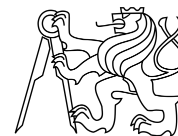
ESKÉ VYSOKÉ U ENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMA NÍCH TECHNOLOGIÍ



ZADÁNÍ BAKALÁ SKÉ PRÁCE

Název:	Návrh datového modelu aplikace pro podporu tvorby odhad pracnosti softwarových projekt
Student:	Milan Vancl
Vedoucí:	Ing. Michal Pet ík
Studijní program:	Informatika
Studijní obor:	Softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2017/18

Pokyny pro vypracování

Cílem práce je navrhnout a implementovat model aplikace pro odhady pracnosti SW projektu na základ metod a postup používaných ve firm Profinit.

1. Analyzujte používané nástroje na tvorbu odhad .
2. Seznamte se s metodikou odhad zadavatele a její implementací v MS Excel.
3. Navrhn te datový model obecného len ní odhadu.
4. Zvolte vhodné technologie a využijte je v implementaci.
5. Vytvo te dokumentaci a aplikaci otestujte.
6. Zhodno te ešení vzhledem k možnosti budoucího rozvoje.

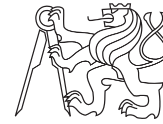
Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D. vedoucí katedry	prof. Ing. Pavel Tvrdík, CSc. d kan

V Praze dne 5. února 2017

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra softwarového inženýrství



Bakalářská práce

Návrh datového modelu aplikace pro podporu tvorby odhadů pracnosti softwarových projektů

Milan Vancl

Vedoucí práce: Ing. Michal Petřík

12. května 2017

Poděkování

Chtěl bych poděkovat Ing. Michalu Petříkovi za odborné vedení, pomoc a rady při zpracování této práce. Dále bych chtěl poděkovat firmě Profinit EU, s.r.o. za poskytnutí prostředků, které mi při zpracování práce pomohly. V neposlední řadě děkuji také své rodině za jejich podporu při mém studiu.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 12. května 2017 . . . .

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2017 Milan Vancl. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí, je nezbytný souhlas autora.

Odkaz na tuto práci

Vancl, Milan. *Návrh datového modelu aplikace pro podporu tvorby odhadů pracnosti softwarových projektů.* Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2017.

Abstrakt

Tato bakalářská práce se zabývá návrhem a implementací datového modelu systému pro podporu tvorby odhadů pracnosti. Na základě požadavků na systém a poznatků z analýzy existujícího řešení je navržen datový model, který umožňuje hierarchické členění odhadovaných položek, podporuje různé metodiky, varianty, verzování tvorby odhadu a zároveň je dále rozšířitelný. Pro navržený model je implementována příslušná vrstva webové aplikace. Pro uchování dat je navrženo a implementováno schéma databáze odpovídající modelu aplikace. V aplikaci je implementována komponenta pro import odhadů reprezentovaných tabulkami MS Excel. V práci jsou diskutovány možnosti budoucího rozšíření sytému.

Klíčová slova Datový model, odhady pracnosti, softwarový projekt, webová aplikace, tabulkový procesor, centralizace, Spring, Neo4j.

Abstract

This thesis deals with design and implementation of data model for system that supports effort estimate creation. Based on requirements for the system and knowledge from analysis of existing method is data model designed allowing hierarchical distribution of estimated items, supporting different methodics, variants, versioning of estimate creation and is further extensible. An appropriate layer of web application is implemented for the designed model. Schema of database corresponding to the model of application is designed and implemented for data persistence. A component for import of estimates represented by MS Excel spreadsheets is implemented in the application. Future options for extensions of system are discussed in this thesis.

Keywords Data model, effort estimation, software project, web application, spreadsheet, centralization, Spring, Neo4j.

Obsah

Úvod **1**

1 Cíl práce **3**
- 1.1 Rešeršní část 3
- 1.2 Praktická část 4

2 Odhad **5**
- 2.1 Co je a co není odhad? 5
- 2.2 K čemu jsou odhady dobré? 6
- 2.3 Dobrý odhad 6
- 2.4 Kužel nejistoty 8
- 2.5 Metodiky odhadování 8

3 Dosavadní stav **11**
- 3.1 Používaná metodika 11
- 3.2 Používané nástroje 12
- 3.3 Excel 12
- 3.4 Koordinace práce v týmech 12
- 3.5 Sdílení zkušeností mezi kolegy 14
- 3.6 Dobré vlastnosti dosavadního řešení 15
- 3.7 Nedostatky 15

4 Analýza požadavků **17**

5 Případy užití **19**

6 Návrh datového modelu **23**
- 6.1 Koncepty datového modelu 23
- 6.2 Datový model 33

7 Analýza technologií **35**
- 7.1 Databáze 35
- 7.2 Neo4j databáze 37
- 7.3 Rozhraní pro komunikaci s databází 42
- 7.4 Různé poznatky 45
- 7.5 Rozhraní pro import/export 46

8 Realizace Databáze **47**
- 8.1 Schéma 47
- 8.2 Instalace a správa 48
- 8.3 Inicializace 48

9 Modelová vrstva aplikace **51**
- 9.1 Rozhraní pro komunikaci s databází 51
- 9.2 Rozhraní mezi servisní a perzistenční vrstvou 54
- 9.3 Konfigurace 54
- 9.4 Zpracování šablony 54

10 Importer **55**
- 10.1 Struktura souboru 55

11 Realizace logiky servisní vrstvy **57**
- 11.1 Vytvoření odhadu 57
- 11.2 Správa verzí odhadu 57
- 11.3 Import dat 58
- 11.4 Export výsledků odhadu 58
- 11.5 Řízení přístupu k odhadu 58

12 Testování **59**
- 12.1 Rozhraní pro komunikaci s databází 59
- 12.2 Importer 60
- 12.3 Logika servisní vrstvy 60

13 Aktuální stav **61**
- 13.1 Struktura systému 61
- 13.2 Dokumentace 61
- 13.3 Vyhlídky do budoucnosti 62

Závěr **63**

Literatura **65**

A Seznam použitých zkratek **67**

B Obsah přiloženého CD **69**

Seznam obrázků

2.1 Pravděpodobnostní rozdělení odhadu — hustota pravděpodobnosti 7
2.2 Pravděpodobnostní rozdělení odhadu — distribuční funkce 7
2.3 Kužel nejistoty 8

3.1 Excel — přehled 13
3.2 Excel — podíl kategorií v celku 13
3.3 Excel — výsledek 13
3.4 Excel — varianty 14
3.5 Excel — verze 14
3.6 Excel — předpoklady 14
3.7 Excel — kategorie implementace 14
3.8 Excel — kontrolní seznam 14

5.1 Případy užití 19
5.2 Omezení položky předpokladem 22

6.1 Koncept verzování: 1 24
6.2 Koncept verzování: 2 24
6.3 Koncept verzování: 3 24
6.4 Koncept obecné struktury odhadu: 1 25
6.5 Struktura kategorií šablony 26
6.6 Koncept obecné struktury odhadu: 2 26
6.7 Koncept předpokladů: 1 27
6.8 Koncept předpokladů: 2 27
6.9 Koncept řízení přístupu 28
6.10 Koncept seskupování: 1 29
6.11 Koncept seskupování: 2 29
6.12 Koncept zamykání 30
6.13 Diagram řízení a uzamykání verzí 31
6.14 Datový model 32

7.1 Jednosměrná relace v Neo4j 37
7.2 Obousměrná relace v Neo4j 38
7.3 ASCII-art vzor grafu 38
7.4 Neo4j *browser* 41

8.1 Schéma databáze 47
8.2 Ukázka části inicializačního skriptu 49

9.1 Ukázka části třídy kategorie: definice polí 52
9.2 Ukázka části třídy kategorie: bindery 52
9.3 Ukázka části třídy kategorie: setry 53
9.4 Ukázka části rozhraní perzistence 53

10.1 Ukázka části rozhraní importu 56

Úvod

Každý projekt softwarového inženýrství se v dnešní době méně či více potýká s přísnými požadavky zákazníka. O úspěšnosti projektu tak můžou rozhodovat detaily jakékoli části softwarového procesu. Kvůli dopadu na plánování přiřazování zdrojů, časový management, cenu a následný vztah se zákazníkem je jednou z těchto zásadních disciplín i odhadování pracnosti činností. Proto se klade důraz na maximalizaci přesnosti těchto odhadů.

K tomu může sloužit mnoho různých nástrojů, na které se kladou nároky např. na dostupnost, podporu sdílení, centralizaci a synchronizaci dat napříč projekty, možnost využití historických dat a v neposlední řadě také na uživatelskou přívětivost. Jedním z používaných nástrojů jsou např. tabulkové procesory, které využívají data na lokálních discích. Takový postup může být ideální pro použití v malém rozsahu. Avšak ve firemním prostředí, kde běžně funguje několik projektů zároveň, se projeví nedostatečná podpora zmíněných organizačních nároků. Jedno z možných řešení, které by požadavky plnilo lépe, je systém skládající se z webové aplikace umožňující tvoření a správu odhadů ukládaných v centralizovaném úložišti.

Systém řešený touto bakalářskou prací je vyvíjen v rámci projektu firmy Profinit EU, s.r.o. (dále jen firma) a je používán právě jejími zaměstnanci, kterým pomáhá snadněji a efektivněji pracovat a celkově přispívá ke zlepšení průběhu softwarového procesu na projektech. Z těchto důvodů jsem se rozhodl účastnit vývoje systému a tato práce tedy dokumentuje mou účast na tomto projektu.

Konkrétně se v práci zabývám návrhem a implementací datového modelu, který odpovídá kritériím odhadů jako jsou např. hierarchické členění položek, podpora různých variant či verzování tvorby. Dále se zabývám návrhem databáze pro uložení modelovaných dat a implementací částí aplikace zajišťujících komunikaci s databází a podporující import/export dat pro navržený model.

V práci nejdříve shrnu dosavadní řešení tvorby odhadů – jeho nedostatky, které by měly být napraveny, ale také v čem je dobré a co je zachováno v novém systému. Zmíním přehled technologií použitelných pro realizaci systému s odů-

vodněním finální volby nejvhodnějších z nich. Dále se budu zabývat samotným návrhem, implementací, testováním a dokumentací zmíněných oblastí. Nakonec shrnu výsledky mé práce.

Ve své práci se nevěnuji návrhu a implementaci architektury a uživatelského rozhraní vzniklé aplikace. Touto problematikou se zabývá ve své práci student a kolega Juraj Polačok. Pro lepší pochopení kontextu se však budu na jeho práci odkazovat.

Kapitola 1

Cíl práce

1.1 Rešeršní část

Cílem rešeršní části práce je seznámení se s problematikou tvoření odhadů pracnosti (dále jen odhady), analýzou požadavků a případů užití zadavatele a dostupných technologií.

Nejdříve tedy pojednávám obecně o odhadech jako takových — co to znamená, k čemu jsou dobré apod.

Pokračuji vystižením dosavadního stavu problematiky, tedy jakým způsobem se odhadování provádí, jakým způsobem se řeší koordinace a sdílení zkušeností mezi pracovníky na jednotlivých projektech ve firmě a jaké nástroje se k tomu používají. Zmíním v čem je dosavadní řešení dobré a bylo by žádoucí to ponechat i do nového systému a naopak v čem je nevyhovující a jaké řešení by tyto nedostatky napravilo.

Následně analyzuji a diskutuji požadavky a případy užití zadavatele.

Dále analyzuji dostupné technologie, které by byly vhodné k vytvoření nového systému.

Na základě nabytých poznatků z analytické části rešerše porovnávám výhody a nevýhody jednotlivých možností a nakonec uvádím důvody výběru daných technologií.

Zaměřuji se hlavně na části systému náležící mé práci. Konkrétně na volbu databáze pro ukládání odhadů a jakým způsobem by s ní aplikace mohla komunikovat. Pro počáteční inicializaci databáze daty zvažuji možnosti importu dat do systému.

1.2 Praktická část

Cílem praktické části práce je samotný návrh a implementace datového modelu a komponent nezbytných k tomu, aby aplikace mohla pracovat s daty uloženými v modelu.

Zprvu tedy řeším návrh modelu tak, aby splňoval požadavky a vyhovoval případům užití rozebíraných v rešerši.

Pro vybranou databázovou technologii a navržený model řeším schéma databáze, ve které budou odhady ukládány. Na straně aplikace implementuji modelovou vrstvu, rozhraní pro komunikaci s databází a část servisní vrstvy související s perzistencí.

Nakonec implementuji funkčnost importu dat a exportu výsledků. Import slouží k získání dat z tabulek MS Excel do modelu aplikace. Export převádí výsledky odhadů do obecného formátu, který je využitelný i mimo systém.

Kapitola 2

Odhad

„Je velice těžké energicky a věrohodně riskovat obhajobu odhadu, který není postaven žádnou uznávanou metodou, je podložen málo daty a manažery schválený pouze ústně.“

— Fred Brooks (překlad vlastní)

2.1 Co je a co není odhad?

V této kapitole shrnuji, co říká o odhadech Steve McConnell[1, s. 3–4]:

Definice odhadu dle slovníku zní: odhad je předpověď jak dlouho bude projekt trvat nebo kolik bude stát.

V kontextu softwarových projektů však souvisí také s tématy jako jsou: obchodní cíle, závazky, řízení a plánování.

Stanovení obchodních cílů probíhá nezávisle na odhadech. Nicméně i přes to, že určité cíle mohou být žádoucí ba i nutné neznamená, že jsou dosažitelné.

Oproti tomu závazek je příslib dodat funkcionalitu v definovaném časovém horizontu a kvalitě. Takový závazek by zdánlivě měl být stejný jako odhad, ale není.

Více provázané téma s odhady je projektové plánování. Plánování z veliké části vyplývá z odhadu, avšak je mezi nimi zásadní rozdíl. Zatímco odhadování by mělo být nezaujaté, plánování už by mělo být zaujaté vlivem hledání cílů.

Jestliže se odhad rozchází s cílem, musí následný plán postihovat mezery a počítat s vysokou úrovní rizika. Na druhou stranu, když odhad je blízko cíli, plán může předpokládat méně rizika.

U softwarového projektu můžeme odhadovat jeho velikost, pracnost[1], trvání[2], cenu a další.

[1] Pracnost, také úsilí, vyjadřuje počet pracovních jednotek potřebných na dokončení činnosti nebo úkolu. Jednotkou jsou člověkohodiny, člověkodny . . . [2]

[2] Trvání vyjadřuje počet časových jednotek potřebných na provedení činnosti nebo úkolu projektu. Vyjadřuje se v časových jednotkách — hodiny, dny . . . [2]

2.2 K čemu jsou odhady dobré?

McConnell uvádí následující příklad[1, s. 13]: Představme si situaci vybírání zavazadla při přípravách na cestu. Máme menší a větší kufr. Menší kufr je pohodlnější a praktičtější, ale všechny věci se nám do něj možná nevejdou. Můžeme zkusit balit věci co nejúsporněji a případně je silou zmáčknout, ale pokud neuspějeme, musíme rozhodnout, zdali vyřadit pár věcí nebo vzít velký kufr.

U softwarového projektu je to podobné. Při plánování narazíme na mezery mezi obchodními cíli, odhadnutým rozvrhem a cenou. Pokud mezery nejsou tak velké, můžeme je vyřešit projektovým řízením. Pokud ale velké jsou, projektové cíle musí být přehodnoceny.

Hlavním účelem softwarových odhadů tedy není předpovídat projektový výsledek, ale určit jestli projektové cíle jsou natolik realistické, aby mohl být projekt řízen k jejich naplnění. „Vejdou se nám věci do malého kufru, nebo budeme nuceni vzít velký? Můžeme vzít malý kufr, když uděláme lehké úpravy?"

Vedení chce obdobné odpovědi. Často nechtějí přesný odhad, který jim řekne, že se věci do kufru nevejdou, ale chtějí plán jak pobrat věcí co nejvíce.

Vzhledem k tomu, že okolnosti projektu se obvykle mění a vyvíjí, není nutné a ani možné mít odhad naprosto přesný. Užitečný nám bude ovšem i bez toho.

Máme-li odhad, podle kterého jsme již určili, že cílů můžeme dosáhnout, pomůže nám dále při plánování řešit například tyto aspekty[1, s. 4]:

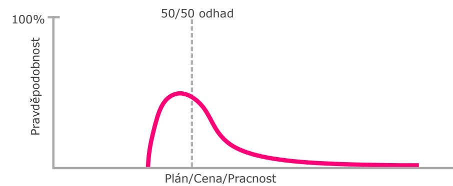
- tvorba detailního plánu (myšleno jako seznam úloh),
- identifikace kritických částí projektu,
- tvorba struktury rozpadu práce,
- nastavení priorit pro nasazení,
- rozdělení projektu do iterací.

2.3 Dobrý odhad

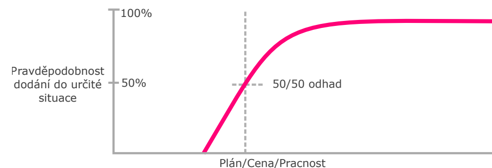
Dobrý odhad je takový, který poskytuje dostatečně čistý náhled na realitu projektu, aby umožňoval projektovému vedení provádět správná rozhodnutí v řízení projektu k dosažení jeho cílů.[1, s. 14]

2.3.1 Pravděpodobnost v odhadu

Jednočíselné odhady obvykle nemají velikou vypovídající hodnotu. Buď můžeme brát, že se odhad v tomto bodě s určitostí naplní, což není realistické, nebo prostě nemáme informaci o tom, s jakou pravděpodobností to nastane. Takové odhady bývají spíše cíle nesprávně vydávané jako odhady. Případně to může být finální výsledek nějakého sofistikovanějšího postupu.[1, s. 13]



Obrázek 2.1: Pravděpodobnostní rozdělení odhadu — hustota pravděpodobnosti[1, s. 8]



Obrázek 2.2: Pravděpodobnostní rozdělení odhadu — distribuční funkce[1, s. 8]

Správný odhad by měl obsahovat informaci o pravděpodobnosti naplnění. Klasicky se v odhadu nachází situace méně, více a nejvíce pravděpodobné. Obvykle se vyskytující normální pravděpodobnostní rozdělení zde není správné, protože četnost pozitivních scénářů rozhodně nezrcadlí situace, kdy projekt něco brzdí. Jinými slovy existuje limit jak moc dobře se může projekt vyvíjet, ale neexistuje limit jak špatně může probíhat.

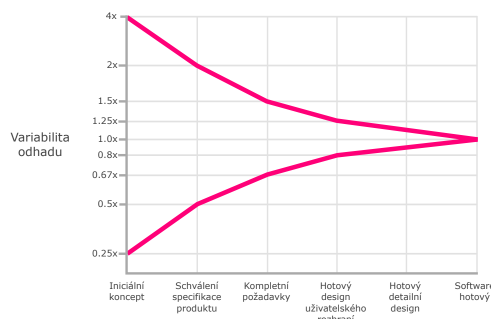
Realistické pravděpodobnostní rozdělení odhadu můžeme sledovat na obrázku 2.1, kde čárkovaná čára představuje medián výsledku. Konkrétněji tedy, že je poloviční pravděpodobnost dokončení projektu lépe a stejná pravděpodobnost, že hůře.

Existuje ještě jiná prezentace výsledků jako můžeme vidět na obrázku 2.2, kde v každém bodu vidíme ohodnocení, s jakou pravděpodobností projekt dopadne lépe.

Pro zjednodušení prezentace výsledků nemusíme používat exaktní graf. Můžeme použít např. jeden z přístupů[1, s. 9]:

- procentní jistota — „s X% jistotou dosáhneme Y“,
- nejlepší/nejhorší případ — „Y dosáhneme v nejlepším případě v X1, v nejhorším v X2“,
- možný rozsah — „Y dosáhneme v X1 až X2“.

Na základě takových odhadů můžeme uzavřít závazek tak, aby byl realistický. Důležité je vědět konkrétní stanovisko pro odpovídající naplánování.



Obrázek 2.3: Kužel nejistoty[1, s. 36]

2.4 Kužel nejistoty

Na začátku projektu nikdy není kompletně předurčen celý jeho průběh, protože nám do jeho dění zasahuje veliké množství různých vlivů. Stejně tak, jako se projekt průběžně vyvíjí, se i odhad zpřesňuje.

Tento jev vystihuje kužel nejistoty, který je výsledkem vědeckého průzkumu odhadů tvořených zkušenými odborníky.[1, s. 35]

Na obrázku 2.3 můžeme pozorovat zpřesňování odhadu v průběhu projektu. Konkrétněji kužel nejistoty vypovídá o rozptylu přesnosti odhadu vzhledem k milníkům projektu.Milníky jsou znázorněny na horizontální ose, zatímco rozptyl je vynesen na vertikální.

Z grafu vyplývá, že v iniciální fázi projektu mohou být odhady až čtyřnásobně odlišné od reality v obou směrech.

Důležité je, že kužel nejistoty neudává limit míry chybovosti.[1, s. 37] Udává pouze nejlepší případ přesnosti odhadů tvořených odborníky. Naopak je velice jednoduché vytvořit odhad s ještě větší mírou chybovosti.

2.5 Metodiky odhadování

2.5.1 Počítání a úsudek

Počítání se nemusí jevit na první pohled jako odhadování samotné. Prakticky se to však využít dá a dokonce je to jedna z nejlepších metodik.

Softwarové projekty poskytují mnoho prvků k počítání. Pro představu můžeme počítat: požadavky, funkčnosti, stránky, obrazovky, databázové tabulky, třídy a mnoho dalších. Pokud nemůžeme počítat přímo, pomohou nám někdy k výsledku závislosti k jiným entitám. Pro příklad berme odhad počtu lidí v restauraci, když víme počet stolů a jejich kapacitu. Jednodušší je rozhodně výpočet přes stoly než počítání jednotlivých osob.[1, s. 84–88]

Samotný počet prvků pro nás ještě neznamená výsledek odhadu. Pokud však máme nějaká statistická data nebo je to zjevné, obvykle není veliký problém převést počet na hledanou veličinu.

Výhody této metodiky jsou nepopiratelné. Počítání nám poskytuje nezaujaté, konzistentní, a přesné výsledky za poměrně málo úsilí.

Z uvedeného vyplývá, že pokud máme co, měli bychom odhad začít počítáním. V jiných případech nám nezbývá než provést odhad vlastním úsudkem, jinak nazývaným „intuitivním expertním odhadem."

2.5.2 Historická data

Na základě historických dat můžeme tvořit či zpřesňovat odhady za předpokladu podobností s předchozími projekty.

Abychom mohli těžit z historických dat, musíme je průběžně sbírat, přičemž platí, že čím aktuálněji to děláme, tím objektivnější data zaznamenáme. Zásadní u těchto dat je jejich konzistence. Sběrem příliš velkých podrobností bychom mohli způsobit nesrovnalosti, vedoucí až ke ztrátě vypovídající hodnoty sbíraných dat. Pro příklad můžeme sbírat alespoň: velikost projektu (v jakémkoli smyslu), pracnost, kalendářní dobu či počet defektů.[1, s. 95]

U této metodiky platí, že čím blíže jsou si porovnávané projekty, tím užitečnější data jsou. Nicméně pokud žádnými takovými daty nedisponujeme, existují veřejné statistiky. Ty ovšem nemusí být dostatečně vypovídající.

Výhodou této metodiky je, že započítává i organizační vlivy na projekt. Takové vlivy mohou být například: stabilita požadavků, pohyby projektových zdrojů, efektivita či používání určitých technik. Použití historických dat zamezuje působení optimistických vlivů jako je přeceňování pracovníků a podobně.[1, s. 92]

2.5.3 Expertní odhad

Expertní odhad, neboli „strukturovaný expertní odhad", vyplývá z myšlenky použití odhadovaného rozsahu, kterou rozšiřuje o další vstupy.

Základní rozsah odhadu je daný nejlepším a nejhorším případem. Abychom byli schopni určit jednoznačný výsledek, obohatíme tento rozsah prostřední hodnotou. Pro tuto hodnotu se jeví střed nejlepšího a nejhoršího případu jako nevhodný, a proto volíme hodnotu nejvíce pravděpodobného odhadu. Tato hodnota obvykle vznikne vlastním úsudkem.[1, s. 108]

Na základě těchto parametrů můžeme odvodit očekávaný případ například následující formulí s koeficienty vyjadřujícími váhy jednotlivých složek:

$$očekávaný = (minimální + 4 * nejpravděpodobnější + maximální)/6$$

Uvedené koeficienty slouží pouze pro příklad. V praxi je nutné koeficienty vypozorovat dle reality, která může být pro každý projekt odlišná.

U expertního odhadu hodně záleží na tom, kdo odhad dělá. Klasicky lidé, kteří budou danou práci vykonávat, podají nejpřesnější odhad.[1, s. 106]

Kontrolní seznam Kvůli vyvarování se zbytečným nedostatkům v odhadu z důvodu opomenutí nějakého celku tvoříme kontrolní seznamy, které obvykle obsahují obecné otázky předcházející opomíjeným situacím. [1, s. 110]

2.5.4 Dekompozice

Dekompozice odhadů spočívá v rozkladu větších tématických celků (kategorií) na menší (podkategorie). U takových podkategorií je snadnější přiblížit se odhadem realitě. Rozklad nám také pomáhá odhalit skryté vlastnosti, které bychom mohli na vyšší úrovni odhadu přehlédnout.

Všechny výhody dekompozice však nejsou tak zjevné.[1, s. 115] Dekompozice zároveň těží ze zákona velkých čísel. Ten jinými slovy říká, že při dostatečném počtu experimentů se odchylka ustálí na středu. Aplikujeme-li zákon velkých čísel na odhad — chybovosti jednotlivých kategorií odhadu budou mít tendenci se vynulovat, protože některé kategorie budou mít chybu pozitivní, zatímco jiné negativní.

2.5.5 Analogie

Odhadovat můžeme také na základě analogie projektu k jinému. Hlavní myšlenka je v rozpoznání odlišností, jejich zhodnocení a následné určení odhadu transformací výsledků srovnávaného odhadu.

O srovnávaném projektu zjistíme informace jako jsou jeho velikost, pracnost, cena, či struktura, byl-li dekomponován.[1, s. 128] Porovnáním jednotlivých částí nového a starého projektu získáme jejich rozdíly. Na základě těchto rozdílů dostaneme ucelenější odhad velikosti nového projektu. Výsledný odhad pracnosti získáme z pracnosti starého projektu jako poměr velikostí projektů.

2.5.6 Další metodiky

Existují ještě další metodiky, které nejsou pro účely tohoto dokumentu důležité a nebudu je tedy popisovat.

Mezi další takové patří metodiky využívající:

- fuzzy logic,
- standardní komponenty,
- story points,
- T-Shirt sizing,
- skupinové odhadování.

Kapitola 3

Dosavadní stav

V následujících kapitolách popisuji dosavadní systém odhadování pracnosti ve firmě Profinit.

3.1 Používaná metodika

Ve firmě se pro co možná nejlepší výsledky odhadování používá kombinace hned několika metodik.

Jako základ slouží metodika expertního odhadu (strukturovaného, viz. 2.5.3) definující složky odhadovaných položek a vztah mezi nimi. Expertní odhad je obvykle dekomponován na jednotlivé kategorie a případně i varianty. Používání dalších metodik se na první pohled neprojevuje tak výrazně, avšak umožňuje-li to odhadovaná tématika a kontext, složky expertního odhadu se určují pomocí počítání či historických dat. Standardní složky (minimum, maximum, nejpravděpodobnější a očekávaný) jsou doplněny o složku průměru minima a maxima.

Rozlišuje se výsledná odhadnutá pracnost a finální závazná pracnost. U výsledku odhadu je v pořádku, ba podle 2.3.1 je dokonce žádoucí, aby byl ve formě rozsahu, zatímco u závazné pracnosti je vyžadováno konkrétní číslo.

U kategorií odhadu se zavádí pojem rizika, které koresponduje s velikostí rozdílu mezi minimální a maximální složkou.

Dále se používá speciální kategorie záruka. Ta vyjadřuje předpokládanou pracnost, kterou si vyžádají opravy v záruce — období po nasazení.

Každý odhad může být podpořen výčtem omezujících podmínek (předpokladů), které slouží pro obhajobu výsledků odhadu.

Firemní kultura disponuje šablonami obsahujícími kontrolní seznamy (2.5.3) pro odhady.

Každý odhad musí projít schvalovacím procesem. Ten může znamenat až několik kol revizí na různých úrovních. Každopádně revizi musí provádět člověk nezaujatý originálním odhadem.

3.2 Používané nástroje

Vzhledem k povaze práce odhadování, která je především o lidském úsudku, se nepoužívá veliké množství různých nástrojů. V podstatě jediný nástroj, který se používá pro samotnou tvorbu odhadů je tabulkový procesor Microsoft Excel (dále jen Excel). Vedle toho slouží několik nástrojů či technologií, které mají podpůrné funkce pro sdílení či verzování.

3.3 Excel

Excel zastává hned několik užitečných funkcí.

Z důvodu, že struktura expertního odhadu odpovídá tabulkovému rozvržení, plní Excel ideálně funkci přehledné reprezentace odhadovaných dat. Díky jeho silným výpočetním a agregačním funkcím můžeme tvořit přínosné statistiky a počítat výsledky. Další schopnost Excelu — makra — nám umožňuje generování výsledků do obyčejného textového formátu.

Dekompozice je v Excelu řešena na úrovni kategorií.

3.3.1 Ukázky použití

Na obrázkových ukázkách můžeme vidět použití Excelu pro:

- souhrn kategorií odhadu 3.1,
- zobrazení statistických informací o odhadu 3.2,
- zformování výsledku odhadu 3.3,
- simulaci variant odhadu 3.4,
- vedení záznamů o vývoji odhadu 3.5,
- zaznamenání omezujících podmínek (předpokladů) odhadu 3.6,
- samotné odhadování pracnosti položek jedné kategorie 3.7,
- zobrazení kontrolního seznamu ke kategorii 3.8.

3.4 Koordinace práce v týmech

Práce na odhadech je ve firmě koordinovaná především dohodou. Dosavadně používané technologie jiný přístup ani neumožňují. Data odhadů jsou sdílena a synchronizována v centrálním úložišti.

Celkový přehled						
	Min (MD)	Max (MD)	Nej. prav. (MD)	Prům. (MD)	Oček. (MD)	Riziko (%)
Analýza	0,3	0,6	0,4	0,4	0,4	21,2%
Design	0,3	0,6	0,6	0,5	0,5	17,6%
Implementace	0,6	1,2	0,9	0,9	0,9	31,7%
Testování	0,3	0,5	0,4	0,4	0,4	14,1%
PM	0,1	0,1	0,1	0,1	0,1	3,5%
Dodávka	0,1	0,3	0,2	0,2	0,2	7,1%
Ostatní	0,0	0,0	0,0	0,0	0,0	0,0%
Záruka	*0,1*	*0,2*	*0,1*	*0,1*	*0,1*	4,8%
Realizace (MD)	**1,3**	**2,4**	**1,9**	**1,8**	**1,9**	
Realizace včetně záruky (MD)	**1,3**	**2,6**	**2,0**	**2,0**	**2,0**	
Celkem bez záruky (MD)	**1,6**	**3,3**	**2,4**	**2,5**	**2,4**	
Celkem včetně záruky (MD)	**1,7**	**3,5**	**2,6**	**2,6**	**2,6**	

Obrázek 3.1: Sekce přehledu obsahuje souhrnné odhadované hodnoty pro jednotlivé kategorie odhadu. V přehledu se také objevuje pojem realizace, která vyjadřuje souhrn vybraných kategorií, typicky upřesněných po analýze.

	Podíl kategorií v celku				
	Min (MD)	Max (MD)	Nej. prav. (MD)	Prům. (MD)	Oček. (MD)
Analýza	14,65%	17,97%	14,65%	16,88%	15,40%
Design	18,32%	17,97%	21,98%	18,08%	20,67%
Implementace	36,63%	34,14%	34,19%	34,96%	34,45%
Testování	14,65%	14,38%	14,65%	14,47%	14,59%
PM	3,66%	3,59%	2,44%	3,62%	2,84%
Dodávka	7,33%	7,19%	7,33%	7,23%	7,29%
Ostatní	0,00%	0,00%	0,00%	0,00%	0,00%
Podíl realizace	73,26%	70,08%	73,26%	71,13%	72,54%

Obrázek 3.2: Sekce se statistickými informacemi o podílu kategorií v celku. Počítá se ze souhrnu včetně záruky.

	Odhad pracnosti - var. 1			
Činnost	Analýza	Realizace (předběžná)		Dodávka
		Minimum	Maximum	
Pracnost v čd	0,4	1,3	2,6	0,2
Cena bez DPH				

Obrázek 3.3: Finální výsledek odhadu členěný na analýzu, realizaci a dodávku. Cena je orientačně vypočítána paušálně za hodinu.

	A	B	C	D	E	F	G
72	**Dostupné varianty**	1	2	3	4	5	6
73	**Uvažované varianty**	1	2				

Obrázek 3.4: Simulace různých variant odhadu. Ve výpočtech započítávají položky podle „uvažovaných variant“.

	C	D	E	F
12	**Datum**	**Verze**	**Autor**	**Poznámka**
13	29.04.2017	1.0	Milan Vancl	iniciální odhad
14	30.04.2017	2.0	Milan Vancl	verze 2

Obrázek 3.5: Záznam o vývoji odhadu.

	A	B
1	**Varianta**	**Předpoklady, omezující podmínky**
2		Platnost odhadu je 3 měsíce od data předání zákazníkovi.
3		V rámci tohoto ZŘ nejsou řešeny výkonnostní požadavky ani měření odezvy systému.
4		
5		Nepočítá se s verzováním šablon
6	2	Počítá se s verzováním šablon
7		

Obrázek 3.6: Předpoklady odhadu. Ve sloupci A může být předpoklad zařazen do odpovídající varianty.

	A	B	C	D	E	F	G
1		**Implementace**					
2	**Varianta**	**Popis činnosti**	**Min (MH)**	**Max (MH)**	**Nej. prav. (MH)**	**Prům. (MH)**	**Oček. (MH)**
3		Tvorba inicializačního skriptu pro databáze	2,00	3,50	2,50	**2,75**	**2,58**
4		Implementace modelové vrstvy aplikace pro šablonu	1,00	2,00	1,50	**1,50**	**1,50**
5	2	Implementace podpory pro verzování šablony v modelové vrstvě aplikace	2,00	4,00	3,00	**3,00**	**3,00**
6						**0,00**	**0,00**
7						**0,00**	**0,00**
8						**0,00**	**0,00**
9						**0,00**	**0,00**
10						**0,00**	**0,00**
11		**Celkově**	**5,00**	**9,50**	**7,00**	**7,25**	**7,08**

Obrázek 3.7: Kategorie implementace — samotný list „Implementace“. Obsahuje položky, které se skládají ze složek expertního odhadu. Ve sloupci A může být položka zařazena do odpovídající varianty.

	X
1	**Checklist implementace**
2	Znalost technologie a dané problémové domény.
3	Vztah řešení ke stávajícím datům v systému - konverze, datafixy, … (viz design).
4	Nastavení vývojového prostředí.
5	Tvorba jednotkových testů, úprava stávajících.
6	Tvorba testovacích dat.
7	Tvorba mock rozhraní pro základní otestování po implementaci.
8	Čas pro otestování vlastní implementace (včetně debugování) - testování po vývoji.
9	Programátorská dokumentace.

Obrázek 3.8: Kontrolní seznam pro kategorii implementace.

3.5 Sdílení zkušeností mezi kolegy

Know how se mezi kolegy sdílí pomocí proprietárních příruček, manuálů a školení. V pokročilé fázi individuálním přístupem.

3.6 Dobré vlastnosti dosavadního řešení

Mezi dobré vlastnosti dosavadního řešení zahrnu bez pochyb celou používanou metodiku z kapitoly 3.1. Metodika samotná je však nezávislá na technologiích, takže nemá veliký smysl ji v rámci této práce řešit. Cílem této práce je neomezit, ba naopak umožnit lepší využití metodiky tak, aby z ní mohlo být těženo ještě více.

Další silné stránky dosavadního řešení korespondují se silnými stránkami používaných technologií.

To převážně znamená silné stránky nástroje Excel, který vyniká v uživatelské přívětivosti, širokými možnostmi počítání a praktické agregaci dat. Tuto problematiku ovšem řeší ve své bakalářské práci Juraj Polačok[3], takže se jí konkrétněji nevěnuji.

Dobrá vlastnost Excelu je na jednu stranu i volnost uživatele provádět cokoli mu umožňuje. Například vepisování různých poznámek či seskupovaní položek odhadu.

Za zmínku jistě také stojí možnost použití maker v Excelu a v neposlední řadě jeho know how v široké veřejnosti. V některých situacích může být vhodné, že s Excelem lze pracovat i bez připojení k síti.

3.7 Nedostatky

Vzhledem k tomu, že metodice není co vytknout, je na místě řešit nedostatky používaných technologií.

I přes to, že Excel je silný nástroj, nativně umožňuje pouze plochou strukturu odhadu, což je limitující pro použití metodiky dekompozice. Co se týče verzování tvorby odhadu: pomocí verzovacího systému sice jsme schopni zaznamenávat jednotlivé verze, ale už nejsme schopni triviálně sledovat rozdíly či přírůstky oproti jiným verzím. Když zajdu do podrobnějších detailů, řešení variant v Excelu sice funguje, ale příliš elegantní není. Nakonec volnost uživatele provádět v Excelu cokoli mu umožňuje je vlastnost dvousečná a způsobuje v odhadech riziko zanesení nekonzistencí a různých skrytých chování.

Nedokonalá je také dostupnost celého systému. K práci na odhadu je potřeba mít přístup ke sdílenému úložišti, specifický software a být domluvený s kolegy, respektive být pověřený vedením.

Dalším nedostatkem je, že organizace odhadů ve firmě není centralizovaná a nic tak nevynucuje synchronizaci dat napříč projekty. To pak může omezovat možnost využití metodiky historických dat.

Kapitola 4

Analýza požadavků

V následující kapitole diskutuji požadavky zadavatele týkající se datového modelu.

P1: Základní jednotka odhadu Základní jednotkou odhadu bude položka obsahující:

- popis úlohy/položky v rámci odhadu,
- ohodnocení pracnosti odpovídající používané metodice,
- variantu, do které spadá,
- verzi, ve které je zaznamenána či revidována.

P2: Podpora více metod odhadování Model bude podporovat více metod odhadování. Kromě možnosti rozpadu odhadu dle dekompozice bude možné položky členit a seskupovat například dle variant či tagů.

P3: Metody definované šablonou Jednotlivé metody odhadu budou definovány šablonou.

P4: Obecná struktura šablony Struktura šablony bude dynamická, díky čemuž bude možné modelovat obecné stromové struktury členění odhadu. Kromě toho šablona bude poskytovat kontrolní seznamy k jednotlivým kategoriím.

P5: Šablona určuje výpočet odhadu Výpočet očekávané hodnoty pracnosti bude udávat šablona přiřazením vah jednotlivým složkám odhadu.

P6: Podpora variant odhadu Model bude podporovat různé varianty odhadu, podle kterých bude možné členit položky. Varianty mohou znamenat například složitostní úrovně projektu.

P7: Okrajové podmínky — předpoklady Odhad bude vždy obsahovat okrajové podmínky, respektive předpoklady.

P8: Provázání předpokladů na položky odhadu Předpoklady mohou být provázány s položkami pro zajištění vztahu, že daná položka je omezena patřičnými předpoklady. Toto provázání je typu M:N.

P9: Podpora verzování a schvalovacího procesu Model bude podporovat verzování odhadu a umožňovat simulaci schvalovacího procesu, který může znamenat i několik revizí odpovědnými osobami.

P10: Řízení uživatelského přístupu Přístup do systému bude podmíněn přihlášením pomocí doménového jména/hesla. Na základě přihlášení bude uživatel disponovat právy k odhadům. Granularita práv bude alespoň následující:

- právo čtení,
- právo úpravy,
- právo revize,
- právo schvalování.

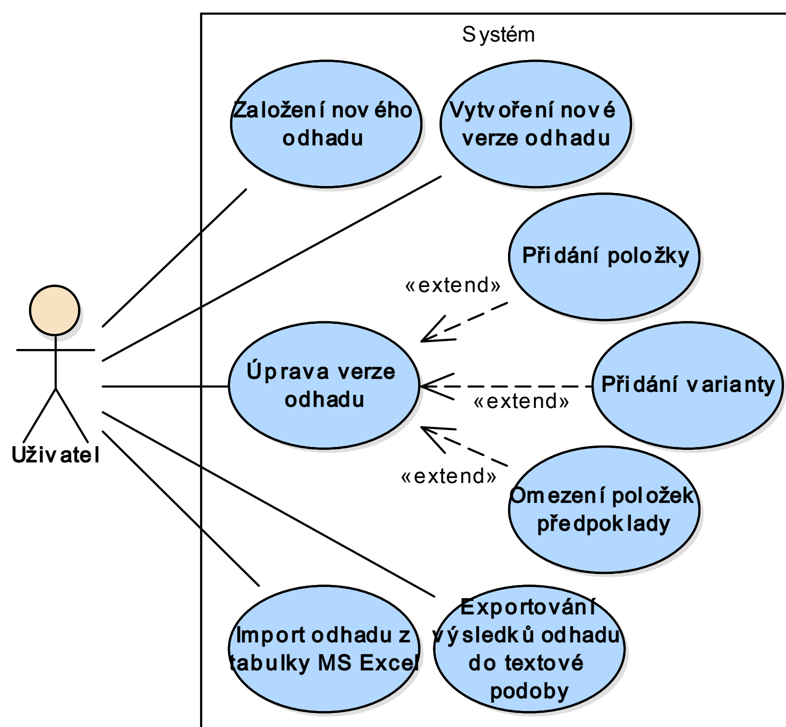
P11: Podpora importu dat Aplikace bude podporovat import dat z dosud používaných Excelů. Pro import je definováno exaktní rozhraní, které musí Excelové soubory k úspěšnému importu splňovat.

P12: Podpora exportu dat Aplikace bude podporovat export dat alespoň do textového formátu, který bude běžně použitelný i mimo systém.

Kapitola 5

Případy užití

Na základě požadavků tvořím základní případy užití. Jejich souhrn je na obrázku 5.1



Obrázek 5.1: Případy užití

PU1: Založení nového odhadu — umožňuje uživateli vytvořit zcela nový odhad. Uživatel musí disponovat právem zápisu pro cílený projekt. Vytvoření nového odhadu je možné na základě výběru patřičné šablony a projektového kontextu. Uživatel může novému odhadu nastavit popis, varianty a iniciální verzi. Nový odhad přejímá strukturu vybrané šablony. Po vytvoření může uživatel vyplňovat položky a předpoklady iniciální verze.

Obecný scénář vytvoření nového odhadu:

1. Přihlášený uživatel volí tvorbu nového odhadu.
2. Uživatel volí odpovídající šablonu a projektový kontext. Dále nastaví ostatní parametry odhadu.
   a) V případě, že odpovídající šablona neexistuje, se uživatel může obrátit na pověřenou osobu s žádostí o vytvoření dané šablony. Scénář zde v této větvi končí.
3. Nový odhad je vytvořen s požadovanou strukturou a uživatel může vyplňovat jeho obsah.

Scénář 1: Uživatel chce vytvořit odhad, který bude mít plochou strukturu obsahující kategorie: návrh, implementace, testování. Dále uživatel postupuje podle obecného scénáře vytvoření nového odhadu.

Scénář 2: Uživatel chce vytvořit odhad, který bude mít hierarchickou strukturu obsahující kategorie: analýza, realizace, dodávka. Kategorie realizace má podkategorie: návrh, implementace, testování. Kromě toho chce mít pro výpočet očekávané složky u expertního odhadu koeficient 3. Dále uživatel postupuje podle obecného scénáře vytvoření nového odhadu.

PU2: Vytvoření nové verze odhadu — umožňuje uživateli vytvořit novou verzi odhadu. Uživatel musí disponovat právy zápisu pro projekt, pod který odhad spadá. V rámci odhadu lze upravovat pouze jednu verzi v jeden čas a to pouze jedním uživatelem. Obsahuje-li odhad jinou verzi, kterou lze upravovat, vytvoření nové verze není možné. Vytvoření nové verze je možné na základě výběru původní verze. Nová verze je kopií původní verze. Po vytvoření nové verze může uživatel upravovat, přidávat, či odstraňovat její položky a předpoklady.

PU3: Úprava verze odhadu — umožňuje uživateli upravovat verzi odhadu. Uživatel musí disponovat právy zápisu pro projekt, pod který odhad spadá. V rámci odhadu lze upravovat pouze jednu verzi v jeden čas a to pouze jedním uživatelem. V rámci úpravy verze může uživatel přidávat, měnit či odstraňovat položky, předpoklady a varianty.

PU4: Přidání položky — umožňuje uživateli přidávat položky v rámci úpravy verze odhadu PU3. Při přidání uživatel zadává popis, složky minimálního, maximálního a expertního odhadu a variantu, do které patří. Po přidání položky je možné ji upravit či smazat.

PU5: Přidání varianty — umožňuje uživateli přidávat varianty v rámci úpravy verze odhadu PU3. Při přidání uživatel zadává označení a popis. Variantě lze libovolně přiřazovat položky a předpoklady. Po přidání varianty je možné ji smazat.

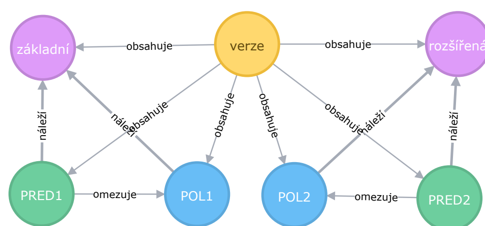
Scénář 1: Přihlášený uživatel pracuje na odhadu projektu podporujícího správu šablon. Uživatel chce 2 varianty odhadu: první šablony neverzuje, druhá ano.

1. Uživatel vytvoří variantu s označením *základní* a popisem „Základní varianta, která nepočítá s verzováním šablony."
2. Uživatel vytvoří variantu s označením *rozšířená* a popisem „Varianta počítá i s verzováním šablony, rozšiřuje základní variantu."
3. Uživatel vytvoří položky: *POL1* = „Implementace modelové vrstvy aplikace pro šablonu", *POL2* = „Implementace podpory pro verzování šablony v modelové vrstvě aplikace".
4. Uživatel přiřadí položku *POL1* do varianty *základní* a *POL2* do varianty *rozšířená*.

PU6: Omezení položek předpoklady — umožňuje uživateli v rámci úpravy verze odhadu PU3 provázat položky na předpoklady ve smyslu jejich omezení.

Scénář 1: Přihlášený uživatel pracuje na odhadu projektu podporujícího správu šablon. Projekt má 2 varianty: *základní* šablony neverzuje, *rozšířená* ano. Situace je zachycena na obrázku 5.2.

1. Uživatel vytvoří pro varianty dva předpoklady: *PRED1* = „šablona je verzovaná", *PRED2* = „šablona není verzovaná".
2. Uživatel vytvoří položky: *POL1* = „Implementace modelové vrstvy aplikace pro šablonu", *POL2* = „Implementace podpory pro verzování šablony v modelové vrstvě aplikace". *POL2* náleží pouze variantě *rozšířená*.
3. Uživatel prováže předpoklady a položky následujícím způsobem: *PRED1* omezuje *POL1* a *PRED2* omezuje *POL2*.



Obrázek 5.2: Omezení položky předpokladem

PU7: Import odhadu z tabulky MS Excel — umožňuje uživateli importovat odhad z tabulky MS Excel. Uživatel musí disponovat právem zápisu pro cílový projekt. Importování odhadu je podmíněno výběrem šablony a projektového kontextu. Importovaný soubor musí splňovat přijímaný formát, který je přesně specifikovaný v dokumentaci. Importovaný odhad musí odpovídat vybrané šabloně, podle které přejímá strukturu. Po úspěšném importu je možné odhad upravovat. V případě chyby během zpracování je uživateli zobrazena informace s detailním popisem příčiny chyby.

PU8: Exportování výsledků odhadu do textové podoby — umožňuje uživateli exportovat výsledky odhadu do obecného textového formátu použitelného mimo systém. Uživatel musí disponovat právy čtení pro projekt, pod který odhad spadá. Exportovaný text obsahuje předpoklady a rozsah pracnosti.

Scénář 1: Odhad určený k exportu je dokončený, proběhla jeho revize a schválení.

1. Přihlášený uživatel zvolí export odhadu, čímž získá výsledky v textové podobě.
2. Uživatel si export uchová.
3. Pověřená osoba může předložit exportované výsledky osobám, které nemají do systému přístup (např. zákazník).

Kapitola 6

Návrh datového modelu

V následující kapitole popisuji vývoj návrhu datového modelu pro nově vznikající systém.

Nejdříve popisuji návrh konceptů modelu a poté jednotlivé entity s jejich vazbami.

6.1 Koncepty datového modelu

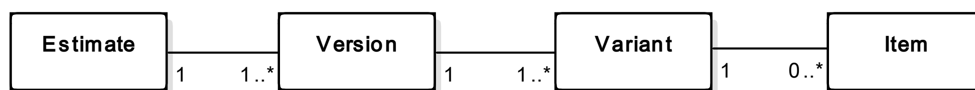
6.1.1 Koncept verzování

Pro zajištění funkce verzování vývoje odhadu dle případu užití PU2 a požadavků P9 a P6 zvažuji 2 schémata:

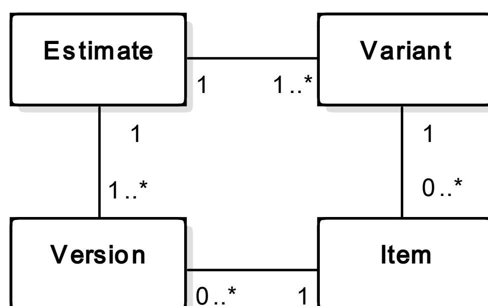
1. Schéma 6.1 řeší koncept na úrovni entity samotné verze (*Version*), tzn.: odhad (*Estimate*) uchovává své verze, které mají nezávisle mezi sebou varianty (*Variant*). Jednotlivé položky (*Item*) jsou vázány pouze na varianty. Problém tohoto řešení je, že varianty nejsou napříč verzemi a celého odhadu sjednocené.

2. Schéma 6.2 řeší koncept tak, že: odhad uchovává verze i varianty. Položka patří nezávisle do verze i varianty. Nezávislost verze a varianty zde způsobuje problém, že nedokážeme implicitně postihnout jaké varianty existují v jednotlivých verzích.

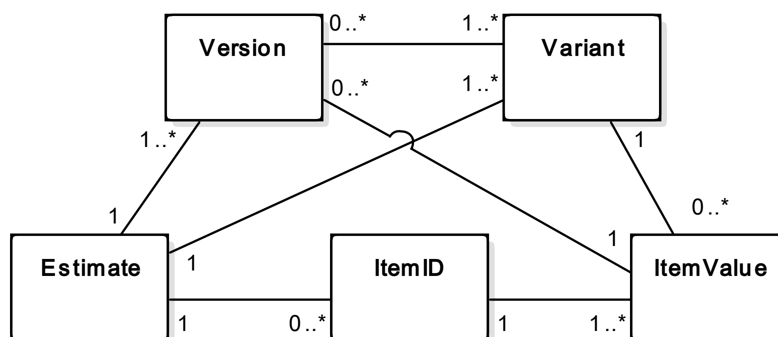
Při verzování je důležité mít možnost porovnávat jednotlivé verze. Abychom mohli sledovat jednoznačný rozdíl mezi položkami v různých verzích, musí položky být jednoznačně identifikovatelné nezávisle na verzi. Schéma na obrázku 6.3 řeší problém rozdělením položky na identifikátor (*ItemID*) a samotnou hodnotu (*ItemValue*), přičemž hodnota zůstává závislá na variantě a verzi, zatímco identifikátor je jednoznačný v rámci celého odhadu. Kromě toho schéma 6.3 kombinuje výhody a zároveň ruší nevýhody 6.1 a 6.2.



Obrázek 6.1: Koncept verzování: 1



Obrázek 6.2: Koncept verzování: 2



Obrázek 6.3: Koncept verzování: 3

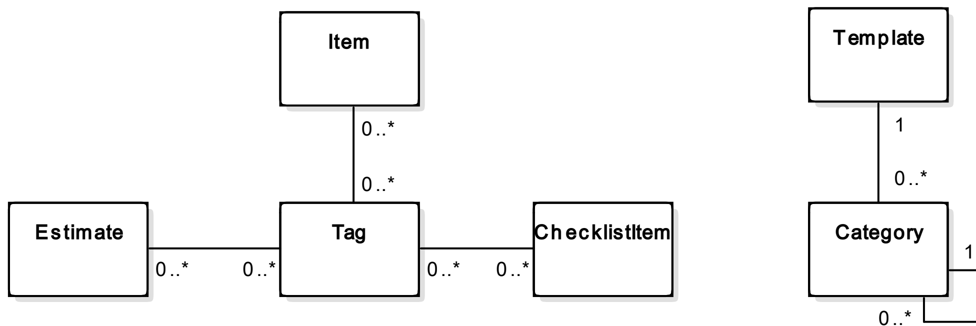
U konceptu verzování je také na místě otázka: Co se stane s položkami při vytvoření nové verze?

V programování existují 2 základní metody kopírování objektů: hluboká[3] a mělká[4] kopie. Vzhledem k tomu, že úspora prostoru databáze není prioritou a implementace mělkého kopírování s sebou nese nutnost implementace metody „kopírování při zápisu“, volím jednodušší hluboké kopírování.

Ve výsledku to znamená, že pro vytvoření nové verze se fyzicky zkopírují všechny položky původní verze.

[3]Hluboká kopie — fyzicky kopíruje všechny data tak, že původní a nová data na sobě jsou zcela nezávislá.

[4]Mělká kopie — kopíruje pouze reference na původní data. S touto metodou je spojen pojem „kopírování při zápisu“ (ang. copy on write), který řeší problém nechtěného přepisování původních a nových dat.



Obrázek 6.4: Koncept obecné struktury odhadu: 1

6.1.2 Koncept obecné struktury odhadu

Pro umožnění tvoření odhadu s libovolnou strukturou dle případu užití PU1 a požadavku P4 zvažuji dvě řešení.

6.1.2.1 Proměnná struktura

První řešení tkví v univerzálním sestavování odhadu pomocí značkování, tzv. tagování. Na schématu 6.4 je to vystiženo značkou (*Tag*), na kterou se váže položka. Kromě položek může být stejným způsobem označen i odhad a položka kontrolního seznamu (*ChecklistItem*). Každá z těchto entit může být označena libovolným počtem značek a naopak libovolný počet entit může být označen stejnou značkou. Na schématu je také vystižena šablona (*Template*) s kategoriemi (*Category*) definujícími samotnou hierarchickou strukturu.

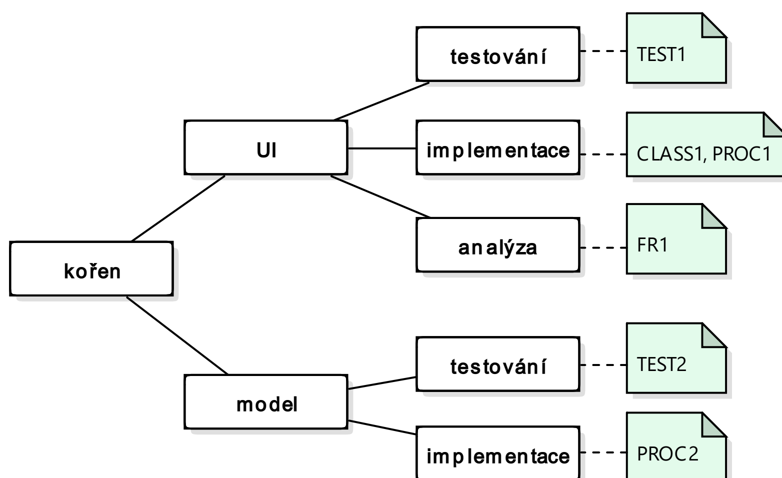
Schéma 6.4 využívá toho, že odhad nemá vlastní pevnou strukturu, nýbrž jeho struktura se buduje při výběru šablony, podle které se má zformovat. Toho je možné dosáhnout na základě označení položek odpovídajícího kategoriím zvolené šablony. Aby bylo zformování jednoznačné, musí být kategorie a označení položek unikátní na cestě strukturou od kořene ke kategorii, do které položka patří.

Stejně jako položky se zde skládají kontrolní seznamy.

Ukázku formování uvádím na příkladu položek z tabulky 6.1 přiřazených kategoriím šablony na obrázku 6.5.

Tabulka 6.1: Položky s označením

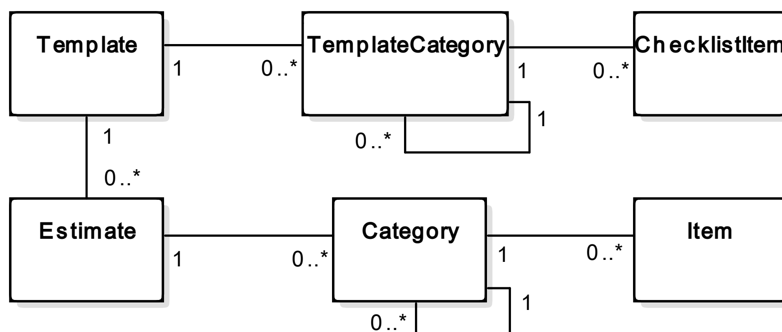
Název	Značky
FR1	UI, analýza
PROC1	UI, implementace
PROC2	model, implementace
CLASS1	UI, implementace
TEST1	UI, testování
TEST2	model, testování



Obrázek 6.5: Struktura kategorií šablony s přiřazenými položkami

6.1.2.2 Pevná struktura

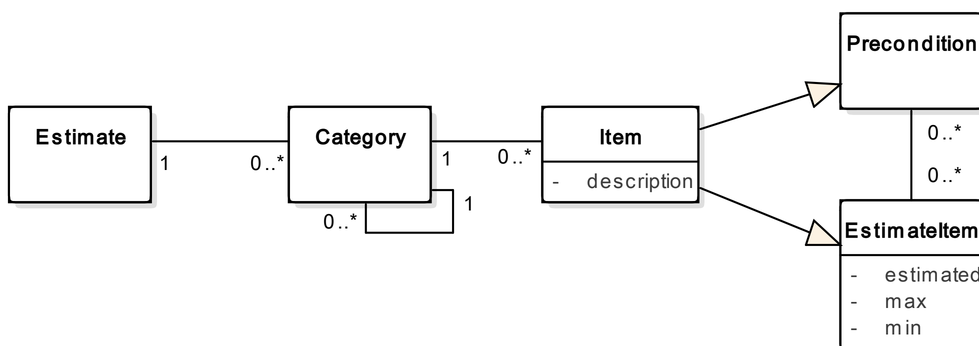
Další řešení je konzervativnější — odhad má strukturu pevně určenou šablonou, dle které je vytvořen. Schéma na obrázku 6.6 vystihuje toto řešení: odhad přímo uchovává celou strukturu pomocí kategorií (*Category*), které obsahují vlastní položky. Struktura odhadu tvořena kategoriemi je přímá kopie struktury kategorií šablony (*TemplateCategory*). Položky kontrolních seznamů jsou vázány na kategorie šablony. Pro možnost dohledání kontrolních seznamů je odhad provázaný s šablonou.



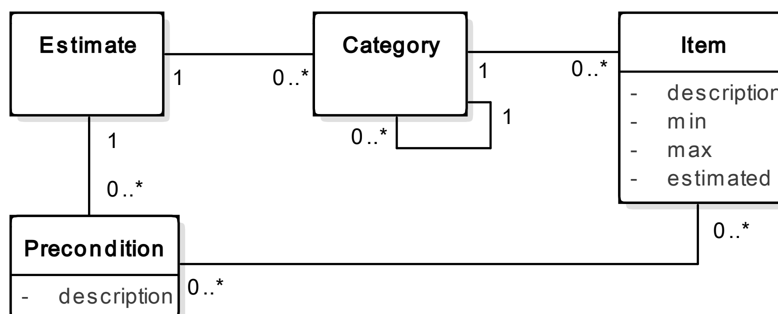
Obrázek 6.6: Koncept obecné struktury odhadu: 2

6.1.2.3 Volba

Z důvodu, že podle případu užití PU1 a požadavku P4 je nadbytečné mít proměnnou strukturu odhadu, volím řešení se strukturou pevnou (6.6), které je zároveň implementačně jednodušší.



Obrázek 6.7: Koncept předpokladů: 1



Obrázek 6.8: Koncept předpokladů: 2

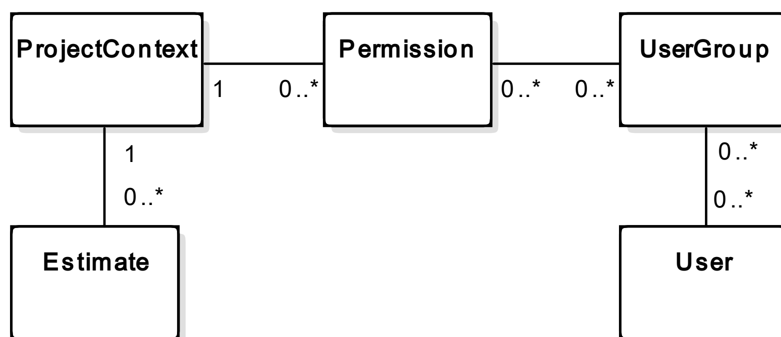
6.1.3 Koncept předpokladů

Odhad dle používané metodiky, probírané v kapitole 3.1, obsahuje kromě položek také předpoklady. Při jejich začlenění do modelu dle požadavků P7, P8 a případu užití PU6 uvažuji dvě následující varianty:

1. Varianta podle schématu 6.7 pojímá předpoklady (*Precondition*) a odhadované položky (*EstimateItem*) jako velice příbuzné entity. Kvůli tomu je využito generalizace, kde nadřazená entita obecné položky (*Item*) uchovává popis. Odhadovaná položka rozšiřuje obecnou položku o odhadované hodnoty. Pro předpoklady existuje speciální kategorie a je možné je libovolně provázat s odhadovými položkami.

2. Varianta zachycená na schématu 6.8 chápe předpoklady (*Precondition*) a položky (*Item*) jako odlišné entity. Zatímco položky patří standardně do kategorií, předpoklady se vážou přímo na odhad. Provázání položek s předpoklady je obdobné jako v první variantě.

Z důvodu, že předpoklady a položky reálně nejsou příbuzné entity, konceptuálně vhodnější varianta je druhá a volím ji.



Obrázek 6.9: Koncept řízení přístupu

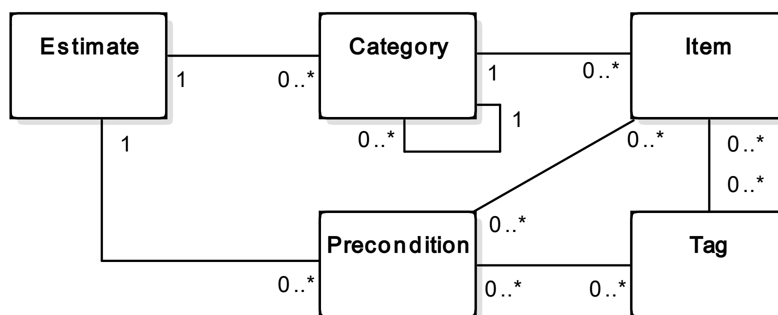
6.1.4 Koncept řízení přístupu

Pro řízení přístupu dle případů užití 5 a požadavku P10 uvažuji pouze jednu variantu vystiženou schématem na obrázku 6.9.

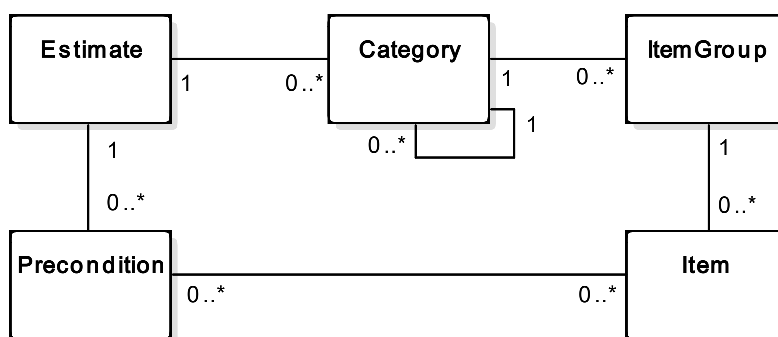
Řízení přístupu je řešeno standardním způsobem pomocí uživatelských skupin a projektových kontextů, který vyžaduje minimum změn pro přidání nového odhadu či uživatele.

Uživatelé (*User*) jsou členy uživatelských skupin (*UserGroup*), které disponují právy (*Permission*) k projektovým kontextům (*ProjectContext*). Každý odhad je veden v rámci jednoho projektového kontextu.

Pro příklad: aby uživatel mohl upravovat daný odhad — odhad musí být veden v kontextu, pro který má právo zápisu skupina, které je uživatel členem. Pro další operace jako je čtení, revize či schvalování je přístup obdobný a nezávislý na ostatních.



Obrázek 6.10: Koncept seskupování: 1



Obrázek 6.11: Koncept seskupování: 2

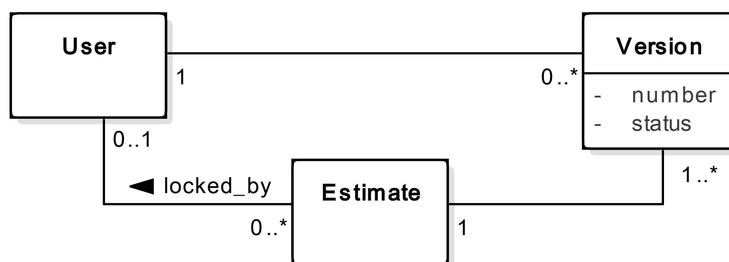
6.1.5 Koncept seskupování

Pro koncept seskupování dle požadavku P2 zvažuji dvě varianty:

1. Varianta podle schéma 6.10 řeší seskupování pomocí označení (*Tag*). Seskupovat lze tímto způsobem položky i předpoklady. Seskupování funguje na základě stejné či stejných značek.

2. Varianta podle schéma 6.11 řeší seskupování odlišně a mění zásadně myšlenku členění položek. Položky jsou seskupovány na základě náležitosti do skupiny (*ItemGroup*). Tímto způsobem lze položku zařadit pouze do jedné skupiny. Rozdíl mezi kategorií a skupinou je takový, že kategorie je pevně daná šablonou, zatímco skupina je libovolně upravovatelná. Koncept skupin by šel rozšířit pro verzování úpravy skupin, hierarchické členění skupin podobně jako u kategorií a aplikaci pro předpoklady.

Z důvodu komplexnosti a náročnosti na implementaci varianty 2 volím variantu 1. Varianta 1 má také oproti variantě 2 výhodu možnosti seskupení podle různých kritérií aniž by byla potřeba změna struktury odhadu. Na druhou stranu varianta 2 má silný potenciál a do budoucna by mohla být uvažována pro implementaci.



Obrázek 6.12: Koncept zamykání

6.1.6 Koncept řízení verzí

Podle případů užití PU2 a PU3 je verzování řízeno na základě stavu (*status*) samotných verzí odhadu. To je zachyceno na schématu 6.12. Zásadní je zde rozlišení stavu verzí na „stabilní“ a „rozpracované“. Stabilní verze jsou již neměnné a poskytují základ pro tvoření nových rozpracovaných verzí. Rozpracované verze mohou být upravovány a nakonec potvrzeny za stabilní.

Nabízí se dvě varianty pro tvoření nových verzí:

1. Rozpracovaná verze může existovat v rámci odhadu pouze jedna. Na té se může účastnit více uživatelů, ale pouze sériovým přístupem.

2. Rozpracovaných verzí může existovat v rámci odhadu více než jedna. Více uživatelů by se tedy mohlo účastnit práce na odhadu paralelně. V takovém případě by to znamenalo nezbytnost řešení spojování jednotlivých rozpracovaných verzí.

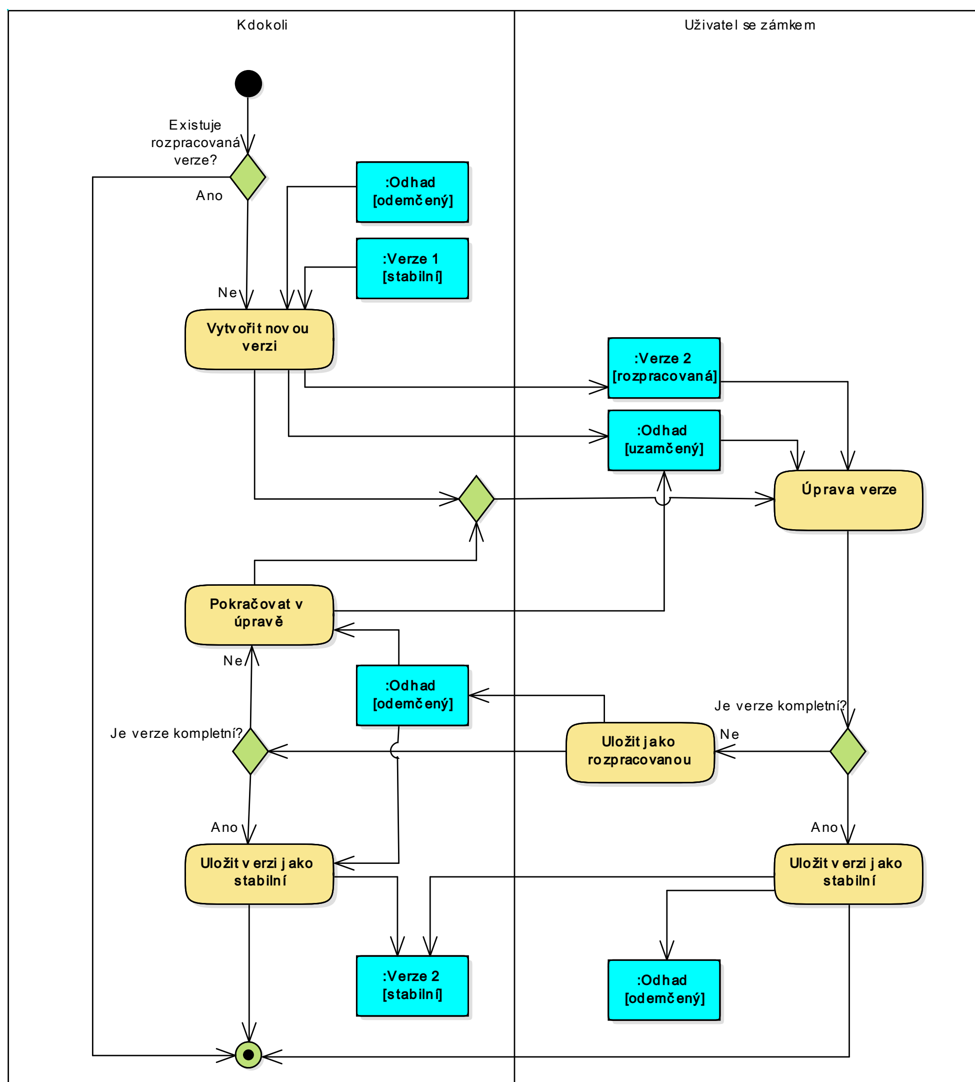
Z důvodu, že paralelní práce na odhadu není prioritou a varianta 1 je jednodušší na implementaci, volím variantu 1.

6.1.6.1 Uzamykání odhadu

Aby nedocházelo k nekonzistencím při paralelní nebo nevalidní úpravě odhadu, aplikuji systém uzamykání. Stěžejní myšlenkou uzamykání je, že pokud chce uživatel provádět úpravy na odhadu, musí nejdříve získat zámek, který získá pouze pokud ho již nemá někdo jiný.

Realizace zámku v modelu je zachycena na schématu 6.12 vazbou *locked_by* a kompletní řízení verzí a uzamykání odhadu je zachyceno na diagramu 6.13:

Pokud odhad neobsahuje žádnou rozpracovanou verzi, je zaručena jeho odemčenost a kdokoli může vytvořit novou verzi. Při vytváření nového celého odhadu se s ním vytvoří i iniciální verze. V jiných případech se vytvoří nová verze jako kopie již existující vybrané stabilní verze. V obou případech je nová verze ve stavu rozpracovaná a uživatel, který ji vytvořil získává zámek na odhad.



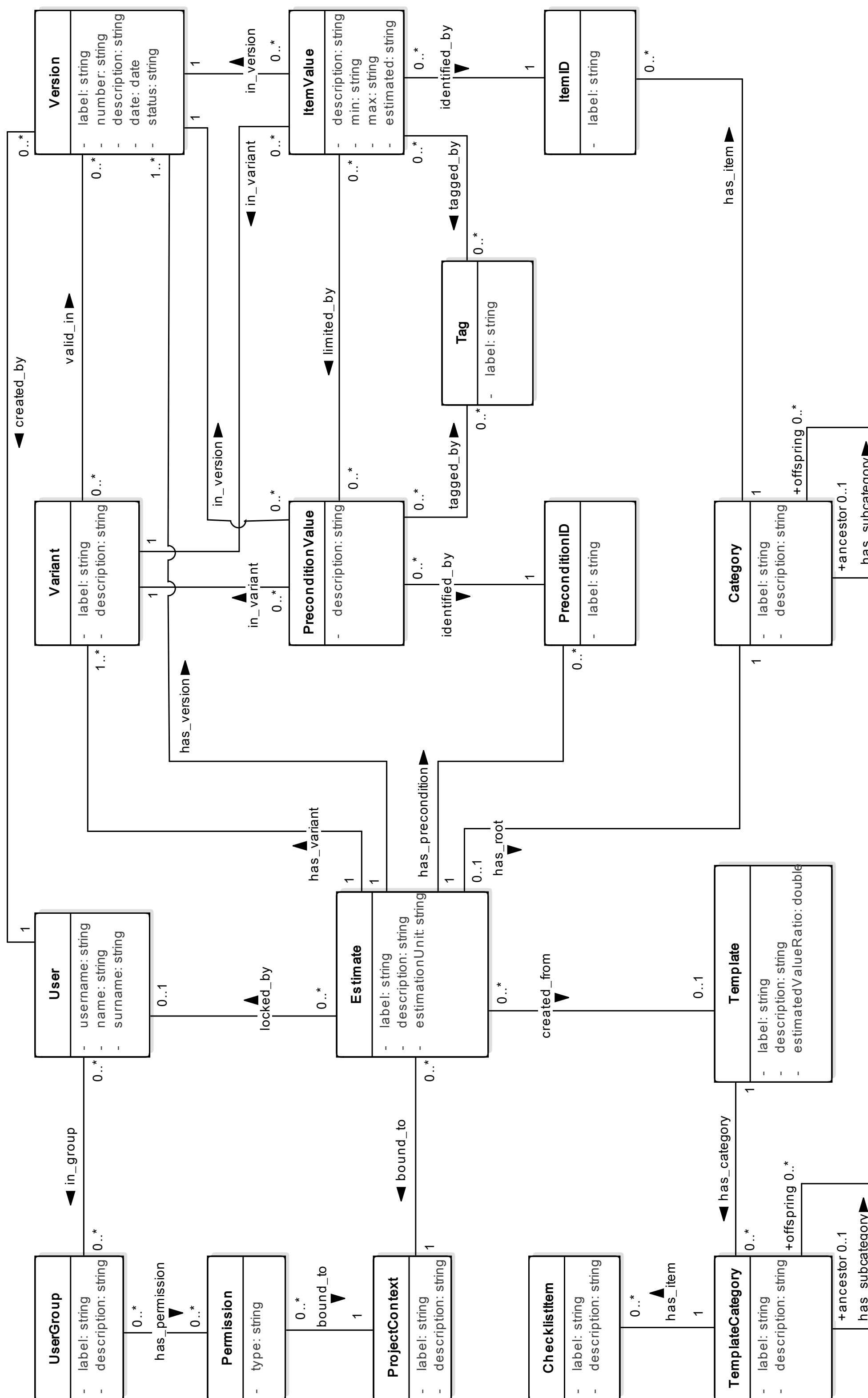
Obrázek 6.13: Diagram řízení a uzamykání verzí

Uživatel se zámkem libovolně upravuje rozpracovanou verzi. Po dokončení svých úprav může buď uložit verzi jako rozpracovanou nebo stabilní. V obou případech uživatel ztrácí zámek.

Po uložení verze jako stabilní nelze dále verzi upravovat.

V případě uložené rozpracované verze může kdokoli navázat další úpravou, čímž dotyčný opět získává zámek. Pokud však již není nutné verzi dále upravovat, lze ji přímo uložit jako stabilní.

V diagramu 6.13 není zachyceno mazání verze, které je možné kdykoli, pokud nikdo nemá zámek. V opačném případě může verzi smazat jen držitel zámku, čímž zámek ztrácí.



Obrázek 6.14: Datový model

6.2 Datový model

V následujících kapitolách popisuji jednotlivé entity schématu 6.14.

Entity mají často označení (*label*) a popis (*description*). Označení není jednoznačný identifikátor a není tak povinné — zastává hlavně funkci identifikátoru pro člověka.

V modelu respektuji standardní konvence pojmenovávání, tj. styl pojmenování entit – *CamelCase*, atributů — *camelCase* a relací — *snake_case*.

6.2.1 Odhad

Odhad (*Estimate*) je základní entitou celého modelu.

Má vazbu na šablonu — ze které je vytvořen (*created_from*), projektový kontext — do kterého spadá (*bound_to*), kořenovou kategorii (*has_category*), verze (*has_version*), varianty (*has_variant*), předpoklady (*has_precondition*) a případně uživatele, který může mít odhad uzamknutý pro úpravu (*locked_by*).

Odhad má navíc atribut jednotky pracnosti (*estimationUnit*), určující jednotku, ve které jsou položky odhadované.

6.2.2 Položky

Položky (*ItemValue*) mají dle požadavku P1 atributy hodnot minimálního (*min*), maximálního (*max*) a odhadovaného (*estimated*) odhadu.

Položka je identifikovaná (*identified_by*) jednoznačným identifikátorem (*ItemID*). Podle identifikátoru se položky řadí do kategorií (*has_item*). Položka náleží do varianty (*in_varitant*) a verze (*in_version*). Kromě toho může být omezena předpokladem (*limited_by*) a seskupována s jinými položkami na základě značek (*tagged_by*).

6.2.3 Předpoklad

Předpoklad (*PreconditionValue*), stejně jako položka, je identifikovaný (*identified_by*) jednoznačným identifikátorem (*PreconditionID*). Podle identifikátoru se předpoklady vážou k odhadu (*has_precondition*). Předpoklad náleží do varianty (*in_varitant*) a verze (*in_version*). Kromě toho může omezovat položky (*limited_by*) a být seskupován s jinými předpoklady na základě značek (*tagged_by*).

6.2.4 Struktura odhadu

Kategorie (*Category*) určují hierarchickou strukturu odhadu pomocí smyčkové relace (*has_subcategory*). Kořenová kategorie struktury je vázána na odhad (*has_root*) a listové kategorie obsahují položky (*has_item*).

6.2.5 Varianty

Varianty (*Variant*) zaobalují položky a předpoklady (*in_variant*), které charakterizují patřičnou variantu projektu. Varianta náleží odhadu (*has_variant*) a je vázána na verze (*valid_in*), ve kterých je s ní počítáno.

6.2.6 Verze

Verze (*Version*) obsahují položky, předpoklady (*in_version*) a varianty (*valid_in*), které jsou v dané verzi platné. Verze je vázána na odhad (*has_version*) a uživatele – autora (*created_by*). Autorem se stává uživatel, který verzi potvrdí a uloží jako stabilní.

Verze obsahuje atributy data poslední úpravy (*date*), stavu ve kterém se nachází (*status*) a číslo revize (*number*).

6.2.7 Šablona

Šablona (*Template*) určuje strukturu vytvářených odhadů pomocí kategorií šablony (*TemplateCategory*), které jsou na ni navázané (*has_category*). Vytvořené odhady podle dané šablony jsou s ní provázány (*created_from*).

Kategorie šablony obsahují (*has_item*) položky kontrolních seznamů (*ChecklitItem*) a díky smyčkové vazbě (*has_subcategory*) tvoří hierarchickou strukturu.

Šablona obsahuje koeficient (*estimatedValueRatio*) pro výpočet očekávané složky odhadu.

6.2.8 Řízení přístupu

Řízení přístupu, řešeného v kapitole 6.1.4, umožňuje následující struktura:

Uživatelé (*User*) jsou členy (*in_group*) uživatelských skupin (*UserGroup*). Projektovému kontextu náleží (*bound_to*) přístupová práva (*Permission*), kterými mohou disponovat (*has_permission*) uživatelské skupiny. Každý odhad je veden (*bound_to*) v rámci jednoho projektového kontextu.

Uživatel může být autorem verze (*created_by*) a může mít zámek (*locked_by*) na odhadu pro zápis. Dále má atributy uživatelského (*username*), křestního jména (*name*) a příjmení (*surname*).

Přístupová práva mají atribut typu (*type*).

Kapitola 7

Analýza technologií

V následující kapitole analyzuji a diskutuji technologie uvažované pro perzistenci dat a podporu importu/exportu.

7.1 Databáze

7.1.1 Relační

Relační databáze považuji za obecně známou tématiku a proto se o nich v práci nerozepisuji.

Pro realizaci projektu připadá v úvahu relační databáze Oracle, která je ve firmě široce používaná.

Další systémy, se kterými bude v budoucnu řešena integrace, používají právě Oracle databázi.

7.1.2 Grafové

V [4] pojednává o grafových databázích: Grafové databáze patří do rodiny databází zvaných NoSQL[5], které vznikly kvůli trendům vzrůstajícího objemu dat, jejich rostoucí propojeností a s tím související ztráty předvídatelné struktury. Těmto trendům se snaží NoSQL databáze přizpůsobit.

Data, jejichž vlastnosti neodpovídají tradičnímu relačnímu modelu, jsou grafové databáze schopné efektivně zpracovávat. Tomu napomáhá také znatelně odlišný dotazovací jazyk.

Jak napovídá název, grafové databáze jsou založeny na grafové[6] struktuře dat. To znamená, že každý objekt má přímé odkazy na své sousedy a díky

[5] NoSQL — zprvu myšlenka databází kompletně zproštěných znaků SQL pro nerelační datová úložiště. Později od myšlenky upouštěno směrem ke konceptu „NOSQL = Not Only SQL“, který naopak rozšiřuje principy tradičního SQL.

[6] Graf je struktura obsahující různě propojené objekty. Objekty se v grafu nazývají vrcholy či uzly a spoje mezi nimi jsou hrany. Každá hrana má 2 koncové vrcholy a pokud jde o orientovaný graf, mluvíme o počátečním a koncovém vrcholu.

tomu není nezbytné používání indexů. To je veliká výhoda oproti relačním databázím, protože odpadá používání výpočetně drahých operací JOIN, bez kterých se chod relačních databází obvykle neobejde.

Grafová databáze nevyžaduje rigidní strukturu datového modelu, což umožňuje volný rozvoj schématu takového modelu.

7.1.2.1 Přehled grafových databází

Grafových databází existuje mnoho, avšak často jsou vyvíjeny pro speciální účely a mohou tak mít různé vlastnosti jako například podle [5]:

- Sones GraphDB — struktura váženého grafu (hrany mají ohodnocení reálnými čísly),
- AllegroGraph — splňující standardy W3C[7] pro manipulaci s propojenými daty a sémantickým webem,
- FlockDB — databáze nepodporuje traverzování[8], využívaná Twitterem.

Databáze poskytující standardní vlastnosti:

- Neo4j — jedna z nejpopulárnějších ve své kategorii, umožňuje ukládání vrcholů, hran a jejich vlastností,
- InfiniteGraph — objektově orientovaná.

7.1.3 Zvolená databáze

Navržený datový model z kapitoly 6 charakterizuje několik známek:

- libovolná hierarchická struktura kategorií šablony a samotného odhadu,
- různé seskupování a členění položek do kategorií či variant,
- provázání položek na předpoklady ve smyslu jejich omezení.

Na základě těchto vlastností datový model vykazuje grafovou strukturu.

Z důvodu lepšího zachycení takového modelu volím grafovou databázi, a to konkrétně databázi Neo4j, která navíc oproti InfiniteGraph podporuje REST[9] přístup a poskytuje verzi zdarma bez omezení, které by měli v rámci projektu vliv.

[7] W3C — World Wide Web Consorcium

[8] Traverzování — systematický průchod grafem po hranách mezi uzly.

[9] REST — Representational state transfer, umožňuje zpracovávání dat na serveru pomocí HTTP volání.

7.2 Neo4j databáze

Neo4j je grafová databáze umožňující efektivní zpracování či dotazování rozsáhlých síťových datových struktur. Data ukládá jako uzly a relace, kde do obou umožňuje přidání vlastností.

Databáze Neo4j podporuje plně transakční přístup dle pravidel ACID[10]

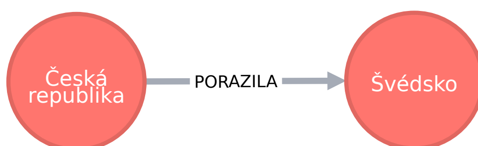
7.2.1 Relace v Neo4j

Podle [6]: Každý vztah, jinak nazýván relací, mezi objekty v Neo4j musí mít definovaný:

1. typ, díky kterému nabývá relace sémantického významu,

2. směr, který udává smysl a bez kterého by byl vztah nejednoznačný.

V grafu na obrázku 7.1 je jednoznačně definovaný vztah — „Česká republika porazila Švédsko.“ Všimněme si, že takto definovaný vztah určuje smysl i v opačném směru: „Švédsko bylo poraženo Českou republikou.“ Proto by bylo přidání další relace vyjadřující tento směr chybou.



Obrázek 7.1: Správně definovaná jednosměrná relace v Neo4j.

Neo4j uchovává odkaz na sousední objekt nezávisle na orientaci relace, díky čemu nás nijak neomezuje jednosměrnost takových relací.

U jednosměrného (orientovaného) vztahu jako je na obrázku 7.1 je správné řešení zřejmé. Problémy však mohou nastat u chápání obousměrných (neorientovaných) vztahů.

U obousměrných vztahů platí naprosto stejná pravidla jako u jednosměrných. To vede k jejich modelování stejným způsobem jako jednosměrné. Díky tomu, že Cypher[11] umožňuje dotazování se na hrany nezávisle na jejich orientaci, prakticky takové modelování neznamená problém.

Například nepochybně obousměrný vztah „je spolužák“ tedy namodelujeme jako na obrázku 7.2.

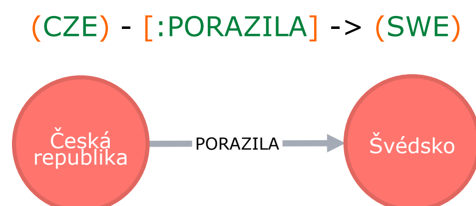
[10]ACID (podle ang. atomicity, consistency, isolation, durability) — množina vlastností databázového transakčního přístupu.

[11]Cypher — dotazovací jazyk pro databázi Neo4j.7.2.2

Obrázek 7.2: Obousměrná relace se v Neo4j správně modeluje jako jednosměrná.

7.2.2 Dotazovací jazyk Cypher

Úvod do Cypheru dle [7]: Neo4j databáze využívá vlastní deklarativní[12] dotazovací jazyk Cypher. Cypher je založen na intuitivním pasování vzorů uzlů a relací na graf. To umožňuje syntaxe inspirovaná klasickým SQL obohacená ASCII-artovým[13] vyjadřováním vzorů grafu, jako můžeme vidět na obrázku 7.3.



Obrázek 7.3: Ukázka ASCII-artového vzoru grafu

Uzly se v Cypheru vyjadřují obyčejnými závorkami. Nepovinně můžeme specifikovat jednotlivě přiřazení uzlu proměnné, jeho typ (ang. používaný label) a vlastnosti (ang. property):

```
( pojmenování:Typ {vlastnost:"hodnota"} )
```

Relace jsou vyjádřeny šipkou `-->` mezi dvěma uzly. Dovnitř šipky můžeme specifikovat podobně jako u uzlu další informace pomocí hranatých závorek:

```
-[ pojmenování:TYP {vlastnost:"hodnota"} ]->
```

Kromě toho můžeme požívat následující:

- relace různých typů `-[ :TYP1 | :TYP2 ]->`,
- informace o vyhovující cestě proměnné délky `-[ *..4 ]->`,
- vynechání informace o směru u obousměrných relací `-[:SOUSED]-`.

Již zmíněné vzory (ang. pattern) jsou sledy[14], které pasujeme na graf. V dotazu je můžeme zapisovat v celku, nebo oddělené čárkami. Vzhledem k tomu,

[12] Deklarativní paradigma — založeno na myšlence definování „co se má udělat“ a ne „jak se to má udělat“ používané imperativním paradigmatem.[8]
[13] ASCII-art — speciální složení znaků za účelem vizuálního výsledku jako celku.
[14] „Sled v grafu je posloupnost vrcholů taková, že mezi každými dvěma po sobě jdoucími uzly je hrana.“[9, s. 23]

že textové vyjádření vzoru je pouze jednorozměrné, potřebujeme vzory skládat v případě, kdy se dotazujeme na složitější strukturu, která nelze vyjádřit jedním vzorem. Napasované vzory můžeme podobně jako relace a uzly ukládat do proměnných. Ukázky použití vzorů na příkladech:

- vzor přítel přítele v celku,
  `(osoba)-[:PŘÍTEL]-(p)-[:PŘÍTEL]-(pp)`
- navigace stromem,
  `(kořen)<-[:RODIČ*]-(list:Kategorie)-[:OBSAHUJE]->(p:Položka)`
- použití grafových algoritmů s uložením do proměnné,
  `cesta = shortestPath( (osoba)-[:ZNÁ*]-(osoba2) )`
- skládání vzorů, např. herci, kteří hrají ve stejném divadle nebo filmu.
  ```
  (herec)-[:HRAJE]->(:Film)<-[:HRAJE]-(herec2),
  (herec)-[:HRAJE]->(:Divadlo)<-[:HRAJE]-(herec3)
  ```

Cypher je case-sensitive[15] pro názvy proměnných. U obou relací i uzlů, které jsou přiřazené proměnné, můžeme přistupovat ke vlastnostem pomocí tečkové notace. [7]

Cypher poskytuje množství klauzulí značně inspirovaných SQL, agregačních, matematických, či funkcí predikátové logiky a jiných. I přes to, že Neo4j nevyžaduje rigidní schéma, můžeme pomocí DDL[16], konkrétně indexů a omezení, schéma grafu definovat.

7.2.2.1 Klauzule jazyka Cypher

Jak už bylo zmíněno — Cypher obsahuje množství klauzulí.

Ke čtení slouží standardně `MATCH`, na který lze navázat, stejně jako v SQL, klauzulemi pro selekci (`WHERE`) a projekci (`RETURN`).

Uzly i relace se vytváří pomocí `CREATE`, odstraňují `DELETE` a upravují `SET`.

Cypher poskytuje speciální klauzuli `MERGE`, která svým způsobem umožňuje čtení i zápis.

Importovat data lze z CSV souborů pomocí `LOAD CSV` klauzule.

7.2.2.2 Schéma

V Neo4j můžeme definovat schéma pomocí indexů[17] a omezení[18].

[15] Case-sensitive — rozlišují se veliká a malá písmena.

[16] DDL — (ang. data definition language) prostředek pro definování struktury dat.

[17] Index v databázi je redundantní informace za účelem jejího efektivnějšího získání. Efektivnější čtení je však za cenu přidané režie způsobující další nároky na datový prostor a pomalejší zápis.[10]

[18] Databázová omezení (ang. constraints) pomáhají vynucovat integritu dat.

Cypher umožňuje vytvoření indexů pro vlastnosti všech uzlů daného typu. Databáze indexy dále sama spravuje a udržuje aktuální při jakékoli změně grafu.

V Neo4j mohou být aplikována omezení na uzly i relace. Existují omezení vynucující unikátnost či existenci vlastností.[11]

Omezení unikátnosti, existence, nebo neexistence se nastavuje na vlastnost v rozsahu jednotlivých typů uzlů. Omezení existence se může aplikovat také na relace.

Na jedné entitě lze mít více omezení, avšak fungují pouze nezávisle na sobě. V Neo4j tedy nelze aplikovat kombinované omezení, které by zamezilo například duplicitám kombinací několika vlastností uzlu.

7.2.2.3 Ukázky Cypher dotazů

V této sekci uvádím ukázky Cypher dotazů v porovnání s ekvivalentními dotazy SQL.

Příklad 1: Jména a ceny 10 nejdražších kol seřazených sestupně.

```
SELECT p.nazev, p.cena
FROM produkt as p
WHERE p.typ = "kolo"
ORDER BY p.cena DESC
LIMIT 10;
```

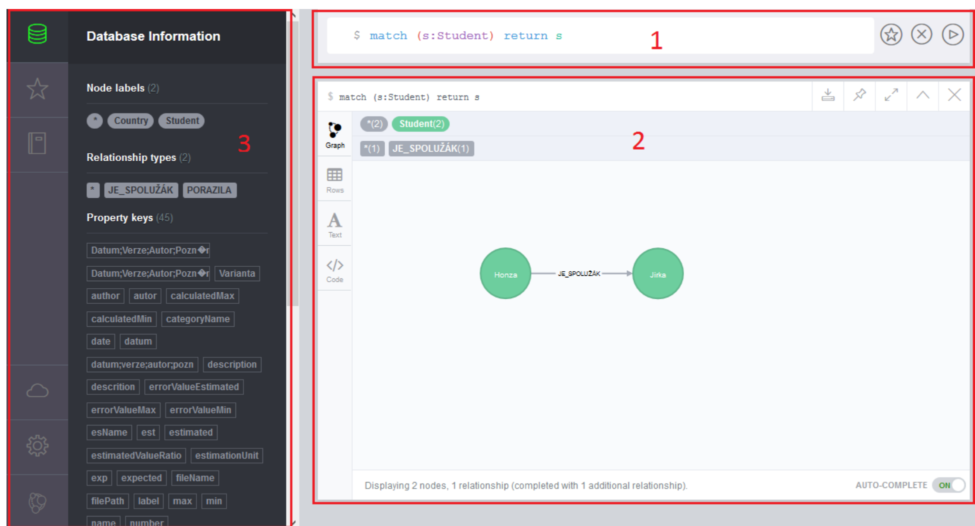
```
MATCH (p:Produkt)
RETURN p.nazev, p.cena
WHERE p.typ = "kolo"
ORDER BY p.cena DESC
LIMIT 10;
```

Příklad 2: Název všech produktů a celková cena, kterou za ně „Kolárna“ utratila.

```
SELECT p.nazev, coalesce(sum(d.cena * d.mnozstvi),0) AS Celkem
FROM produkt AS p
LEFT OUTER JOIN detail AS d ON (d.prod_ID = p.prod_ID)
LEFT OUTER JOIN objednavka AS o~ON (o.obj_ID = d.obj_ID)
LEFT OUTER JOIN zakaznik AS o~ON (z.zak_ID = o.zak_ID)
WHERE z.nazev = "Kolárna"
GROUP BY p.nazev
ORDER BY Celkem DESC;
```

```
MATCH (z:Zakaznik {nazev:"Kolárna"}), (p:Produkt)
OPTIONAL MATCH (p)<-[d:DETAIL]-(:Objednavka)<-[:OBJEDNAL]-(z)
RETURN p.nazev, toInt(sum(d.cena * d.mnozstvi)) AS celkem
ORDER BY celkem DESC;
```

Díky použití `OUTER JOIN` a `OPTIONAL MATCH` získáme i záznamy pro produkty, které nebyly zakoupeny.



Obrázek 7.4: Neo4j *browser* umožňuje práci s databází.

1. *editor*, do kterého lze zadat jakýkoli samostatný Cypher dotaz.
2. *plocha*, kde se zobrazují výsledky dotazů. Plocha umožňuje hned 4 různá zobrazení výsledku:
   a) vizuálně interaktivní zobrazení grafu,
   b) tabulkové zobrazení,
   c) textové zobrazení,
   d) kódové zobrazení odpovídající výsledku rozhraní Bolt[19]formátované jako JSON[20].
3. *postranní lišta* nabízí informace o databázi, výčet jejích objektů, oblíbené dotazy, navigaci pro dokumentaci a správu či nastavení databáze.

7.2.3 Webové uživatelské rozhraní Neo4j

Po nastartování Neo4j stroje je přístupné webové rozhraní pro práci s databází. Dříve dostupný nástroj *webadmin* byl s verzí 2.3.0 prohlášen za zastaralý a byl nahrazen novým nástrojem *browser*, který můžeme vidět na obrázku 7.4.[12]

[16]Bolt — odlehčený síťový klient-server protokol navržený pro databázové aplikace.
[17]JSON — odlehčený datový formát, zpracovatelný člověkem i strojem, určený pro výměnu dat.

7.3 Rozhraní pro komunikaci s databází

Vzhledem k tomu, že zvolená technologie pro jádro aplikace je Spring Boot fungující na programovacím jazyku Java, řeším zde možnosti přístupu k databázi pro jazyk Java.

Neo4j poskytuje proprietární binární „Bolt“ protokol pro komunikaci se serverem. Kromě toho můžeme se serverem komunikovat také pomocí standardního HTTP[21] protokolu. Zvláštní přístup nabízí také embedded[22] instance databáze.[13]

7.3.1 Možnosti přístupů

V následujících podkapitolách uvádím výčet možností přístupů podle [13].

7.3.1.1 Neo4j Java Driver

Neo4j Java Driver je minimalistická knihovna umožňující komunikaci s Neo4j databází použitím binárního protokolu. Zachovává standardní Java koncept pro komunikaci s databázemi.

Umožňuje spouštění parametrizovaných dotazů, které vrací výsledek obsahující namapované hodnoty na názvy proměnných.

7.3.1.2 Spring Data Neo4j

Spring Data Neo4j je způsob integrace přístupu k Neo4j databázi se Spring frameworkem[23].

Využívá mapování objektových grafů Neo4j-OGM(dále jen OGM), na základě čeho poskytuje následující vlastnosti:

- integrace se Spring Boot,
- mapování objektových grafů založené na anotacích,
- rozhraní obsahující perzistenční podporu s anotovanými dotazy,
- skenování metadat tříd,
- optimalizovanou správu čtení a přenosu dat,
- přístup pomocí binárního protokolu, HTTP i embedded databáze,
- správu životního cyklu perzistence.

[21]HTTP — protokol pro přenos strukturovaných dat.
[22]Embedded — jednoúčelový vestavěný systém.
[23]Framework — podpůrná softwarová struktura pro programování a vývoj softwaru.

7.3.1.3 Embedded Neo4j Java API

Embedded Neo4j Java API[24] umožňuje používání Neo4j databáze přímo v paměti podobně jako databázový systém HSQL či Derby.

Embedded databáze poskytuje stejné rozhraní jako standardní server. Díky těmto vlastnostem je embedded databáze vhodná pro testování.

7.3.1.4 Vlastní procedury a funkce

Jazyk Cypher umožňuje použití vlastních procedur a funkcí.

Po nasazení oanotované Java třídy do instalace Neo4j můžeme v dotazovacím jazyku přistupovat k vlastním procedurám a funkcím.

7.3.1.5 Rozšíření Neo4j serveru s REST API

Po rozšíření Neo4j serveru o REST API můžeme komunikovat se serverem čistě přes REST protokol.

Neo4j server se stará o zapouzdření funkcionality definovaných přístupových bodů, oproti kterým můžeme posílat HTTP požadavky.

7.3.1.6 Neo4j server s JDBC

Vzhledem k tomu, že Cypher je stejně jako SQL textový parametrizovatelný dotazovací jazyk, který vrací tabulkové výsledky, je možné využívat přístup k Neo4j přes JDBC driver, který je veřejně široce známý.

7.3.2 Neo4j-OGM

Zásadní význam pro komunikaci aplikace s databází má mapovací framework OGM dokumentovaný v [14].

OGM je rychlá knihovna pro mapování objektových grafů, optimalizovaná pro serverové instalace využívající Cypher. Zaměřuje se na zjednodušení vývoje softwaru využívajícím Neo4j databázi. Funguje na základě anotací jednoduchých POJO[25] doménových objektů.

Zaměření na výkon OGM podporuje následujícími metodami:

- perzistence proměnné hloubky, která umožňuje vyladění dotazů pro charakter grafu,
- optimalizované mapování objektů zamezující nadbytečné požadavky na databázi, zlepšující latenci a minimalizující zbytečnou spotřebu výkonu,
- definovatelný životní cyklus sezení7.3.2.2, který pomáhá dosáhnout rovnováhy mezi využitím paměti a efektivitou požadavků na server.

[24] API — aplikační programové rozhraní, poskytuje rozšiřující funkce, které může programátor používat.

[25] POJO — plain old Java object

7.3.2.1 Anotování entit

Základní anotací OGM je `@NodeEntity`, která označuje POJO jako entitu ukládanou pod uzlem v databázi.

Pole (ang. field) oanotované entity jsou standardně mapovány jako vlastnosti (`@Property`) uzlů. Pole, která jsou referencí (`@Relationship`) na další entity jsou propojené relací. Anotace `@Transient` zamezuje mapování pole.

Anotace `@Relationship` může definovat typ a orientaci relace. Vícenásobné relace OGM dokáže mapovat na standardní kolekce. Pro obousměrné vztahy slouží nedefinovaný směr relace, který zaručí unikátnost relace stejného typu mezi sousedními uzly. Problematiku vztahů řeším v kapitole 7.2.1. Relace stejného typu u různých polí entity jsou možné za podmínky, že typy koncových uzlů jsou rozdílné.

Bohaté relace — relace s vlastnostmi — mohou být mapovány na POJO anotované `@RelationshipEntity`. Takové POJO musí obsahovat pole vyjadřující počáteční a koncový uzel relace.

OGM vyžaduje u každé entity pole anotované jako grafový identifikátor (`@GraphId`). Na základě toho přiřazuje uzly v databázi k objektům. Objekt s nedefinovaným ID je při uložení do databáze vytvořen, zatímco objekt s definovaným ID pouze změněn.

Vlastnosti uzlů mohou být mapovány pro primitivní datové typy a typy převeditelné na `String` (datum, čísla, výčtový typ). Mapovány jsou také kolekce těchto typů.

Pro používání indexů existuje anotace `@Index`.

7.3.2.2 Perzistenční operace

Podle konfigurace se vytvoří sezení (ang. session), v rámci kterého můžeme spouštět dotazy a volat předdefinované operace uložení, načtení a mazání.

Pro uložení a načtení můžeme určit hloubku, do které má operace zasahovat. Operace načtení podporují řazení a stránkování.

Neo4j umožňuje transakční přístup, který můžeme v rámci sezení ovládat.

OGM umožňuje navázání událostí před či po operaci uložení a mazání.

7.3.2.3 OGM a Spring Data Neo4j

Díky integraci se Spring Data Neo4j můžeme definovat libovolný parametrizovaný Cypher dotaz jako metodu rozhraní perzistence.

OGM je schopné mapovat cokoli vracíme dotazem. Chceme-li například získat namapovanou část grafu, musí být ve výsledku dotazu všechny uzly a relace hledaného grafu. Toho docílíme například takto:

```
@Query("MATCH cesta=(koren)<-[:RODIC*]-(list:Kategorie) " +
    "WHERE id(koren) = {id} RETURN nodes(cesta), rels(cesta)")
Kategorie getStrom(@Param("id") int idKorenu);
```

7.4 Různé poznatky

7.4.1 Náročnost technologie OGM

I přes to, že technologie OGM je optimalizovaná na výkon, narážím na zvláštní chování a problém s efektivitou.

Mapování pomocí OGM je silně ovlivňováno hloubkou operací.

Na příkladu grafu s přibližně 350 uzly, 750 hranami a poloměrem 3 zkoumám operaci načítání. Do tabulky zaznamenávám závislost paměťové a časové náročnosti na hloubce hledání. Omezení paměti virtuálního stroje je 2200 MB.

Driver	Hloubka	Využitá paměť	Čas
HTTP	3	400 MB	okamžitě
	4	1800 MB	35 s
	5	2200 MB	inf
Embedded	3	zanedbatelná	okamžitě
	4	900 MB	5 s
	5	2100 MB	35 s
	6	2200 MB	inf

Každý experiment, který doběhl, dosáhl stejného výsledku — namapování celého grafu, avšak s razantně rozdílnými nároky.

Z experimentu zjišťuji, že není vhodné načítat obecně široký graf. Naopak je nutné striktní řízení načítání grafu z databáze.

7.4.2 Výběr kolekce pro mapování

Při průzkumu technologie narážím na otázku, zdali je lepší využít kolekci *Set*, nebo *List* pro mapování relací.

Na základě otázky zkouším experiment s měřením času mapování na obě kolekce.

Experiment čítá 42 000 uzlů a podobné množství relací. Zkouším uložení takových dat do databáze a jejich následné načtení.

Výsledky experimentu ukazují, že se efektivita mapováni na obě kolekce v průměru rovná. Kromě toho zjišťuji, že zápis a čtení z databáze se náročností velice liší. Zatímco jedna iterace zápisu souboru dat do databáze trvá přibližně 6 minut, jejich zpětné načtení okolo 80 sekund. Zajímavé je, že při zápisu vykazuje výkon většinu času databáze, zatímco u načítání naopak OGM.

Kromě otázky efektivity také řeším otázku konzistence načítání dat.

Mapování na kolekci *Set* nezaručuje žádné uspořádání. Pro opakovaná načítání mohou být výsledky různě zpřeházené. Oproti tomu mapování na kolekci *List* zaručuje konzistentní pořadí načtených objektů ve výsledku.

Na základě tohoto průzkumu usuzuji, že z kolekcí *List* a *Set* je pro mapování vhodnější použít právě kolekci *List*.

7.5 Rozhraní pro import/export

Import dat přímo souvisí se zpracováním excelových souborů.

K tomu existuje pro jazyk Java více technologií. Mezi nejvýznamnější patří *JExcelApi* a *Apache POI*.

JExcelApi je cílené přímo na zpracování excelových souborů, avšak dnes je již zastaralé a nepodporuje formáty novější než Excel 2000.[15]

Apache POI je technologie univerzálnější a umožňuje tak zpracování celého spektra produktů Microsoft Office. Oproti *JExcelApi* podporuje i novější formáty Excel 2007, které se ve firmě běžně používají.[16]

Z uvedených důvodů volím právě *Apache POI* pro podporu funkce importu.

Stěžejní význam exportu je obyčejný textový výstup dat, pro což není potřeba žádné nadstandardní technologie. Pro příležitostný export dat do excelového souboru je použito stejně jako pro import *Apache POI*.

7.5.1 Apache POI

Dle [17]: Technologie Apache POI poskytuje, krom jiných, rozhraní HSSF a XSSF pro zpracování tabulek MS Excel.

Formát Excel 2003 a dřívější (přípona .xls) je schopné zpracovat rozhraní HSSF, zatímco XSSF pracuje se soubory Excel 2007 a pozdějšími (přípony .xlsx a .xlsm).

Obě verze používají svoji reprezentaci rozhraní *Workbook*, *Sheet*, *Row* a *Cell*. Pomocí těch můžeme jednotlivě přistupovat k listům, řádkům a buňkám tabulek MS Excel.

Z jednotlivých buněk můžeme zpracovávat data různých typů:

- text,
- numerické hodnoty,
- data,
- formule,
- formulemi určené hodnoty,
- logické hodnoty boolean,
- hodnota chybné buňky.

Dle kapitoly 3.3 potřebuji pro import načítat právě text, čísla, data a hodnoty určené formulemi.

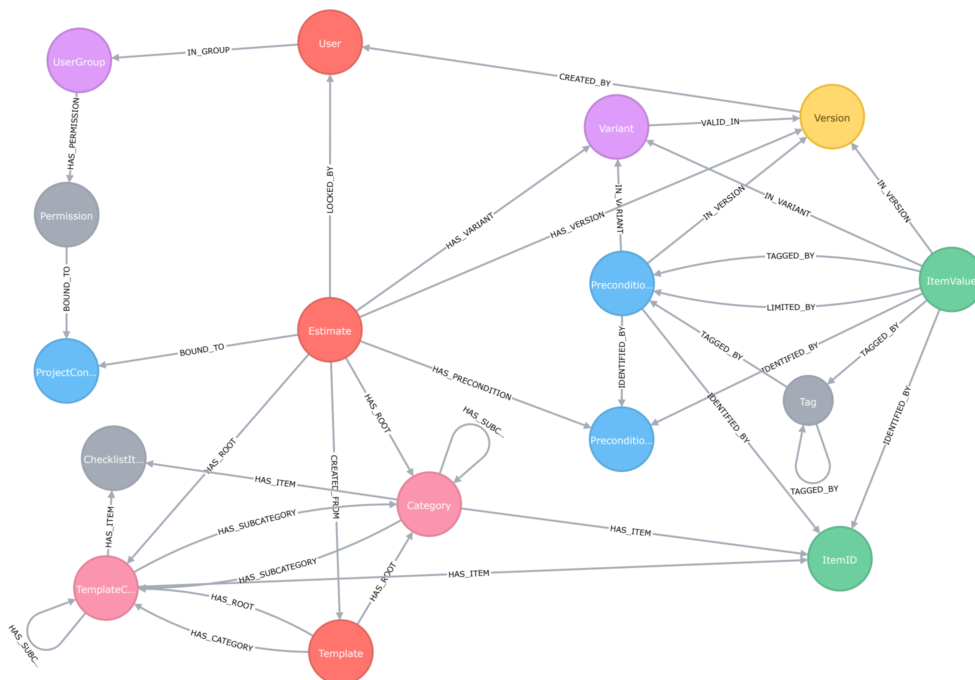
Kapitola **8**

Realizace Databáze

8.1 Schéma

Díky výběru grafové databáze Neo4j ukládaná data tvoří schéma odpovídající datovému modelu 6.14.

Na obrázku 8.1 je schéma dat v databázi vyvolané: `CALL db.schema()`. Datovému modelu odpovídá věrně až na několik relací, které, jak jsem vypozoroval, jsou způsobené chybou *browseru*. Ten zobrazuje neexistující relace na základě jejich stejného pojmenování s jinými existujícími.



Obrázek 8.1: Schéma databáze zobrazené Neo4j *browserem*7.2.3.

8.2 Instalace a správa

Neo4j verze 3.1.1 nabízí pro operační systém Windows dvě možné instalace:

1. aplikaci instalovanou standardním instalátorem, která umožňuje spouštění instance databáze a přístup pomocí *cypher-shellu*[26],
2. ZIP archiv, obsahující produkt, který umožňuje spouštět databázi jako konzolovou aplikaci, nainstalovat jako službu systému, přístup pomocí *cypher-shellu* a staršího *neo4j-shellu*.

Z důvodu, že *cypher-shell* mi na vývojové stanici podle [18] nedokáže spouštět Cypher skripty, volím podle [19] instalaci Neo4j jako služby systému Windows ze ZIP archivu dle [20]. Tato instalace poskytuje pomocí nástroje *neo4j-shell* funkční spouštění Cypher skriptů.

8.3 Inicializace

Pro vývojové účely je nutné mít data konzistentně a opakovatelně použitelná pro testování. Kvůli tomu potřebuji opakovatelnou inicializaci databáze.

Toho dosáhnu pomocí inicializačních skriptů, které jsem schopen díky nástroji *neo4j-shell* spouštět.

Cypher dokáže číst data z CSV souborů pomocí `LOAD CSV` 7.2.2.1. Toho využívám a zpracovávám odhady ze souboru MS Excel tak, aby mi zbyla jen data ve formátu CSV.

Spuštěním vzniklých skriptů přes *neo4j-shell* jsem schopný do databáze načíst požadovaná data.

Pro Unit testování však tento postup nestačí, protože není možné při Unit testování spouštět *neo4j-shell*. Aby inicializace mohla proběhnout pro každé spuštění testů, je nutné programově zpracovat skripty a spustit je pomocí Spring Data Neo4j zmiňovaného v kapitole 7.3.1.2. Pomocí Spring Data Neo4j je možné spouštět pouze jednotlivé dotazy. Kvůli tomu zavádím do skriptů oddělovače, podle kterých je možné programově skript rozdělit na jednotlivé dotazy.

Takovým způsobem už jsem schopný při každém spouštění testů inicializovat databázi.

Z důvodu, že manuální spouštění *neo4j-shellu* je nepraktické, vytvářím dávkový soubor `db_init.bat`, který spouští jednotlivé skripty na základě nastavených cest k databázi a daným skriptům.

Ukázka dotazu inicializačního skriptu 8.2.

[26] Cypher-shell — nástroj pro příkazovou řádku umožňující připojení k databázi a spouštění Cypher dotazů.[18]

```
//RETURN "Version";
//Version
MATCH (tmp:Temp {name:"tmp1"})-[:TMP_REL]->(es:Estimate)
MATCH (tmpVer:Temp {name:"tmpVer"})
LOAD CSV WITH HEADERS
FROM tmp.filePath + tmpVer.fileName
AS row FIELDTERMINATOR ";"
WITH split(row.datum,".") AS cd, row, tmp, es
CREATE (n:Version)
SET
  n.label = row.verze,
  n.number = 1,
  n.date = cd[2]+"-"+cd[1]+"-"+cd[0]+"T00:00:00.000Z",
  n.description = row.pozn,
  n.status = "STABLE"
CREATE (es)-[:HAS_VERSION]->(n),
 (tmp)-[:TMP_REL]->(n);
//_END
```

Obrázek 8.2: Ukázka části inicializačního skriptu: dotaz na základě nastavení načteného z dočasných uzlů načte z CSV souboru data o jednotlivých verzích a vytvoří pro ně uzly. //_END na posledním řádku je oddělovač jednotlivých dotazů.

Kapitola 9

Modelová vrstva aplikace

Jak jsem naznačil v kapitole 7.3, pro každou entitu modelu implementuji na straně aplikace třídu. Každou z těchto tříd anotuji odpovídajícím způsobem pro kompletní mapování grafu na model.

Z důvodu, že relace jsou realizovány referencí na sousední entitu po obou stranách, implementuji koncept „navazování“ (tzv. *binderů*), pro zamezení vzniku nekonzistencí v modelu. Tyto *bindery* nahrazují obyčejné *setry*.

Pro ukázku uvádím část třídy kategorie (*Category*) s definicemi jednotlivých polí na figuře 9.1 a *bindery* na figuře 9.2. Smyčková relace kategorie by mohla být standardním způsobem mapována nejednoznačně, proto je nutné anotovat i setry zajišťující mapování této relace, jako uvádím na figuře 9.3.

9.1 Rozhraní pro komunikaci s databází

Komunikaci s databází zajišťuje technologie Spring Data Neo4j diskutovaná v kapitole 7.3.1.2.

Pro každou třídu modelu, pro kterou potřebujeme provádět operaci čtení či zápisu do databáze, vytvářím rozhraní rozšiřující *GraphRepository<>*. Takové rozhraní standardně poskytuje operace *save* a *delete* pro zápis do databáze a operace čtení na několik způsobů: *findOne* hledající podle grafového identifikátoru, *findAll* hledající všechny entity odpovídajícího typu, *findBy-XXX*, kde *XXX* je predikát skládající se z vlastností uzlu, podle kterého hledáme entitu.

Kromě předdefinovaných operací můžeme rozhraní rozšířit o vlastní dotazy, kterým můžeme předávat parametry. Aby OGM mohlo namapovat celý výsledný graf, musíme vracet všechny jeho uzly i hrany. Toho docílíme funkcemi *nodes()* a *rels()*.

Každá operace či dotaz je přístupná jako metoda rozhraní. Díky tomu tato rozhraní tvoří samotnou perzistenční vrstvu aplikace.

Pro ukázku uvádím rozhraní pro třídu odhadu na figuře 9.4.

```
@NodeEntity
public class Category {

    @GraphId private Long id;

    private String label;

    private String description;

    @Relationship(type = "HAS_ROOT",
        direction = Relationship.INCOMING)
    private Estimate estimate;

    @Relationship(type = "HAS_SUBCATEGORY",
        direction = Relationship.INCOMING)
    private Category parent;

    @Relationship(type = "HAS_SUBCATEGORY",
        direction = Relationship.OUTGOING)
    private List<Category> subcategories
        = new LinkedList<>();

    @Relationship(type = "HAS_ITEM")
    private List<ItemID> itemIDS
        = new LinkedList<>();
```

Obrázek 9.1: Ukázka části třídy kategorie: definice polí — jednotlivé pole a anotace odpovídají entitě *Category* modelu na obrázu 6.14. Navíc třída obsahuje pole grafového identifikátoru *id*, které je nutné pro funkčnost mapování.

```
    public void bindWithParent(Category parent){
        this.setParent(parent);
        parent.getSubcategories().add(this);
    }
    public void bindWithSubcategories
        (List<Category> categories) {
        this.setSubcategories(categories);
        categories.forEach(i -> i.setParent(this));
    }
```

Obrázek 9.2: Ukázka části třídy kategorie: bindery — nahrazují funkci obyčejných *setrů* tak, že navážou obě strany relace.

```
@Relationship(type = "HAS_SUBCATEGORY",
    direction = Relationship.INCOMING)
public void setParent(Category parent) {
    this.parent = parent;
}

@Relationship(type = "HAS_SUBCATEGORY",
    direction = Relationship.OUTGOING)
public void setSubcategories
    (List<Category> subcategories) {
    this.subcategories = subcategories;
}
```

Obrázek 9.3: Ukázka části třídy kategorie: setry — kvůli smyčkové relaci u kategorií je nutné anotovat i setry, aby OGM jednoznačně namapovalo relaci.

```
public interface EstimateRepository
    extends GraphRepository<Estimate> {

    Estimate findByLabel(String label, @Depth int depth);

    @Query("MATCH␣p=(es:Estimate␣{label:{label}})" +
        "-[␣]->␣(ver:Version␣{label:{version}})" +
        "<-[␣]-␣(:ItemValue)␣-[␣]->" +
        "(:ItemID)␣<-[*]-␣(es)␣" +
        "RETURN␣nodes(p),rels(p),es")
    Estimate getEstimateByLabelAndVersion(
        @Param("label") String label,
        @Param("version") String version);
```

Obrázek 9.4: Ukázka části rozhraní perzistence — rozhraní perzistence pro třídu *Estimate*. Metoda *findByLabel* vrací odhad a na něj namapovaný graf do hloubky uvedené parametrem *depth*. Metoda *getEstimateByLabelAndVersion* vrací odhad označený parametrem *label* s namapovanou verzí podle druhého parametru *version*, položkami patřícími do verze rozčleněnými do struktury kategorií.

9.2 Rozhraní mezi servisní a perzistenční vrstvou

Uvedené rozhraní v předešlé kapitole 9.1 využívá servisní vrstva aplikace.

Servisní vrstva aplikace se stará o samotnou logiku operací jako vytvoření nového odhadu či verze, uložení, smazání verze, import a podobně. Tyto operace typicky vyžadují přístup k datovému úložišti, kterou umožňuje právě perzistenční vrstva zpřístupněním definovaných metod.

Logiku servisní vrstvy využívá vyšší vrstva — prezentační, kterou řeší v rámci své bakalářské práce Juraj Polačok [3].

9.3 Konfigurace

Aby technologie OGM mohla fungovat, vyžaduje konfiguraci pro správu sezení a transakcí.

Proto nastavuji konfiguraci pomocí souboru `ogm.properties`. Konkrétně v souboru nastavuji adresu databáze, driver a přístupová pověření, která se používají pro komunikaci s nastavenou databází.

Na výběr máme mezi HTTP, Bolt a embedded drivery diskutovanými v kapitole 7.3. Pro běh aplikace vybírám standardní HTTP driver.

Soubor s nastavením využívá komponenta tvořící sezení implementovaná jako tovární metoda *sessionFactory* v hlavní třídě aplikace (*EstimateApplication*).

Podobně je řešena i správa transakcí — metoda *transactionManager*.

9.4 Zpracování šablony

Šablona s datovým modelem jsou pro tvorbu odhadu neoddělitelné. Každý odhad z šablony získává informace o způsobu výpočtu očekávané složky odhadu. Příležitostně se z šablony získávají kontrolní seznamy k předpokladům a kategoriím.

Nejdůležitější je však samotná struktura kategorií, kterou určuje také šablona. Jakýkoli vznik odhadu (vytvoření nového, import) je provázen převzetím struktury šablony.

Zkopírování struktury obstarává tovární metoda třídy *CategoryFactory*, která vrací kořenovou kategorii zkopírované struktury.

Samotné kopírování je realizováno při průchodu struktury kategorií šablony do šířky. Při každém nalezení kategorie šablony se vytvoří odpovídající kategorie odhadu a prováže se s nadřazenou kategorií.

Kapitola 10

Importer

V systému zavádím komponentu tzv. importeru sloužícího k importování dat z excelových souborů dle případu užití PU7 a požadavku P11.

Importer očekává na vstupu definovanou šablonu, projektový kontext a soubor, ze kterého načte všechna ostatní data. K načítání využívá technologii Apache POI, diskutovanou v kapitole 7.5, díky které importer podporuje formáty souborů Excel 2007 a novější (XLSX, XLSM).

Importer vytvoří odhad, který prováže s šablonou a kontextem. Podle šablony se okopíruje struktura odhadu. Ze souboru jsou načteny verze, varianty, položky a předpoklady. Protože jednotlivé verze v Excelu od sebe nelze rozlišit, importer přiřadí všechny položky a předpoklady do každé načtené verze a patřičné varianty. Položky jsou členěny do kategorií na základě listu, na kterém se v souboru nachází.

Importer využívá komponentu *EstimateWorkbook*, která rozšiřuje standardní Apache POI *XSSFWorkbook*, jako rozhraní pro načítání dat z excelových souborů.

EstimateWorkbook umožňuje modulární přístup k datům, díky čemu lze jednotlivé části jako varianty, verze, položky a jiné načíst nezávisle na sobě. Tyto části dat importer poskládá dohromady a vytvoří celý odhad reprezentovaný datovým modelem aplikace.

10.1 Struktura souboru

Pro úspěšný import musí soubor na vstupu splňovat přesně definovaný formát.

Ten určuje předpoklady existence určitých listů s daty na daných souřadnicích nebo v sekcích, jejichž rozsah je jednoznačně určený.

Pokud soubor definovaný formát nesplňuje, import se neprovede a je vyhozena odpovídající chyba, výše zpracovaná odpovědnou vrstvou aplikace.

Při splnění definovaného formátu je zaručeno úspěšné naimportování následujících dat:

- popisu odhadu,
- označení variant,
- informace o verzích: datum, označení a popis,
- informace o předpokladech: popis a varianty, do které patří,
- informace o položkách: popis, minimální, maximální a odhadovaná složka a varianta či skupina, do které patří.

Ukázku zpracování dat z *workbooku* uvádím na figuře 10.1:

```
public Estimate getEstimate() {
    Estimate estimate = new Estimate();
    estimate.setEstimationUnit(EstimationUnit.MAN_HOUR);

    DateFormat dateFormat =
        new SimpleDateFormat("yyyyMMdd_HHmmss");
    Date date = new Date();
    estimate.setLabel("EST_" + dateFormat.format(date));

    Sheet sheet = getSheet(Constants.SheetName.HEADER);
    if (sheet == null)
        throw new UnsupportedWorkbookFormatException(
            "Soubor neobsahuje list: ",
            Constants.SheetName.HEADER);

    Row row = sheet.getRow(
        Constants.Estimate.DESCRIPTION_ROW);
    if (row!=null) {
        Cell cell = row.getCell(
            Constants.Estimate.DESCRIPTION_COLUMN);

        String description = getTextValueFromCell(cell);
        estimate.setDescription(description);
    }
    return estimate;
}
```

Obrázek 10.1: Ukázka části rozhraní importu — metoda *getEstimate*, která vytvoří novou instanci odhadu, nastaví jeho označení skládajícího se z aktuálního data a času. V *EstimateWorkbooku* je přistoupeno k listu představujícím hlavičku, na kterém je z určité buňky načten popis odhadu.

Kapitola 11

Realizace logiky servisní vrstvy

11.1 Vytvoření odhadu

Při vytvoření nového odhadu se nastaví zadané informace, vytvoří se struktura prázdného odhadu a zformují kategorie podle šablony.

Samotnému odhadu se nastaví zadané označení a popis. Odhad se naváže na projektový kontext a šablonu, podle které se vzápětí zkopíruje struktura kategorií. Dále se na odhad váží varianty zadané uživatelem. Pro odhad se vytvoří iniciální verze, která získává stav *rozpracovaná*. Odhad se uzamkne pro zápis uživatele, který ho vytvořil. Nakonec se vytvořený odhad uloží do databáze.

11.2 Správa verzí odhadu

11.2.1 Vytvoření nové verze

Vytvoření nové verze odhadu je podmíněno tím, že pro odhad neexistuje jiná *rozpracovaná* verze.

Při vytvoření verze dostane uživatel zámek na odhad. Vytvoření nové verze navazuje na původní *stabilní* verzi jako její *rozpracovaná* kopie.

Pro zkopírování původní verze využívám vlastnosti mapování OGM, které v databázi vytváří uzly pro objekty, které nedokáže spárovat na základě identifikátoru, viz. kapitola anotace 7.3.2.1. Díky tomu nemusím fyzicky kopírovat data, ale stačí vynulovat identifikátory původní verze, položek a předpokladů.

Nová verze se uloží do databáze, čímž teprve fyzicky vzniká.

11.2.2 Přístup k úpravě verze

Povolení úpravy verze je podmíněno jejím statusem, který musí být *rozpracovaná*, a volným zámkem odhadu.

Povolením úpravy verze získává uživatel zámek, což je ihned uloženo do databáze.

11.2.3 Uložení verze

Uložením verze uživatel ztrácí zámek. Podle situace existují dvě možnosti:

1. uložení verze jako *rozpracované* pro pozdější úpravu, podmíněné pouze zámkem uživatele,
2. uložení verze jako *stabilní* pro zamezení dalších úprav verze, podmíněné validitou položek, stavem verze jako *rozpracovaná* a volným zámkem.

11.2.4 Smazání verze

Při smazání verze dochází k odstranění položek, předpokladů a uzlu verze z databáze. Je podmíněno zámkem uživatele a stavem verze, která musí být *rozpracovaná*.

11.3 Import dat

Import dat je v aplikaci napojený na tzv. uploader, který nahrává soubory na server jako datový proud.

Tímto způsobem nahraný soubor zpracovává komponenta *Importer*, řešená v kapitole 10. Pro úspěšný import musí být data validní, což je ověřeno komponentou *kalkulátoru*.

11.4 Export výsledků odhadu

Pro naplnění případu užití PU8 podloženého požadavkem P12 implementuji komponentu tzv. *exporteru*, využívaného servisní vrstvou.

Výsledky pro export jsou nejdříve vypočítané komponentou *kalkulátoru*, kterou řeší ve své práci Juraj Polačok[3].

Vypočítané výsledky jsou v podobě odhadu a verze předané *exporteru*. Na základě výsledků *exporter* vybuduje textový řetězec obsahující předpoklady a pracnost reprezentovanou složkami minimálního, maximálního a očekávaného odhadu. Vzniklý řetězec *exporter* vrací jako návratovou hodnotu a následně je předán prezentační vrstvě aplikace.

11.5 Řízení přístupu k odhadu

Pro umožnění řízení přístupu k odhadu dle případu užití PU3 a požadavku P10 implementuji získávání oprávnění uživatele k operacím s odhadem.

Množina oprávnění je získána na základě uživatelova členství ve skupinách, které disponují právy k projektu, pod který spadá cílový odhad.

Na základě této množiny oprávnění je možné dále řídit přístup k jednotlivým operacím.

Kapitola 12

Testování

12.1 Rozhraní pro komunikaci s databází

Na rozdíl od běhu aplikace, u testování rozhraní pro komunikaci s databází je vhodné použít driver pro použití embedded databáze.

Testovací framework JUnit umožňuje vlastní konfiguraci pro spouštění testů. Nastavuji proto OGM konfiguraci pro testování na driver pro embedded databázi.

Jak už jsem zmínil v kapitole 8.3, embedded databáze se inicializuje při každém spouštění testů pomocí inicializačních skriptů.

Na základě inicializace jednotkové testy ověřují správnou funkčnost mapování grafu na model aplikace. Testování ověřuje jak operace čtení, tak i zápisu. Testy ověřují mapování kontrolou hodnot atributů a struktury namapovaného grafu — zdali namapované objekty obsahují odpovídající počet a typ referencí.

Jednotlivé testy testují například:

1. načtení širokého grafu do určité hloubky od hledaného uzlu odhadu,
2. načtení všech uzlů odhadů,
3. načtení odhadu pomocí obecného dotazu,
4. načtení určité verze odhadu obsahující položky,
5. vytvoření a následného smazání uzlu odhadu s jeho variantami,
6. konverze data při uložení a zpětného načtení.

Při vývoji často nastává situace, kdy potřebuji spustit test oproti reálné databázi. V takových situacích nastavuji konfiguraci na HTTP driver a adresu reálné databáze. Oproti reálné databázi nechci provádět inicializaci a tak vytvářím speciální soubor experimentálních testů, které jsou ignorovány při pravidelném spouštění jednotkových testů.

12.2 Importer

Pro zajištění správné funkce importu implementuji jednotkové testy komponenty importeru.

Tyto testy ověřují správné fungování načítání jednotlivých modulů dat, stejně jako kompletní import celého odhadu.

Aby bylo na čem testovat funkčnost importeru, vytvářím soubor testovacích dat, které obsahují různé validní i nevalidní formáty souborů tak, aby pokryly celou funkčnost importeru.

Testy jednotlivých modulů ověřují v pozitivním případě hodnoty načtených entit. V negativním případě vyhození relevantní výjimky.

Testy celého importu ověřují strukturu naimportovaného odhadu — převzatá struktura kategorií ze šablony, počty verzí, variant, předpokladů, položek a jejich rozčlenění do kategorií a přiřazení k variantám.

12.3 Logika servisní vrstvy

Jednotkové testy servisní logiky implementuji pro zajištění správné funkčnosti dle případů užití z kapitoly 5.

Testy servisní vrstvy využívají embedded databázi inicializovanou skripty řešenými v kapitole 8.3.

Implementuji pozitivní a negativní testy pro každou operaci:
vytvoření (11.2.1), úpravu (11.2.2), uložení (11.2.3) a smazání (11.2.4) verze.

Pozitivními testy vždy ověřuji správné uzamykání odhadu, stav verzí a persistenci, zatímco negativními testy ověřuji vyhození výjimek odpovídajících chybové situaci.

Kapitola 13

Aktuální stav

13.1 Struktura systému

Z důvodu, že mnou realizované části jsou přímou součástí aplikace stejně jako části, které realizuje Juraj Polačok, je implementace nás obou ve sdíleném repozitáři.

Mnou realizované části aplikace jsou konkrétně:

- datový model,
- perzistenční vrstva,
- servisní logika související s perzistencí (vytvoření nového odhadu a verze, uložení či smazání verze, import odhadu, systém uzamykání) a exportem výsledků,
- komponenta importeru,
- komponenta exporteru.

Juraj Polačok ve své práci [3] řeší realizaci částí potřebných pro funkčnost prezentační vrstvy aplikace.

13.2 Dokumentace

Datový model a řízení verzování je dokumentováno pomocí schémat a diagramů v projektu nástroje Enterprise Architect.

Dokumentace rozhraní importeru obsahují specifikaci přijímaného formátu souborů MS Excel je uložena ve sdíleném firemním úložišti.

Zdrojový kód aplikace je dokumentován systémem Javadoc[27] a komentářovými poznámkami kritických částí kódu.

[27] Javadoc — nástroj automaticky generující dokumentaci na základě speciálních komentářů v kódu.

13.3 Vyhlídky do budoucnosti

Datový model by v budoucnu mohl být rozšířen o variabilní část struktury odhadu, kterou by bylo možno měnit nezávisle na šabloně.

Dále by mohly být verzovány i další entity, např. šablona s jejími kategoriemi a kontrolními seznamy.

V databázi by mohly být využity indexy a omezení pro zajištění lepšího výkonu a konzistence dat.

Aplikace bude rozšířena o logování servisní vrstvy.

Dále by mohla být rozšířena o možnost správy šablon a export odhadů do dalších formátů.

Systém by mohl být integrován s dalšími firemními systémy, například se systémem vykazování.

Závěr

V práci jsem se zabýval návrhem datového modelu a implementací částí systému pro podporu tvorby odhadů pracnosti, čímž jsem splnil cíle této bakalářské práce.

Zpočátku jsem se seznámil se samotnou problematikou a různými metodikami odhadování.

Následně jsem analyzoval dosavadní existující řešení, požadavky a případy užití nového systému. Na základě analýzy jsem poté řešil samotný návrh modelu. Dále jsem analyzoval dostupné technologie vyhovující požadavkům a umožňující realizaci navrženého modelu. Za pomoci vybraných technologií jsem implementoval modelovou vrstvu webové aplikace a vytvořil schéma databáze pro uchování odhadů. V aplikaci jsem implementoval komponentu importu odhadů reprezentovaných tabulkami MS Excel a funkci exportu výsledků do textové podoby.

Navržený model umožňuje systému vyhovovat cíleným požadavkům a případům užití. Odhad je možné libovolně hierarchicky strukturovat a položky lze členit do různých variant. Model podporuje verzování odhadu v průběhu jeho tvorby a umožňuje seskupování položek na principu značkování.

Z kandidátních technologií jsem pro uchování dat zvolil grafovou databázi Neo4j. S databází aplikace komunikuje pomocí technologie Spring Data Neo4j využívající objektově grafové mapování Neo4j-OGM. Aplikace umožňuje import odhadů z formátu tabulek MS Excel a export výsledků odhadu do textové podoby, která je použitelná pro jejich prezentaci mimo systém.

Literatura

[1] McConnell, S.: *Odhadování softwarových projektů.* Brno: Computer Press, vyd. 1. vydání, 2006, ISBN 8025112403.

[2] Pitaš, J.: *Národní standard kompetencí projektového řízení verze 3.2.* Brno: Společnost pro projektové řízení, vyd. 3., dopl. a aktualiz. vydání, 2012, ISBN 9788026023258.

[3] Polačok, J.: *Aplikácia pre podporu tvorby odhadov pracnosti softwarových projektov.* Bakalářská práce, Fakulta informačních technologií, České vysoké učení technické v Praze, Katedra softwarového inženýrství, Praha, 2017, vedoucí bakalářské práce Michal Petřík.

[4] Ramba, J.: Grafová terminologie a dostupné technologie. *Zdroják* [online], 21. 10. 2013, [cit. 2017-05-05]. Dostupné z: `https://www.zdrojak.cz/clanky/grafova-terminologie`

[5] Ramba, J.: Přehled grafových databází. *Zdroják* [online], 8. 11. 2013, [cit. 2017-05-05]. Dostupné z: `https://www.zdrojak.cz/clanky/prehled-grafovych-databazi/`

[6] Ramba, J.: Modelování dat v Neo4j – Metodika a obousměrné hrany. *Zdroják* [online], 8. 1. 2014, [cit. 2017-05-05]. Dostupné z: `https://www.zdrojak.cz/clanky/modelovani-dat-v-neo4j-metodika-obousmerne-hrany/`

[7] Neo Technology, Inc.: *Intro to Cypher* [online]. ©2017, [cit. 2017-05-05]. Dostupné z: `https://neo4j.com/developer/cypher-query-language/`

[8] Chao, J.: Imperative vs. Declarative Query Languages: What's the Difference? *Neo4j Blog* [online], September 19, 2016, [cit. 2017-05-05]. Dostupné z: `https://neo4j.com/blog/imperative-vs-declarative-query-languages/`

[9] Kolář, J.: *Teoretická informatika*. Praha: Česká informatická společnost, druhé vydání, 2000, ISBN 8090085385.

[10] Neo Technology, Inc.: *Indexes* [online]. ©2017, [cit. 2017-05-05]. Dostupné z: http://neo4j.com/docs/developer-manual/current/cypher/schema/index/

[11] Neo Technology, Inc.: *Constraints* [online]. ©2017, [cit. 2017-05-05]. Dostupné z: http://neo4j.com/docs/developer-manual/current/cypher/schema/constraints/

[12] Neo Technology, Inc.: *Release Notes: Neo4j 2.3.0* [online]. ©2017, [cit. 2017-05-05]. Dostupné z: https://neo4j.com/release-notes/neo4j-2-3-0/

[13] Neo Technology, Inc.: *Using Neo4j from Java* [online]. ©2017, [cit. 2017-05-05]. Dostupné z: https://neo4j.com/developer/java/

[14] Neo Technology, Inc.: *Chapter 3. Reference*. ©2017, [cit. 2017-05-05]. Dostupné z: https://neo4j.com/docs/ogm-manual/current/reference/

[15] Java Excel API - A Java API to read, write, and modify Excel spreadsheets [online], 2017, [cit. 2017-05-05]. Dostupné z: http://jexcelapi.sourceforge.net/

[16] The Apache Software Foundation: *POI-HSSF and POI-XSSF - Java API To Access Microsoft Excel Format Files* [online]. ©2017, [cit. 2017-05-05]. Dostupné z: http://poi.apache.org/spreadsheet/index.html

[17] www.codejava.net: *How to Read Excel Files in Java using Apache POI* [online]. 2015, [cit. 2017-05-09]. Dostupné z: http://www.codejava.net/coding/how-to-read-excel-files-in-java-using-apache-poi

[18] Neo Technology, Inc.: *10.2. Cypher Shell* [online]. ©2017, [cit. 2017-05-07]. Dostupné z: https://neo4j.com/docs/operations-manual/current/tools/cypher-shell/

[19] Hunger, M.: *Access to the neo4j-shell in NEO4J CE 3.x* [online]. Neo Technology, Inc., ©2017, [cit. 2017-05-07]. Dostupné z: https://neo4j.com/developer/kb/using-neo4j-shell-neo4j-ce-3x/

[20] Neo Technology, Inc.: *2.4. Windows installation* [online]. ©2017, [cit. 2017-05-07]. Dostupné z: https://neo4j.com/docs/operations-manual/current/installation/windows/

Příloha A

Seznam použitých zkratek

ACID Atomicity, Consistency, Isolation, Durability

API Application Programming Interface

ASCII American Standard Code for Information Interchange

CSV Comma-Separated Values

DDL Data Definition Language

HTTP Hypertext Transfer Protocol

JDBC Java Database Connectivity

JSON JavaScript Object Notation

MS Microsoft

NoSQL No Structured Query Language

NOSQL Not Only Structured Query Language

OGM Object-Graph Mapping

POJO Plain Old Java Object

REST Representational State Transfer

SQL Structured Query Language

W3C World Wide Web Consortium

Příloha B

Obsah přiloženého CD

```
readme.txt ............................ stručný popis obsahu CD
src
├── impl .............................. zdrojové kódy implementace
└── thesis ............................ zdrojová forma práce ve formátu LaTeX
doc ................................... dokumentace implementace
text .................................. text práce
└── thesis.pdf ........................ text práce ve formátu PDF
```