkrétní účel a vymezují její místo a platnost v systému. Pro ilustraci uvedu příklady takovýchto vlastností:

- spočítání výdajů za aktuální rok

- autentizace uživatele oproti zadaným přihlašovacím údajům
- vytištění souhrnu na síťové tiskárně
- vypsání formuláře pro editaci údajů

Druhým stěžejním pojmem v oblasti FDD je modelování. Modelování je součástí prakticky všech fází FDD. Na začátku se tvoří celkový model a v každé iteraci se vytváří podrobnější lokální model popisující danou vlastnost. Díky tomuto se FDD také někdy říká *přístup řízený modelem*.

3.4.3 Role

FDD celkem striktně udává, jakým způsobem se mají vlastnosti systému popisovat, jak se mají vést záznamy o dokončených vlastnostech a celkové rozpracovanosti projektu. Stejným způsobem rozděluje role ve vývojovém týmu. Každá role má přesně definováno, co je jejím úkolem v jednotlivých fázích vývoje. Jeden člen týmu může zastupovat více rolí, stejně tak jednu roli může zastávat více lidí. FDD rozlišuje 3 skupiny rolí: *klíčové*, *podpůrné* a *další*. Dříve než se pustím do popisu procesů v rámci FDD, uvedu charakteristiky jednotlivých rolí.

Klíčové role

Projektový manažer (Project Manager) je administrativním i finančním vedoucím projektu. Jednou z jeho úloh je zajišťovat celému projektovému týmu odpovídající pracovní podmínky, aby všichni byli spokojeni, měli z práce radost a nic jim jejich pracovní efektivitu nenarušovalo. Má vždy poslední slovo v otázkách rozsahu projektu, časového harmonogramu a personálního obsazení týmu.

Hlavní architekt (Chief Architect) je zodpovědný za celkový návrh systému. Pořádá s týmem schůzky a workshopy ohledně návrhu a má také právo posledního rozhodnutí ve všech otázkách týkajících se návrhu. Pokud je to nutné, může být tato role rozdělena do dvou rolí, konkrétně doménového architekta a technického architekta.

Vývojový manažer (Development Manager) má na starosti aktivity spojené s každodenním vývojem. Jeho úkolem je řešit jak problémy se zdroji, tak problémy, které vzniknou uvnitř týmu. Často se jeho úkoly kryjí s úkoly projektového manažera a hlavního architekta, proto bývá často tato role obsazena stejným člověkem, který má jednu ze dvou výše popsanych rolí.

Hlavní programátor (Chief Programmer) je většinou zkušený vývojář, který se podílí na analýze požadavků a návrhu systému. Je odpovědný za vedení

malých programátorských týmů, které v iteračních cyklech provádějí analýzu, návrh a vývoj nových vlastností. Z množiny všech vlastností vybírá ty, které se budou implementovat v další iteraci a určuje vlastníky jednotlivých tříd. Spolupracuje s dalšími hlavními programátory při řešení technických problémů mezi jednotlivými týmy. Každý týden podává zprávu o pokroku v projektu za svůj tým.

Vlastník třídy (Class Owner) je pod vedením hlavního programátora a jeho prací je návrh, kódování, testování a dokumentace jednotlivých konkrétních vlastností. Je odpovědný za vývoj třídy, které je vlastníkem. Z členů týmu nesoucích tuto roli se formují *týmy pro vlastnosti* (feature teams) a v každé iteraci se složení těchto týmů mění podle toho, které třídy právě implementovaná vlastnost ovlivňuje. To FDD označuje jako techniku *dynamických týmů*.

Doménový expert (Domain Expert) může být koncový uživatel, klient, sponzor, obchodní analytik, obecně člověk, který dokonale rozumí cílové oblasti, ve které bude daná aplikace nasazena. Díky tomu dobře ví, jaké požadavky by systém měl mít a předává tyto poznatky vývojářům, aby zajistil, že se vyvíjí opravdu to, co se očekává. Ve vývojovém týmu zastupuje onu nepostradatelnou zpětnou vazbu, která kontroluje, jestli se vývoj neodklonil od své původní linie.

Podpůrné role

Manažer pro dodávky (Release Manager) kontroluje pokrok ve vývoji, shromažďuje do hlavních programátorů zprávy o tom, co už se udělalo a v jaké fázi je vývoj, a předává je projektovému manažerovi.

Jazykový specialista (Language Guru) musí důkladně ovládat danou potřebnou technologii, a musí být schopen své znalosti poskytnout dále vývojovému týmu. Příkladem může být programovací jazyk. Tato role je zvláště důležitá, pokud má tým začít používat nějakou novou technologii.

Konstruktér (Build Engineer) je zodpovědný za nastavení a provedení procesu sestavení výsledné aplikace nebo některé z jejích prototypů či betaverzí. Má také za úkol spravovat systém pro kontrolu verzí (příkladem takového systému může být Subversion nebo CVS) a zveřejňovat dokumentaci.

Nástrojář (Toolsmith) má za úkol programovat nebo obstarávat podpůrné nástroje pro potřeby vývoje nebo testování. Může také být pověřen udržováním databáze nebo webových stránek týkajících se projektu.

Administrátor (System Administrator) spravuje a konfiguruje servery, síť a pracovní stanice, které vývojáři potřebují ke své práci. Udržuje softwarové vybavení nutné k vývoji.

Další role

Tester (Tester) může pracovat sám nebo být součástí testovacího týmu, jehož úkolem je testovat a ověřit, že vyvíjený systém opravdu pokrývá požadavky zákazníka.

Správce nasazení nových verzí (Deployer) se stará o konverzi dat do formátů odpovídajících novému systému a podílí se na vydávání nových verzí (včetně betaverzí).

Písař (Technical Writer) je autorem uživatelské dokumentace a manuálů.

3.4.4 Procesy

Metodika FDD se snaží zabránit chaosu a přinést více řádu do vývoje zavedením pěti procesů, u kterých uvádí:

1. popis procesu
2. vstupní kritéria
3. úkoly
4. verifikace
5. výstupní kritéria

Tento model pro popis procesů se někdy v literatuře označuje zkratkou ETVX (Entry criteria, Tasks, Verification, Exit criteria).

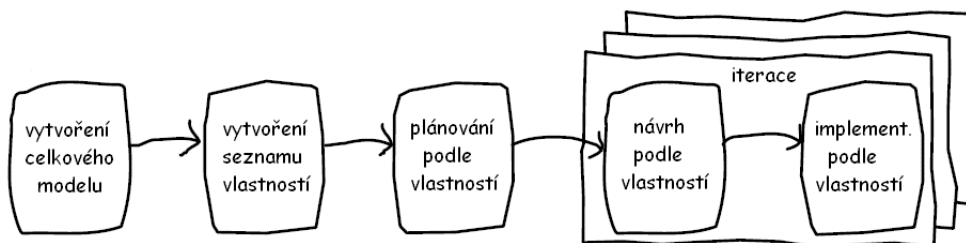
Ještě před tím, než uvedu a popíšu jmenovaných pět procesů, je potřeba se zmínit o rozsahu a úrovni podrobností, se kterými FDD popisuje prováděné činnosti. Rational Unified Process jako jedna z nejpropracovanějších metodik nejen že podrobně rozebírá veškeré procesy a postupy, ale také definuje konkrétní metody, průvodce, šablony a řešené příklady, které vývojáře vedou celým procesem od úvodní studie, až po udržování hotového produktu. Nic takového není obsahem FDD. Metodika popisuje pouze obecný rámec, kterého by se vývojář měl držet, ale spoustu dalších menších nezbytností už nedefinuje a nechává je na vývojáři samotném. Pokud tedy metodika předepisuje vytvořit návrh a provést jeho inspekci, pak nic neříká o tom, jak konkrétně by se daný návrh měl provádět a jak by se mělo přistoupit k inspekci. Metodika určí pouze *co* se má dělat, ale *jak* se to má dělat, to už je v rukou hlavního programátora, který musí být dostatečně zkušený, aby dokázal zvolit vhodné odpovídající nástroje a metody. Sami autoři metodiky tuto obecnost vysvětlují tím, že metodiky s přesně vyspecifikovanými kroky a postupy vycházejí z optimistického předpokladu, že vývoj je dobře předvídatelný. To ovšem ve většině případů neplatí a pokud vývojář sklouzne k bezmyšlenkovitému následování

metodiky, nedokáže přizpůsobit vývoj aktuální situaci a potřebě. Proto se také FDD drží podstaty agilního vývoje, tedy flexibility a volnosti při rozhodování, kterou metodika ponechává.

FDD definuje následujících pět procesů (v závorce vždy uvedu pojmenování procesu, které se používá v anglické literatuře):

1. Vytvoření celkového (globálního) modelu (Develop an Overall Model)
2. Vypracování seznamu vlastností (Build a Features List)
3. Plánování podle vlastností (Plan By Feature)
4. Návrh podle vlastností (Design By Feature)
5. Implementace podle vlastností (Build By Feature)

Seřazení jednotlivých fází znázorňuje obrázek 3.3.



Obrázek 3.3: Posloupnost vývojových fází v metodice FDD

Nyní každý z těchto procesů popíšu podrobněji podle schématu uvedeného výše.

1. Vytvoření celkového modelu

Pro první fázi je potřeba mít k dispozici doménové experty a mít vybraného hlavního programátora a hlavního architekta. Úkolů pro první fázi je několik, ale ty hlavní by se daly rozdělit do dvou skupin: studium domény a vytvoření globálního modelu. Poté co projektový manažer sestaví tým pro vytvoření modelu, musí se prozkoumat doména, tedy prostředí, ve kterém bude daná aplikace provozována. To v sobě nese jednak studium relevantních dokumentů, ale hlavně je úkolem doménových expertů, tedy lidí od zadavatele, aby celý tým seznámili se zaměřením projektu, co od něho očekávají, jaké jsou jejich požadavky, atd. Informace jsou předávány abstraktní formou náčrtků, komentářů a popisů v přirozeném jazyce. Není požadováno (a ani vítáno) přílišné zabíhání do podrobností nebo snaha o formalizaci požadavků.

Vstupní kritéria: byli vybráni doménoví experti, hlavní programátor a hlavní architekt

Úkoly: sestavit tým pro vytvoření modelu, podrobně prozkoumat doménu, prostudovat dokumenty (volitelné), vytvořit globální model, prověřit a vyladit globální model, napsat poznámky k modelu

Verifikace: posudková řízení: interní (v rámci týmu) i externí (se zákazníkem)

Výstupní kritéria: je vytvořen objektový model – diagram tříd s identifikovanými atributy a metodami, sekvenční diagramy, detaily zachyceny v poznámkách

2. Vypracování seznamu vlastností

V této fázi máme za úkol vytvořit podrobný seznam vlastností. Ten má za úkol specifikovat a vymezit systém. V této chvíli určitě nebude možné sestavit kompletní a do konce vývoje neměnný seznam vlastností. Tým by se ale měl snažit sestavit seznam co nejobsáhlejší, aby pokryl co nejvíce požadavků na systém. Vlastnosti nejsou udržovány jako homogenní seznam, ale jsou rozděleny do skupin podle toho, jak spolu souvisí a na kterou oblast systému se zaměřují. Při vypracovávání seznamu vlastností tým vychází nejen z modelu vytvořeného ve fázi 1, ale také ze všech dostupných dokumentů a informací, jakými mohou být kupříkladu případy užití. Nakonec musí ještě tým ve spolupráci se zákazníkem, případně s doménovými experty, stanovit minimální množinu vlastností, která musí být implementována, aby se produkt dal považovat za hotový a zákazník ho byl ochoten převzít a zaplatit.

Vstupní kritéria: byli vybráni doménoví experti, hlavní programátor a hlavní architekt

Úkoly: sestavit tým pro vytvoření seznamu vlastností, vytvořit seznam vlastností

Verifikace: posudková řízení: interní (v rámci týmu) i externí (se zákazníkem)

Výstupní kritéria: je vytvořen seznam vlastností – seznam oblastí domény, pro každou oblast je vytvořen seznam obchodních procesů týkajících se domény, pro každý krok každého obchodního procesu je identifikována vlastnost, která daný krok realizuje

3. Plánování podle vlastností

Cílem třetí fáze je vytvořit plán, podle kterého bude možné včas a dobře implementovat všechny potřebné vlastnosti. Vlastnosti jsou seřazeny podle jejich priorit a závislostí. Součástí plánu je i stanovení data ukončení vývoje.

Každé skupině vlastností je přiřazen hlavní programátor. V rámci každé skupiny vlastností jsou jednotlivé třídy přiřazeny jejich vlastníkům, kteří budou zodpovídat za implementaci dané třídy. Pro jednotlivé skupiny vlastností jsou naplánovány milníky, podle kterých se bude snáze měřit pokrok vývoje. Tyto milníky se později můžou posouvat a měnit, ale datum ukončení vývoje by mělo zůstat neměnné, protože každé další průtahy stojí zákazníka další peníze navíc, které jistě nebude projektu věnovat s radostí.

Vstupní kritéria: je sestaven seznam vlastností (dokončena fáze 2)

Úkoly: sestavit tým pro plánování, sestavit pořadí, v jakém se vlastnosti budou implementovat, přiřadit obchodní procesy hlavním programátorům, přiřadit třídy vývojářům (vlastníkům tříd)

Verifikace: průběžná interní posudková řízení

Výstupní kritéria: plán vývoje – přibližná data dokončení (měsíc a rok) pro všechny obchodní procesy, hlavní programátoři jsou přiřazeni k obchodním procesům, rozdělení seznamu vlastností na oblasti podle obchodních procesů, seznam tříd s jejich vlastníky

4. Návrh podle vlastností

Tato fáze společně s fází následující se iteračně opakují a zaměřují se už na konkrétní vývoj nalezených vlastností. Hlavní programátor nejprve vybere množinu vlastností, která se bude v dané iteraci implementovat. Měl by při výběru mít na paměti jednak priority a závislosti vlastností, ale také dobu potřebnou k vývoji daných vlastností. Doba implementace vybrané skupiny vlastností by měla být mezi jedním a dvěma týdny. Hlavní programátor následně kontaktuje vlastníky tříd, kterých se skupina vlastností týká. Ti pak vytvoří dočasný tým a vypracují podrobný návrh pro implementaci daných vlastností včetně časového harmonogramu realizace.

Vstupní kritéria: proces plánování (fáze 3) je dokončen

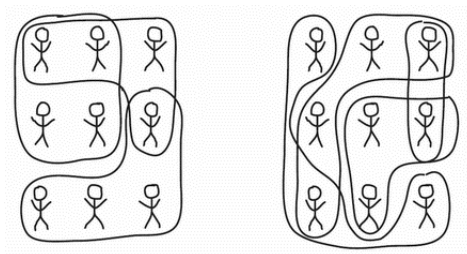
Úkoly: sestavit z vlastníků tříd tým pro realizaci dané vlastnosti, volitelně se může zařadit zkoumání domény a relevantních dokumentů, vytvořit sekvenční diagram pro danou vlastnost, vyladit objektový model (například přidat třídy nebo metody na základě zkoumání dané vlastnosti), napsat hlavičky tříd a metod

Verifikace: inspekce návrhu

Výstupní kritéria: vytvořen *balíček návrhu* (Design Package) – obecné informace o vlastnosti, diagramy posloupností, případné alternativy v návrhu, objektový model dané vlastnosti, hlavičky tříd a metod, harmonogram prací na implementaci vlastnosti

5. Implementace podle vlastností

Dostávám se k poslední fázi, která má za úkol implementaci vybraných vlastností. Za tu jsou odpovědní vlastníci tříd patřící do týmu pro danou skupinu vlastností. Ti musí napsat nejen metody svých tříd, které pokryjí funkcionality vlastnosti, ale provádějí také testování jednotek. Jakmile je vlastnost dokončena, je na hlavním programátorovi, aby ji integroval do systému. Jeden člověk (vlastník třídy) může být současně ve více týmech. Týmy se vytvářejí a rozpouštějí dynamicky podle toho, jaká skupina vlastností se právě implementuje. Sestavování týmů má na starosti hlavní programátor, který koordinuje jednotlivé členy a přiděluje je do jednotlivých týmů. Dynamické týmy jsou znázorněny na obrázku 3.4.



Obrázek 3.4: Dynamické týmy v metodice FDD

Vstupní kritéria: proces návrhu podle vlastností (fáze 4) byl dokončen, tzn. balíček návrhu úspěšně prošel inspekci

Úkoly: implementovat třídy a metody pro danou vlastnost, provést inspekci napsaného kódu, testovat jednotky, připravit projekt na integraci a sestavení

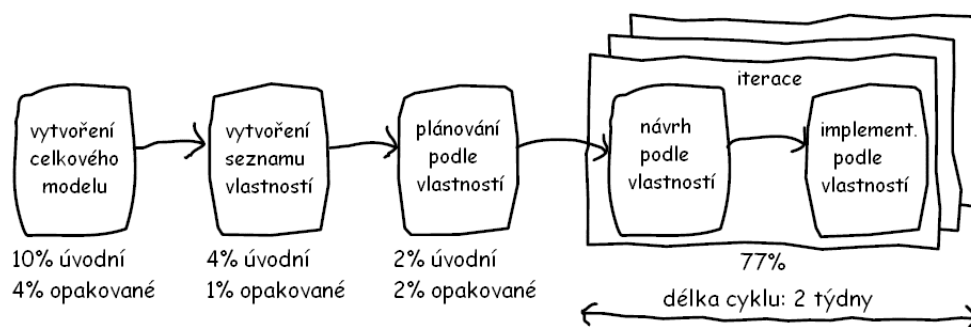
Verifikace: průběžné inspekce kódu, testování jednotek

Výstupní kritéria: třídy a metody úspěšně prošly inspekci, případně další třídy potřebné pro správnou funkčnost vlastnosti, dokončena implementace vlastnosti (tzn. funkcionality přinášející hodnotu pro klienta)

Dobu strávenou v jednotlivých fázích znázorňuje obrázek 3.5.

3.4.5 Praktiky

Metodika FDD se skládá z několika praktik, jejichž dodržování slibuje úspěšné dodání fungujícího produktu v dohodnutém čase za dohodnutou cenu, přesně podle motta FDD. Vývojový tým se musí ztotožnit se všemi těmito praktikami, aby dostal pravidlům vývoje podle FDD. Autoři však říkají, že není nutné přijmout



Obrázek 3.5: Rozložení času do vývojových fází metodiky FDD

všechny praktiky najednou a okamžitě. Je možné je do vývojového procesu zahrnovat postupně podle toho, jak zkušený tým je.

Nyní krátce představím zmíněné praktiky.

Objektové modelování domény

Doménu, tedy oblast, ve které bude výsledná aplikace nasazena, je dobré modelovat objektově. Objektové vidění světa je bližší lidskému uvažování. V objektovém modelu můžeme využít (narozdíl od ERD diagramů) vztahů mezi třídami, dědičnosti, specializace a hlavně interakce mezi třídami pomocí metod. Doporučuje se vytvořit navíc ještě sekvenční diagramy, které popisují vzájemnou komunikaci tříd. V této fázi se musejí analytici oprostít od jakéhokoliv zabíhání do detailů a od podvědomého formování architektury aplikace. Objektový model má být abstraktní a má vývojářům poskytnout úvodní seznámení s doménou a uvedení do problematiky. Detailní architektura se řeší až v dalších fázích a její rozebírání v této fázi by bylo zbytečné, skoro až nežádoucí.

Vývoj podle vlastností

FDD se zakládá na vývoji *vlastností*, omezuje tak množinu funkčních požadavků jen na funkce, které přinášejí hodnotu uživateli. Tím se FDD liší od neagilních metodik, které většinou vyvíjejí po modulech nebo po funkcích. V tomto případě by se totiž snadno mohlo stát, že se vývojář příliš soustředí na funkcionalitu daného modulu na úkor toho, co vlastně zákazník opravdu chce.

Vlastnictví tříd

Individuální vlastnictví tříd se nevyskytuje jen ve FDD, ale je součástí mnoha dalších agilních metodik. Vlastník odpovídá na úspěšnou implementaci třídy, za

její otestování a celkovou konceptuální integritu. Ta je důležitá, pokud je již třída napsána, ale při dalším vývoji s ní má spolupracovat třída nová. V tom případě musí vlastník zajistit, aby jeho třída byla schopna tuto komunikaci poskytnout a tak zachovat integritu systému jako celku. Vlastník třídy je často jediný, kdo jí pořádně do detailu rozumí a je tedy pro něj snadnější přidat novou funkcionalitu než pro někoho, kdo daný kód vidí poprvé.

Dynamické týmy sestavované podle vlastností

Úkolem týmů je implementovat danou vlastnost. Většinou se vyvíjí více vlastností paralelně, přitom funkcionalita jedné vlastnosti obvykle zasahuje do více tříd. Proto se vytváří konkrétní tým pro konkrétní vlastnost. Jeden člověk proto může být zároveň ve více týmech. Vlastníky tříd si do týmu vybírá šéf týmu implementujícího vlastnost, i když jsou daní lidé členy jiných týmů. S dokončením vlastnosti tým zaniká.

Inspekce

Inspekce mají napomáhat odstraňování chyb vzniklých při vývoji. Inspekci neprochází jen samotný kód, ale také modely, které se vytvářejí ve fázích návrhu.

Pravidelné dodávky nových verzí a správa konfigurací

Třídy musejí být do výsledného produktu integrovány tak, aby v každém okamžiku bylo možné z nich sestavit funkční (ne nutně kompletní) ukázkovou aplikaci. Ta se může předvést zákazníkovi a slouží jednak ke sběru zpětné vazby a také k ujištění zákazníka o tom, že vývoj jeho produktu jde stále kupředu. Správa konfigurací slouží k jednoznačné a snadné identifikaci souborů se zdrojovými kódy a k uchovávání historie jejich změn. Aplikací na správu verzí a konfigurací je celá řada, těmi nejznámější a nejpoužívanějšími jsou Subversion a CVS.

Zprávy o stavu projektu

Metodika FDD si na přehledném a detailním monitorování stavu vývoje projektu hodně zakládá. Jsou definovány šablony, tabulky a schémata, do kterých vývojový tým zapisuje, co všechno je již hotovo, stupeň rozpracovanosti toho, na čem se právě pracuje, a co se plánuje pro další iteraci. Souhrny pořízené z těchto informací jsou pak distribuovány celému dodavatelskému týmu, od nejvyššího managementu až po samotné programátory. V dnešní době se často k účelu sledování stavu projektu využívají připravené softwarové aplikace.

3.4.6 Závěr

Metodika FDD přistupuje k vývoji software zajímavým a originálním způsobem. Definuje pojem *vlastnost*, což je malý kousek funkcionality přinášející hodnotu pro uživatele. Po vytvoření celkového modelu se vypracuje seznam vlastností, kolem kterého se pak celý vývoj točí. Definuje se společně se zákazníkem minimální množina vlastností, která může být považována za hotový produkt. Jakmile ani zákazník ani vývojový tým nemůže přijít na žádnou další vlastnost, která by se měla implementovat, vývoj se ukončí. Tím je zaručeno, že zákazník dostane přesně to, co očekává. Metodika používá pět základních procesů, z nichž poslední dva se opakují v iteracích trvajících přibližně dva týdny. Tím je zaručena dostatečná flexibilita a prostor pro to, aby mohl zákazník zasáhnout do vývoje. Unikátní je ve FDD dynamické vytváření týmů. Metodika nepokrývá celý vývojový proces, pouze ukazuje schéma vývoje, podle kterého by se mělo vyvíjet, konkrétní postupy, metody a nástroje už jsou v kompetenci hlavního programátora. FDD se hodí na celou škálu projektů, od malých jednoúčelových projektů, až po zakázky pro velké korporace.

3.5 SCRUM Development Process

Metodika SCRUM Development Process (dále jen Scrum) patří k těm mladším agilním metodikám. Přišli s ní Ken Schwaber a Mike Beedle v roce 2002 a prezentovali ji v [17]. Pro pojmenování metodiky právě termínem „scrum“ se autoři nechali inspirovat rugby, kde „scrum“, v češtině překládáno jako „mlýn“, znamená týmovou strategii, jak dostat míč do hry. Metodika jako taková se opírá o objektové vidění světa a přináší zcela nové a ojedinělé praktiky do vývojového procesu.

3.5.1 Charakteristika

Scrum byl vyvinut pro podporu řízení vývojového procesu. Není to tedy metodika stylu „programátorské kuchařky“, vůbec neuvádí konkrétní nástroje, technologie a postupy, které by měli vývojáři použít, ale zabývá se tím, jak by měl celý tým při práci komunikovat a spolupracovat. Scrum je založen na poznání, že vývoj s sebou přináší spoustu nepředvídatelných událostí a tím se stává složitým. Je tedy potřeba mít metodiku, které dokáže flexibilně a rychle reagovat na změny. Scrum je převážně manažerská metodika, která se snaží vylepšit existující softwarově inženýrské praktiky, a zaměřuje se na velice časté sledování a řešení všech překážek, které by mohly stát v cestě úspěšnému vývoji aplikace. Obecné vlastnosti metodiky Scrum se v podstatě shodují s ostatními agilními metodikami, které také vycházejí z Agilního manifestu. To znamená, že je iterativní, flexibilní, dbá na rychlé dodávky částí aplikace nebo prototypů a následné sbírání zpětné

vazby od zákazníka a snaží se rychle reagovat na měnící se požadavky a změny během vývoje.

3.5.2 Základní pojmy

Dříve než popíšu fungování metodiky podrobně, uvedu pár klíčových pojmů, které jsou s ní spojeny.

Sprint

Jedná se o první typ iterace. Trvá přibližně jeden měsíc a jejich počet se podle projektu různí. Většinou vývoj probíhá ve třech až osmi sprintech. Na začátku každého sprintu je plánovací schůzka, kde se třídí požadavky a vybírá se množina, která se bude implementovat. Na konci sprintu je opět schůzka celého týmu, kde se probere, co všechno se na projektu za tento sprint událo, jaké požadavky se povedlo splnit, jaké ne, a na jaké nové požadavky k zapracování se během vývoje přišlo.

Scrum

Jde o druhý typ iterace, který je o mnoho kratší, trvá jeden den. Na začátku každého pracovního dne je uspořádána schůzka, tzv. Scrum Meeting (podrobnosti viz dále). Díky takto krátkým iteracím je tým nejen neustále dobře informován, v jakém stádiu jsou jednotlivé práce na projektu, ale také je schopen případné problémy operativně řešit, protože na ně přijde brzy.

Backlog

V rámci metodiky Scrum existují dva druhy těchto backlogů. Je to Product Backlog a Sprint Backlog. Ten první obsahuje seznam veškeré funkcionality požadované od výsledné aplikace. Vlastník produktu tento seznam uspořádá podle priorit a logických souvislostí, tým pak z něj vybere část, kterou přesune do Sprint Backlogu. Někdy se může použít technika zobrazování položek backlogů pomocí tzv. uživatelských scénářů známých z Extrémního programování. Často se používá seznam udržovaný jako tabulka v nějakém editoru podobném Microsoft Excelu. V dalším textu budu používat originální anglické názvy pro tyto seznamy, protože nemají adekvátní české pojmenování a ani v česky psané literatuře nejsou nijak překládány. Následují ukázky Product Backlogu (obrázek 3.6) a Sprint Backlogu (obrázek 3.7), tak jak je používá firma Mountain Goat Software, která se zabývá implementací pomocí metodiky Scrum.

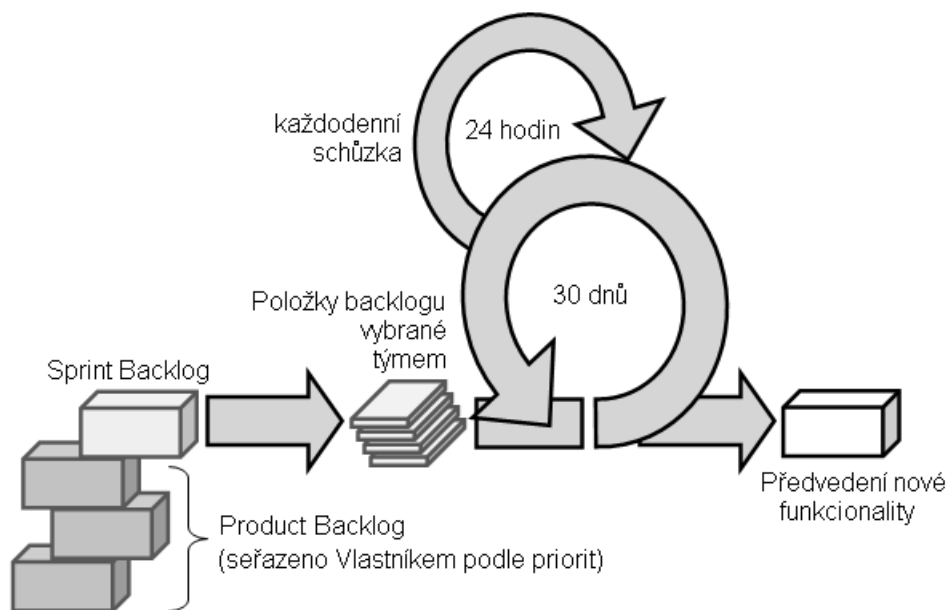
Zevrubný pohled na filozofii celé metodiky ukazuje obrázek 3.8, který v originále pochází z [17].

	Item #	Description	Est	By
Very High				
	1	Finish database versioning	16	KH
	2	Get rid of unneeded shared Java in database	8	KH
		- Add licensing	-	-
	3	Concurrent user licensing	16	TG
	4	Demo / Eval licensing	16	TG
		Analysis Manager		
	5	File formats we support are out of date	160	TG
	6	Round-trip Analyses	250	MC
High				
		- Enforce unique names	-	-
	7	In main application	24	KH
	8	In import	24	AM
		- Admin Program	-	-
	9	Delete users	4	JM
		- Analysis Manager	-	-
	10	When items are removed from an analysis, they should show up again in the pick list in lower 1/2 of the analysis tab	8	TG
		- Query	-	-
	11	Support for wildcards when searching	16	T&A
	12	Sorting of number attributes to handle negative numbers	16	T&A
	13	Horizontal scrolling	12	T&A
		- Population Genetics	-	-
	14	Frequency Manager	400	T&M
	15	Query Tool	400	T&M
	16	Additional Editors (which ones)	240	T&M
	17	Study Variable Manager	240	T&M
	18	Haplotypes	320	T&M
	19	Add icons for v1.1 or 2.0	-	-
		- Pedigree Manager	-	-
	20	Validate Derived kindred	4	KH
Medium				
		- Explorer	-	-
	21	Launch tab synchronization (only show queries/analyses for logged in users)	8	T&A
	22	Delete settings (?)	4	T&A

Obrázek 3.6: Ukázka Product Backlogu

		Days Left in Sprint				
		15	13	10	8	
		F				
		7/22/2002				
		7/24/2002				
		7/26/2002				
		7/31/2002				
Who	Description					
Total Estimated Hours:		554	458	362	270	0
-	User's Guide	-	-	-	-	-
SM	Start on Study Variable chapter first draft	16	16	16	16	
SM	Import chapter first draft	40	24	6	6	
SM	Export chapter first draft	24	24	24	6	
Misc. Small Bugs						
JM	Fix connection leak	40				
JM	Delete queries	8	8			
JM	Delete analysis	8	8			
TG	Fix tear-off messaging bug	8	8			
JM	View pedigree for kindred column in a result set	2	2	2	2	
AM	Derived kindred validation	8				
Environment						
TG	Install CVS	16	16			
TBD	Move code into CVS	40	40	40	40	
TBD	Move to JDK 1.4	8	8	8	8	
Database						
KH	Killing Oracle sessions	8	8	8	8	
KH	Finish 2.206 database patch	8	2			
KH	Make a 2.207 database patch	8	8	8	8	
KH	Figure out why 461 indexes are created	4				

Obrázek 3.7: Ukázka Sprint Backlogu



Obrázek 3.8: Schéma metodiky SCRUM

3.5.3 Role

Metodika Scrum rozlišuje ve vývojovém týmu celkem šest rolí podle jejich povinností a pravomocí. Následuje popis těchto rolí tak, jak je popisují autoři metodiky v [17].

Scrum Master je odpovědný za to, že proces vývoje probíhá v souladu s pravidly a praktikami metodiky Scrum. Komunikuje jak s vývojáři, tak také se zákazníkem a s managementem. Jeho úkolem je zaručit, že všechny překážky zabráňující plynulému a produktivnímu vývoji budou co nejdříve odstraněny. Pokud je Scrum Master nedokáže odstranit sám, musí být schopen najít v týmu kompetentního a schopného člověka, který problému rozumí a je schopen ho odstranit.

Vlastník produktu (Product Owner) je vybrán Scrum Masterem, zákazníkem a managementem a odpovídá za chod projektu, řídí jeho průběh a upravuje a publikuje Product Backlog. Má hlavní slovo v rozhodování o položkách Product Backlogu. Vzhledem k tomu, že mu nejvíce záleží na úspěšném dokončení projektu, zavazuje se tak Vlastník produktu nepřidávat další požadavky do Sprint Backlogu v průběhu sprintu. Položky Product Backlogu se sice často mění nebo přidávají, ale pouze mimo sprinty.

Scrum tým (Scrum Team) je samoorganizující projektový tým, který má za úkol splnit cíle daného sprintu. Dalšími jeho úkoly je odhad pracnosti a času

potřebného k implementaci dané vlastnosti, spolurozhodování o položkách Sprint Backlogu nebo nacházení překážek, které by mohly narušit průběh projektu nebo nějak negativně ovlivnit jejich práci.

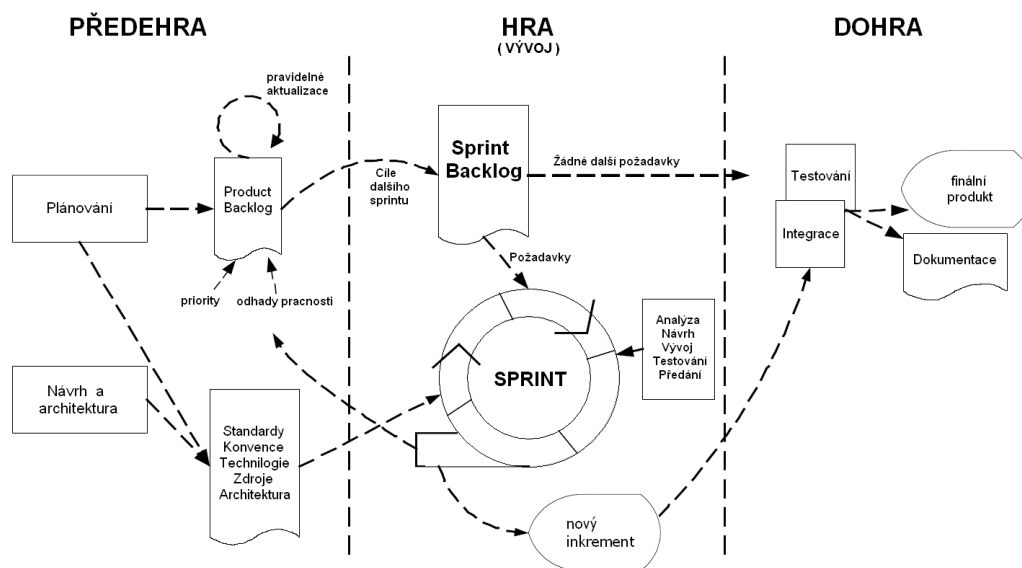
Zákazník (Customer) se podílí na sestavování seznamu položek pro Product Backlog.

Management má na starost konečná rozhodnutí o projektu v oblasti uzavírání smluv a následování standardů a konvencí. Podílí se na sestavování požadavků a cílů celého projektu i jednotlivých etap.

3.5.4 Procesy

Proces metodiky Scrum se dělí do tří hlavních fází. Ty se příznačně jmenují předehra, hra a dohra. První a třetí fáze je lineární, prostřední fáze, skládající se ze sprintů, se iterativně opakuje. Scrum v průběhu celého vývoje nařizuje spoustu postupů, ale neuvádí jejich definici. Říká například, kdy a kdo má naplánovat aktivity pro další sprint, ale jakým způsobem se má toto plánování provádět, to už součástí metodiky není a daný člen týmu už si musí konkrétní prostředky zvolit sám.

Obrázek 3.9, převzatý z [1] znázorňuje celý proces metodiky Scrum.



Obrázek 3.9: Proces metodiky SCRUM

Fáze 1: Předehra

Tato fáze se dělí do dvou podfází: *Plánování* a *Navrh a architektura*.

Plánování obsahuje definici systému, který má být vytvořen. Je sestaven Product Backlog obsahující všechny známé požadavky. Ty mohou přicházet od zákazníka, obchodního nebo marketingového oddělení, zákaznické podpory nebo přímo od vývojářů. Požadavky jsou uspořádány podle priorit a je odhadnuto úsilí potřebné k jejich implementaci. Tyto odhady jsou v počátcích samozřejmě nepřesné, všechny hodnoty se v průběhu vývoje upravují a zpřesňují podle aktuální situace. Úkolem fáze Plánování je také sestavit projektový tým, definovat vývojové nástroje, analýzu rizik, zdroje a postup schválení výsledného produktu.

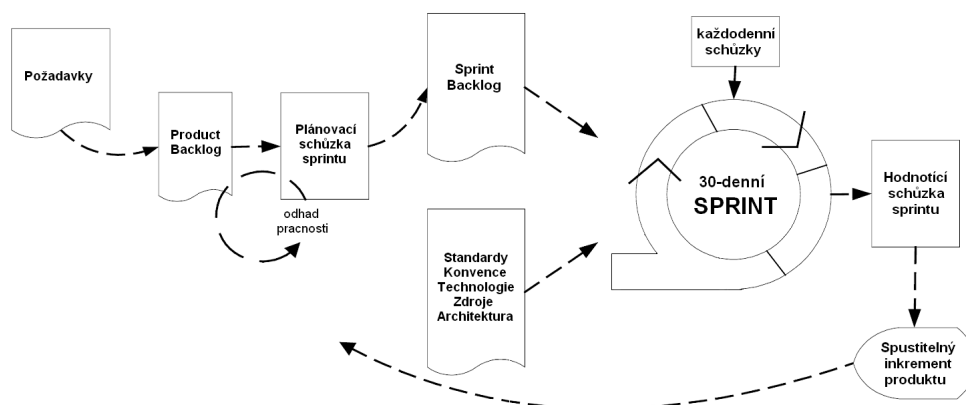
Návrh a architektura se zaměřuje na vytvoření návrhu systému na vysokém stupni abstrakce. Jako základ se bere Product Backlog vytvořený v předchozí fázi. Provádí se doménová analýza a předběžné plány pro obsah jednotlivých verzí systému.

Fáze 2: Hra

Tato fáze je agilní částí celého procesu Scrum. Je iterativně opakována ve sprintech. O této vývojové fázi autoři mluví jako o fázi, kde nepředvídatelné je očekáváno. Různé proměnné, které Scrum definuje, jako například čas, kvalita, požadavky, zdroje, technologie a nástroje nebo vývojové metody, se mění během vývoje, takže jsou sledovány neustále v průběhu každého sprintu. Metodika tyto hodnoty nenastavuje na počátku vývoje a nesnaží se je pak udržet po celou dobu konstantní. Místo toho je kontroluje neustále, aby byla schopna pružně reagovat na změny a přizpůsobovat se měnícím podmínkám. Samotný vývoj, jak už jsem řekl, probíhá se sprintech, ve kterých se implementují nové funkcionality. Sprints můžou být plánovány na časové období od jednoho týdne až po jeden měsíc. V rámci jednoho sprintu pak probíhají všechny klasické vývojové fáze: sbírání požadavků, analýza, návrh, vývoj, testování, předání. Středně velký projekt může mít běžně tři až osm sprintů. Obrázek 3.10 ukazuje jednotlivé kroky, vstupy a výstupy v rámci jednoho sprintu.

Fáze 3: Dohra

Pokud zákazník ani vývojový tým nemůže přijít na žádné další požadavky nebo nové funkcionality, nepokračuje už vývoj dalším sprintem, ale spustí se poslední fáze – dohra. Úkolem této fáze je připravit produkt na finální uvolnění do oběhu nebo předání zákazníkovi. Provádí se integrace celého systému, testování mezi moduly, akceptační testy a vytváří se dokumentace.



Obrázek 3.10: Průběh jednoho sprintu metodiky SCRUM

3.5.5 Praktiky

Nyní popíšu jednotlivé praktiky, které dělají Scrum unikátním a možná proto také tak úspěšným.

Flexibilní předměty dodání (Flexible Deliverables)

Obsah dodávek není předem přesně dán, ale je závislý od typu projektu a prostředí, ve kterém nebo pro které se vyvíjí. Metodikou není striktně nařízeno, že výsledek analýzy musí být zpracován podle nějaké normy, protože v některých případech může být daleko užitečnější například objektový model nebo prototyp.

Odhad pracnosti (Effort Estimation)

Pracnost jednotlivých položek Product Backlogu je odhadována průběžně. Podle informací z průběhu vývoje se hodnoty neustále zpřesňují. Určování odhadů má na starost Vlastník produktu společně se členy týmu.

Plánovací schůzka sprintu (Sprint Planning Meeting)

Tuto schůzka je svolávána na začátku každého sprintu, organizuje ji Scrum Master a má dvě fáze. Té první se účastní zákazník, uživatelé, management, Vlastník produktu a Scrum tým. Jejich úkolem je stanovit cíle nadcházejícího sprintu a funkcionality, které budou implementovány. Položky vybírají z Product Backlogu a sestaví z nich Sprint Backlog. V něm nemusí být pouhá podmnožina položek Product Backlogu, ale některé z nich mohou být detailněji rozpracovány do více položek. Druhé fáze se účastní jen Scrum Master a Scrum tým a zaměřují se na konkrétní implementaci daných položek.

Každodenní schůzka (Daily Scrum Meeting)

Tato aktivita je nejtypičtější pro metodiku Scrum. Svolává ji Scrum Master a koná se každý den ve stejnou dobu na stejném místě a trvá přibližně 15 až 30 minut. Schůzky se aktivně účastní Scrum tým, pasivně pak mohou být přítomni členové managementu, zástupci zákazníka, apod. Ti však mohou jen poslouchat a nesmějí se aktivně podílet na diskuzi.

V rámci schůzky jsou každému členovi týmu pokládány tři otázky:

1. Co jsi udělal včera?
2. Co budeš dělat dnes?
3. Jsou nějaké překážky, které ti brání v práci?

Nalezené problémy se ovšem neřeší přímo na schůzce, pouze se zaznamenají a jejich řešení má později po skončení schůzky na starost skupina lidí, kterých se problém týká a jsou schopni ho vyřešit. Veřejné odpovídání na tyto otázky má tu kladnou vlastnost, že je celý tým každodenně informován, jaké práce na projektu právě probíhají, co už je hotovo a co se ještě má implementovat. Není cílem, aby šéf sbíral informace o tom, kdo je pozadu oproti plánu. Spíše jde o vzájemné svěřování se ostatním členům týmu s tím, na čem právě dělám, kdy to asi budu mít hotové, atd. Je velmi důležité, že si tyto názory sdělují vývojáři mezi sebou, má to úplně jiný efekt, než kdyby podávali správu o stavu projektu zákazníkovi nebo třeba account managerovi.

Hodnotící schůzka sprintu (Sprint Review Meeting)

Svolává ji Scrum Master společně se Scrum týmem poslední den sprintu. Účastní se jí Vlastník produktu, management, zákazník a zástupci uživatelů. Je vedena neformálně a jejím úkolem je prezentovat práci týmu v uplynulém sprintu, většinou formou nějaké demoverze. Sleduje se splnění cílů oproti jejich definování v průběhu plánovací schůzky sprintu. Schůzka může také přinést nové položky do Product Backlogu nebo usměrnit vývoj celého produktu podle vyjádření managementu, zákazníka a dalších kompetentních osob.

3.5.6 XP@Scrum

Rád bych se ještě krátce zmínil o snaze spojit Extrémní programování a Scrum do jednoho fungujícího celku. Možná se může zdát, že nejde kombinovat dvě samostatné metodiky, ale u těchto dvou to opravdu jde. Je to dáno tím, na jakou část vývojového procesu se jednotlivé metodiky primárně zaměřují. Extrémní

programování poskytuje integrované inženýrsko-programátorské praktiky, zatímco Scrum je zastřešuje metodami agilního managementu. Proto se tyto dvě metodiky nekříží, ale dobře se doplňují a rozšiřují tak svou působnost ve spektru procesu vývoje software.

Tato kombinace je s úspěchem používána v rámci firem spoluautorů metodiky Scrum, konkrétně Advanced Development Methods Inc., kde je prezidentem Ken Schwaber, PatientKeeper Corp., jehož CTO je Jeff Sutherland. I Mike Beedle, CEO firmy e-Architects Inc. a také spoluautor metodiky Scrum, začal vyvíjet metodiku spojující XP a Scrum, kterou nazval XBreed.

3.5.7 Závěr

Scrum je agilní metodika unikátní v tom, jak přistupuje k řízení celého procesu vývoje software. Zaměřuje se více než na samotné postupy implementace na manažerské řízení, spolupráci uvnitř týmu a organizaci jeho práce. Scrum se hodí spíše pro menší projekty a malé týmy, i když stále zvládá větší projekty než například Extrémní programování. Typická velikost Scrum týmu je 5-10 lidí, ovšem jsou i případy použití v daleko větších skupinách. Kupříkladu Mike Cohn, zakladatel firmy Mountain Goat Software, která se věnuje vývoji podle metodiky Scrum, použil Scrum v týmu více než 100 lidí, a Jeff Sutherland metodiku aplikoval dokonce v týmu čítajícím přes 600 lidí.

Hlavní výhodou metodiky Scrum je schopnost pružně reagovat na změny, které vznikají v průběhu vývoje projektu. Díky každodennímu reflektování změn a hledání rizik z nich vyplývajících je velmi snadné změnit směřování projektu podle přání zákazníka nebo pro zvýšení efektivity vývojového procesu. Nevýhodou metodiky Scrum je její zaměření na management projektu, tudíž obsahem metodiky nejsou žádné konkrétní implementační postupy. Sami autoři uvádějí, že nasazení Scrum je nejvhodnější v případech, kdy firma již nějakou metodiku využívá a zavedením Scrum se snaží zlepšit způsob vedení projektu, přičemž konkrétní postupy provádí podle své původní metodiky.

3.6 Test-Driven Development

Každý, kdo se někdy pustil do vývoje jakéhokoliv software, musí potvrdit, že testování je důležitou a nezastupitelnou činností. Ovšem autoři metodiky Test-Driven Development (dále jen TDD) nepovažují testování za jednu z fází vývoje, ale staví ho před všechno ostatní a přiřazují mu nejvyšší důležitost. TDD vlastně obrací klasický proces vývoje úplně naruby, když říká, že nejprve danou funkci otestujeme a až pak ji teprve napíšeme. Možná se to může zdát jako extrémní představa, jakpak by ne, když metodiky na svět přivedl Kent Beck, autor Extrémního programování. Poprvé TDD představil v [4].

3.6.1 Charakteristika

Test-Driven Development prosazuje myšlenku, že pro každou sebemenší funkcionalitu je nutné nejdříve napsat test, který ji dokáže dostatečně otestovat. Proto se TDD také někdy říká Test-First Development, tedy vývoj s testováním na počátku. Po tom, co máme napsán test, se teprve pouštíme do programování samotné funkcionality, přičemž se snažíme implementovat jen přesně takové množství, které je schopno projít testem, nic navíc. Následně hotový kód (jak funkce tak i testu) optimalizujeme, zpřehledníme a odstraníme duplicity. To je jedna iterace ve vývoji podle TDD. Stejně jako v Extrémním programování se i zde postupuje po velice malých iteracích trvajících dokonce i jen několik málo minut.

Ať už napsání testu pro neexistující funkcionalitu může znít jakkoliv zvláštně, ještě to není všechno. Správným postupem je nejprve napsat test, který selže. Selže v důsledku toho, že ještě není implementována funkcionalita, kterou má testovat. Pak se teprve pustíme do implementace testované funkce a skončíme právě ve chvíli, kdy funkce testem projde. Poté nastává fáze refaktORIZACE, neboli úprava zdrojového kódu bez změny funkčnosti, abychom dosáhli přehlednějšího a efektivnějšího kódu.

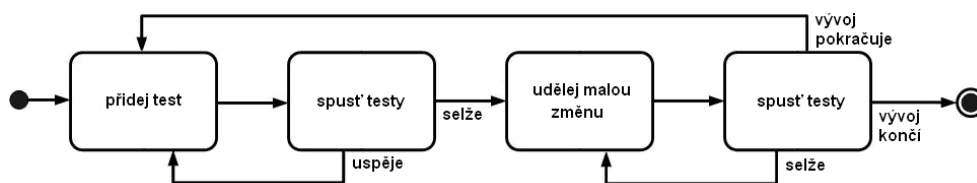
V rámci metodiky TDD se mluví o tzv. *testování jednotek*. Co to tedy ta jednotka je? Je to velmi malý, specifický kousek funkcionality. Pokud chceme tuto jednotku kvalitně otestovat, musí test splňovat několik jednoduchých zásad:

- **jednoduchost** – test musí být specifický, zaměřovat se právě jen na danou funkcionalitu, kterou má testovat, na nic víc; pokud je funkcionalita příliš složitá, rozdělíme ji do více logických částí a pro každou píšeme test zvlášť
- **rychlost** – test musí proběhnout rychle; vzhledem k tomu, že ho nespouštíme samostatně, ale vždy všechny testy najednou, nemůžeme při pětiminutových iteracích čekat půl hodiny na výsledek série testů
- **nezávislost** – testy musí být nezávislé mezi sebou, nezávislé na prostředí, ve kterém jsou spuštěny, a také nezávislé na pořadí, do kterého jsou při spouštění seřazeny
- **kvalita** – kód testu musí být stejně kvalitní, jako samotný produkční kód aplikace; test není jen něco, co se provizorně narychlo napíše; pokud má kvalitně testovat, musí být kvalitně napsán, proto se také v poslední fázi iterace (refaktORIZACE) zefektivňuje nejen kód funkce, ale také kód samotného testu
- **automatizace** – jak už jsem říkal, testy se nespouštějí samostatně, ale vždy ve velkém balíku s ostatními testy; není přípustné, aby se test zastavil a čekal na nějaký vstup od uživatele, vše musí být připraveno tak, aby to mohlo proběhnout automatizovaně

Co však nemůže být považováno za test jednotky, je test, který neběží izolovaně od okolí. Pokud se test kupříkladu připojuje k databázi, k síti nebo pracuje s fyzickým souborem, pak nemůže být považován za test jednotky. Potřebujeme-li testovat i takovéto situace, pak kvůli testování v izolaci používáme objekty, kterým se říká „falešné končetiny“ nebo „pahýly“. To jsou simulace reálných zdrojů, které se tváří navenek funkčně, i když pravou funkčnost daného objektu nemají.

3.6.2 Procesy

Jednotlivé fáze, kterými vývojář pracující v duchu TDD prochází, přehledně znázorňuje diagram 3.11 (obrázek je převzatý z [2]).



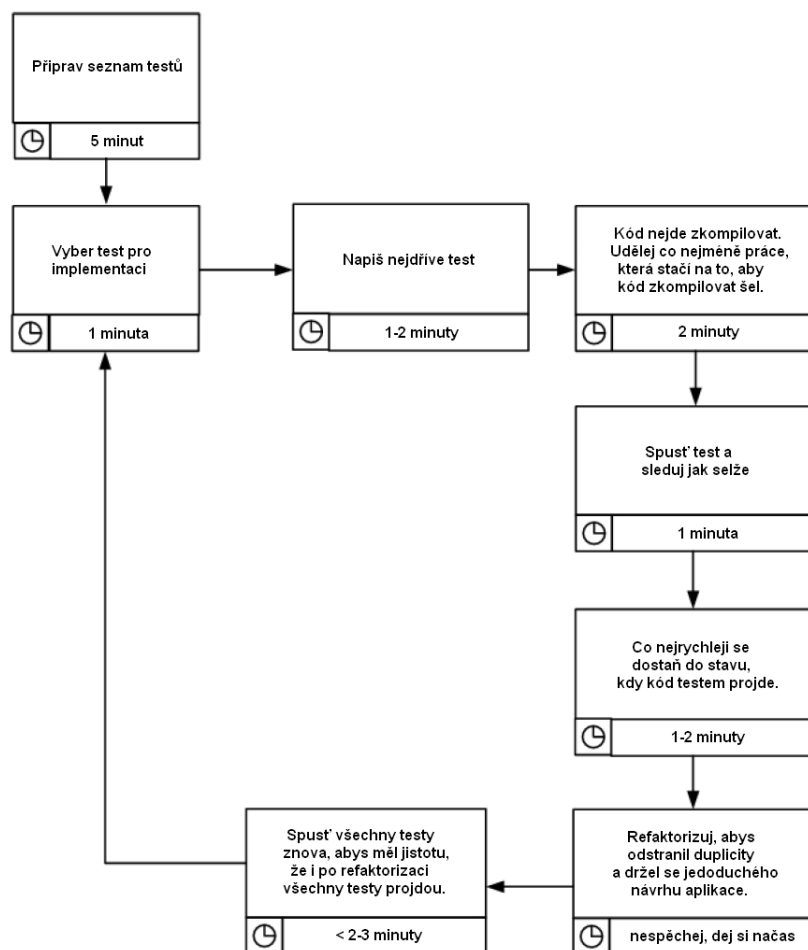
Obrázek 3.11: Schéma vývoje podle metodiky TDD

Nyní popíšu jednu iteraci v procesu vývoje podle TDD:

1. Co nejrychleji vytvoříme test, konkrétně test, který selže díky chybějící funkcionalitě.
2. Spustíme všechny dosavadní testy, abychom se přesvědčili, že test napsaný v bodě 1 opravdu selže a zároveň nekoliduje s žádných z předchozích testů.
3. Teprve v této fázi napíšeme potřebnou funkcionalitu (třídu, funkci, modul) a zajistíme, že tento kód projde testem, který jsme napsali v bodě 1.
4. Spustíme všechny testy nad tímto novým kódem. Pokud některý z nich selže, upravíme daný kus kódu a spustíme testy znova. Takto pokračujeme až do doby, kdy všechny testy uspějí.
5. Upravíme testovací případ i funkční kód, abychom odstranili duplicity a přesuneme testovací případ do sady testů spouštěných automaticky.

Již v úvodu jsem zmiňoval, že iterace v rámci TDD jsou velice krátké, v délce řádově několik minut. Velice pěkně ilustruje časový průběh iteračního cyklu obrázek 3.12, který pochází z [10].

Jistě si lze všimnout, že proces začíná napsáním testu pro nějakou funkcionalitu, kterou chceme implementovat. TDD se vývojem zabývá až od této chvíle. Žádné



Obrázek 3.12: Časový průběh vývoje podle metodiky TDD

praktiky a postupy, jak vůbec najít chybějící funkcionalitu nebo jak vybrat k implementaci právě tuto, bychom v metodice TDD hledali marně.

Kent Beck definoval následující dvě pravidla pro použití TDD (která popsal v [4]):

1. Nový produkční kód píšeme jen tehdy, když automatizovaný test selhal.
2. Musíme odstranit všechny duplicity, které najdeme.

Autor k těmto dvěma pravidlům také publikoval pár dalších zásad, které mají vysvětlit jejich opodstatnění a pomoci při vývoji podle TDD:

- Testy pro svůj kód si píší vývojáři sami, protože ti nejlépe dokáží stanovit, co má test dělat a jakou funkcionalitu má testovat. Při tak krátkých iteracích,

jaké TDD prosazuje, by bylo značně neefektivní, kdyby musel programátor čekat, až pro něj někdo jiný napíše test.

- Vývojové prostředí musí být dostatečně rychlé, aby dokázalo poskytovat patřičnou odezvu na časté změny. TDD nelze použít v případě, kdy přidání funkcionality trvá dvě minuty, přičemž dokončení všech testů zabere další půlhodinu.
- Je dobré mít kvalitní návrh architektury aplikace, ideálně nezávislé komponenty spojené přes vhodný middleware. Díky dobré architektuře je nejen testování snazší, ale výsledný produkt je také lépe udržovatelný a rozšiřitelný.

Klíčová technika, kterou by si všichni vývojáři pracující podle TDD měli osvojit, je umění vytvářet efektivní testy. Jaké vlastnosti by tedy měl mít takový kvalitní a efektivní test?

- je rychlý a nemusí se nijak složitě konfigurovat
- jeho pořadí v sadě testovacích případů je nezávislé na ostatních testech, každý test musí běžet samostatně, nemělo by záviset na pořadí, ve kterém jsou jednotlivé testy spouštěny
- vstupem i výstupem jsou data, která jsou snadno pochopitelná a ověřitelná
- používají reálná data, například kopie produkčních dat
- mělo by být z testu jasné, jakou novou funkcionalitu systému přináší a jak posunuje vývoj směrem k cíli

3.6.3 Praktiky

Důležitou praktikou, kromě psaní testů před psáním kódu, je udržování tzv. *to-do seznamu*. Zde se uchovávají jednak příklady testů, které chceme napsat, a pak také veškeré úpravy v rámci refaktORIZACE, která je potřeba udělat. To-do seznam by měl být udržován stále aktuální, připraven poskytnout informaci o tom, jaké testy a refaktORIZACE jsou právě prováděny, a jaké jsou do budoucna naplánovány. Měl by ochránit programátora před chaosem a záplavou různými testovacími případy a dát vývoji nějaký řád. V neposlední řadě poskytuje to-do seznam informaci o pokroku ve vývoji projektu.

Je důležité, a v prvních fázích nasazení TDD také dost obtížné, nesklouznout od principů TDD ke starým zvykům. Programátor si musí dávat pozor, aby nikdy nepsal více selhávajících testů najednou, nepsal nový test bez odstranění duplicit v tom předchozím nebo dokonce nenechal v aplikaci kód, který neprošel testem úspěšně.

I když se TDD zaměřuje primárně na testy, není to jak říká Ward Cunningham testovací technika. Je to programátorská technika, takže i při použití TDD musíme projít například testováním uživatelského rozhraní, integračními testy nebo akceptačními testy u zákazníka. Automatizovaně testovat uživatelské rozhraní je téměř nemožné, proto by se mělo programovat co nejtenčí, aby nebylo tak náchylné na chyby.

Mohlo by se zdát, že v rámci TDD neexistuje žádná fáze návrhu. Opak je pravdou. Návrh je podle autorů přítomen v každé iteraci právě díky testům, které jsou psány ještě před samotnou funkcionalitou. Když má totiž člověk nejdříve napsat test, musí si dobře promyslet, jak bude daný modul nebo komponenta fungovat, aby věděl, jak má daný test vlastně napsat. Nejdříve si programátor musí rozmyslet co testovat, potom jak to testovat a nakonec teprve napíše příslušný test. A právě v tomto kroku je schován návrh aplikace. Je tedy přítomen v celém procesu, v každé iteraci, vymýšlením testů vývojář de facto navrhuje daný systém. Marcus Baker, hlavní vývojář firmy Wordtracker pracující podle metod Extrémního programování a TDD, uvádí jako ideální životní cyklus software posloupnost testování – implementace – návrh, tedy postup přesně opačný než jak ho definují všechny tradiční metodiky.

Jak jsem popisoval výše, testy by měly probíhat automatizovaně. To se těžko dá při větších projektech uhlídat ručně, proto samozřejmě existují různé nástroje pro podporu automatizovaného testování. Mezi nejznámější patří rodina aplikací známých jako xUnit. Existuje více než 30 implementací pro nejrůznější programovací jazyky, některé z nich jsou komerční, jiné open-source. První byla implementace pro SmallTalk nazvaná SUnit, později přibýly další jako JUnit pro Javu, VUnit pro Visual Basic, cppUnit pro C++, PyUnit pro Python, NUnit pro platformu .NET, a spousta dalších.

3.6.4 Závěr

Test-Driven Development je zajímavá metodika, která je založená na myšlence nenapsat jediný kus kódu bez toho, aniž by pro tento kód neexistoval předem napsaný test. Při použití TDD neexistuje fáze hledání chyb, která se běžně označuje jako *debugging*. Autoři dokonce mluví o metodě *pre-bugging*, tedy odstraňování chyb ještě dříve než vzniknou.

Vzhledem k tomu, že TDD je celý orientován na zdrojový kód, tak si nemyslím, že by mohl být použitelný jen sám o sobě. Často se o TDD mluví jako o jedné z technik Extrémního programování. Podle mého názoru je to ale tak úzce zaměřená metodika, která nepříliš ovlivňuje ostatní procesy, že může být použitelná v souvislosti s většinou dalších metodik, které jí poskytnou potřebné procesní řízení a manažerské zázemí.

Kapitola 4

Případové studie

V následující kapitole nejprve nadefinuji ukázkový projekt a pak uvedu postupy, které použijí tři výše popsané metodiky při realizaci tohoto projektu.

4.1 Ilustrační projekt

Cílem projektu bude realizovat níže specifikovaný informační systém, který usnadní správu a sledování dostupnosti lidských zdrojů ve firmě zadavatele. Půjde o informační systém vytvořený na zakázku, nikoliv o sériově vyráběný software nebo customizovaný produkt.

Zadavatel

Zadavatelem projektu je grafické studio, které poskytuje služby v oblasti zpracování reklamy, DTP služeb, interaktivních multimediálních prezentací, výroby propagačních předmětů apod. Firma má stále pracovníky, většinou na pozicích project manager nebo account manager. Na grafické a designerské práce si najímá externí specialisty. Ti mají spoustu dalších povinností, často jsou to studenti, takže nejsou dostupní po celou pracovní dobu, tak jako interní zaměstnanci. Proto je důležité sledovat jejich časové možnosti a snažit se přizpůsobovat projektové plány omezeným a měnícím se časovým možnostem.

Řešitel

Řešitelem projektu je firma, která se specializuje na vývoj software na zakázku. Nejčastěji jsou to informační systémy pro správu organizací a řízení činností ve firmách.

Současný stav

V současnosti je správa externích lidských zdrojů realizována bez podpory softwarového informačního systému. Dosud ve firmě zadavatele pracovalo jen málo externistů, takže se jejich správa vedla pouze v papírové podobě. S tím, jak se firma rozrůstá a prosperuje, přijímá stále více externích pracovníků a cítí potřebu vést evidenci a plánování jejich práce elektronicky.

Protože původní systém nepracoval s mnoha daty, mohl se o plánování a organizaci práce starat jeden člověk. Většinu údajů byl schopen spravovat sám, takže žádná centrální evidence nebyla potřeba. V případě příchodu nového člena týmu pouze rozeslal všem stávajícím členům zprávu o tomto novém člověku, spolu s kontakty na něj a případně i s komentářem o jeho zkušenostech a plánovaném zapojení do projektů firmy.

Pokud tedy ve firmě působilo 5-6 externistů a přibližně stejný počet manažerů, nebyl problém se vzájemným sdílením informací. Jak ale počet lidí ve firmě rostl a nejen externisté, ale postupně i manažeři začali být najímáni z celé republiky, vzrůstala také tendence lépe sdílet kontakty mezi jednotlivými členy týmu, a v neposlední řadě také usnadnit plánování projektů tím, že se budou detailněji sledovat a uchovávat informace o časové dostupnosti jednotlivých pracovníků.

Na základě tohoto rozhodnutí byla učiněna poptávka na projekt.

Poptávka

Cílem realizovaného systému bude poskytování informací o projektech, externích lidských zdrojích (EZ) a evidování prací externistů na jednotlivých projektech. Systém umožní spravovat projekty a EZ z pohledu stanovování termínů, přerozdělování kapacit a sledovat a v reálném čase měnit rozdělení kapacit EZ. Systém bude sledovat aktuální přiřazení jednotlivých EZ k projektům a hlídat kolize v termínech, upozorňovat na nedostatek hodinové kapacity některého EZ, nabízet nejbližší volné termíny, apod.

Realizovaný informační systém by měl vycházet se současného způsobu vedení evidence lidských zdrojů. V podstatě by měl zachovat funkcionalitu původního systému a navíc poskytnout funkce, které umožňuje elektronický způsob ukládání dat.

Celý systém by měl být jednoduše použitelný, bez velkých nároků na hardware. Měl by respektovat to, že projektoví manažeři jsou často na cestách a externisté pracují ve svém prostředí. Proto nelze spoléhat na homogenitu prostředí, ze kterého budou uživatelé do systému vstupovat. Systém by se měl vyvíjet s ohledem na pozdější úpravy a vylepšení, která budou přicházet společně s tím, jaké bude jeho použití v praxi.

Prvotní specifikace

Dokument s prvotní specifikací byl sestaven řešitelským týmem po prozkoumání poptávky a úvodní schůzce se zadavatelem. Je záměrně napsán neformálním jazykem, aby byl lépe přístupný zadavateli, jehož zapojení do tvorby tohoto dokumentu je pro projekt klíčové. Vzhledem k tomu, že cílem této práce není podrobná specifikace projektu a přesných postupů jeho řešení, ale popis aplikace dané metodiky, je tento dokument obsahem přílohy A.

Změny ve specifikaci požadavků po přezkoumání návrhu systému zadavatelem

Po kontrole a přezkoumání celého principu byly zadavatelem navrženy změny, které byly zpracovány jako dodatek k prvotní specifikaci. Stejně tak byl pro celý systém vypracován ERD diagram popisující datové závislosti jednotlivých objektů a use-case diagram znázorňující uživatele a jejich interakci se systémem. Všechny tyto dokumenty jsou součástí přílohy B.

4.2 Řešení projektu dle zvolených metodik

V minulé kapitole jsem nadefinoval ukázkový projekt, na kterém nyní ukážu řešení podle tří metodik, které byly teoreticky popsány v kapitole 3. Proces vývoje software v sobě zahrnuje řadu fází a činností. Ne všechny metodiky se zaměřují na vývoj od počátku až do konce. Zaměřují se na různé části životního cyklu software, takže některé aktivity, které jsou při vývoji nutné, nejsou v dané metodice vůbec popsány. V některých metodikách je přímo uvedeno, na jaké aspekty se zaměřují a co již obsahem konkrétní metodiky není. Jinde je pouze naznačen doporučený seznam postupů, kterých by se tým měl při vývoji držet, a cílů, které by měl sledovat. Jak ale těchto cílů dosáhnout už metodika nespecifikuje. Málokterá metodika je tak obsáhlá, aby pokrývala kompletní životní cyklus budoucího SW produktu. Příkladem takové metodiky je Rational Unified Process, ale většina volně dostupných metodik takto podrobně rozpracována není.

4.2.1 Řešení podle FDD

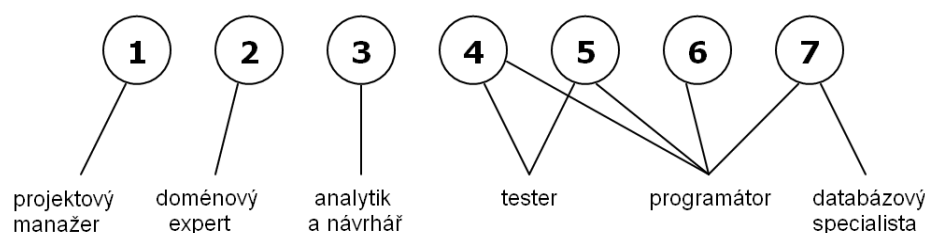
Metodika FDD popisuje 5 fází vývojového procesu: vytvoření celkového modelu, podrobný seznam vlastností, plánování podle vlastností, návrh podle vlastností a implementace podle vlastností. Nejdříve popíšu složení vývojového týmu, pak jednotlivé fáze.

Vývojový tým

Vývojový tým má 7 členů, kteří se dělí o 6 rolí. Uvedu seznam rolí a u každé z nich v závorce počet osob zastupujících danou roli:

- Doménový expert (1)
- Projektový manažer (a současně hlavní programátor) (1)
- Analytik a návrhář (hlavní architekt) (1)
- Databázový specialista (1)
- Programátor (4)
- Tester (2)

Celkový počet osob pro všechny role je 10, v týmu je ale jenom 7 lidí, což značí, že někteří členové týmu musí zastávat více než jednu roli. Jeden z programátorů je zároveň i databázovým specialistou, další 2 programátoři současně testují. Rozdělení rolí v týmu je naznačeno na obrázku 4.1.



Obrázek 4.1: Rozdělení rolí v týmu

Fáze 1: vytvoření celkového modelu

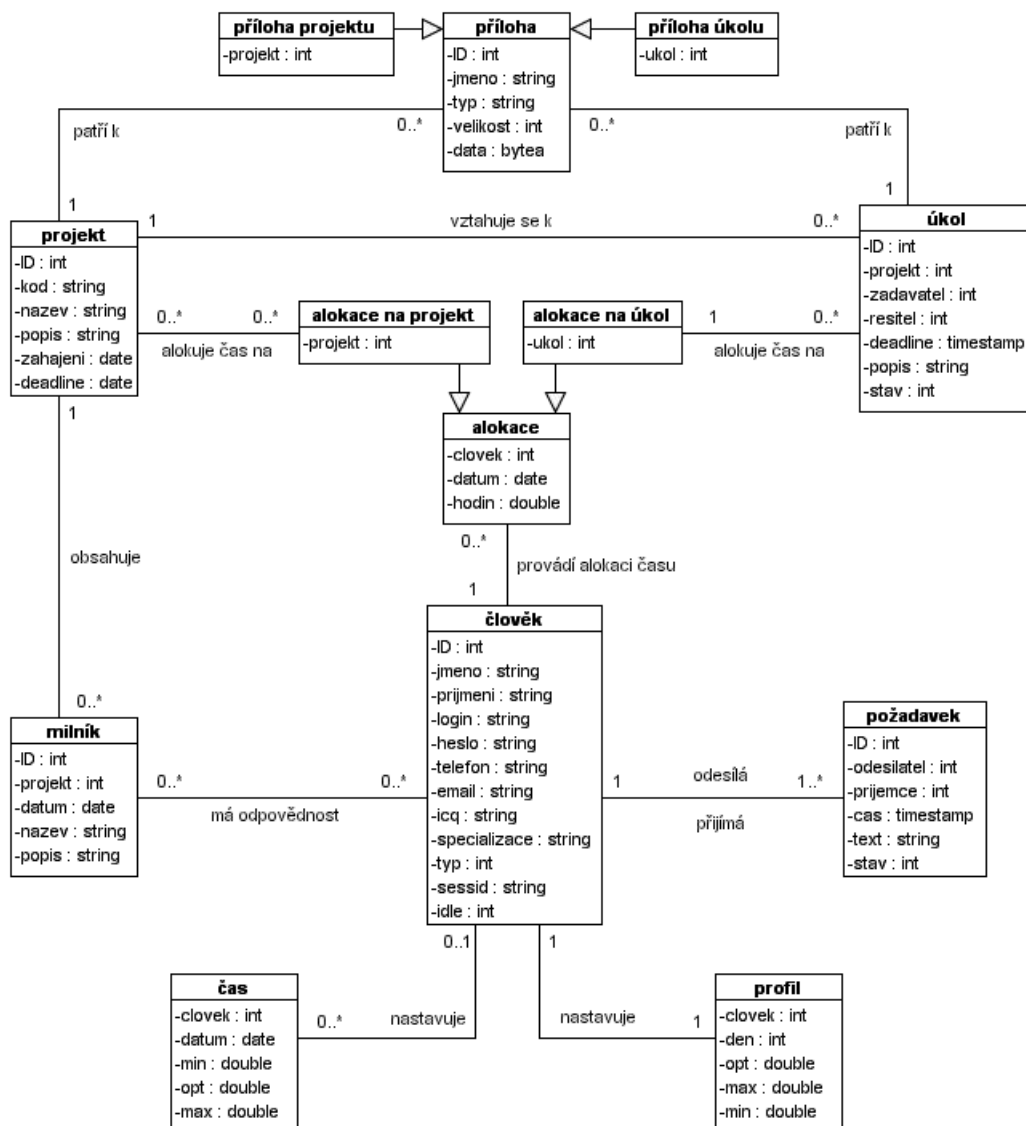
Celkový (globální) model by měl poskytovat pohled na systém jako celek a demonstrovat jeho základní účel. Nejčastěji je reprezentován jako diagram tříd. Model se vytváří po tom, co doménový expert seznámí vývojový tým s cílem projektu a hlavními požadavky na systém.

Hlavní architekt ve spolupráci s doménovým expertem navrhl následující model zakreslený pomocí diagramu tříd (obrázek 4.2).

K diagramu byl také zpracován následující popis všech objektů.

Člověk

Vzhledem k tomu, že systém má sloužit ke sledování externích lidských zdrojů,



Obrázek 4.2: Diagram tříd

je třída „člověk“ hlavní a nejdůležitější v celém systému. Pracuje s lidmi, kteří v systému jakýmkoliv způsobem figurují, ať už aktivně nebo jen pasivně. Úlohu a práva člověka k systému určuje atribut „typ“. Většina atributů uchovává nej-různější kontakty na daného člověka, seznam lidí slouží pro všechny zaměstnance jako jakýsi adresář kontaktů.

Projekt

Sem se ukládají informace o projektech, která daná firma dělá, a na kterých pracují její zaměstnanci.

Milník

Milníky jsou kontrolní body projektu, identifikované svým datem a názvem. Může to být například „dokončení prvotního návrhu“, „dodání designu webu od grafika“, „začátek testování“, apod.

Úkol

Pojmem úkol se rozumí nárazová, většinou krátkodobá, práce na určitém projektu, kterou ale nevykonává člen daného projektového týmu. Slouží například k zadávání ad hoc úkolů konkrétním lidem, které není vhodné dlouhodobě vést jako členy projektového týmu. U úkolu se sleduje jeho „stav“, který mění řešitel úkolu, a který nabývá hodnot „zadán“, „akceptován“ a „dokončen“.

Profil

Jedná se o časový profil patřící danému člověku. Každý člověk má v profilu 7 záznamů, pro každý den v týdnu jeden. Každý záznam pak obsahuje 3 hodnoty udávající minimální, optimální a maximální počet odpracovaných hodin.

Čas

Třída podobná třídě „profil“, s tím rozdílem, že zde jsou 3 výše zmíněné časové hodnoty vztaženy už k určitému danému datu. Uchovává se zde časová dostupnost jednotlivých lidí.

Požadavek

Požadavek je krátká textová zpráva, kterou jakýkoliv člověk může poslat jinému člověku. Projektový manažer může přes tento systém kupříkladu poslat požadavek člověku, starajícím se o lidské zdroje ve firmě, že mu kapacita jeho týmu nestačí pro všechny práce na projektu a požaduje tedy rozšíření svého projektového týmu o dalšího člověka. Stejně jako úkoly mají i požadavky svůj „stav“, konkrétně je to jedna z hodnot „poslán“, „akceptován“ nebo „vyřízen“. Stav požadavku může měnit pouze jeho příjemce. Odesílatel má pak přehled o tom, jestli se jeho požadavku adresát věnuje.

Příloha

Uchovává přílohy (soubory), které může projektový manažer uložit do systému k danému projektu nebo úkolu. Můžou to být nejrůznější podklady a dokumenty, jako například návrh grafického designu aplikace, specifikační dokument, apod. Všichni členové týmu je díky tomu mají neustále k dispozici ke stažení.

Příloha projektu

Specializace třídy „Příloha“, která určuje konkrétní projekt, ke kterému je příloha určena.

Příloha úkolu

Specializace třídy „Příloha“, která určuje konkrétní úkol, ke kterému je příloha určena.

Celkový model může být ještě podroben inspekci a schválení zadavatelem, ale tento konkrétní model je natolik jednoduchý, že stačila kontrola mezi jednotlivými členy

vývojového týmu.

Fáze 2: vytvoření seznamu vlastností

Nalezení a sestavení seznamu vlastností mají na starost vedoucí programátoři (chief programmers), kteří se podíleli na sestavování celkového modelu ve fázi 1. V tomto projektu je vedoucí programátor jen jeden, ale obecně jich u větších projektů může být více.

Seznam vlastností je rozdělen do skupin podle toho, které vlastnosti spolu logicky souvisejí. Takový seznam může být poměrně dlouhý, u větších projektů může čítat i stovky položek, proto zde uvedu jen jeho část. Vezměme kupříkladu část systému zabývající se nastavením času a jeho alokací na projekty.

Nejdříve se naleznou skupiny vlastností:

- Nastavení časového profilu
- Nastavení času
- Alokace času na projekt

Poté se v každé skupině naleznou a popíší jednotlivé vlastnosti:

1. Nastavení časového profilu
 - 1.1. Vypsání časového profilu na obrazovku
 - 1.2. Uložení modifikovaného profilu do databáze
2. Nastavení času
 - 2.1. Vypsání aktuálního nastavení času do formuláře
 - 2.2. Uložení nastavení času do databáze
3. Alokace času na projekt
 - 3.1. Vypsání výběru projektů na obrazovku
 - 3.2. Zobrazení formuláře pro alokaci času s možností volby měsíce
 - 3.3. Uložení nastavených alokací do databáze

Podobným způsobem se sestaví kompletní seznam vlastností.

Fáze 3: plánování podle vlastností

Úkolem v této fázi je seřadit činnosti do posloupnosti, v jaké se budou jednotlivé vlastnosti implementovat, předběžně rozvrhnout práce, včetně stanovení datumu ukončení vývoje. Dále se jednotlivým programátorům mají přiřadit třídy, za jejichž úspěšné dokončení budou odpovídat.

Tým pro plánování (sestavající z projektového manažera a všech vedoucích programátorů) stanovil posloupnost vlastností a přiřadil každé vlastnosti její prioritu (tabulka 4.1).

pořadí	vlastnost	priorita
1.	(2.1.) Vypsání aktuálního nastavení času do formuláře	vysoká
2.	(2.2.) Uložení nastavení času do databáze	vysoká
3.	(3.1.) Vypsání výběru projektů na obrazovku	vysoká
4.	(3.2.) Zobrazení formuláře pro alokaci času s možností volby měsíce	vysoká
5.	(3.3.) Uložení nastavených alokací do databáze	vysoká
6.	(1.1.) Vypsání časového profilu na obrazovku	střední
7.	(1.2.) Uložení modifikovaného profilu do databáze	střední

Tabulka 4.1: Seznam vlastností

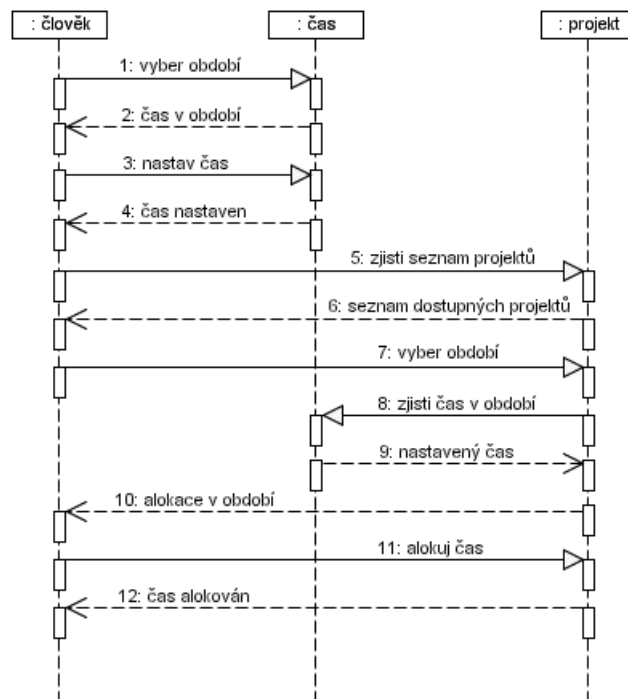
Vysoká priorita znamená, že daná vlastnost je nutná k fungování systému a musí být implementována před jeho nasazením. Vlastnosti týkající se časového profilu budou implementovány až nakonec, mají také nižší prioritu, protože nejsou pro funkčnost systému jako celku nezbytně nutné. Ve specifikačním dokumentu je časový profil uveden jako požadavek, tato funkce také bude implementována, ale plánovací tým správně odhadl, že odložení prací na této části systému neohrozí jeho dokončení.

Fáze 4: návrh podle vlastností

Tato fáze se zabývá už jednotlivou vlastností, kterou vybere hlavní programátor ze seznamu vlastností, vybírá podle priorit a vzájemných závislostí. Jediným výstupem může být sekvenční diagram nebo diagram spolupráce. V tomto případě byl pro skupinu vlastností uvedenou ve fázi 2 sestaven sekvenční diagram, viz obrázek 4.3. Znázorňuje komunikaci tříd při zadávání času a jeho alokaci na konkrétní projekt.

Nesmíme ještě zapomenout ověřit celkový model oproti detailnímu návrhu vlastností, může se totiž stát, že díky tomu budeme potřebovat celkový model vytvořený v první fázi upravit. V takovém případě se na chvíli vrátíme do fáze 1, upravíme model, a projdeme znovu fáze 2 a 3, abychom zapracovali všechny vzniklé změny.

V tomto konkrétním případě k žádným změnám při návrhu podle vlastností nedošlo, takže globální model může zůstat beze změny.



Obrázek 4.3: Sekvenční diagram

Fáze 5: implementace podle vlastností

Nastává fáze, kdy se tým pouští do funkcí potřebných pro implementaci vybrané vlastnosti. Metodika FDD specifikuje naprogramovat, provést inspekci, otestovat jednotky, integrovat, sestavit. Jak konkrétně a za použití jakých nástrojů by tyto činnosti měly být prováděny už metodika nespecifikuje. Volba je na hlavním programátorovi, který by měl mít dostatek zkušeností, aby pro vývojový tým zvolil ty správné prostředky a postupy.

4.2.2 Řešení podle SCRUM

SCRUM je metodikou, která docela přesně definuje co se má ve které fázi vývoje dělat, kdo to má na starosti a co by mělo být výsledkem. Tady ovšem veškerý dostupný popis končí. Dozvíme se co máme udělat, ale už se nedozvíme, jak bychom toho měli dosáhnout. Konkrétní formu realizace jednotlivých kroků musí stanovit Scrum Master, to je ovšem na jeho znalostech a zkušenostech, metodika jako taková mu v tom nijak nepomůže. Na tento fakt lze nahlížet ze dvou pohledů. Buď to můžeme brát jako nedostatek a proto nám může metodika připadat nekompletní, nebo se na to budeme dívat jako na příležitost jak do metodiky vnést své vlastní postupy a vývojové procesy a možnost přizpůsobit si metodiku vlastním potře-

bám. V tom případě nám SCRUM dává volnou ruku a nesvazuje nás žádnými striktními pravidly.

Vývojový tým

Ze strany zadavatele je v týmu přítomna osoba označována jako *zákazník* (customer). Na straně řešitele je nejdůležitější rolí *Scrum Master*. Ten se musí starat o to, aby vývoj probíhal podle praktik a zásad metodiky SCRUM. *Vlastník* (product owner) je zodpovědný za projekt jako takový, úzce spolupracuje se Scrum Masterem na sestavování Product Backlogu, při vybírání, které vlastnosti budou mít jakou prioritu a kolik času bude věnováno jejich implementaci. Pokud není na tuto roli stanoven samostatný člověk, pak tuto roli vykonává ten, kdo je zároveň Scrum Master. To je náš případ. Posledním článkem je *Scrum Team*, tedy tým podílející se na sestavování Sprint Backlogu a na návrhu a implementaci jednotlivých vlastností. Detailnější rozdělení tohoto týmu metodika neuvádí, jen dodává, že Scrum Team by měl být samoorganizující tým s dobrou komunikací, který je schopen se dohodnout na tom, kdo bude například odpovědný za testování nových vlastností, kdo bude mít na starosti zakreslování návrhů nových vlastností v UML, apod. Vývojový tým pro tento projekt se tedy skládá ze 7 lidí. Jeden z nich je zákazník, další je v roli Scrum Mastera a zároveň Vlastníka produktu. Zbývajících 5 členů tvoří Scrum Team.

Vývoj podle SCRUM probíhá ve třech po sobě jdoucích fázích, kterým se říká předehra, hra a dohra. První a třetí fáze jsou lineární, prostřední fáze je iterační.

Fáze 1: předehra

Formálně se tato fáze skládá ze 2 kroků: prvním je *Plánování*, druhým pak *Návrh a architektura*. Při *plánování* má nejvíce úkolů Scrum Master. Musí sestavit harmonogram, zajistit potřebné zdroje (ať už lidské, softwarové nebo materiální), zvolit vývojové nástroje, provést analýzu rizik. Ovšem nejdůležitější dokument, který z této fáze vzejde je Product Backlog. Udržuje ho Vlastník produktu (často společně se Scrum Masterem) po celou dobu vývoje aplikace, většinou ve formě listu v některém tabulkovém procesoru jako například OpenOffice.org Calc nebo Microsoft Excel. Příklad takového backlogu uvádím v tabulce 4.2. Pro ukázkou postačí jeho část, v kompletní verzi by Product Backlog měl obsahovat co nejvíce požadavků vycházejících ze specifikace. Předpokládá se, že se Product Backlog bude v průběhu vývoje podle situace postupně doplňovat a rozrůstat, to má na starosti Scrum Master.

Každé položce je přiřazena priorita, očekávaná doba implementace (vyplňují členové týmu) a člověk, který je za implementaci zodpovědný (vzhledem k tomu, že vývojový tým není příliš velký, pro přehlednost se používají jen iniciály). Předpokládané doby trvání jsou pouze orientační, musí se počítat s tím, že jednotliví

priorita	položka	popis	doba	kdo
velmi vysoká				
	–	návrh databáze	–	–
	1	ERD diagram	20	MR
	2	struktura databáze	8	MR
	3	kontrola omezení (cizí klíče)	4	LP
	–	kostra aplikace	–	–
	4	práce s databází	32	VF
	5	obecná kostra všech stránek	16	JS
	6	definice hlavního menu	6	JS
vysoká				
	–	přehledy	–	–
	7	přehled lidí	7	LP
	8	přehled projektů	12	LP
	9	přehled úkolů	8	LP
	10	přehled požadavků	3	LP
	–	admin statistiky	–	–
	11	vytíženost EZ	20	VF
	12	vytíženost projektů	20	VF
	13	zaměstnanost EZ	18	VF
	14	přehled alokací	26	LP
střední				
	–	design	–	–
	15	hlavní menu rozbalovací	10	MR
	16	volba individuálního designu	5	MR

Tabulka 4.2: Product Backlog

členové týmu nejsou na počátku vývoje schopni přesně určit, jak dlouho bude vývoj dané položky trvat. Scrum Master to musí mít na paměti a brát tyto hodnoty s rezervou, používají se jen pro hrubé plánování.

V kroku *Návrh a architektura* se navržená architektura může modifikovat například podle toho, jak proběhla analýza rizik v průběhu Plánování. Dále se zde provádí doménová analýza, tzn. podrobné zkoumání prostředí zadavatele, kde bude vyvíjený software nasazen, se všemi jeho obchodními procesy, kterých se daná aplikace týká. S tím souvisí i studium všech dostupných a relevantních dokumentů, které většinou poskytne vývojovému týmu zadavatel.

Fáze 2: hra

Herní fáze sestává z několika sprintů. Jejich počet a délka závisí na druhu a složitosti aplikace, na velikosti vývojového týmu, a dalších parametrech. Na začátku

každého sprintu vybere manažer projektu skupinu položek z Product Backlogu, které se budou implementovat v rámci tohoto sprintu. Tato skupina tvoří Sprint Backlog. Vývojový tým pak jednotlivé položky Sprint Backlogu podrobněji rozebere a stanoví plán vývoje. V našem případě není projekt příliš složitý, proto zvolíme délku trvání sprintu 14 dní. Příklad Sprint Backlogu uvádím v tabulce 4.3.

	Dnů do konce sprintu	14	12	8	3	1
kdo	popis	Zbývá hodin				
		1. 3. 2005	3. 3. 2005	7. 3. 2005	12. 3. 2005	14. 3. 2005
	Databáze					
LP	přidat tabulku pro logování	4	4			
MR	upravit referenční integrity u úkolů	8	8	8		
LP	vytvořit indexy	2	2	2	2	
	Bezpečnost					
JS	šifrování hesel při přihlášení	12	12	8		
LP	logovat neúspěšné pokusy o přihlášení	14	7	7	2	
	Prostředí					
VF	doinstalovat do PHP podporu knihovny pro odesílání hromadných e-mailů s přílohou	10				
VF	testování nové knihovny	3	3			
	Design					
JS	nastavení hlavního menu do designu aplikace	6	4	4	4	
JS	nečtvercovým ikonám přidat průhledné pozadí	2	2			

Tabulka 4.3: Sprint Backlog

V průběhu celého sprintu tento backlog udržuje Scrum Master, zanáší do něho aktualizace času odpracovaného na jednotlivých úkolech, případně doplňuje o nově vzniklé úkoly.

Další pro SCRUM klíčovou činností je kromě udržování backlogů také pořádání každodenních setkání vývojářů (Daily Scrum Meetings). Jejich pořádání a vedení má na starost opět Scrum Master. Pokládá vývojářům sadu předem připravených otázek. Uvedu příklad pár otázek a jejich možných odpovědí.

Co jste udělali od poslední schůzky?

Byly dokončeny formuláře pro výběr projektu a následnou alokaci času na projekt.

Vyskytly se při realizaci nějaké překážky?

Musel být upraven CSS styl pro zobrazení seznamu dostupných projektů.

Detekovali jste nějaká nová rizika?

Nebyly dostatečně kontrolovány hodnoty předávané skriptu v URL, což způsobilo, že člověk mohl přistupovat i k projektu, ke kterému by neměl mít přístup.

Na čem budete pracovat do příští schůzky?

Odstranění chyby v předávání hodnot přes URL. Následně pak zpracovat analogický formulář pro alokaci času na úkoly.

Napadají vás nějaké možnosti ke zlepšení nebo zefektivnění práce?

Vytisknout ERD diagram celé aplikace na velký formát a vyvěsit ho v místnosti na viditelné místo, aby ho všichni měli na očích a dostupný pro případné poznámky.

Fáze 3: dohra

V této fázi se již opustil iterační cyklus, protože vývoj dalších funkcí a vlastností již není potřeba, a pokračuje se lineárně. Výsledný produkt prochází integrací a testováním, jakým způsobem se bude testovat metodika nepopisuje, záleží to na vývojovém týmu. Jako poslední bod se vytváří dokumentace, například uživatelské manuály, apod.

4.2.3 Řešení podle TDD

Jak bylo popsáno v kapitole 3.6, Test-Driven Development je metodika, která zasahuje do poměrně malé části životního cyklu software. Jde spíše o soubor pravidel a postupů, kterých by se programátor měl při vývoji držet. Pokud je v metodice řečeno „naprogramujte kus kódu“, neříká se tam, jakým způsobem se má daný kód programovat. To je záležitost konkrétního programátora, jeho znalostí, zkušeností a zvyklostí.

Pokud většina agilních metodik klade mimo jiné důraz na kvalitní a srozumitelný zdrojový kód, pak TDD se nezabývá ničím jiným než zdrojovým kódem. Cílem TDD je přimět programátora produkovat funkční, bezchybný, čistý, kvalitní a kdykoliv znova testovatelný zdrojový kód. Proto se někdy také o TDD mluví ne jako o metodice, ale jako o programovací technice.

TDD stejně jako ostatní agilní metodiky těží z iterativního přístupu. Jednotlivé kroky, které se v každé iteraci provádějí, jsem popsal v kapitole 3.6.2. Nyní je aplikuji na náš konkrétní projekt. Pro příklad vyberu část systému, na které budu demonstrovat použití TDD. Může to pro snazší srovnání být opět část aplikace zabývající se nastavením času a jeho alokací na projekt.

V následujícím tedy budu postupně procházet jednotlivé kroky iteračního procesu, u každého z nich uvedu řešení nebo výsledek, ke kterému vývojář dospěl.

1. iterace

1. *Přemýšlejte o tom, co chcete udělat.*

Chci implementovat nastavení času uživatelem. Začnu tedy tím, že vypíšu do formuláře nastavení času na aktuální týden.

2. *Přemýšlejte o tom, jak to otestovat.*

Vyberu si určitého uživatele a určitý den a otestuji, jestli nastavený čas, který z databáze čte metoda třídy „Čas“, vrací správnou hodnotu.

3. *Napište malý test. Přemýšlejte o požadovaném API.*

Napišu test, který z databáze načte hodnotu nastaveného času na 10. března 2005 pro člověka s ID=2. Tato hodnota je 4, proto porovnám načtenou hodnotu s číslem 4 a podle toho vrátím výsledek testu.

4. *Napište jen tolik kódu, aby test selhal.*

K implementaci této vlastnosti je potřeba třída „Čas“, proto implementuji nejnutnější základ této třídy. Nebude obsahovat žádné parametry ani metody, bude jen existovat, ale nebude nic provádět.

5. *Spusťte test a dívejte se, jak selže.*

Test samozřejmě selže, protože třída „Čas“ neobsahuje žádné metody, které by načítaly čas z databáze.

6. *Napište jen tolik kódu, aby test prošel.*

Implementuji ve třídě „Čas“ metodu „getCas“, která bude mít 2 parametry, ID člověka a datum, a bude vracet hodnotu nastaveného času, kterou přečte z databáze.

7. *Spusťte všechny dosavadní testy a dívejte se, jestli projdou. Pokud aspoň jeden selže, udělali jste něco špatně, proto to opravte okamžitě, protože chyba bude v něčem, co jste právě napsali.*

Test selhal, je tedy někde chyba. Tu ale musím hledat výhradně v metodě „getCas“, protože to je jediný kus kódu, který jsem napsal. Opravím chybu (překlep ve volání funkce) a spustím test znovu. Nyní už uspěje.

8. *Pokud se v kódu vyskytnou duplicity nebo nevýznamný kód, refaktorizujte, abyste odstranili duplicity a zefektivnili kód; to zahrnuje redukci propojení a zvýšení soudržnosti.*

Zefektivním kód tím, že parametry funkce pro práci s databází předám odkazem, aby se ušetřilo místo v paměti.

9. *Spusťte testy znovu, měly by stále všechny projít. Pokud ne, udělali jste chybu při refaktorizaci, opravte ji a znovu spusťte testy.*

Všechny testy tentokrát proběhly v pořádku, při refaktorizaci nevznikla žádná chyba. Vlastnost načtení času z databáze je úspěšně implementována.

Nyní tedy máme třídu „Čas“, která umí z databáze načíst údaj o nastavení času. Protože chceme čas i ukládat, musíme třídu rozšířit, čímž se dostáváme k další iteraci. Projdeme opět celou posloupnost bodů, stejně jako u implementace první vlastnosti. Nyní už nebudu znovu vypisovat pokyny, jsou pro každou iteraci stejné, u každého bodu jen uvedu výsledek činnosti.

2. iterace

1. Chci rozšířit třídu „Čas“ tak, aby uměla nastavená data z formuláře správně uložit do databáze.
2. Testovat budu tak, že nejdříve do databáze vložím předem určenou hodnotu, kterou pak zpětně načtu a ověřím výsledek.
3. Test bude spočívat v tom, že uložím uživateli s ID=2 na 9. 3. 2005 do databáze hodnotu nastaveného času 5, následně použiji metodu „getCas“, a srovnám její výsledek s číslem 5.
4. Napíšu v třídě „Čas“ hlavičku metody „setCas“.
5. Test selže, protože metoda „setCas“ zatím nic nedělá, takže nemůže vracet požadované výsledky.
6. Implementuji metodu „setCas“. Bude mít 3 parametry, ID člověka, datum a hodnotu času, který chci do databáze uložit. Návrátová hodnota bude typu boolean, tedy „true“ nebo „false“ podle toho, co vrátí databázový server při pokusu o vložení hodnoty.
7. Test uspěl. Metodu „setCas“ jsem implementoval správně.
8. Při pokusu o refaktORIZACI jsem nenašel žádné duplicity, kód tedy ponechávám beze změny.
9. Spouštět testy znovu není nutné, protože jsem od jejich posledního úspěšného dokončení zdrojový kód nijak neupravoval.

Teď již máme dokončenu třídu „Čas“, svými metodami nám poskytuje přesně to, co pro práci s nastaveným časem potřebujeme. Nic méně, ale také nic zbytečného navíc. To je jedna z důležitých zásad TDD. Kód, který není nutný pro naplnění nějaké funkcionality, nemá v aplikaci co dělat.

Umíme pracovat s nastavením času, začneme tedy implementovat jeho alokaci na projekt. Opět projdeme celým iteračním cyklem.

3. iterace

1. Jako první se při alokaci času vypisuje seznam dostupných projektů, na které si čas můžu alokovat. Cílem je tedy zjistit a vypsát tento seznam.
2. Test bude probíhat tak, že si u předem vybraného uživatele zjistím seznam jemu dostupných projektů. Pak se budu snažit tento seznam získat metodou „getDostupneProjekty“ třídy „alokace-projekt“ a porovnáím výsledek se známou hodnotou.
3. Napíšu test, který použije metodu „getDostupneProjekty“ pro uživatele s ID=3 a vrátí seznam ID dostupných projektů. Tento seznam (pole) pak porovnáím s polem [7, 11, 23, 28], což je skutečný seznam dostupných projektů pro uživatele s ID=3.
4. Implementuji základ třídy „alokace-projekt“ a hlavičku metody „getDostupneProjekty“.
5. Zkontroluji, že test skutečně selže, protože zatím nenapsaná funkce nemůže vrátit správný výsledek.
6. Implementuji metodu „getDostupneProjekty“. Bude mít jen 1 parametr – ID uživatele a bude vracet pole hodnot představujících ID projektů, které má tento uživatel k dispozici pro alokaci času.
7. Spustím test a vidím, že selhal. Funkce dokáže vrátit seznam projektů, ale není to přesně ten seznam, který by správně měl být vrácen. Zapomněl jsem brát v úvahu již skončené projekty, které nesmím zařadit do výsledku. Musím tedy počítat i s aktuálním datem a vracet jen dostupné projekty, které jsou „v běhu“, tzn. aktuální datum je větší než datum zahájení projektu a zároveň menší než datum jeho ukončení. Opravím tedy metodu a spustím test znovu. Test uspěje.
8. Metoda sice pracuje správně, ale ne příliš efektivně. Nemusím z databáze vybírat všechny dostupné projekty a pak kontrolovat, jestli jde o projekty aktivní, nebo už skončené. Díky schopnosti databáze pracovat s aktuálním datem a časem můžu tyto omezující podmínky zahrnout přímo do SQL dotazu a nechat databázový stroj, aby vrátil jen dostupné aktivní projekty.
9. Spustím testy a vidím, že metoda „getDostupneProjekty“ vrátila chybu, která pochází z volání databázového dotazu. Způsobila to chybná SQL syntaxe. Opravím databázový dotaz a spustím testy znovu. Tentokrát všechny testy uspějí. Úspěšně jsem implementoval další vlastnost. Nyní by následovala implementace načtení aktuálních hodnot alokací daného uživatele na vybraný projekt, následně uložení vyplněných hodnot z formuláře do databáze.

Takto, po velmi malých krůčcích se postupuje dál a dál. Vlastnost po vlastnosti až do doby, kdy budeme chtít nastartovat novou iteraci, ale už nebudeme schopni vymyslet další test, který by selhal v důsledku chybějící funkcionality. V tomto stavu je implementace aplikace dokončena a pokračuje se dalšími fázemi vývojového cyklu software, kterými jsou testování jednotek, integrace, psaní dokumentace, akceptační testy, předání hotového produktu zákazníkovi, údržba. Popis těchto fází už není součástí metodiky TDD. V ní se jenom uvádí, že po skončení iterační fáze se postupuje „klasicky sekvenčně“. Jaké konkrétní metody vývojový tým zvolí pro další vývoj záleží na něm, zde už se nemůže spoléhat na Test-Driven Development, ale jen na své znalosti a zkušenosti.

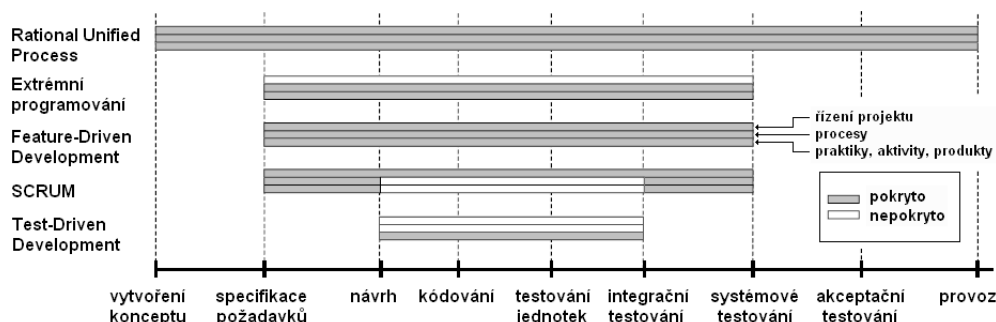
4.2.4 Srovnání jednotlivých řešení

Díky řešení ilustračního projektu popsanému v předcházejících kapitolách je vidět, že všechny popisované metodiky lze úspěšně využít k vývoji softwaru. Jak je to ale s vývojem softwaru z pohledu kompletního procesu, to už je věc jiná. Každá metodika se totiž zaměřuje na jinou část životního cyklu a dívá se na něj z jiného pohledu. FDD implementuje po vlastnostech a považuje za stěžejní seznam vlastností systému, který se snaží v produktu obsáhnout, Scrum je více orientována na management a řízení projektu, zatímco TDD je vyloženě programovací technika, která se zabývá čistě jen zdrojovým kódem.

Jednotliví autoři většinou vyvíjeli své metodiky s cílem zlepšit některou z fází životního cyklu vývoje softwaru ve firmě, kde působili jako vývojáři nebo analytici. Daná firma nemusela mít nutně problém se všemi aspekty vývoje, proto byly metodiky vyvíjeny cíleně a specializovaly se jen na určitou část vývojového procesu. Jejich vzájemné srovnávání je z tohoto důvodu obtížné.

Přesto se pokusím nabídnout srovnání z pohledu řešení jednotlivých fází projektu. Na obrázku 4.4 (který v originálním znění pochází z [1]) je zakresleno, kterým částem životního cyklu projektu se daná metodika věnuje. Kromě tří metodik, kterými se zabýváme podrobně, jsou pro srovnání uvedeny i dvě další – Rational Unified Process jako zástupce tradičních metodik a Extrémní programování kvůli přirovnání k jedné z nejrizičnějších agilních metodik. Z obrázku je vidět, že RUP je opravdu komplexní metodika, která pokrývá celý projekt nejen z hlediska času, ale také z hlediska aktivit prováděných během vývoje. Pojdme si nyní rozebrat stěžejní body našeho ilustračního projektu a srovnat přístup tří popisovaných metodik.

Všechny tři metodiky předpokládají, že úvodní koncepce projektu je již stanovena, takže se touto fází vůbec nezaobírají. FDD a Scrum se o projekt začínají zajímat ve fázi specifikace požadavků a návrhu systému. V případě FDD jde o úvodní schůzku se zadavatelem, společné vypracování celkového modelu a sestavení počátečního seznamu vlastností. Scrum pro tuto fázi předepisuje vytvořit Product



Obrázek 4.4: Pokrytí fází projektu jednotlivými metodikami

Backlog a model systému na vysoké úrovni abstrakce. TDD se tímto krokem explicitně nezabývá, ale někteří odborníci tvrdí, že fáze návrhu je skryta v přemýšlení nad novými testy. Následující fázi kódování se věnují všechny tři metodiky, každá ale z trochu jiného pohledu. TDD přímo předepisuje, jakými postupy se má programovat, celá metodika je vlastně na těchto postupech založena. FDD má sestaven plán implementace a díky Sprint Backlogu vymezuje skupinu funkcionalit, které se budou implementovat. Jakými konkrétními prostředky a programovacími technikami bude implementace probíhat už neříká, resp. nenařizuje, nechává volnou ruku vývojářům. Scrum má v této části jen organizační a manažerskou kontrolu nad projektem, vlastní programování si řídí jednotlivé týmy samy. Po kódování (nebo často současně s ním) probíhá testování jednotek a integrační testování. V případě TDD je to jednoduché, testování jednotek probíhá průběžně, střídavě s psaním produkčního kódu. Vývoj probíhá po velmi malých kouscích, takže není potřeba integrovat nějaké větší moduly nebo komponenty. Přidávané funkce jsou integrovány průběžně. Při postupu podle Scrum je testování jednotek skryto v aktivitách prováděných v rámci jednoho sprintu. Vývojáři, kteří píší kód, si ho taky sami testují. Zpětnou vazbu získávají na každodenních schůzkách a také na hodnotící schůzce sprintu. Testování jednotek je ve FDD součástí poslední fáze – Implementace podle vlastností. Opět si sami vývojáři testují svůj vlastní kód. Následnými fázemi – akceptačními testy a provozem systému – se ani jedna z metodik nezabývá podrobně. Většinou jen říká, ve které fázi a po provedení jakých aktivit je systém připraven k provozu, ale už dále nespecifikuje jednotlivé kroky, které jsou nutné ke skutečnému spuštění v ostrém provozu.

Z obrázku i z komentáře je vidět, že nejkomplexnější z těchto tří metodik je FDD. Věnuje se všem vývojovým aktivitám, od řízení projektu až po samotné programování. Naopak u TDD je jasné, že musí být použita v kombinaci s nějakou jinou metodikou. Je to zřejmé z toho, jak je TDD cílena na zdrojový kód a jeho testování. Napadá mě zajímavé spojení s metodikou Scrum, protože jejich aktivity jsou takřka komplementární, takže by se mohly úspěšně doplňovat. Metodika Scrum by obstarávala řízení projektu a komunikaci v týmu, TDD zase produkování kvalitních zdrojových kódů.

Kapitola 5

Závěr

Hlavním cílem práce bylo získání přehledu v oblasti agilních metodik vývoje software a popsání tří vybraných zástupců. To bylo splněno v kapitole 3, kde jsem popsal jak agilní metodiky obecně, tak také podrobně rozpracoval postupy a praktiky metodik Feature-Driven Development, SCRUM Development Process a Test-Driven Development. Navíc byly ještě v kapitole 2 představeni tři důležité zástupci tradičních metodik, kvůli srovnání a pochopení historických souvislostí. Neméně důležitým cílem práce byly i případové studie zvolených tří metodik. Tyto studie jsou součástí kapitoly 4, kde jsem nejdříve nadefinoval projekt a poté jsem ukázal jeho řešení pomocí jednotlivých metodik. Nakonec jsem uvedl srovnání jejich přístupu k vývoji projektu.

Vzhledem k tomu, že téma agilních metodik je celkem nové, potýkal jsem se v některých částech práce s nedostatkem dostupné literatury. Většina pramenů je v současnosti jen v podobě článků a diskuzí na Internetu. Knižní publikace byly o jednotlivých metodikách vydány většinou jejich tvůrci, ale systematických ucelených rozborů a objektivních srovnání je zatím jen pár. Při studiu pramenů k této problematice jsem nenarazil na situaci, kdy by autor chtěl postupy své metodiky před světem tajit. Drtivá většina autorů přistupuje ke svým metodikám podobným stylem jako k open-source produktům, to znamená, že maximum svých poznatků publikuje na Internetu v podobě článků, blogů nebo encyklopedií na principu wiki. Dává tak prostor k veřejné diskuzi nad úpravami a směřováním vývoje metodiky. Ctí tedy jedno ze základních pravidel agilního vývoje samotného, tedy těsnou a častou komunikaci s uživateli. Uživateli jsou v tomto případě samotní vývojáři, kteří budou danou metodiku používat.

Podle mého názoru je perspektiva agilních metodik slibná. Přestože některé z nich existují jen krátce, tak stihly dokázat, že mohou být v praxi velice úspěšné. V minulosti byl vývoj software záležitostí pár schopných nadšenců a postupem času se dostal k širší veřejnosti, takže dnes se například do programování může pustit téměř každý technicky zdatný člověk. Pokud se agilní metodiky razantně prosadí,

tak budu se zájmem sledovat, jestli se tato činnost nevrátí zpět do rukou hrstky zasvěcených, kteří budou schopni vyvíjet kupříkladu ještě extrémněji než Extrémní programování. Čas ukáže, kam dovedou agilní metodiky požadavky klientů na rychlost a kvalitu vývoje.

V oblasti agilního vývoje je spousta možností dalšího výzkumu. Jednak jsou to snahy o spojování vlastností jednotlivých metodik ve snaze eliminovat nedostatky jedné metodiky a importovat výhody druhé. Jako příklad můžeme uvést XP@Scrum popsanou v kapitole 3.5.6. Dále je možné zkoumat problémy škálovatelnosti software, které agilní přístupy většinou neřeší. Nakonec by možná bylo dobré přizpůsobovat dobré vlastnosti agilního vývoje pro větší projekty a týmy, aby z jeho výhod mohly těžit i velké firmy produkující sériově vyráběný software. K tomuto kroku se ale musí přistupovat uvážlivě, aby se nakonec agilní metodiky nezačaly potýkat s problémy, které měly metodiky tradiční a nemusely tak být brzy nahrazeny opět zcela novým přístupem.

Literatura

- [1] Abrahamsson, Pekka, Salo, Outi, Ronkainen, Jussi & Warsta, Juhani: Agile software development methods: Review and analysis, VTT Publications 2002
- [2] Ambler, Scott W.: Agile Database Techniques, Wiley 2003
- [3] Astels, David: Test Driven Development: A Practical Guide, Prentice Hall 2003
- [4] Beck, Kent: Test Driven Development, Addison-Wesley 2002
- [5] Boehm, Barry: Get Ready for Agile Methods, with Care, časopis Computer, číslo 35, leden 2002
- [6] Boehm, Barry: Spiral development: Experience, principles and refinements, Carnegie Mellon University, Software Engineering Institute 2000
- [7] Buchalceková, Alena: Agilní metodiky, příspěvek v konferenci Objekty 2002
- [8] Coad, Peter: Java Modeling in Color with UML, Prentice Hall 1999
- [9] Doshi, Gunjan: Test-driven Development Quick Reference Guide
- [10] Doshi, Gunjan: Test-driven Development Rhythm
- [11] Highsmith, Jim: Agile Software Development Ecosystems, Addison-Wesley 2002
- [12] Král, Jaroslav: Informační systémy, SCIENCE 1998
- [13] Kruchten, Phillipe: The Rational Unified Process: An Introduction, Addison-Wesley 2000
- [14] Meade, Erik: Design Principles in Test First Programming, článek publikovaný na serveru objectmentor.com, 2000, <http://www.objectmentor.com/resources/articles/DesignPrin.pdf>
- [15] Palmer, Stephen R., Felsing, John M.: A Practical Guide to Feature-Driven Development (The Coad Series), Prentice Hall 2002

- [16] Richta K., Sochor J.: Softwarové inženýrství I., skriptá ČVUT 1996
- [17] Schwaber, Ken, Beedle, Mike: Agile Software Development with SCRUM, Prentice Hall 2002
- [18] Wideman, R. Max: The Role of the Project Life Cycle (Life Span) in Project Management, <http://www.maxwideman.com/papers/plc-models/intro.htm>
- [19] Aliance pro agilní vývoje software, The Agile Alliance, <http://www.agilealliance.org>
- [20] Manifest agilního programování, Manifesto for Agile Software Development, <http://www.agilemanifesto.org>
- [21] The Scrum Development Process, popis metodiky Scrum na webových stránkách společnosti Mountain Goat Software, <http://www.mountaingoatsoftware.com/scrum>
- [22] Schwaber, Ken: články na webových stránkách společnosti Advanced Development Methods, Inc., <http://www.controlchaos.com>
- [23] Popis a zkušenosti s metodikou TDD na webových stránkách společnosti Adaption Software, Inc., <http://www.adaptionsoft.com>
- [24] De Luca, Jeff: články o FDD na webu společnosti Nebulon Pty. Ltd., <http://www.nebulon.com>
- [25] Články o FDD na webu <http://www.featuredrivendevelopment.com>
- [26] Popis vodopádového modelu v Portland Pattern Repository's Wiki, <http://c2.com/cgi/wiki?WaterFall>
- [27] Popis spirálového modelu v Portland Pattern Repository's Wiki, <http://c2.com/cgi/wiki?SpiralModel>
- [28] Popis spirálového modelu ve volně šiřitelné encyklopedii Wikipedia, http://en.wikipedia.org/wiki/Spiral_model
- [29] Webové stránky společnosti IBM zabývající se vývojem software pro metodu RUP, <http://www.ibm.com/rational>

Příloha A

Prvotní specifikace

A.1 Systém správy externích lidských zdrojů (SEZ)

Celý systém bude zpracován dostupnými webovými technologiemi. Zajistí se tak použitelnost z jakéhokoli místa, kde je k dispozici připojení k Internetu a webový prohlížeč. Data budou uchovávána v databázi na straně serveru, aby byla zajištěna maximální aktuálnost dat a efektivita sdílení informací.

Základní objekty

- **Lidé**

Sem patří všichni lidé, kteří mají nějaký vztah k systému, ať už aktivní (vkládání, editace, interakce se systémem) nebo pasivní (pouze evidování pro potřeby systému, nemají žádná práva k operacím se systémem). U všech lidí se ukládají tyto informace: ID, jméno, příjmení, typ, pozice, telefon, e-mail. U aktivních uživatelů navíc login a heslo. „Typ“ je stupeň oprávnění uživatele k funkcím systému, každý typ má všechna práva typů předchozích a navíc nějaká další.

Existují 4 typy (úrovně) uživatelů:

- **none**: žádná práva, pouze evidován pro potřeby statistik a souhrnů
- **read-only**: má právo nahlížet na statistiky, přehledy, ale nemůže vkládat a editovat údaje
- **full-access**: má právo vkládat a editovat údaje, které se týkají jeho samotného nebo entit, na nichž má podíl nebo za ně nese zodpovědnost
- **board**: administrátor, má přístup ke všem funkcím a spravuje systém

- **Externí zdroje (EZ)**

Lidé, o nichž se sledují údaje o práci na projektech. Entity „Lidé“ a „Externí zdroje“ jsou zde uváděny zvlášť, protože vztah mezi nimi je 1:n, tzn. každý „Externí zdroj“ patří mezi „Lidi“, ale ne všichni „Lidé“ jsou také „Externí zdroje“.

U každého EZ se eviduje: ID EZ, ID člověka, základní hodinová sazba, minimum, optimum, maximum odpracovaných hodin za týden. MIN, OPT a MAX bude v profilu EZ výchozí hodnota, bude se moct upravovat¹ individuálně podle situace pro jednotlivé týdny.

EZ bude mít ke zpřesnění dostupného času „rozvrh“ dnů v týdnu, kde bude moci uvést počty hodin. Takovýto profil by mělo být možné snadno aplikovat na další týdny bez nutnosti psát každý týden. Hodnoty „OPT, MIN, MAX“ budou udávány pro časový úsek „týden“, hodnota „OPT/týden“ může být součtem časů za jednotlivé dny (ty si nastaví EZ v denním odhadu času).

- **Projekty**

Eviduje se u nich:

- ID, kód, název, project manager (PM), datum zadání, datum ukončení (deadline)
- předpokládaná hodinová zátěž (stanovuje EZ)
- milníky (stěžejní kontrolní body v projektu)
- PT = projektový tým (kdo vše patří do týmu: ID člověka, jméno, pozice, telefon, e-mail).
- login a heslo

Operace s projekty:

- zakládání a definování všech položek (členů týmu)
- editování stávajících projektů
- editovat projekt může jen příslušný PM

Na operace s projektem má právo pouze jeho PM, nebo uživatel typu „board“.

- **Týmy**

Spojovací entita mezi „Lidmi“ a „Projekty“. Každý projekt má kolem sebe tým lidí účastnících se prací na projektu. Jeden člověk může být ve více týmech.

Ukládá se: ID projektu, ID člověka, pozice.

Pozice se následně zobrazuje u člověka v PT (projektovém týmu), momentálně existují následující pozice:

- PM – project manager
- AC – account manager

¹úpravu MIN/OPT/MAX času by mělo být možné provádět kdykoli. Když dojde k úpravě času v dnech, kdy už je přidělen projekt, pošle se automaticky zpráva odpovídajícímu projektovému manažerovi (PM)

- EZ – externí zdroj
- OS – ostatní

- **Práce**

Zaznamenává skutečně odpracovanou dobu jednotlivých EZ na konkrétních projektech.

Eviduje tyto údaje: ID EZ, ID projektu, datum, odpracované hodiny, poznámka.

Požadavky na systém

- každý, kdo na to má potřebné oprávnění, může založit nový projekt, stanovit jeho mezní termíny a přiřadit mu tým EZ, editovat atributy projektu smí pouze jeho PM (nebo kterýkoliv „board“)
- PM nebo EZ může u konkrétního projektu v rámci stanovených termínů zadat délku realizace určitých prací EZ na projektu (EZ nejdříve stanoví dle reálného odhadu, PM buď ponechá, nebo upraví po dohodě s EZ)
- PM může měnit nastavení minima, optima a maxima týdenní hodinové kapacity u EZ, který je v „jeho“ projektu, primárně však stanovuje svoje kapacity každý EZ, při každé změně MIN/OPT/MAX musí být rozeslán automatický e-mail jednotlivým PM, dle toho které projekty to ovlivní (struktura zprávy: Došlo k úpravě časové alokace osoby XY v projektu XY. Prověřte si projektový plán a termíny dokončení)
- hlášení kolizí termínů projektů nebo kapacit EZ při přerozdělování a plánování (v jednodušších případech automatická změna), hlášení o kolizi by mělo obsahovat všechna důležitá data pro rozhodování jako např.: jaké jiné projekty ovlivňuje, kdo je PM (dostupnost na kontakty), termíny, PT (kdo je v projektovém týmu)
- u každého EZ se sleduje jakýsi jeho pracovní kalendář: tzn. na čem právě pracuje, jaká je jeho aktuální vytíženost, jaké projekty jsou pro něho naplánovány do budoucna a na jaký termín

Admin sekce „board“

Sekce bude přístupná jen správci systému typu „board“. V sekci admin bude možné zakládat a nahlížet na následující položky (možnost editace všech jejích parametrů):

1. Lidé
2. EZ (stanovením této vlastnosti u „lidé“)
3. Projekty

4. Definovat týmy
5. Sledovat časy provedených prací nad všemi projekty

Admin sekce „PM“

Sekce přístupná projekt manažerům pro účely:

1. zakládání projektů (PM může vkládat všechny parametry až na EZ, které přidělí „board“)
2. sledování všech projektů z pohledu vytíženosti atd.
3. editování vlastních projektů

Filosofie zadávání času a alokace na projekty

Každý EZ si nastavuje svůj osobní časový profil. Ten se skládá z hodnoty MIN, OPT a MAX pro každý den v týdnu. Slouží EZ k tomu, aby nemusel vyplňovat tyto hodnoty pro každý týden opakovaně, pokud nastavuje svou časovou dostupnost. Nastavení času už probíhá pro konkrétní datum, typicky několik týdnů dopředu, pro každý den je možno nastavit výše uvedené 3 hodnoty. Ty postupně označují, kolik hodin EZ minimálně věnovat odpracuje, kolik hodin by chtěl optimálně práci věnovat a jaké je jeho časové maximum, přes které nemůže jít. Tyto hodnoty pak budou sloužit PM, pro časové plánování projektů, sledování vytíženosti jednotlivých EZ, atd. Alokace času se vztahuje nejen ke konkrétnímu datu, ale i ke konkrétnímu projektu. EZ stanoví počet hodin, které bude danému projektu věnovat. Může si tak rozvrhnout svůj čas na jednotlivé projekty. PM díky tomu dostáváji poslední informace pro stanovení vytíženosti EZ a projektů jako podklad k dalšímu rozhodování.

Typické otázky kladené na systém (výstupy a sestavy)

- přehled všech projektů, plánovaných, probíhajících i dokončených
- přehled EZ + přiřazení k projektům, řazení podle abecedy, týmů, vytíženosti
- informace o projektu: kdo je jeho PM, které EZ jsou v týmu pro tento projekt, termíny projektu, případně kolize s ostatními projekty
- informace o EZ: jakých projektů se v daném období účastní, kolik jeho času je pro jednotlivé projekty alokováno, kolik zbývá pro případné další projekty, je EZ zároveň PM nějakého jiného projektu, ...
- dělá některý EZ na více projektech a je tedy jeho zátěž nad rámec jeho možností, nebo naopak nemá co na práci, tzn. vytíženost jednotlivých EZ za určité období: týden, měsíc, kvartál, rok

Příloha B

Změny ve specifikaci systému

B.1 Dodatek k prvotní specifikaci

Úprava práv

V SEZu budou tyto skupiny práv:

- BO – board
- PM – project manager
- EZ – externí zdroj
- OS – ostatní (můžou být dočasně lidé zapojení do týmu, ale nemá cenu je dlouhodobě označovat jako EZ)

Co komu se bude v menu zobrazovat je uvedeno níže a ke každému linku v menu je výčet zkratk (BO, PM, EZ, OS) podle toho kdo k tomu může mít přístup + případné upřesnění.

Dále bych viděl vhodné zvolit kromě těchto pevně nastavených principů přístupu možnost nastavit položky do menu ještě individuálně. Tzn. že v sekci admin z pozice BO by byl celkový výčet lidí v SEZu a na stránce práv by byl celkový výčet možností v menu ke každé osobě. Pokud bude PM, tak budou zaškrtnuty checkboxy defaultně tak jak nastaveno v hard profilu. Mohu mu však zaškrtnutím dalšího checkboxu přidat další položku menu a to nad rámec nastavených práv PM.

Nová entita – Úkol

Úkol může zadat jakýkoliv PM nebo BO jakémukoliv EZ nebo OS. Jde vlastně o nárazové práce, které se vztahují k určitému projektu, ale vykonává je kupříkladu člověk, který není v daném projektovém týmu.

Ukládá se zde: ID zadavatele, ID řešitele, ID projektu, ke kterému se úkol vztahuje, popis úkolu, deadline, stav úkolu (odeslán / potvrzen / dokončen).

Struktura hlavního menu

Navrhuji přeskupit menu dle principu níže a jinak rozvrstvit princip zobrazování komponent jednotlivým členům.

1. OSOBNÍ NASTAVENÍ

- 1.1. osobní údaje (BO, PM, EZ, OS)

2. ALOKACE ČASU, POŽADAVKY, NASTAVENÍ

- 2.1. čas individuálně (EZ, OS)
- 2.2. alokace času (EZ, OS)
- 2.3. časový profil (EZ, OS)

3. PŘEHLEDY

- 3.1. vystavení požadavku/úkolů (BO, PM, EZ, OS)
- 3.2. vytíženost EZ (BO)
- 3.3. vytíženost projektu (BO, PM) (BO vidí vše, PM jen svoje projekty)
- 3.4. zaměstnanost (BO, PM, EZ, OS)
- 3.5. alokace (BO, PM, EZ, OS) (EZ + OS má jen svoji alokaci, BO+EZ má možnost náhledu alokace na všechny EZ+OS a to formou seznamu kde osoba bude linkem na celkový přehled, avšak logicky bez možnosti editace)
- 3.6. úkoly (BO, PM, EZ, OS) (EZ + OS vidí jen svoje úkoly, PM vidí všechny úkoly které zadal+možnost editace, BO vidí všechny úkoly s informací kdo je zadal a komu+možnost editace)
- 3.7. projekty (BO, PM, EZ, OS) (EZ + OS vidí jen projekty, do kterých je zapojen, EZ vidí všechny své projekty, BO vidí všechny projekty a má možnost editace)
- 3.8. týmy (BO, PM) (PM vidí jen své týmy a nemá možnost editace, má však možnost poslat požadavek na BO, aby mu přidělil buď nového EZ nebo vyjmul z týmu, BO vidí všechny týmy a má možnost editace)
- 3.9. lidé (BO, PM, EZ, OS) (příčemž PM, EZ, OS mají jen možnost náhledu položek bez možnosti změny, změnu může dělat jen BO)

4. NASTAVENÍ SEZU

- 4.1. atributy SEZu (BO)
- 4.2. editace práv uživatelů (přidání či ubrání položek jednotlivým členům)
- 4.3. editace práv pro skupiny (BO, EZ, EZ, OS), zde by bylo možné pomoci checkboxu zaškrtnat a upravit profil zobrazování práv pro dané skupiny (BO EZ ... atd.)

Musí platit následující logika

- A. po zadání úkolu se pošle EZ e-mail, případně SMS se zprávou, že je po něm vyžadován nový úkol, aby potvrdil jeho termín v SEZu (z toho plyne aby po přihlášení měli EZ ihned přehled úkolů a projektů, úkoly by bylo možné potvrdit, označit jako hotové apod.)
- B. PM po potvrzení úkolu od EZ v SEZu obdrží automaticky e-mail, že úkol s tímto názvem pro daného EZ byl akceptován (stejně tak se v SEZU v přehledu zadaných úkolů u každého PM zobrazí přehled úkolů se stavem (zadáno, potvrzeno, dokončeno))
- C. Board by měl mít přehled o všech úkolech, kteří PM je zadali jakým EZ, jejich stav apod. (tříděno dle atributu: termín, stav (zadáno, potvrzeno, dokončeno), EZ, OS, PM)

Informace na homepage systému, hned po přihlášení

Stránku „home“ ihned po přihlášení bych navrhoval udělat s těmi nejdůležitějšími údaji podle toho co je ke každému typu práv nejrelevantnější.

Home po přihlášení pro BO – board

- Požadavky od PM (úpravy týmu, přidání / odebrání lidí), přehled s informací od jakého PM k jakému projektu (prokliky do karet o projektu, atd.)
- Požadavky od EZ (detail na specifikaci)
- Aktuální stav:
 - % vytíženosti celého týmu: XX (jedno číslo, jinak celkový přehled dále v menu)
 - Celkový počet EZ: XX (+ link na přehled EZ)
 - Počet EZ bez nastavení alokace času: XX (link na seznam těchto lidí)
 - Počet EZ bez nastavení profilu času: XX (link na seznam těchto lidí)
 - Počet EZ bez přidělení k projektu/týmu: XX (link na seznam těchto lidí)

Home po přihlášení pro PM – project manager

- Úkoly zadané EZ (stav, komu, link na detail a editaci)
- Úkoly/požadavky od EZ (detail na specifikaci)
- Úkoly zadané BO (stav, link na detail)
- Přehled rozpracovaných projektů (vyčet jako v sekci projekty, ovšem jen jeho projekty)

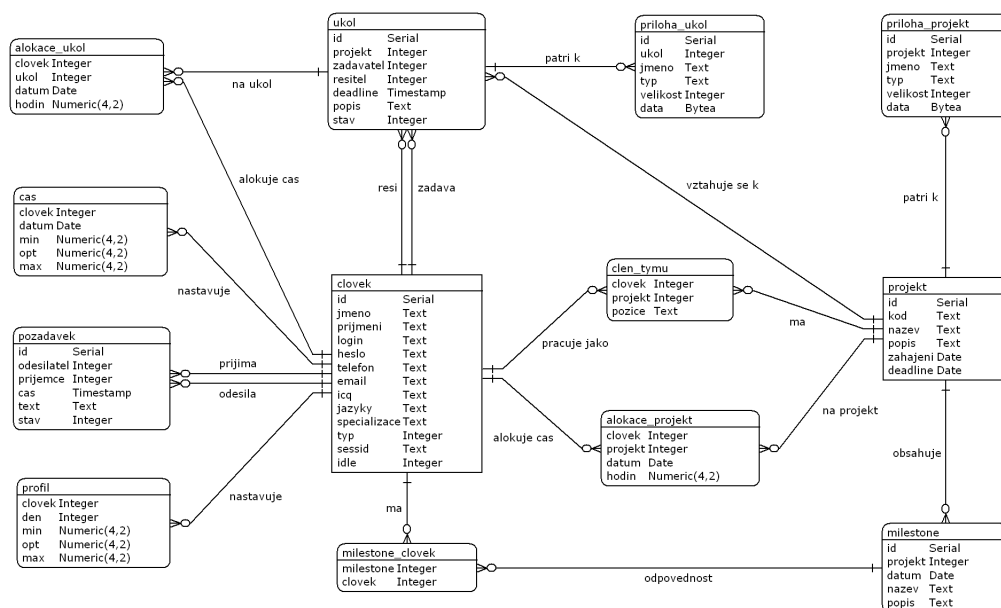
Home po přihlášení pro EZ – externí zdroj

- Úkoly přijaté od PM (stav, termín, link na detail)
- Požadavky zadané na BO (stav, link na detail)
- Přehled projektů ve kterých je zapojen (výčet jako v sekci projekty, pozn.: jen jeho projekty)

Možná by stálo za to implementovat princip evidence a zobrazování úkolů/požadavků do jedné tabulky s nějakým přehledným členěním, než to mít v jednotlivých odstavcích (například rozdělit ikonami).

B.2 ERD diagram

Pro lepší orientaci v datových závislostech mezi jednotlivými objekty byl zpracován ERD diagram (viz obrázek B.1).

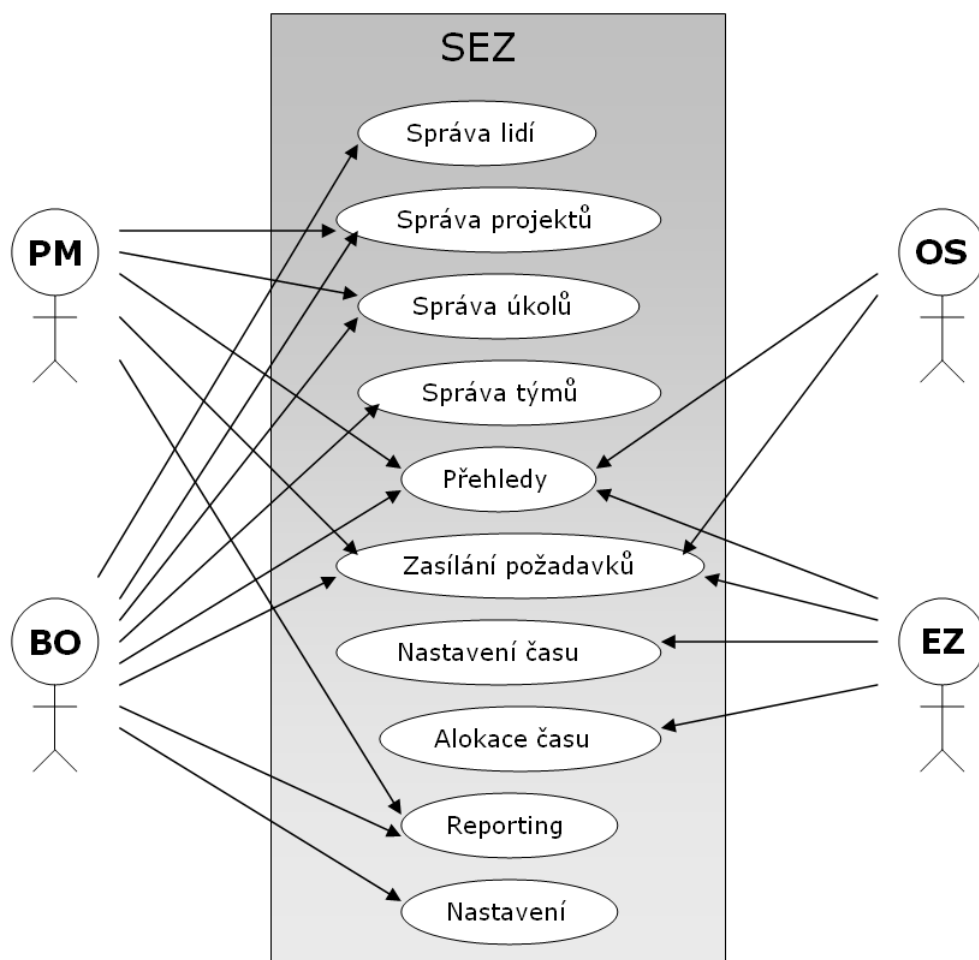


Obrázek B.1: ERD diagram

B.3 Use-case diagram

Na základě požadavků vyplývajících z prvotní specifikace a jejích úprav byly objeveny případy užití, které jsou znázorněny v diagramu B.2. Uživatelské role korespondují s typy uživatelů, které systém rozlišuje. Jednotlivé případy užití většinou

reflektují funkce, které jsou systémem nabízeny, některé však zahrnují více funkcionalit týkajících se stejné aktivity. Například nastavení časového profilu se řeší v rámci případu užití „Nastavení času“, obdobně je správa milníků součástí případu užití „Správa projektů“.



Obrázek B.2: Use-case diagram