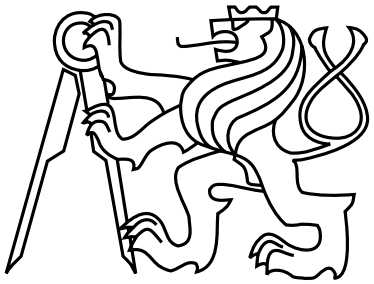
Na tomto místě bude oficiální zadání vaší práce

- Toto zadání je podepsané děkanem a vedoucím katedry,
- musíte si ho vyzvednout na studiijním oddělení Katedry počítačů na Karlově náměstí,
- v jedné odevzdané práci bude originál tohoto zadání (originál zůstává po obhajobě na katedře),
- ve druhé bude na stejném místě neověřená kopie tohoto dokumentu (tato se vám vrátí po obhajobě).

České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra počítačů



Bakalářská práce

GTD vývojářský úkolovník

Ondřej Šatera

Vedoucí práce: Ing. Macek Ondřej

Studijní program: Softwarové technologie a management, Bakalářský

Obor: Softwarové inženýrství

29. dubna 2012

Poděkování

Zde můžete napsat své poděkování, pokud chcete a máte komu děkovat.

Prohlášení

Prohlašuji, že jsem práci vypracoval samostatně a použil jsem pouze podklady uvedené v přiloženém seznamu.
Nemám závažný důvod proti užití tohoto školního díla ve smyslu §60 Zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Jihlavě dne 29. 4. 2012 .....

Abstract

The purpose of this work is to create tool for developers, which allows them to manage tasks from different services. The tool will be built on a platform of Titanium. An integral part of developing this application will be testing. The aim is to facilitate the application users' work with tasks from different resources / services.

Abstrakt

Obsahem této práce je vytvoření nástroje pro vývojáře, který umožní správu úkolů z různých služeb. Nástroj bude postavený na platformě Titanium. Nedílnou součástí vývoje bude i testování aplikace. Cílem aplikace je usnadnit uživateli práci s úkoly z různých zdrojů/služeb.

Obsah

Seznam obrázků

Seznam tabulek

Kapitola 1

Úvod

Zadáním mojí bakalářské práce je vytvoření nástroje primárně pro vývojáře, kteří pracují na projektech hostovaných na jednom z těchto tří verzovacích serverů: Google Code, Assembla a GitHub. Nástroj ale poslouží i ostatním vývojářům, byť nejsou primární cílovou skupinou. Využití najde i u neprogramátorů, a to u těch kteří svůj čas organizují metodikou „Getting Things Done“. Hlavním přínosem mojí práce je usnadnění práce vývojářům, kteří pro správu projektů používají jednu z výše zmíněných služeb. Aplikace bude kompletně vytvořena za pomocí vývojového prostředí Titanium Studio v rámci platformy Titanium. Výstupem práce bude kromě samotné aplikace i rešerše této platformy a zhodnocení práce s ním.

Kapitola 2

Motivace

V této kapitole je popsána motivace, kvůli které je tento nástroj vyvíjen.

2.1 Očekávaná funkcionalita

Stěžejním úkolem aplikace je správa úkolů a jejich třídění do projektů. Úkoly lze filtrovat podle jim přiřazených štítků a je možné si zvolený výběr vytisknout pro další zpracování. K úkolům lze přiřadit uživatele, kteří mají na starosti splnění daného úkolu. Zároveň je možné sledovat procentuální splnění jednotlivých projektů pomocí grafů a přehledů.

V tomto nástroji existují v zásadě čtyři typy projektů:

- obyčejný GTD projekt
- projekt hostovaný na serveru Assembla
- projekt hostovaný na serveru Google Code
- projekt hostovaný na serveru GitHub

V další části jsou tyto typy projektů popsány podrobněji.

2.1.1 Obyčejný GTD projekt

Obyčejný projekt neposkytuje žádné speciální funkce, poskytuje pouze možnost roztřídit si úkoly podle nějaké jejich společné vlastnosti. Uživatel si tak může například rozdělit nějaký rozsáhlejší úkol na menší části. Tento projekt ani nemusí souviset s vývojem software.

2.1.2 Hostované projekty

Projekty, které jsou uloženy na některém ze serverů, jsou vždy pevně svázány s některým z repozitářů uložených na daném serveru. Poskytují tak možnost synchronizace úkolů uložených na serveru s těmi v aplikaci. Při synchronizaci se zároveň stáhne seznam uživatelů, milníků a štítků, které je pak možné dále využívat. Například je možné do milníku přiřadit nový úkol a změna se automaticky projeví i na serveru. To samé je možné při přiřazování úkolů uživatelům.

Funkce	Assembla	GitHub	Google Code
Definování milníku	ano	ano	ano
Přiřazení štítků k issue	ne	ano	ano
Více různých stavů issue	ano	ne	ano
Nastavení priority	ano	ne	ano
Uložení aktivity u issue	ano	ne	ne
Přidání přílohy k issue	ano	ne	ano
Sledování aktivity v issues	ano	ne	ano
Cena za měsíc (pro komerční projekty)	od 9 (1GB) do 99 dolarů (20GB)	od 7 (0,6GB) do 22 dolarů (2,4GB)	-

Tabulka 2.1: Rozdíly mezi verzovacími servery

2.1.2.1 Porovnání verzovacích serverů

Každý ze serverů poskytuje jiné funkce, tzn. i API jednotlivých služeb nebudou stejná. To přináší komplikace při vývoji nástroje, který má v sobě integrovat správu všech tří služeb.

Každé z trojice používaných API poskytuje jiný přístup k datům uloženým na serveru. Assembla a Google Code se spoléhá na rozhraní REST. Github používá pro přenos dat JSON pole, která jsou snadněji zpracovatelná JavaScriptem a pomalu se dostávají i do jiných programovacích jazyků. Nejlépe zdokumentované rozhraní má jednoznačně Github. V dokumentaci lze najít i chybové hlášky a všechny typy návratových polí. Github zároveň poskytuje největší paletu nabízených služeb a klientská aplikace se tak dostane kamkoliv. Samotná webová aplikace Githubu běží nad tímto API.

Nejhůře je na tom s dokumentací Assembla, kde polovina údajů chybí a postup práce s API je tak spíše pokus-omyl. Nikde například nejsou k nalezení chybové hlášky, které API vrací pokud požadavek z nějakého důvodu nevyhovuje. Vývojáři tak nezbývá nic jiného, než odpověď hledat pomocí vyhledávače.

Google Code má dokumentaci obstojnou i když pod úrovní té na Githubu. Některé informace jsou zapsány poměrně nelogicky a v místech, kde by je člověk nehledal. Svůj účel ale dokumentace plní a na většinu otázek dokáže odpovědět.

Služby jsou rozdílné i z jiného úhlu pohledu. A sice z pohledu uživatele. Pro lepší přehlednost jsou tyto rozdíly uvedeny v následující tabulce:

Kvůli těmto rozdílům jsou nutné různé kompromisy, aby bylo možné aplikaci používat konzistentně bez ohledu na to, kde je projekt hostován. Ukazuje se, že GitHub má sice nejlepší dokumentaci, ale ve funkcionalitě pokulhává. Assembla, jejíž dokumentace je nejhorší, naopak poskytuje spousty funkcí navíc. Assembla je ale spíše zaměřená na komerční projekty, kdežto zbylé dvě služby jsou orientovány spíše na open-source vývojáře. Google Code neumožňuje hostovat komerční projekty vůbec. GitHub má i placený hosting projektů, ale jeho možnosti jsou mnohem menší než u konkurenční Assembly.

2.1.3 Getting Things Done

Tato metoda byla vytvořena americkým koučem Davidem Allenem, který ji popsal ve stejnojmenné knize. Neslouží přímo k organizaci času, ale orientuje se spíše na organizaci práce

a její plánování. Hlavní myšlenkou Allenovi práce je fakt, že lidský mozek není diář a není uzpůsoben k tomu, aby si pamatoval každý úkol a závazek, který je nutno splnit. Člověk pracuje lépe, pokud se nemusí věnovat tomu, aby si vzpomenul, co všechno musí udělat. Jádrem této metody jsou proto různé seznamy, které obsahují veškeré úkoly, které je nutno vyřešit. Mozek se tak může soustředit čistě na práci a není rozptylován vzpomínáním na jiné nesouvisející úkoly.

Celou metodu lze rozdělit do pěti kroků:

- sběr úkolů
- zpracování
- zorganizování
- zhodnocení
- vykonání

V aplikaci jsou zachyceny pouze prostřední tři kroky. Sběr úkolů je nutné provádět průběžně, tzn. nemusí to být v dosahu počítače. Tomuto účelu bohatě postačí nějaký papírový zápisník, příp. poznámky uložené v telefonu. Poslední krok „vykonání“ je zase plně v režii člověka, tam už aplikace nepomůže.

V kroku „zpracování“ dochází k přesunu úkolů z různých zdrojů (zápisník, poznámky) do jedné schránky - aplikace. Nesplnitelné úkoly se buď zahodí nebo se uloží na později. Splnitelné úkoly se buď vykonají, přiřadí někomu jinému nebo se uloží na později. O tom, zda se úkol vykoná hned nebo se uloží, rozhoduje pravidlo 2 minut. Pokud vykonání úkolu zabere víc času než dvě minuty, je uložen k pozdějšímu zpracování. Zároveň pokud je krok komplexnější a k jeho splnění je potřeba víc než jeden krok, je tento rozdělen na víc částí a uložen jako projekt.

V další fázi - zorganizování - dochází k rozdělení úkolů do těchto pěti oblastí:

- další kroky - realizovatelné, fyzicky viditelné činnosti, které vedou k nějakému výsledku
- delegované úkoly - přiřazené jiným lidem, u kterých čekáme na zpracování
- projektové úkoly
- úkoly uložené na později
- naplánované úkoly - pevně dané datum splnění (deadline)

Čtvrtá fáze, která je zde označena jako zhodnocení, probíhá paralelně se všemi ostatními. Během ní člověk přehodnocuje, zda dělá to, co by dělat měl. K tomu mu poslouží seznam nesplněných úkolů a projektů. Ke zhodnocení dochází také jednou za týden, kdy se upravuje seznam úkolů tak, aby byl aktuální.

2.2 Důvody pro zvolení desktopové aplikace

V dnešní době už desktopové aplikace vycházejí z módy. Všechna data a aplikace se přesouvají do webového prostoru, kde je obsah přístupný odkudkoliv a je jedno přes jaké zařízení se k němu přistupuje. Dokumenty tak lze vytvářet na stolním počítači a pak je upravovat na svém smartphonu, který má připojení k internetu. Používání webových úložišť má ale i svá úskalí - bezpečnost a spolehlivost.

2.2.1 Bezpečnost

Data na internetu jsou uložena na nějakém vzdáleném serveru a člověk nemá jistotu v tom, že k nim nemá přístup někdo nepovolaný. U desktopových aplikací lze bezpečnost snadno ohlídat minimálně pomocí hesla, příp. šifrováním obsahu na pevném disku. Pokud je ale obsah uložený na jednom pevném disku, zvyšuje se pravděpodobnost ztráty dat, takže je nutné pravidelné zálohování. U vzdálených serverů je obvykle o zálohování postaráno automaticky.

Jak vidno, obě varianty mají svá pro a proti. Důvody pro zvolení desktopové jsou v zásadě dva - rešerše práce s vývojovým prostředím Titanium Studio a experiment, zda je možné napsat kompletní aplikaci pouze pomocí JavaScriptu.

2.2.2 Využití JavaScriptu

JavaScript se v poslední době opět vrací na výsluní a je k nalezení téměř na každé webové stránce či aplikaci. Na rozdíl od ostatních technologií jako je Flash nebo Silverlight, není závislý na platformě a funguje stejně dobře na operačním systému Windows, Linux nebo Mac. Jeho podpora je pevně zabudována do drtivé většiny webových prohlížečů a v poslední době se jeho podpora rozšiřuje i na mobilní zařízení. Žádná jiná technologie nemá takovou podporu. JavaScript je nejsnadnější způsob, jak udělat stránku interaktivní. Změnit to může snad jedině větší rozšíření HTML 5, ale to bude ještě nějakou dobu trvat.

Kapitola 3

Analýza

Pro vývoj aplikace v této bakalářské práci byla zvolena novější metodika vývoje, která je modernější a v mnoha ohledech jednodušší než dříve používaný Rational Unified Process (RUP). Hlavní výhoda spočívá v tom, že se nevytváří hromady dokumentů, jejichž vytvoření zabere spoustu času a ve finále je většina z nich zbytečná. RUP má smysl spíše u velkých projektů, na kterých pracuje větší tým lidí a které jsou hodně rozsáhlé. RUP a agilní metodiky se liší už ve způsobu sběru požadavků. Zatímco RUP je orientovaný na use-case modely, vytvářené obvykle v jazyce Unified Modeling Language (UML), agilní metodiky evidují požadavky tzv. user stories.

3.1 User stories

Požadavky v rámci agilních metodik se nezapisují jako prosté úkoly, ale mají formát tzv. user story. Jejich tvar je pevně daný a skládá se ze tří částí:

> Jako **Uživatel**, chci **Funkcionalitu** abych dostal **Bussiness Value**

Takto formulovaný požadavek je pro člověka mnohem srozumitelnější, protože vidí proč daný úkol splnit a co to komu přinese. Má tak jistotu, že nedělá něco zbytečného. User story je takový krátký příbeh a jako takový je lidským mozkem snáz vnímaný.

User stories poslouží i po ukončení vývoje - při akceptačních testech. Stačí projít seznam user stories a pokud je v aplikace nalezen jejich „odraz“, test byl úspěšný. Proto je lepší, aby user stories prošly schválením od zadavatele ještě před začátkem vývoje.

V rámci této bakalářské práce byl vytvořen následující seznam user stories, který je pro lepší přehlednost rozdělen do bloků podle cílové skupiny. Každý blok je uveden názvem role, kde lomítko značí dědičnost. Jsou v něm zachyceny všechny požadavky, které by měla aplikace po ukončení vývoje splňovat.

3.1.1 Uživatel

Jako užívatel chci:

- používat aplikaci co nejjednodušeji, abych se mohl plně soustředit na přesné zadání úkolu
- vytvářet úkoly, abych na nic nezapomněl
- vytvářet projekty, abych si mohl úkoly třídit
- vkládat okažité nápady do inboxu s tím, že je později zatřídím do projektu
- označit úkol jako splněný
- uložit úkol do archivu, kdybych se k němu chtěl někdy později vrátit
- vyhodit úkol do koše, pokud se rozhodnu, že ho nebudu realizovat
- přidávat úkolům štítky, abych si mohl odfiltrovat úkoly z určité oblasti
- na konci týdne vědět, co všechno jsem za týden stihl, abych si mohl lépe naplánovat úkoly na příští týden
- své nápady, na které teď nemám čas, přiřadit do skupiny s delším časovým horizontem, abych na ně nezapomněl a mohl je později rozvíjet
- vytisknout svoje úkoly, abych je mohl mít neustále na očích

3.1.2 Uživatel /vývojář

Jako vývojář chci:

- importovat projekty a úkoly z GitHubu (dále jen GH)
- být informován o blížící se deadline úkolů
- aby aplikace neumožňovala přístup neoprávněným osobám k mým úkolům
- spárovat úkol s konkrétním commitem do repozitáře
- exportovat úkol a poslat ho jednoduše kolegovi v týmu, aby ho nemusel ručně přepisovat

3.1.3 Uživatel /vývojář /senior

Jako senior vývojář chci:

- párovat repozitáře z GH s mými projekty, abych mohl sledovat, jak práce na projektu pokračuje
- párovat issues z GH s mými úkoly, abych s nimi mohl párovat commity do GitHubu
- nastavit úkolům čas, kdy mají být splněny
- delegovat úkoly junior vývojářům

- přiřadit úkol do konkrétního milníku, abych měl přehled o tom, kolik toho ještě zbývá dokončit
- být informován o změnách v mých repozitářích na GH
- uzavřít projekt, po jeho dokončení příp. ukončení
- přidávat štítky k úkolům, abych je mohl lépe třídit a filtrovat
- být upozorněn na neaktivní otevřené projekty, abych mohl urgovat dokončení úkolů na junior vývojářích
- mazat úkoly, které se nakonec realizovat nebudou nebo které již byly dokončeny
- zpřesňovat zadání úkolů, pokud dojde k nejasnostem

3.1.4 Uživatel /vývojář /junior

Jako junior vývojář chci:

- být informován o nově přiřazených úkolech od senior vývojářů
- označit úkol jako splněný

3.2 Doménový model

Z výše zmíněných user stories byl dalším kroku vytvořen seznam entit, které byly logicky provázány asociacemi. Pro lepší znázornění těchto entit a vztahů mezi nimi byl vytvořen doménové model v jazyce UML.

3.2.1 Issue

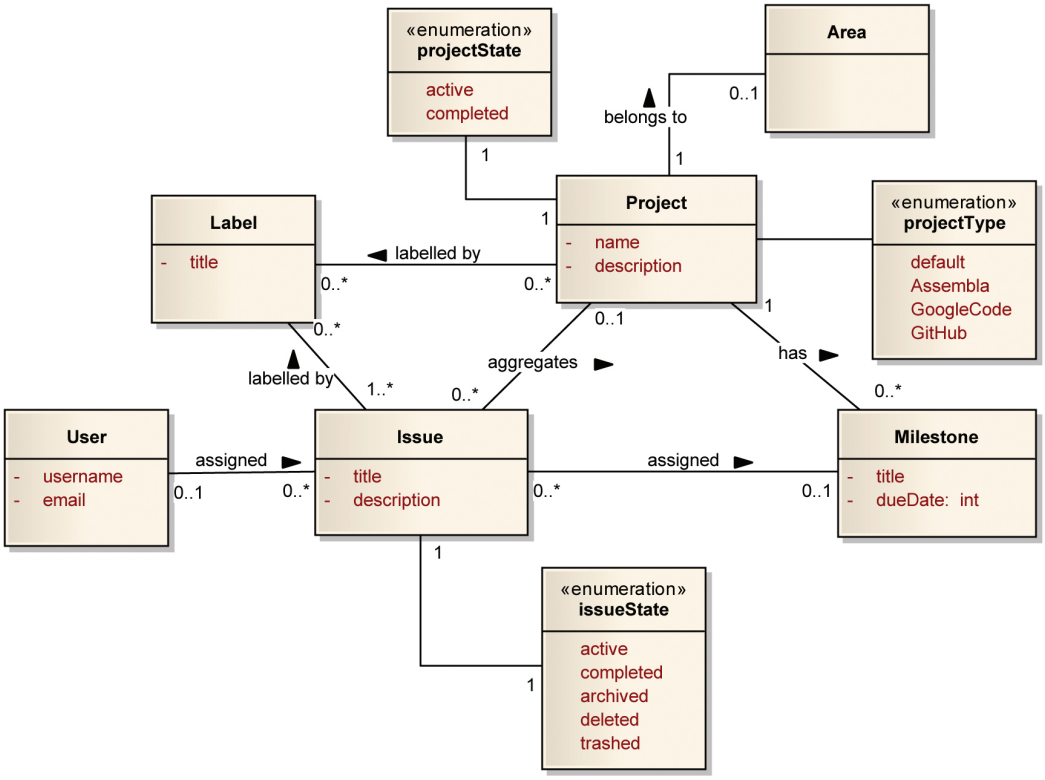
Entita reprezentující jeden úkol, uložený v aplikaci. Má definované jméno (název problému) a popis, který obsahuje konkrétní popis problému

3.2.2 issueState

Úkoly (issues) existují v aplikaci v různých stavech. Každý stav určuje jak bude aplikace s daným úkolem zacházet.

3.2.3 Label

Štítky (label) slouží k filtraci úkolů (issues) a projektů (projects). Obvykle popisují nějakou obecnější vlastnost dané entity, podle které má smysl je filtrovat. To může být například priorita, programovací jazyk nebo náročnost.



Obrázek 3.1: Doménový model

3.2.4 Milestone

Milestone se do češtiny překládá jako milník. Je to nějaký bod v čase, do kterého musí být dokončená určitá množina úkolů (issues). Lze sledovat procentuální dokončení.

3.2.5 User

Uživatel (user) vystupuje v aplikaci jako člen týmu, kterému je možné přiřadit nějaký úkol (issue).

3.2.6 Project

Úkoly (issues) jsou uspořádané do projektů, které mají definované jméno a popis. I z úkolu se může stát projekt pokud je k jeho dokončení potřeba víc než jeden krok (podle GTD).

3.2.7 projectState

Projekt může být buď aktivní, tzn. že se na něm pracuje, nebo dokončený.

Atributy	Poznámky
active	na úkolu se pracuje
completed	úkol byl splněn
archived	úkol byl uložen do archivu
deleted	úkol byl smazán
trashed	úkol byl přesunout do koše

Tabulka 3.1: Vlastnosti entity IssueState

Atributy	Poznámky
title	text štítku

Tabulka 3.2: Vlastnosti entity Label

Atributy	Poznámky
title	název milníku
dueDate	do kdy musí být milník splněn

Tabulka 3.3: Vlastnosti entity Milestone

Atributy	Poznámky
username	uživatelské jméno
email	e-mailová adresa uživatele

Tabulka 3.4: Vlastnosti entity User

Atributy	Poznámky
name	název projektu
description	popis projektu

Tabulka 3.5: Vlastnosti entity Project

Atributy	Poznámky
active	aktivní
completed	dokončený

Tabulka 3.6: Vlastnosti entity ProjectState

Atributy	Poznámky
default	obyčejný GTD projekt
Assembla	projekt hostovaný na serveru Assembla.com
GoogleCode	projekt hostovaný na serveru Google Code
GitHub	projekt hostovaný na serveru GitHub.com

Tabulka 3.7: Vlastnosti entity ProjectType

3.2.8 projectType

Každý projekt má definovaný nějaký typ, podle kterého se určí na jaký server se má synchronizovat. Pokud jde o obyčejný projekt (default) nesynchronizuje se nikam, všechny úkoly zůstávají pouze na lokálním úložišti.

3.2.9 Area

Projekty (project) lze zařadit do nějaké oblasti (area), což může být např. škola, práce, vzdělávání apod. Cílem je zpřehlednění seznamu projektů.

Kapitola 4

Architektura

4.1 Rozdělení do balíčků

Aplikaci lze rozdělit do několika balíčků, i když toto rozdělení není tak patrné jako například u Javy. Jako vzor pro celkovou architekturu bylo zvoleno MVC (model-view-controller), kde je oddělená prezentační část od výkonné a obsahové. Nicméně to vyžadovalo pár kompromisů protože JavaScript není primárně objektový jazyk.

Celá aplikace využívá ke svému běhu knihovnu PrototypeJS, díky které lze snadno definovat objekty a omezeně i dědičnost mezi nimi. V prezentační části se navíc využívá knihovna Script.aculo.us, která je postavena na dříve zmíněném PrototypeJS a která umožňuje snadnější vytváření efektů a různých animací.

Následující diagram znázorňuje rozdělení aplikace do balíčků.

Class model.eps

4.1.1 Controller

Balíček tříd, které se starají o samotný běh aplikace a synchronizaci se vzdálenými servery.

4.1.1.1 Application

Tato třída je využívána hlavně při startu aplikace. Má dva hlavní úkoly - načíst a naparsovat konfigurační soubor, který je uložený ve formátu JSON, a založit připojení k databázi.

4.1.1.2 Sync

Synchronizace mezi aplikací a vzdálenými servery má na starosti právě tato třída. Přijímá nové objekty z dialogů, předává je do modelové části a stará se o volání odpovídajících API (podle typu projektu). Žádná velká logika v ní není, stará se hlavně o předávání požadavků do jiných vrstev aplikace (model a view).

4.1.2 Model

Balíček tříd, které se starají o komunikaci s databází.

4.1.2.1 Model

V této třídě jsou umístěny všechny metody, které přímo komunikují s databází. Největší skupinou metod jsou ty, které poskytují CRUD (create, read, update, delete) nad všemi entitami zastoupenými v systému. Zároveň je zde několik metod, které usnadňují často používané operace, jako je načítání štítků ke konkrétnímu úkolu nebo zjištění procentuálního dokončení jednotlivých projektů.

4.1.3 API's

Balíček tříd, které jsou používány při komunikaci se vzdálenými servery.

4.1.3.1 xxxAPI

Třídy se sufixem „API" jsou v systému celkem tři (AssemblaAPI, GCodeAPI a GitHubAPI). Každá odpovídá jednomu verzovacímu serveru, se kterým je systém synchronizovatelný. Všechny mají společného předka - abstraktní třídu API - a to z důvodu usnadnění budoucího rozšíření aplikace o další servery. Bohužel v JavaScriptu není implementace dědění úplně dokonalá, takže jde spíš o doporučení než povinnost. Třídy využívají každá svého klienta pro volání vzdálených serverů. U Assembly a Google Code je to REST_client, GitHub používá Ajax_client. Zároveň mají na starost naparsování entit do odpovídajícího formátu (JSON nebo XML), aby se dali předat klientovi, který je přepošle ven.

4.1.3.2 Ajax_client

Tato třída je spolu s tou následující dalším usnadněním budoucího rozšiřování aplikace o spolupráci s dalšími servery. Slouží k odesílání požadavků v podobě AJAXových volání. V momentálním stavu aplikace ji využívá pouze API služby GitHub. Na vstup získává požadavek ve formátu JSON a odesílá ji na stanovenou URL adresu pomocí zvolené metody (POST, PUT, DELETE). Po přijetí odpovědi předá získaný výsledek zpět API do metody, jejíž jméno klient získal při úvodním volání.

4.1.3.3 REST_client

Protože zbylé dva servery, tzn. Assembla a Google Code, fungují na architektuře označované jako REST (REpresentational State Transfer) bylo nutné vytvořit druhého klienta, který usnadní komunikaci s těmito servery. Požadavky jsou narozdíl od předchozí třídy ve formátu XML souboru. Zpracování odpovědi se příliš neliší od předchozí třídy. Jediný rozdíl tkví v naparsování odpovědi do objektu reprezentujícího XML dokument.

4.1.4 View

Balíček tříd, které mění vizuální stránku aplikace.

4.1.4.1 Viewer

V rámci dodržení architektury MVC došlo k oddělení vytváření vizuální stránky aplikace do samostatné třídy. Ta se stará o výpis seznamu úkolů, projektů a dalších grafických prvků. Grafická stránka aplikace je vytvářena na základě několika šablon, které jsou uloženy v samostatných souborech, aby bylo možné je v budoucnu snadno pozměnit bez ovlivnění funkčnosti aplikace. Takže například výpis úkolů je složen z x částí, kde každá část pochází z jedné šablony, která se opakuje.

4.1.5 Entity

Balíček tříd, které reprezentují jednotlivé entity v systému. Třídy byly již popsány v sekci Metodika - Doménový model.

4.2 Bezpečnost

U webových aplikací je kladen velký důraz na zabezpečení aplikace proti vnějšímu zásahu ať už za účelem získání soukromých informací nebo poškození aplikace. Mnou vytvářená aplikace sice není přímo webová ale i tak byla otázka zabezpečení důležitá. Nejzranitelnější částí aplikace je samotná komunikace se vzdálenými servery jednotlivých verzovacích systémů. Není totiž možné použít pokročilé zabezpečovací techniky jako je OAuth2 (používá Github) nebo AuthSub proxy (Google Code). V obou případech je totiž nutnou podmínkou fixní URL, na které klientská aplikace běží. Mnou vytvářená aplikace běží přímo na uživatelově počítači a ne někde na vzdáleném serveru a jako taková nemá přidělenou globálně přístupnou URL. Jedinou možností zabezpečení aplikace tak zůstala Basic authentication, u které stačí připojit speciální hlavičku „Authorization“ přímo do odesílaného požadavku. Jejím obsahem je slovo „Basic“ a zahashované spojení uživatelského jména a hesla kódováním Base64. Tento hash lze snadno přeložit zpět na čitelnou formu, protože účelem toho algoritmu není šifrování přenášených dat ale pouze možnost zapsat binární data do tisknutelných znaků ASCII. Každý z trojice verzovacích systémů (Assembla, Github, Google Code) používá jiný způsob zabezpečení komunikace (tzn. autentizaci a autorizaci).

4.2.1 Zabezpečení serveru Assembla

U tohoto serveru existuje jediný způsob zabezpečení a sice použití Basic Authentication, které vlastně žádné zabezpečení neposkytuje, slouží pouze k autentizaci požadavku na API.

4.2.2 Zabezpečení serveru GitHub

V případě služby Github je situace komplikovanější. Upřednostňovaná forma přihlášení k serveru je OAuth2, což je protokol sloužící externím aplikacím k požádání o autorizaci bez toho, aby získaly heslo uživatele. Je preferována před Basic Authentication protože umožňuje omezit přístup jen k určitým datům a uživatel může tento přístup kdykoliv zrušit. Postup získání tohoto povolení je jednoduchý. Vývojář nejdřív svou aplikace zaregistruje a tím získá dva údaje: unikátní ID klienta a tajné heslo, které by nemělo být nikde zveřejněno. Pomocí

těchto dvou údajů se klientská aplikace autentifikuje u serveru a po vyplnění přihlašovacího jména a hesla je uživatel přesměrován zpět do klientské aplikace s náhodně vygenerovaným tokenem (buď přímo v URL, nebo v hlavičce odpovědi). Tento token je pak používán u každého požadavku na server až do ukončení session. Tento postup je ale pro aplikace běžící na desktopu nepoužitelný protože není kam uživatele přesměrovat. Proto Github zároveň podporuje i Basic Authentication.

4.2.3 Zabezpečení serveru Google Code

Google Code používá službu založenou na podobném principu jako je OAuth2. Zde ale není nutná registrace aplikace přímo na serveru. Aplikace, která vyžaduje přístup k soukromým datům uživatele, a nemůže tedy využit anonymního přístupu, přesměruje uživatele na speciální URL, kde se vyplní uživatelské jméno a heslo a uživatel je posléze přesměrován zpět do klientské aplikace s vygenerovaným AuthSub tokenem uloženým v odpovědi. Obsahem požadavku jsou čtyři údaje:

1. next - URL, na kterou má být uživatel přesměrován (URL aplikace)
2. scope – určí, že je požadován vstup do Google Code
3. secure – určuje, zda klient požaduje zabezpečený token
4. session – určuje, zda může být token konvertován na multi-use (session) token

Pro aplikace běžící na desktopu je ale nutné použít nižší úroveň zabezpečení – ClientLogin. Ten spočívá v odeslaní požadavku ve specifikovaném formátu na danou URL. V odpovědi, pokud je autentizace úspěšná, jsou navrácena tři alfanumerické kódy. Klientskou aplikace ale zajímá pouze ten poslední, který je použit jako autorizační token při odesílání požadavku (podobně jako se u Basic authentication odesílá Base64 hash). Kamenem úrazu této metody je ale úvodní požadavek, ve kterém je odesláno uživatelské jméno a heslo v čitelné formě přímo v požadavku (konkrétně v POST), takže pokud by provoz na síti někdo odposlouchával, získá snadno přístup do účtu uživatele. Bohužel jiná forma autentizace pro desktopové aplikace neexistuje ani u jedné z těchto tří služeb.

4.3 Titanium studio

4.3.1 Představení

Samotné Titanium Studio je pouze vývojové prostředí, které usnadňuje práci s platformou Titanium. Tato není závislá na operačním systému. Je tak možné vytvářet aplikace zároveň pro Windows, Linux nebo Mac. Stejně tak pokud se rozhodneme vytvořit mobilní verzi dané aplikace, je možné některé části kódu znovupoužít a předělat jen grafickou stránku aplikace. Jádrem celé platformy je API, které je přístupné přes globální objekt Titanium. Přes něj se dají snadno vytvářet další okna aplikace, různá menu a poskytuje snadný přístup k souborům uloženým na filesystému nebo k tabulkám v databázi.

4.3.1.1 Databáze

Jako databázový engine se používá SQLite, kde je celá databáze uložena v jednom souboru a tím se snadno zálohuje nebo přenáší na jiný počítač. O připojení k databázi se postará globální objekt, nám stačí znát její jméno, které si můžeme předem určit. Po připojení k ní získáme objekt, na kterém už lze pokládat dotazy na databázi v jazyce SQL. Jak může takové volání vypadat, ukazuje následující příklad:

```
DB.execute("INSERT INTO images (title, description) VALUES (?, ?)", 'test',
'description');
```

V tomto příkladě je vidět obrana proti útoku SQL Injection, které je realizováno voláním metody s argumenty, které jsou následně escapovány.

4.3.1.2 AJAX

Další velmi důležitou součástí API je mechanismus pro asynchronní volání vzdálených serverů - AJAX. Provádění těchto volání má jednu velkou výhodu oproti tomu samému volání v prohlížeči – není zde problém s cross-domain policy, což je v zásadě ochrana proti vykonávání JavaScriptu na jiném serveru, kterou mají prohlížeče zabudovánu v sobě.

Během volání je možné sledovat odpovědi serveru a můžeme si definovat metody, které na tyto události zareagují. Máme také plnou kontrolu nad tím jaká data na server posíláme a jakou metodou v rámci protokolu HTTP se tak stane. API verzovacích serverů totiž rozlišují požadavky i podle této metody. Takže pokud chceme něco na serveru smazat musíme použít metodu DELETE. Pokud naopak chceme založit něco nového (úkol, projekt, milník), použijeme metodu PUT. K běžnému stahování obsahu - čtení - nám postačí metoda GET příp. POST.

Protože API služeb Assembla a Google Code běží na architektuře REST, je nutné mít možnost posílat voláním i soubory, konkrétně ve formátu XML. Tento požadavek byl poměrně problematický, protože v dokumentaci API nebylo toto dostatečně popsáno. Naštěstí jsou na internetu různé tutoriály, které pomohou a navedou člověka správným směrem. Samotné sestavování AJAXového volání je hodně low-level a člověk musí řešit spousty technickým věcí. Například u zmiňovaného posílání souborů musíme ručně sestavit hlavičku požadavku, aby došel ve správném tvaru na server. Server Assembla například soubor nepřijme, pokud ve volání nepřijde hlavička:

```
Accept : application/xml
```

4.3.1.3 Vytváření oken a nabídek

Každá aplikace na desktopu sestává z více částí. Hlavní viditelnou částí je samotné okno aplikace. V prostředí Titanium Studio si můžeme definovat jak toto okno bude velké, zda bude maximalizované a jak s ním bude moct uživatel manipulovat. Všechny tyto volby jsou uloženy v XML souboru, takže se dají snadno upravovat.

Pokud potřebujeme za běhu aplikace vytvořit nové okno, umožní nám to zmiňované API. Nastavíme si jeho velikost, polohu, obsah (obvykle HTML soubor) a další volby. Titanium ho

pak vykreslí a my s ním můžeme dále pracovat. Všechny Javascriptové soubory, které chceme v novém okně využít, musíme vložit do toho HTML souboru. A to i když jsou již vloženy do hlavního okna aplikace. Jediné, co se vloží automaticky, je globální objekt Titanium, který nám zpřístupňuje API.

Většina aplikací na desktopu mívá hlavní menu v šedém pruhu hned na vrchu okna aplikace. I v rámci Titanium Studia je možné toto menu vytvořit. Funguje to jednoduše. Na API zavoláme metodu Titanium.UI.createMenu, která nám vytvoří objekt, reprezentující celé menu. Do něj pak můžeme buď přidávat další podmenu nebo přímo jednotlivé položky. Menu funguje na principu událostí, takže pokud uživatel na některou z položek klikne, dojde k zavolání metody definované při zakládání dané položky.

4.3.2 Výhody a nevýhody

Mezi hlavní výhody využívání Titanium API patří rozhodně:

- snadná práce s databází
- možnost posílat AJAXová volání na vzdálené servery bez omezení
- snadná práce s UI aplikace (vytváření dalších oken a nabídek)

Velkou nevýhodou je rozhodně debugování (nalézání a oprava chyb) aplikace. V rámci běhu aplikace sice máme možnost sledovat výpis hlášení v okně konzole, ale mnohé chyby se zde neobjeví a aplikace prostě zamrzne. Vývoj se tak občas dost zpomaluje, protože hloupé chyby se hledají obtížně a jeden malý překlep může způsobit velké problémy.

4.3.3 Využití API v aplikaci

Vytvořená aplikace je pevně spojená s Titanium API a nemůže bez něj prakticky existovat. Mezi nejdůležitější využívané funkce patří rozhodně přístup k databázi a filesystému. To jsou funkce, které by byly bez API poměrně obtížně realizovatelné. Je toho ale mnohem víc. Vzhledem k tomu, že jedním z poslání aplikace je synchronizace se vzdálenými servery, je hojně využívána i část síťová, tzn. AJAXová volání.

Protože JavaScript není úplně objektový jazyk a některé konstrukce nejsou možné, pomohlo Titanium API i zde. Jde například o předávání globálních objektů, aby nebylo potřeba je pokaždé vytvářet znovu, což by bylo velké plýtvání prostředky a nejspíš by to výrazně zpomalilo celou aplikaci. Objekty se proto po vytvoření uloží do globální úložiště (ač je to v rozporu s principy objektového programování), které je plně v režii Titanium API a odkud je možné si je kdykoliv vyžádat a vykonávat na nich operace. Takto je například uložené spojení s databází nebo objekt starající se o synchronizaci se vzdálenými servery.

Jak je API v aplikace využíváno, bude nejlépe patrné z nějakého příkladu. Následující seznam ukazuje workflow přidání nového úkolu do projektu, který je svázán s repozitářem na serveru GitHub. Pro lepší přehlednost je vždy nejdříve uvedena entita, která danou operaci provede.

1. uživatel: klikne na tlačítko „New issue“

2. systém: zavolá funkci, která obsluhuje dané tlačítko
3. API: zobrazí dialog s obsahem, který definoval handler v předchozím kroku
4. (a) API: načte z databáze uživatele a milníky daného projektu
5. uživatel: vyplní formulář v dialogu a odešle ho tlačítkem "Save"
6. systém: vytáhne z formuláře vyplněná data a vyžádá si od API objekty reprezentující zvoleného uživatele a/nebo milníku
7. (a) API: načte z databáze uživatele a předá ho modelové třídě
8. (a) systém: modelová třída uživatele naparsuje do objektu a předá ho zpět
9. systém: uzavře dialog zavoláním API
10. systém: nově vytvořený objekt reprezentující úkol je předán synchronizační třídě
11. (a) systém: objekt je naparsován do formátu JSON, který používá server GitHub pro komunikaci
12. (a) API: pomocí objektu Titanium.Network.HTTPClient je řetězec s JSON odeslán na server a vrácený výsledek je poslán zpět do synchronizační třídy
13. (a) systém: záznam o úkolu je aktualizován v databázi o údaje vrácené ze serveru
14. systém: aktualizuje výpis úkolů daného projektu

Jak je vidět z příkladu, API je skutečně integrální součástí aplikace a poskytuje spousty důležitých funkcí, které hodně zrychlí a usnadní vývoj celé aplikace.

Kapitola 5

Nasazení aplikace

Titanium Studio je deklarované jako multiplatformní a neměl by tedy být problém s přenosem vytvořené aplikace na jiný operační systém. Bohužel dle získané zkušenosti tomu tak není. Aplikace byla primárně vyvíjena na operačním systému Windows 7, kde se neobjevily žádné větší problémy. Ty ale nastaly při pokusu o spuštění na jiné platformě. Pokus o nasazení byl proveden na dalších dvou různých operačních systémech. Šlo o operační systém Debian, jakožto zástupce Linuxu, a Mac OS X. Ani na jednom ze zmiňovaných systémů se nepodařilo aplikaci plně zprovoznit.

5.1 Linux

Test nasazení byl proveden na virtuálním stroji s nainstalovaným operačním systémem Debian ve verzi 6.0.3 (i386).

Zde se ukázalo jako nemožné samotné spuštění vývojového prostředí Titanium Studio, ve kterém se aplikace kompiluje. Možnou příčinou je fakt, že aplikace je distribuována v archivu typu zip, který není primárně určen pro Linux a nezachovává symbolické odkazy. Veškeré pokusy o nápravu skončily neúspěchem. Podle oficiálního fóra má tyto problémy více uživatelů, takže by snad mělo někdy dojít k nápravě. Zatím se tak nestalo.

5.2 Mac OS X

Testování na této platformě bylo poměrně problematické, protože odpovídající operační systém nebyl k dispozici. Pokusy o nasazení byly prováděny pouze na soukromém počítači vedoucího této práce a lze říci, že nebyly 100% úspěšné. Na rozdíl od Linuxu sice bylo možné nainstalovat a spustit vývojové prostředí, ale se samotnou aplikací už to bylo horší. Ani po mnoha pokusech se ji nezdařilo plně zprovoznit. Neustále se objevovaly chyby na úrovni OS, které byly obtížně odstranitelné.

O multiplatformnosti Titanium Studia lze tedy oprávněně pochybovat.

Kapitola 6

Testování

Každý správný vývojový cyklus software by měl obsahovat fázi testování. Aplikace lze testovat ve více vlnách, kdy se pokaždé provádí jiný typ testů. Obecně lze testy rozdělit do těchto dvou hlavních skupin - whitebox a blackbox.

Blackbox testy testují aplikaci zvenku a nepotřebují vědět, co se děje uvnitř. Proto se testy označují jako „blackbox“ - aplikace je pro nás černou skříňkou a máme k dispozici pouze dva otvory - vstup a výstup. Do aplikace pošleme nějaký vstup a očekáváme určitý výstup. Pokud výstup splňuje očekávání, test byl úspěšný. Hlavní výhodou těchto testů je jejich jednoduchost, protože nepotřebujeme znát konkrétní obsah tříd a metod uvnitř aplikace. Do blackbox testů můžeme zařadit tyto testy:

- testy rozhraní (Selenium)
- akceptační
- zátěžové

Naproti tomu whitebox testy vyžadují znalost kódu aplikace a jsou pevněji svázány s implementací uvnitř. Tyto testy jsou mnohem konkrétnější a obvykle testují menší celky aplikace (balíčky -> třídy -> metody). Postup testování může být dvojí. Buď testujeme nejdřív největší celky a postupně se zanořujeme, nebo naopak postupujeme od nejmenších jednotek po ty největší. Těmto testům se říká jednotkové (angl. unit testy). Do kategorie whitebox testů spadají také integrační testy, které ověřují, zda spolu jednotlivé komponenty aplikace komunikují tak, jak mají.

6.1 Použitý způsob testování

Testování této aplikace nebylo tak snadné jako u webových aplikací. Některé testy nebyly ani realizovatelné (např. testy rozhraní pomocí nástroje Selenium), protože aplikace není spustitelná v prohlížeči. Toto omezení způsobuje právě ten globální objekt Titanium, který není v prostředí prohlížeče k dispozici a aplikace se tak stává nepoužitelnou.

Pro testování byla využita JavaScriptová knihovna jsUnity, pomocí které jdou poměrně snadno vytvářet jednotkové testy. Poskytuje jednak běhové prostředí a hlavně assertovací

metody, které ověří výsledek testu. Knihovnu je potřeba trochu poupravit, aby výsledek testů vypsala do okna a do výstupu, kde by se výsledek ztratil v záplavě runtime hlášení. Není to nijak velký zásah, stačí překrýt metodu log tímto způsobem:

```
jsUnity.log = function(message) { document.write(message + " extbackslashnbr");
};
```

Testy jsou seskupeny v objektech, které se spouští metodou *jsUnity.run()*. Aby nebyly při každém testu znovu zakládány všechny objekty, vytvoří se předem a během testů už se na nich pouze volají metody. Korektní postup je sice jejich zakládání před každým testem v metodě *setUp()*, ale tento způsob se ukázal jako hodně pomalý a bylo od něj upuštěno. Nejpomalejší operace je určitě založení spojení s databází, která je potřeba u většiny testů a protože testy mají být hlavně rychlé, byl nutný nějaký kompromis.

6.2 Testování asynchronních volání

V aplikaci se mnoho operací děje asynchronně, aby aplikace nezamrzala (hlavně při spojení se vzdálenými servery). Tento způsob běhu aplikace bohužel znemožňuje testování jednotkovými testy. Asynchronní volání by se dalo rozepsat do těchto kroků:

1. handler údalosti zavolá metodu na controlleru
2. controller získá potřebná data z databáze a naparsuje je pro potřeby daného serveru
3. pak zavolá metodu *send()* nebo *sendFile()* (pokud se odesílá soubor) na vytvořeném HTTP klientovi. Zároveň mu předá název funkce, kam chce dostat výsledek volání, tzv. callback
4. klient zavolá server s danými parametry a čeká na odpověď
5. po tom, co získá odpověď od serveru, ji naparsuje a pošle zpět controlleru na callback, který od něj získal
6. controller zpracuje odpověď od klienta

Problém, který zabraňuje testování, vzniká v kroku č.3 - předání callbacku. Během testování se o volání metod stará běhové prostředí a není možné volat jednotlivé metody (testy) samostatně. Proto se nemůžeme „vrátit“ z HTTP klienta zpět do testu a vyhodnotit správnost odpovědi - nemáme jak.

6.3 Zátěžové testy

(grafy zavislosti poctu issues na jejich vypis)

6.4 Doporučení do dalšího vývoje

Zátěžové testy nám ukázaly, že aplikace se stává poměrně pomalou s přibývajícími úkoly v jednotlivých projektech. Problém tkví ve čtení dat z databáze. Řešením by mohla být nějaká vyrovnávací paměť (cache), která by byla umístěna mezi databází a zbytkem aplikace. To s sebou nutně přinese mnoho dalších komplikací a zrychlení aplikace tak může být poměrně drahé. Mezi největší problémy patří určitě volba úložiště vyrovnávací paměti a také její invalidace (smazání neaktuálních dat). Úložiště musí být dostatečně rychle přístupné, aby vůbec mělo smysl vyrovnávací paměť implementovat. Nasnadě je využití souborové cache, ale čtení dat z filesystému nemusí být zrovna nejrychlejší. Lepší by bylo ukládání dat do operační paměti, jenže k té nemá JavaScript přístup. Dalším problémem je invalidace cache, tedy odstranění neaktuálních dat z paměti. Je totiž poměrně velký problém určit, kdy už data nejsou aktuální.

Kapitola 7

Závěr

7.1 Závěrečné zhodnocení aplikace

Zhodnocení aplikace z pohledu splnění požadavků je snadné, stačí si projít user stories z části týkající se metodiky a hned bude jasné, které byly naplněny a které nikoliv. Pokud tedy použijeme tento postup, dostaneme 95% splnění požadavků. V aplikaci chybí párování úkolů a jednotlivých commitů ze vzdáleného serveru. Analýza totiž ukázala, že tuto možnost má pouze GitHub, ostatní ji neposkytují, proto od ní bylo upuštěno a nebyla zahrnuta do aplikace.

7.1.1 Práce s IDE Titanium Studio

Součástí zadání této práce byla i rešerše Titanium Studia. Jejím cílem bylo ověřit možnost použití této platformy k vývoji desktopových aplikací. Jedna strana jsou oficiální specifikace a na druhé její reálné používání, jak se ukázalo mnohokrát během vývoje. Ve fázi testování se ale ukázalo, že spoustě problémů jsem se vyhnul tím, že jsem vyvíjel v operačním systému Windows a ne na Linuxu nebo Macu, kde už jen samotné zprovoznění IDE se ukázalo jako problém.

Nicméně Titanium API poskytuje spousty funkcí, které by se jinak museli doprogramovávat ručně. Není tak nutné zahrnovat do aplikace skripty napsané v jiném jazyce (Titanium Studio podporuje vývoj v PHP, Ruby a Pythonu) a využít připravené rozhraní. Nedokážu si například představit, jaké by to bylo vytvořit spojení s databází jen pomocí JavaScriptu nebo aplikační menu, které bude snadno rozšiřitelné a použitelné pro uživatele.

Všechno má ale jednu velkou vadu na kráse - vychytávání chyb (debugging). To je v tomto IDE velmi špatně zpracované. Autoři Titanium Studia sice nabízí rozšířený editor, který by měl mít debugging zpracovaný lépe, ale to už není poskytováno zdarma, a to ani ke studijním účelům. Ve většině případů jsem tak byl odkázaný na metodu pokus-omyl, kdy i oprava banálního překlepu může trvat velmi dlouho. Během vývoje této aplikace sice došlo k několika aktualizacím a IDE tak hlásí aspoň některé chyby, ale ve spoustě případů prostě jen zamrzne a neposkytne vůbec žádnou zpětnou vazbu o tom, co a kde se vlastně stalo.

Na základě mých zkušeností mohu Titanium Studio doporučit dalším vývojářům, kteří by chtěli vyvíjet desktopové aplikace v jiném jazyce než v Javě nebo C#. Protože jsem

členem redakce Programuje.com, což je momentálně nejčtenější IT magazín v ČR, využil jsem této příležitosti a sepsal jsem menší článek, představující Titanium Studio a práci s ním. Dle reakcí čtenářů lze usoudit, že platforma má před sebou budoucnost a má smysl ji dále rozvíjet.

7.1.2 Psaní aplikace zcela v JavaScriptu

Další teorií, kterou měla tato aplikace za cíl potvrdit nebo vyvrátit, byla otázka, zda je možné vytvořit aplikaci zcela v JavaScriptu bez pomoci dalších programovacích jazyků. Ukázalo se, že je to možné, ale zahrnuje to poměrně dost úskalí a kompromisů. Není například možné vytvářet rozhraní (interface) ve smyslu Javy nebo i PHP. To samé se týka abstraktních tříd. Rozšiřování aplikace o další moduly tak není tak snadné, jak by mohlo teoreticky být.

Dalším problémem je to, že JavaScript není primárně objektový jazyk, ale spíš procedurální a některé konstrukce se vytváří hodně neohrabaně. Problém, se kterým jsem se často potýkal, je ztráta kontextu objektu. Nebylo tak možné přímo volat metody objektu, i když se zrovna prováděl kód v jiné z jeho metod. Toto se stávalo hlavně při obsluze asynchronního volání, kdy si metoda musela získat svého vlastníka z globálního kontejneru, kam byly všechny velké třídy (Application, Sync, Viewer a Model) ukládány.

Neduhem aplikací napsaných v JavaScriptu je také přehršel funkcí, které je nutné zakládat velmi často a celý kód se tak znepřehledňuje kvůli velkému množství závorek. Tento problém by částečně mohla vyřešit knihovna CoffeeScript, která používá hlavně odsazování a spousty závorek nepotřebuje, protože je schopná si je "domyslet". Bohužel je určena hlavně pro Linux a její zprovoznění na Windows se ukázalo jako velmi problematické. Také by to znamenalo nutnost učit se novou syntaxi a to by vývoj pravděpodobně zpomalilo.

7.1.3 Nové funkce do budoucna

Primárním cílem této práce bylo potvrzení výše zmíněných teorií a splnění zadaných user stories. Aplikace ale určitě není dokonalá a spousta užitečných funkcí by se dala doimplementovat. Mezi ně určitě patří nějaká vyrovnávací paměť mezi aplikací a databází, která se ukázala jako dobrý nápad do budoucna během zátěžových testů. Další funkce, na které se přišlo během vývoje a které byly zařezeny do kategorie "hezké mít" (nice-to-have) jsou tyto:

- sledování i cizích projektů
- vytváření vlastních přehledů úkolů (kombinace štítků a projektů)
- automatické sledování commitů do synchronizovaného repozitáře
- modulárnější vizuální stránka s využitím knihovny MustacheJS

Tyto funkce se ukázaly buď jako obtížně realizovatelné (vyrovnávací paměť, vytváření vlastních přehledů) nebo poměrně zbytečné (MustacheJS, sledování commitů), ale přesto byly uloženy, aby se na ně nezapomnělo. Plánuji se do budoucna vývoji v Titanium Studiu dále věnovat a možná dojde i na rozšíření této aplikace.

7.2 Závěr

Výsledkem této práce je nástroj, který v sobě integruje správu tří verzovacích serverů. Protože se poskytovaná API hodně liší, byla nutná spousta kompromisů pro zachování konzistence ovládání aplikace. Tím je sice uživatel ochuzen o několik extra funkcí poskytovaných těmito servery, ale to hlavní je v aplikaci umožněno. Na čem by se dalo určitě ještě zapracovat je vizuální stránka aplikace, která dost často rozhoduje o úspěchu aplikace. Bohužel jako programátor nemám moc velké výtvarné nadání a barevné cítění.

Důležitou součástí vývoje software je testování ve všech různých podobách. Při vývoji tohoto nástroje to nebylo jinak. Během vývoje byly prováděny jednotkové (unit) testy, po dokončení implementační fáze bylo provedeno zátěžové testování výpisu úkolů, což se ukázalo jako časově velmi náročná operace. Úplně nakonec byly provedeny akceptační testy na základě sepsaných user stories. V původním plánu bylo i provedení testu s uživateli, ale na ten už bohužel nezbyl čas.

7.3 Kódy programu

Chceme-li vysázet například část zdrojového kódu programu (bez formátování), hodí se prostředí `verbatim`:

```
         (* nickname2 *)
Lego> Refine in1
             (do_reg (nickname1 h));
Refine by  in1 (do_reg (nickname1 h))
   ?4 : pcdata
   ?5 : pcdata
          (* surname2 *)
Lego> Refine surname1 h;
Refine by  surname1 h
   ?5 : pcdata
          (* email2 *)
Lego> Refine undo_reg (email1 h);
Refine by  undo_reg (email1 h)
*** QED ***
```

7.4 Další poznámky

7.4.1 České uvozovky

V souboru `k336_thesis_macros.tex` je příkaz `\uv{}` pro sázení českých uvozovek. „Text uzavřený do českých uvozovek.“

Kapitola 8

Seznam použitých zkratek

IDE Integrated Development Environment

GTD Getting Things Done

API Application Programming Interface

JSON JavaScript Object Notation

REST Representational State Transfer

HTML HyperText Markup Language

RUP Rational Unified Process

UML Unified Modeling Language

⋮

Kapitola 9

UML diagramy

Tato příloha není povinná a zřejmě se neobjeví v každé práci. Máte-li ale větší množství podobných diagramů popisujících systém, není nutné všechny umísťovat do hlavního textu, zvláště pokud by to snižovalo jeho čitelnost.

Kapitola 10

Instalační a uživatelská příručka

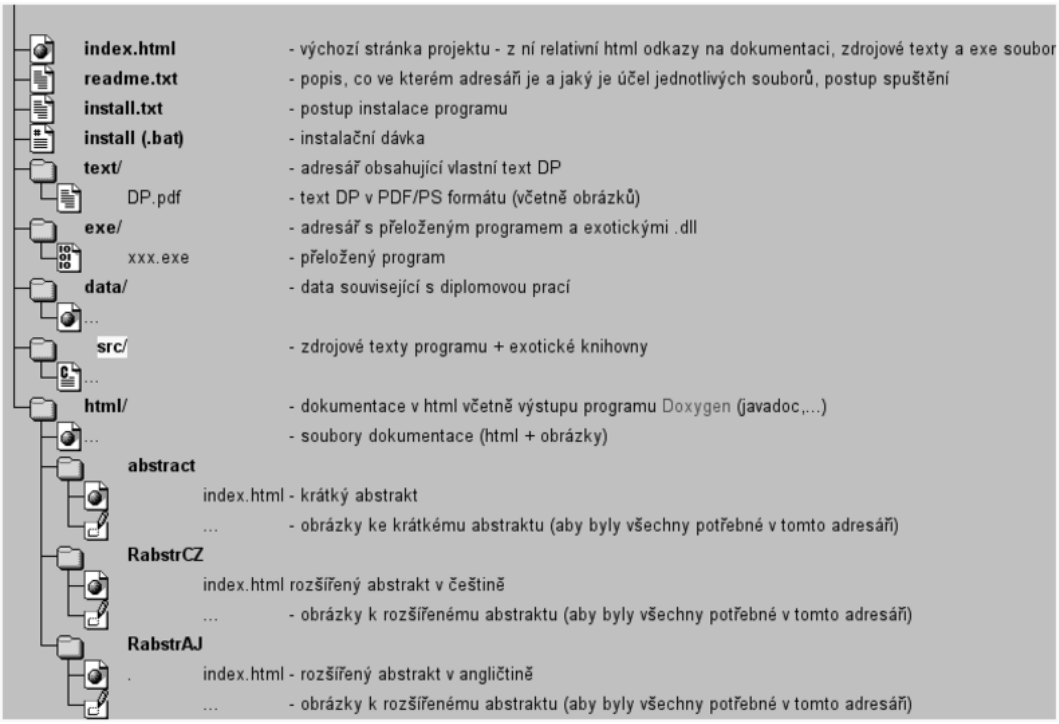
Tato příloha velmi žádoucí zejména u softwarových implementačních prací.

Kapitola 11

Obsah přiloženého CD

Tato příloha je povinná pro každou práci. Každá práce musí totiž obsahovat přiložené CD. Viz dále.

Může vypadat například takto. Váš seznam samozřejmě bude odpovídat typu vaší práce. (viz [**?**]):



Obrázek 11.1: Seznam přiloženého CD — příklad