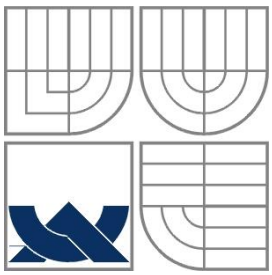
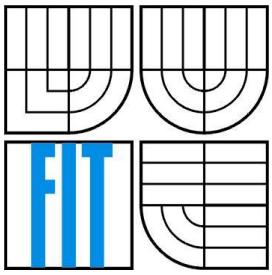


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



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

METODY A ORGANIZACE TESTOVÁNÍ SOFTWARE

SOFTWARE TESTING ORGANIZATION AND METHODS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. MIROSLAV KAJAN

BRNO 2015

VEDOUCÍ PRÁCE
SUPERVISOR

doc. JITKA KRESLÍKOVÁ, CSc.

BRNO 2015

Abstrakt

Tato práce přednáší návrh na zlepšení organizace a metodik testování za pomoci studia nejrůznějších přístupů a následné aplikace v reálném prostředí firmy Siemens CZ. V první části práce rozebírá vývojové modely z hlediska segmentu testování a v části následující se zaměřuje speciálně na metodiky agilního testování a vývoje jako celku včetně použitelných metrik pro evaluaci jednotlivých aspektů software vystupujícího z fáze testování. V třetí části práce seznamuje čtenáře se způsobem fungování organizace ve firmě Siemens a analýzou reálného projektu hodnotí pozitiva i nedostatky testovacího procesu a jeho řízení. Praktická část této práce spočívá v návrhu a implementaci zásuvného modulu pro prostředí JIRA. Nástroj dokáže přehlednou a interaktivní formou v dlouhodobém měřítku pojmenovat a vyčíslit status projektových fází agilního vývoje a testování, přičemž výstupem jsou jasně definované problémy, které lze pak snadněji ošetřit a argumentovat další postup.

Abstract

This document's objective is to propose a set of methods for improvement of the organization and methodology of software testing and subsequent application of those in a real environment of the Siemens CZ company. The first part discusses the development models in terms of testing segment and the following section focuses specifically on the methodology of agile testing and development as a whole, including applicable metrics for evaluating various aspects of software exiting the testing phase. The third major section lets the reader get acquainted with the Siemens organization and methods of their software testing and by analyzing real-world project it assesses the strengths and weaknesses of the particular testing process and its management. The practical part of this thesis lies in the design and implementation of a plug-in for the JIRA environment. This tool is able to identify and quantify the long term status of the project phases of agile development and testing in a clear and interactive way, while the outputs are represented by clearly defined problems that can be more easily treated, as one can argue further progress.

Klíčová slova

testování softwaru, metodika vývoje, vývojový model, vývoj řízený testováním, ADD, agilní metodika, Scrum, JIRA, automatizace testování, zásuvný modul, optimalizace testování, optimalizace projektu, evaluace projektu, datová analýza, automatizace testování, procesní řízení

Keywords

software testing, development methodology, development model, driven development testing, ADD, Agile, Scrum, JIRA, testing automation, plug-in, testing optimization, project optimization, project evaluation, data analysis, automation testing, process management

Citace

Kajan Miroslav: Metody a organizace testování software, diplomová práce, Brno, FIT VUT v Brně, 2015

Metody a organizace testování software

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením doc. RNDr. Jitky Kreslíkové, CSc.
Další informace mi poskytl Ing. Jakub Řezáč z brněnské divize firmy Siemens CZ.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Miroslav Kajan
20.5.2015

Poděkování

Tímto děkuji za cenné rady z dlouholeté praxe doc. RNDr. Jitky Kreslíkové, CSc. a Ing. Jakubovi Řezáčovi z brněnské divize firmy Siemens CZ za kvalitní průpravu organizací a procesním řízením v rámci zvoleného předmětu diplomové práce.

© Miroslav Kajan, 2015.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

1 Úvod

Pokročilá doba uvedla do chodu řadu technik v oblasti vývoje software, které jednak inovují či upravují metody staré, nebo reprezentují skutečně nový pohled na již tak komplexně procesovou disciplínu, jakou vývoj software bezesporu je. S obecným zkvalitněním vzdělávacího systému a turbulencí dnešního světa se na vývojářské pozice dostávají lidé s markantnější dostupností k informacím, ale menší porcí času než kdy předtím. Důsledkem toho je dlouhá řada programátorů či softwarových architektů, kteří pracují rychle, efektivně, ale stále ne bezchybně. Dalším problémem samotného vývoje je strana zákazníka, která si s vědomím vlády nad celým procesem poroučí splnitelné i nesplnitelné, a to pravděpodobně ještě na poslední chvíli.

Do popředí softwarového procesu se tak i díky těmto problémům dostává pozice obecně využívaná jako poslední před odevzdáním výsledného produktu zákazníkovi – pozice testerů. Rozvoj nejrůznějších testovacích metodik, prostředí, automatizačních softwarů a softwaru pro organizaci testování se dostal do fáze, kdy se i dobrý tým vývojářů může cítit v jistých aspektech své práce ztracený.

Předmětem tohoto projektu je v první řadě nabídnout ucelený pohled na metody a typy organizace softwarového testování s ukázkou škály možností a doporučení. V druhé řadě pak práce zahajuje kooperaci s konkrétním vývojářským týmem a v rámci získaných znalostí provádí experimentální zhodnocení reálného projektu, zejména z hlediska testovacích metrik se snahou navrhnout vylepšení firemních testovacích procesů jako celku.

Praktická část této práce spočívá v návrhu a implementaci zásuvného modulu UpAnalyzer, který slouží k sledování definovaných metrik a celkového vyhodnocení projektové práce testovacího týmu brněnské divize firmy Siemens. Nástroj dokáže přehlednou formou v dlouhodobém měřítku pojmenovat a vyčíslit status projektových fází agilního vývoje a testování, přičemž výstupem jsou jasně definované problémy, které lze pak snadněji ošetřit, argumentovat další postup a nalézt odpovídající množství nutných investic.

Nástroj tedy dokáže nejen analyzovat a vizualizovat data do přehledného a interaktivního reportu, ale také upozornit na vznikající problém a napomoci nalezení jeho podstaty. Mezi dodatečná rozšíření modulu patří možnost vlastního nastavení striktnosti hodnocení projektu dle různých metrik, a dále částečně automatický modul pro vytvoření vlastní metriky či analýzy.

2 Metodiky testování softwaru

Testování softwaru dle odborné literatury představuje technický výzkum kvality testovaného produktu nebo služby prováděný za účelem poskytnutí těchto informací všem zainteresovaným stranám [1].

Proces testování je podmnožinou procesu ověřování a plánování kvality. Proto mohou být úkoly testovacího týmu dosti široké a na modelech životního cyklu pozorujeme, že testovací disciplína nejen prokládá fáze celého vývoje, ale často nahrazuje zajišťování kvality. Součástí zjišťování informací o kvalitě je reportování nalezených problémů či chyb.

Proces testování začíná stanovením vize a cílů testování. Dále se určí rozsah testování, tedy co vše je třeba testovat, vybírají se testy, sbírají data a připravují nástroje, které tým k testování potřebuje. Samotné testování probíhá zkoumáním produktu na několika úrovních a reportováním nalezených skutečností.

2.1 Životní cyklus vývoje softwaru

Pro bezprecedentní snahu o poznání jednotlivých metod a organizací testování je nutné se předem podívat na pojem obecnější – metodiky vývoje softwaru. Pojem metodika označuje nějaký pracovní postup, způsob provádění nějaké činnosti. Pokud se mluví o metodikách testování, z velké části se tím myslí metodika vývoje softwaru, jejíž součástí je i testování.

Snahou kvalitní softwarové metody je vytvořit rámec, v němž jsou definovány jednotlivé činnosti, jejich návaznosti, přesahy a zodpovědnosti. Proces vývoje by měl být schopen každému aktérovi projektu sdělit co je jeho úkolem, co musí být splněno, aby mohl zahájit svou činnost, jaký je její očekávaný výstup a komu má být předán.

2.1.1 Kaskádový přístup

Jedná se patrně o nejznámější metodu, jejíž principem je postupné navazování jednotlivých fází vývoje, přičemž každá další fáze začíná v okamžiku, kdy ta předcházející skončí. Nejobvyklejší fáze vývoje softwaru můžeme sepsat do následujícího seznamu: Sběr dat, Analýza, Vývoj, Testování. Testování tedy začíná až v okamžiku, kdy je dokončen Vývoj aplikace.

V této metodice je možný pohyb pouze ze shora dolů. Pokud v dané fázi byly vykonány všechny činnosti, které vykonány být měly, a byly dodány všechny požadované výstupy, považuje se tato fáze za úspěšně ukončenou.

Tato metoda se může na první pohled jevit jako snadno použitelná a především odpovídající skutečnému postupu vývoje. Navíc umožňuje snadnější plánování zdrojů i termínů. Díky rozsáhlé dokumentaci, kterou tato metoda předpokládá, se zdá, že i kontrola správnosti vývoje musí být relativně snadná.

Každá fáze pracuje s výstupy fáze předcházející a musí předpokládat, že tyto výstupy jsou správné. Přestože tato metoda podporuje kontrolní mechanismy právě na úrovni přechodů mezi fázemi, tyto kontroly jsou díky povaze metody spíše zaměřené na formální stránku věci. Zákazník do procesu na jedné straně vloží své požadavky a na druhé straně získá hotovou aplikaci. Pokud své požadavky nespecifikoval dostatečně přesně, nebo na některé zapomněl, pak výsledná aplikace neodpovídá tomu, co očekával. Celý proces vývoje v tuto chvíli musí začít znovu. Podobně je tomu v okamžiku, kdy zákazník začne svoje požadavky modifikovat tehdy, kdy vývoj již opustil fázi sběru požadavků. Díky nepružnosti metody dochází v takovéto situaci k „zahození“ dosavadních výsledků a proces začíná od začátku. Testování tu spíše než úlohu garanta kvality plní úlohu ověření, že výsledek je správně.

2.1.2 Iterativní přístup

Iterativní přístup se snaží vylepšit nedostatky přístupu kaskádového. Každá iterace produktu se vyvíjí jako samostatný proces se všemi fázemi, které vývoj předpokládá. Proces vývoje jedné iterace násobený počtem iterací se pak rovná vývoji aplikace jako celku. Největší plusové body získává iterativní přístup na faktu, že lze na reálném výstupu mnohem dříve ověřovat, zda odpovídá očekávání. Obvyklý postup u iterativního přístupu představuje rozšiřování funkčností.

Nejznámější implementací iterativního přístupu je tzv. spirálový model. Ten rozlišuje základní čtyři činnosti: Analýzu, Hodnocení rizik, Vývoj/testování, Plánování další iterace. Tyto činnosti je možné vynést na osy grafu. Vývoj začíná v bodě nula, kdy neexistuje aplikace a začíná se sběrem požadavků. Každý posun do další fáze znamená růst aplikace. Při několika iteracích a jejich grafickém znázornění vývoje aplikace získáme spirálu.

2.1.3 Agilní metodiky

V dnešní době se jedná o pravděpodobně nejprosazovanější řešení ukryté pod obecnějším obalem. Jde o další vývojový stupeň navazující na předchozí přístupy. Snahou je minimalizovat formálnost ve vývoji softwaru a tím tento vývoj zrychlit. Základ agilní metodiky tvoří krátký iterační

cyklus a vysoká úroveň komunikace se zákazníkem. Agilní metodika je zaměřená na malé týmy. Cílem veškerého snažení v rámci agilních metodik je spokojenost zákazníka a proto důležitou roli hraje právě komunikace se zákazníkem a jeho zapojení do procesu vývoje.

Funkční vlastnosti, jak už z jejich názvu vyplývá, se týkají samotného účelu testované aplikace. Testování funkčních vlastností má ověřit, že aplikace správně vykonává úkoly, pro které byla vytvořena. To ale neznamená pouze to, že při korektním zacházení vykoná odpovídající operace. Testuje se také to, že aplikace je schopná vypořádat se i s nekorektním chováním ze strany uživatele. Jinak řečeno, že obsahuje správně implementované validace uživatelských vstupů a že případný nevalidní vstup vyvolá předem definovaný chybový stav (GUI testy nebo tzv. *smoke* testy, bude o nich dále ještě psáno).

Jako nefunkční (kvalitativní) vlastnosti aplikace můžeme označit všechny ty, které se týkají její instalace, výkonu, dostupnosti a podobně. Tento typ testování odpovídá na otázky, jako zda nárůstem objemu zpracovávaných dat není negativně ovlivněn výkon aplikace, nebo zda je vhodné vzhledem k důležitosti aplikace implementovat bezpečnostní testy, jejichž úkolem je ověřit odolnost aplikace proti vnějším útokům. Následují příklady nejpoužívanějších agilních metodik dneška:

- **Scrum** – tato metodika bude vzhledem k její důležitosti a rozsahu pro tuto práci popsána v následující podkapitole.
- **Extrémní programování** – jedná se o jednu z nejpoužívanějších agilních metodik. Pro tuto metodiku jsou běžné časté dodávky výstupu v krátkých cyklech. Vedlejší aktivitou definující tuto metodiku je pak tzv. párové programování, tzv. *test-first development*[1] (nejdříve se píše test a následně až daná funkčnost) a *refaktoring*[2].
- **Vývoj řízený testy** – neboli také *test-driven* vývoj (angl. zkr. **TDD**) navrhuje psaní testů před samotným kódem a následně naprogramovat samotný kód. Implementuje se přesně takové množství kódu, jaké dokáže projít testem.
- **Vývoj řízený vlastnostmi (FDD – Feature Driven Development)** – FDD začíná vytvořením doménového modelu popisujícího celý systém. Ten se převede do seznamu vlastností (elementární funkcionality, které přináší hodnotu uživateli). Vývoj má celkem pět fází (první tři sekvenční, další dvě iterativní). Iterace trvá většinou dva týdny. Během každé iterace se implementují konkrétní užitné vlastnosti systému. Zákazník průběžně dostává mezivýsledky a nové verze produktu. Na rozdíl od XP nebo SCRUM je jednotlivým programátorům práce přidělena – nevybírají si ji sami.

[1] nejdříve se píše test a následně až daná funkčnost
[2] častý návrat programátora k úpravě kódu tak, aby byl co nejefektivnější a nejsrozumitelnější

- **Vývoj a design řízený atributy (ADD – Attribute Driven Design) –** ADD představuje metodiku založenou na kvalitativních atributech softwaru. Počítá se vstupy v podobě funkcionálních požadavků, nutných kvalitativních požadavků a výjimek v návrhu (tzv. *design constraints*). V podstatě se nejedná čistě o metodiku vývoje, ale spíše o metodiku architektonického návrhu [16]. V současnosti však bývá častokrát zařazována do různých etap vývojového cyklu. Podstata metody tkví v rekurzivním procesu dekompozice – v každé fázi se nastaví množina scénářů plnění kvalitativních atributů a popíší se metody, jak tohoto plnění dosáhnout. Výstupem metodiky by pak měla být řada úrovní dekompozice včetně různých pohledů na systém.

2.1.3.1 Metodika Scrum

Metodika Scrum se v současné době řadí mezi nejpoužívanější agilní metodiky, není-li nejpoužívanější agilní metodikou vůbec. Klade důraz na pružné reakce na změny, průběžnou a zevrubnou analýzu rizik a jakousi volnost ve využívání zásad zvyklostí a know-how v daném konkrétním týmu, zejména díky které vešel ve všeobecnou oblíbenost. Má propracovaný způsob odhadu náročnosti jednotlivých problémů (známých spíše pod anglickým označením *issues*).

Mezi nevýhody využití metodiky Scrum patří fakt, že se spíše jedná o souhrn úspěšně použitých vzorů než přesnou specifikaci kroků. Jednou z možností je tak například situace, kdy tým vychází z existující metodiky ve firmě a Scrum používá při pouhém vedení projektů. Další z toho plynoucí nevýhoda značí, že Scrum bývá v drtivé většině nasazován pro malé týmy a nepříliš rozsáhlé projekty.

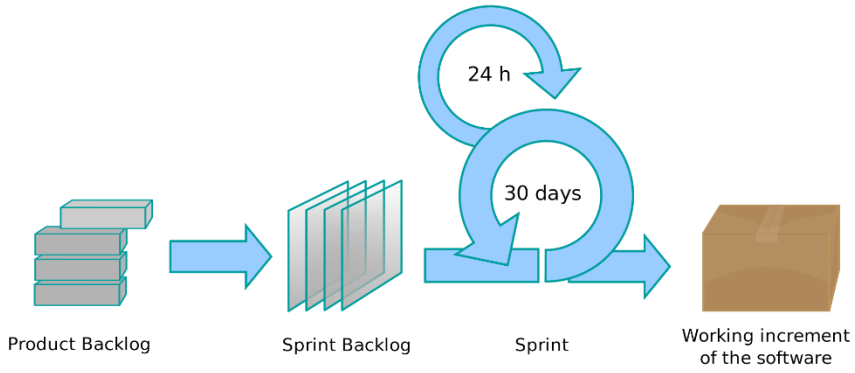
Jeden člen Scrum týmu se stává tzv. ScrumMaster a pozicí se blíží běžnému projektovému manažerovi. Další významnou osobou je ProductOwner, který zastupuje pozici zákazníka, který si daný produkt objednal. Ostatními členy týmu jsou vývojáři a testeři přiřazení na vývoj daného software**.** Důraz kladený na týmovou spolupráci je zjevný, nicméně každý člen má svou jasně definovanou volnost**.**

V souvislosti s touto metodikou nastává potřeba zmínit několik pojmů, které budu dále ve své práci využívat:

- *Sprint* (někdy také iterace) je krátký časový úsek (často uváděno mezi dvěma až čtyřmi týdny, ale je to stanovení ryze individuální), ve které vývojářský tým implementuje a otestuje jasně definovanou část výsledného produktu (milník).
- *Backlog* je seznam chyb, vlastností a úkolů produktu nebo sprintu. Rozlišujeme dva typy těchto seznamů – produktový backlog, jež obsahuje skutečně všechny požadavky na jednom místě bez

větší hierarchie, a sprintový backlog, který vymezuje požadavky pouze v rámci konkrétního sprintu.

- *Story points* jsou body, které hodnotí komplexitu daného záznamu (chyby, úkolu a podobně). V zásadě se při určování míry neužívá posloupnost lineární nýbrž Fibonacciho, a to zejména pro jasné a zřetelné oddělení důležitosti (hodnocení 13 a 8 působí na člověka extrémněji než 5 a 4). Někdy se tyto body přidělují individuálně každým členem týmu a následně se rozhodne o jeho skutečné důležitosti. Předchází se tak „kopírování“ názoru kolegů a podporuje se tak přístup samostatného názoru a jedinečnosti osoby v týmu.



Obrázek 1: Proces metodiky Scrum (převzato z [8])

Začátek každého sprintu začíná výběrem požadavků, které se budou v dané iteraci plnit. Každý den si tyto požadavky ujasní celý tým v krátké konferenční poradě. Prostor dostává každý člen týmu a řekne ostatním, na čem minulý den pracoval a na čem bude pracovat dnes.

Naprosto hlavní výsada agilní metodiky Scrum spočívá v tom, že se stále jedná o iterativní vývojovou metodu, a proto na konci každého sprintu tým nutně odevzdává funkční produkt, který sice neimplementuje všechny požadavky, ale přináší zákazníkovi alespoň částečnou možnost nasazení.

2.2 Typologie testování

V této kapitole se zaměřím na charakterizaci nejpoužívanějších typů testů při vývoji software. Je nutné upozornit, že se stále orientuji na testování jako obecnou položku při vývoji software; v tomto výčtu se tak vyskytují i položky, které si při konkrétních technikách agilního testování nemusí nutně využívat. Rozsah typologie v této práci omezuji na následující podskupiny:

- Testy z hlediska chronologizace
- Testy dle přístupu ke kódu

- Testy dle běhu programu

2.2.1 Testy z hlediska chronologizace

Nejčetnějším pohledem využívaným na kategorizaci softwarových testů je pohled časový, tzn., kdy se daná testovací aktivita z pohledu vývojového modelu nastává. Jednotlivé typy testování budu řadit chronologicky od prvotního až do posledního.

2.2.1.1 Testování vývojářem

Při vývoji jsou prvotní testy označovány jako *assembly* testy a dle literatury [3] je provádí sám vývojář. Doba agilních metodik však pokročila k tomu, že spíš nastává připodobnění situace k párovému programování, tzn. situaci, kdy jeden programátor kontroluje kód druhého a naopak. Veškerá kontrola tak probíhá na úrovni zdrojového kódu. Při *assembly* testování může docházet k nejrůznějším druhům latence či dokonce zavádění dalších chyb do kódu, ať již je to dáno nejednotným přístupem k psaní kódu, nepozorností, či nedostatkem času. Obecně platí, že pozice testera je k tomuto určená a sám tester by měl mít přinejmenším slušné znalosti jazyka, v němž je software tvořen, a proto se někdy *assembly* testování z testovací hierarchie úplně vylučuje. Takovéto postupy zbytečně zabírají čas i ostatním členům týmu. Pro kvalitní *assembly* testování je potřeba mít mezi programátory ucelený přístup k psaní kódu.

2.2.1.2 Testování jednotek

Po ověření kódu programátorem přichází na řadu test jednotek, tzv. *unit testing*. U objektově orientovaného programování se jedná o testování jednotlivých tříd a metod. Testovanou jednotkou v tomto případě rozumějme samostatně testovatelnou část aplikačního programu. Testy těchto jednotek se zapisují ve formě programového kódu. Proto jej z pravidla obsluhují vývojáři. Pro vytváření testů se využívá nástrojů na bázi různých rozhraní (tzv. *frameworks*). Obecně platí, že testování jednotek nelze aplikovat na běžící projekty. Jedná se o formu testování, která přestála veškerý vývoj agilních metodik bez větší změny, a to zejména proto, že má při test-driven vývoji význačnou a přední pozici z hlediska testovacího ROI[3].

Testy jednotek se velmi špatně aplikují na již zaběhlých projektech. U již vytvořených aplikací se většinou musí provést kompletní refaktoring kódu či dokonce mnohem hlubší úpravy. Takováto

[3] Význam ROI (Return Of Investment) se ve spojitosti s testováním používá jako metrika vyjádřená poměrem průměrného času stráveným tvorbou testu k průměrnému počtu chyb nalezených tímto testem.

časová investice se u menších projektů většinou nevyplatí, ale ani u velkých projektů takovýto zásah není příliš šťastný a často se nesetkává s podporou u vedoucího projektu. Proto je vhodné zabývat se těmito testy již v etapě návrhu aplikace a v té době se rozhodnout, zda tyto testy budeme využívat. Jelikož obecně platí, že čím dříve (v rámci životního cyklu software) chybu nalezneme a opravíme, tím méně času nad touto opravou strávíme. Proto se obecně doporučuje této úrovni věnovat maximální pozornost a to již před samotným vývojem aplikace.

2.2.1.3 Integrační testování

Integrační testy nepřipravuje programátor, ale výhradně testovací tým. Jedná se o sadu testů, které zkoumají správnost vnitřní integrity aplikace. Testuje se převážně bezchybná komunikace mezi jednotlivými komponentami uvnitř aplikace. Integraci však lze ověřovat nejen mezi komponentami, ale také mezi komponentou a operačním systémem, hardwarem či rozhraním různých systémů. V této fázi se tak testuje integrace dosud jednotlivě ověřených částí. Začínáme testovat integraci mezi dvěma komponentami a postupně přidáváme další. Integrační testy mohou být jak manuální, tak i automatizované.

Úroveň integračního testování je svým způsobem obsažena ve většině testovacích postupů softwaru. U menších projektů je však na tyto testy kladen velmi malý důraz. Má to své logické odůvodnění. Integrační testování lze v testovacím cyklu zcela vynechat. Na výslednou bezporuchovost softwaru to přitom nebude mít žádný vliv, tedy alespoň za předpokladu, že korektně provedeme následující úrovně testování. Chyba, kterou bychom odhalili během integračních testů, se zcela jistě projeví v průběhu dalších úrovní testování. Jak již bylo zmíněno dříve, během testování však platí: „čím dříve chybu objevíme, tím méně úsilí nás stojí její oprava“. Proto integrační testy mají svůj význam, nicméně nelze jejich použití nikterak přeceňovat.

2.2.1.4 Systémové testování

Spojení integračních i systémových testů je označována jako fáze SIT – *System Integration Tests*. Po ověření správné integrace nastává ten pravý čas na systémové testování. Během těchto testů je aplikace ověřována jako funkční celek. Tyto testy jsou používány v pozdějších fázích vývoje. Ověřují aplikaci z pohledu zákazníka. Podle připravených scénářů se simulují různé kroky, které v praxi mohou nastat. Nalezené chyby jsou opraveny a v dalších iteracích jsou tyto opravy opět otestovány. Součástí této úrovně jsou jak funkční tak nefunkční testy. Poslední úroveň testů, které se provádějí před předáním produktu zákazníkovi, jsou tedy systémové testy. Tato úroveň testů tak většinou slouží jako výstupní kontrola softwaru. Systémové testování je obsaženo prakticky v každém procesu testování. Bez této úrovně by celé testování softwaru nemělo žádný význam. Bezporuchovost výsledného produktu by byla

významně ohrožena. Proto tuto úroveň testů považuji za stěžejní v celém postupu testování software. Na realizaci těchto testů by se mělo myslet již v raném stádiu návrhu postupu testování, tak aby bylo možné obsah testů co možná nejvíce přizpůsobit očekávanému softwaru.

2.2.1.5 Akceptační testování

Akceptační testy probíhají na straně zákazníka. Pokud všechny předchozí etapy testů proběhly bez větších nedostatků, je možné předat aplikaci zákazníkovi. Ten se svým týmem testerů provede akceptační testy dle připravených scénářů, které připravil zákazník s dodavatelem. Testy probíhají na testovacím prostředí u zákazníka. Nalezené nesrovnalosti mezi aplikací a specifikací, jsou reportovány zpět vývojovému týmu. Otázkou častého řešení bývá způsob, jak zabezpečit opravení chyb v co nejkratší době. Velké prodlevy v nalezení a opravení chyby v uvedené úrovni testování mohou vést ke zpoždění termínu nasazení softwaru do provozu, což se leckdy pro projekt ukazuje jako fatální.

Specifikem akceptačního testování může být jen ujištění, např. že aktuální verze není vysloveně chybná, a to na základě několika fundamentálních scénářů („nástřelů naslepo“, tzv. *smoke* testů). V případě pochybností se přikládá v závěru důraz na testy regresní. Ty ověřují širokou škálu testů, i u méně často používaných funkcí. V případě využívání iterativní testovací metodiky představují regresní testy naprostou nutnost, kdy se testují i testy z předchozích iterací, které byly úspěšně složeny (ale v aktuální verzi tomu tak být nemusí).

2.2.2 Testy dle přístupu ke kódu

Dalším hlediskem, dle kterého lze testy rozdělit, je hledisko viditelnosti zdrojového kódu ze strany samotného testera. V praxi se běžně používají tři následující kategorie testů:

- **„*Black box*“ testování** – při použití těchto testů nemá tester žádný přístup k programovému kódu. Pro tento případ si produkt můžeme představit jako černou skříňku, jejíž obsah není pro pracovníka viditelný – tester neví, jak přesně systém pracuje s daty. Jediné, čím se při takovém testování zodpovědný pracovník zabývá, je sledovat a hodnotit jaký výsledek získáme po vložení různých vstupních dat.

- **„*White box*“ testování –** v tomto případě zná tester vnitřní strukturu software. Mimo vhodnosti výstupů se testují průchody zdrojovým kódem, reakce kódu na prázdné nebo nestandardní hodnoty a podobně.

- **„*Grey box*“ testování –** jedná se o kombinaci předchozích dvou kategorií. Může se jednat o situaci, kdy software testujeme přes uživatelské nebo API rozhraní.

2.2.3 Testy dle běhu programu

Posledním hlediskem, které v této práci uvádím, je hledisko rozdělení dle nutnosti běhu programu. Tímto chci upozornit, že samozřejmě existuje řada dalších rozdělení, tyto dvě uvedené se však využívají nejvíce a poměrně dobře charakterizují testování „ze všech stran“.

- **Statické** testování – v zásadě nevyžaduje běh programu. Využívá se v prvotních fázích životního cyklu software. Používá se na kontrolu specifikace požadavků a rutinních revizí/kontrol kódu a ze statické analýzy kódu.
- **Dynamické** testování – vyžaduje běh aplikace (spustitelný software). Využívá se v pozdějších fázích vývoje a je zaměřeno na bezproblémovost provozu. Právě v souvislosti s dynamickým testováním můžeme mluvit o testování černou skříňkou a dalších výše zmíněných metodách.

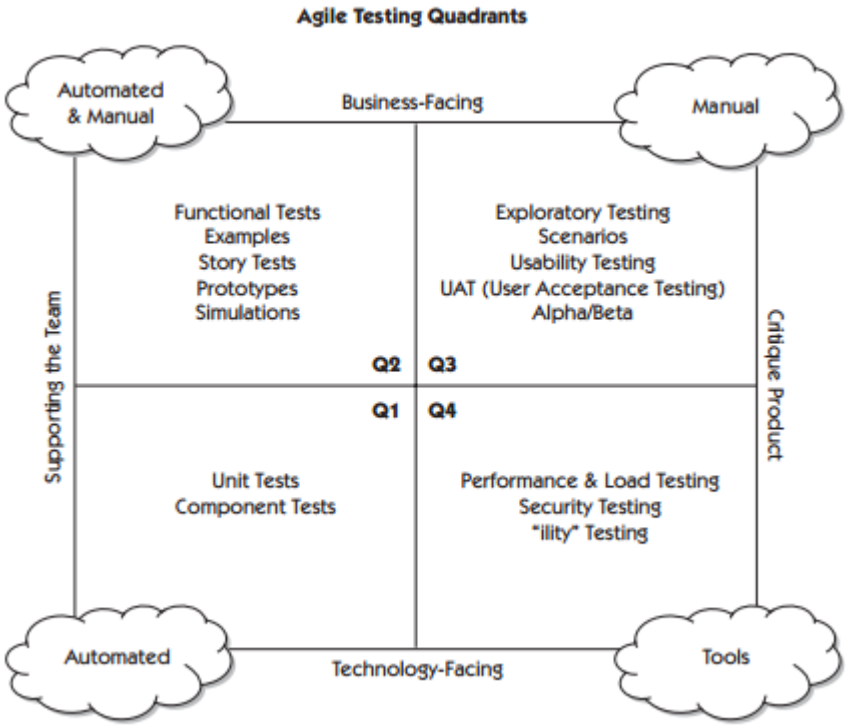
2.3 Metody testování při agilním vývoji distribuovaných systémů

Metody testování s využitím moderních agilních technik přinášejí jisté a leckdy nemalé změny do celého procesu. Tato kapitola se zabývá právě těmito výjimkami a předpokládá znalost typologií testů z kapitoly předchozí.

2.3.1 Chronologizace testů používaných u agilních metodik

Testování programátorem obecně odpadá a programátor soustředí své úsilí na psaní kódu, který je efektivní. Jednotkové, systémové a integrační testování je zde považováno za jednotný celek na různých úrovních. Obecně se tyto tři kategorie shrnují do pojmenování testy komponent.

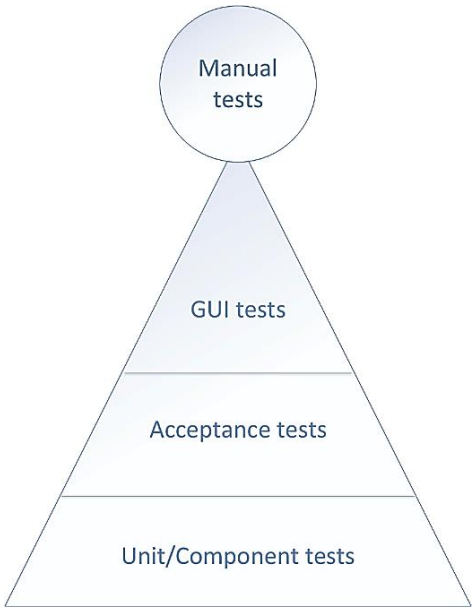
Pojem agilního testování funguje dokonce osamoceně. Tzv. explorační testování (angl. *exploratory testing*) je striktně definováno jako agilní typ testu a může být využito i v neagilním prostředí. Na straně zákazníka existují rovněž testeři, kteří připravují akceptační testy a úzce spolupracují s testery vývojářského týmu. Obecně nejlépe vysvětluje principy agilních testů koncept, jež byl ve své době poměrně převratnou sumarizací všech typů agilních testů:



Obrázek 2: Princip agilních testovacích kvadrantů (převzato z [1])

Na jedné ose rozděluje Crispin a Gregory [1] testy na ty, které podporují práci týmu, a na ty, které konstruktivně kritizují vzniklý produkt. Horizontální osa pak dělí testy na byznys testy a technologické testy. Z obrázku lze jednoduše vyčíst, že všechny testy podporující práci týmu mohou být automatizované. Čtvrtý kvadrant pak zahrnuje sadu utilit, které je možné rovněž automatizovat. Jediným neautomatizovaným kvadrantem v této agilní metodice zůstává kvadrant č. 3.

Jeden z protinázorů plynoucí z článku dalšího z odborníků na agilní metodiky Adzice [13], rozebírá aktuální vztah k testovacím kvadrantům. Dle autora se díky velkým změnám v informatických konceptech během posledních pěti let (obsese projektových manažerů pojmem analýza velkých dat – potažmo angl. *big data analysis*, kontinuální integrace a doručení produktu zákazníkovi prakticky v jakémkoli momentu a podobně) nedá nad čtyřmi kvadranty testů uvažovat stejně. Separace testů dle vertikální osy je stále dobře použitelná (byznys testy versus technologické testy), ale například filosofie metodiky TDD (a její konkrétní odnož s anglickým názvem *acceptance test-driven development*) prolíná jednotkové testování se akceptačním testováním příběhu (angl. *story acceptance testing*).



Obrázek 3: Princip pyramidy agilního testování (koncepčně převzato z [1])

Pokud se tým rychle sžije s principy *test-driven* vývoje, stává se pro něj základní stavební kámen pyramidy testů tím časově nejméně náročným a nejefektivnějším testovacím prostředkem. Někdy do základních testů můžeme zahrnovat i ty byznysové, pokud se jedná o takové funkcionality, které sám zákazník nemusí vidět. Druhý stupeň pyramidy obsahuje testy aplikační vrstvy, kterými si odpovídáme na otázku: *„Tvoříme vlastně tu správnou věc?“*. Třetí stupeň testuje viditelnou stránku věci – uživatelské rozhraní.

2.4 Testovací metriky

Metriky mohou být kontroverzním tématem a vývojářský tým může jejich probíráním strávit hodně času. Je třeba definovat, do jaké míry je dolování metrik plýtváním týmového úsilí. Primární vlastností metrik však bývá určovat aktuální stav produktu, měření pokroku vývojářského týmu a dosažení cílů. Někdy se můžeme setkat s názorem, že pokud je chyb při analýzách mnoho, je dobré doplnit tým o další vývojáře a opačném případě lze přesunout kapacity na jiné projekty. Tento názor je poměrně jednostranný a nikterak nezohledňuje to, že by se programátoři měli maximálně soustředit na to, aby se chyby v kódu vůbec nevyskytovaly. Z hlediska kategorizace testů se můžeme dostat až do příliš složité situace. Některé zdroje dělí metriky prvotně dle toho, kde se provádějí jednotlivá měření:

- metriky na zdrojovém kódu
- metriky na testech
- metriky testováni na chybách

V následujícím textu rozeberu některé konkrétní metriky vycházející z výše uvedených tříd.

2.4.1 Metriky na zdrojovém kódu

Tento typ metrik se soustředí na samotný zdrojový kód, a tudíž má za úkol software hlavně kvantifikovat a nepřímo navrhovat, kde by bylo vhodné provést refaktorizaci či kód nějak upravit. Nezasahujíedo funkčních vlastností softwaru a řadí se tak čistě do přístupu bílé skříňky. Vhodné je používat následující metriky v kombinaci s metrikami z předchozích tříd [3]. Pro ukázku uvádím několik nejčastěji využívaných metrik zdrojového kódu:

- Cyklomatická složitost – určení lineárně nezávislých cest v rámci funkce. Výsledné ohodnocení pomáhá zjednodušit vyloženě složité rutiny či odstranit zbytečné cykly. Hodnota cyklomatické složitosti se inkrementuje po každém klíčovém slově z množiny `{if, while, repeat, for, and, or, case}`.
- Halsteadovy metriky – měří výskyt syntaktických prvků v programu (počet operátorů, počet operandů, počet jedinečných proměnných a podobně). Výrazně pomáhají při stanovení programové náročnosti.
- Metriky živosti kódu – popisují počet změn, ke kterým v modulu dojde během určitého časového období. Kromě vývoje nových funkcionalit se na nich podílejí rovněž opravy nalezených chyb. Vysoké procento oprav ve skutečnosti chybu nemusí odstranit a může navíc vytvořit jinou. Celkovou živost kódu lze spočítat např. následujícím způsobem:

$$\check{Z}ivost\ k\acute{o}du = Počet\ změn\ v\ souboru + počet\ smazaných\ \check{r}\acute{a}dk\mathring{u} + počet\ vytvořených\ \check{r}\acute{a}dk\mathring{u} + počet\ editovaných\ \check{r}\acute{a}dk\mathring{u}$$

Rovnice 1: Výpočet metriky Živost kódu

- Objektově orientované metriky – vztahují se k třídám a jejich struktuře v objektově orientovaných jazycích (C++, Java a další). Nejznámější z nich zohledňují například velikost třídy, váhu metod pro třídu, spřažení mezi třídami, počet metod, počet metod volaných třídou, hloubku stromu dědičnosti, počet potomků ve třídě a podobně. Minimálně pro zajímavost uvádím, že dle různých výzkumů (např. [15]) mají pro predikci chyb v daném software největší váhu parametry počtu metod volaných třídou, počtu řádků ve třídě a celkovou složitost metod

ve třídě. Naopak metriky týkající se dědičnosti či polymorfismu nejsou příliš častým zdrojem chyb. To poukazuje na fakt, že nejčastějším zdrojem chyb jsou spíše ty syntaktické (přehlédnutí, přepsání) než sémantické (skutečná chyba v logice).

2.4.2 Metriky na testech

V případě, že přidělujeme každému testu nějakou nosnou váhu, je možné v rámci komplexního testování dle sumy takové hodnoty určit, v jaké fázi testování se aktuálně nacházíme, počítat váhu testů za každou iteraci a podobně. Pokud rozlišujeme více různých kategorií testů, může být však taková hodnota zavádějící. Proto se často užívá jen v případě funkčních testů, popř. testů jednoho agilního kvadrantu (viz kapitola o agilních testovacích kvadrantech).

2.4.3 Metriky testování na chybách

Určit samotný počet chyb v softwaru nestačí, ale při rozdělení na jednotlivé funkční oblasti (GUI, výpočtový mechanismus, komunikační protokol apod.) a dalším dělení dle závažnosti chyb se následující metriky dají kvalitně využít pro rozdělení odpovědností a prioritizaci. Tyto metriky obvykle vycházejí z databáze pro sledování chyb a jsou užitečné pro vyhodnocování kvality softwaru [15]. Samotné statistiky chyb by ale neměly být používány k měření výkonu jednotlivce, protože jsou ovlivněny mnoha dalšími faktory (např. složitostí testované funkcionality, schopnostmi autorů kódu, kompletností specifikací). Názorné příklady metrik této kategorie následují:

- *„Bug fix rate“* metrika vyjadřuje, kolik procent z celkového počtu nalezených chyb již bylo opraveno. Do čitatele ve vzorci se v praxi dosazuje počet jakkoliv vyřešených chyb. Tedy například i chyby, které byly zamítnuty s odůvodněním, že se jedná o vlastnost aplikace.

$$Bug\ fix\ rate\ = \frac{množství\ opravených\ chyb}{množství\ nalezených\ chyb}$$

Rovnice 2: Výpočet metriky Bug Fix Rate

- Průměrný čas pro vyřešení – měří rychlost odezvy testovacího a vývojářského týmu.

$$Průměrný\ čas\ pro\ vyřešení = \frac{\sum doba\ vyřešení\ jednotlivých\ problémů}{počet\ problémů}$$

Rovnice 3: Výpočet metriky Průměrný čas pro vyřešení

- Průměrný čas pro uzavření – měří celkovou reakci týmu na odhalené chyby a čas potřebný k provedení celého procesu zpracování chyb. Je opět nutné jasně rozlišovat mezi termínem vyřešení a uzavření problému. Vyřešení představuje podmnožinu k uzavření; uzavření problému skýtá další kroky (zejména akceptaci výsledku reportu z testování vývojářem).

$$Průměrný\ čas\ pro\ uzavření = \frac{\sum doba\ k\ uzavření\ jednotlivých\ problémů}{počet\ problémů}$$

Rovnice 4: Výpočet metriky Průměrný čas pro uzavření

- Efektivnost testu – určuje, kolik procent chyb bylo nalezeno za pomoci testu. Tento údaj slouží k porovnání jednotlivých testů nebo k porovnání s časem stráveným prováděním testu. Primárně touto metrikou můžeme odstranit neefektivní testy.

$$Efektivnost\ testu = \frac{množství\ nalezených\ chyb\ daným\ testem}{množství\ chyb\ celkem}$$

Rovnice 5: Výpočet metriky Efektivnost testu

- Rychlost objevení chyby – informuje o tom, kolik chyb je průměrně nalezeno za časovou jednotku. V praxi je tato hodnota nejvyšší na počátku testování projektu a postupně klesá s blížícím se koncem testování.

$$Rychlost\ objevení\ chyby = \frac{množství\ nalezených\ chyb}{doba\ vykonávání\ testu}$$

Rovnice 6: Výpočet metriky Rychlost objevení chyby

- Rychlost odhalování chyb v průběhu času – příliš vysoké nebo příliš nízké hodnoty mohou indikovat problém a měly by být objasněny.
- Chyby podle závažnosti – chyby kategorizované jako nejzávažnější by měly být nacházeny v úvodních fázích vývoje.

- Podle místa výskytu – porozumění tomu, ve kterých částech softwaru jsou nacházeny chyby, může odhalit rizikové oblasti produktu.
- Míra reaktivace chyb – naznačuje, jak kvalitní opravy programátoři vytvářejí. Ke konci projektu se zpravidla zvyšuje kvůli rychlejšímu opravování chyb.
- Počet chyb podle typu testovací aktivity – zjištění, které testovací postupy vedou k nalezení velkého množství chyb, může zaměřit testování tímto směrem. Jedná se např. o integrační testy, automatizované testy, regresní testy, akceptační testy apod.

2.4.4 Ostatní nezařazené metriky

Obecně platí, že metriky navrhují testeři takové, jaké poslouží pro získání potřebných informací o daném projektu, tzn. informace, které management projektu, firmy či zákazníka potřebuje znát. Je jistým posláním softwarového inženýra tyto potřeby za pomoci managementu definovat a následně najít cesty, jak na ně odpovědět. Nejen proto se mohou metriky ke každému danému projektu dle aktuálních potřeb měnit a žádná vyšší kategorizace pro ně nemusí platit. Zde uvádím některé další metriky, jež se objevují v odborné literatuře [1, 3], ale vzhledem k vybrané kategorizaci je nebylo možné do výše uvedených skupin zařadit:

- Jak byla chyba zjištěna
- Zda byla zjištěna dříve, než způsobila problém
- Zda byla zjištěna náhodně nebo prostřednictvím řízeného procesu
- Zda o chybě předem někdo věděl nebo ji tušil
- Jak bylo nákladné chybu zjistit
- Jak bylo nákladné chybu odstranit
- …

3 Integrace poznatků do firmy Siemens

Praktická část této práce má za úkol nabyté informace zužitkovat při optimalizaci testovacích procesů v jedné z poboček firmy Siemens. Následující kapitola tak předkládá zevrubný popis původního stavu testování ve výše zmíněné firmě, popis užívaných metodik a nástrojů a prvotní analýzu vstupních dat, jež mají sloužit jako evaluační vzorek k výběru a výpočtu metrik pro ohodnocení kvality testování.

3.1 Představení firmy Siemens

Firma Siemens patří v rámci celosvětového měřítka mezi největší elektrotechnické firmy v České republice. Zabývá se vývojem technologií a produktů v oblasti energetiky, průmyslové infrastruktury a informačních technologií. Konkrétní divize, se kterou blíže spolupracuji v rámci této práce, má na starosti oblast průmyslu a automatizace (Divize Digital Factory, Process Industries and Drives). Konzultantem z firemní strany se stal zaměstnanec, který má v podstatě výhradní slovo při komunikaci mezi týmem vývojářů a testerů. Veškeré níže uvedené informace jsou firemním materiálem, jež mi byl při konzultačních schůzkách blíže představen.

3.1.1 Popis užívaných metodik

Brněnská divize firmy Siemens používá vlastní upravenou metodiku pro vývoj a testování svých softwarových produktů. Konkrétně se jedná o spojení klasické agilní metody Scrum s metodologií ADD, které jsem popisoval v předchozích kapitolách.

Přes všechny zmíněné výhody metodiky Scrum se její úložiště problémů (*backlogs*) orientuje pouze na požadavky funkcionální, tedy ty zadané a specifikované samotným zákazníkem. Z výchozího procesu této metodiky je tak vyřazen apel na požadavky kvalitativní. Neúspěšnost softwarového projektu určuje právě především nespokojenost s kvalitativními vlastnostmi (výkon, použitelnost a další, viz blíže kapitola o metodice ADD).

Se snahou o celistvost v získávání požadavků využívá Siemens tým právě kombinace dvou zmíněných technik, které jsou pospolu ještě upraveny[4].

Metoda Scrum obsahuje dle procesního řízení Siemensu dvě fáze v rámci jednoho sprintu – spánek (SLEEP) a běh (RUN), které se vzájemně v některých oblastech časově prolínají. Fáze spánku, která slouží k setkání se zákazníkem, přednesení nových požadavků, prezentaci aktuálního stavu a větším týmovým setkáním. Fáze běhu spočívá v klasickém sprintu, tzn. výběru požadavků z *product backlog*, jejich zpracovávání a denní týmová setkání. Prolínání těchto dvou fází má smysl v rychlejší reakci na změny a v principu kontinuální integrace, který bude rozebrán v další kapitole.

Metodika ADD má čtyři hlavní fáze, které v podstatě jen upřesňuji poněkud vágní definice metodiky v odborných příručkách:

- Tvorba stromu kvalitativních měřítek (tzv. *utility tree*), což vlastně odpovídá fázi dekompozice
- Příprava a rozbor scénářů pro jednotlivé listy stromu
- Tvorba taktiky a strategie (pro Siemens odpovídá termín strategie přípravě sady testů)
- Vývoj

Veškeré modelování a plánování svázané v metodě ADD probíhá v nástroji Enterprise Architect, který bude ještě v práci dále popsán.

3.1.2 Procesní řízení v rámci testování

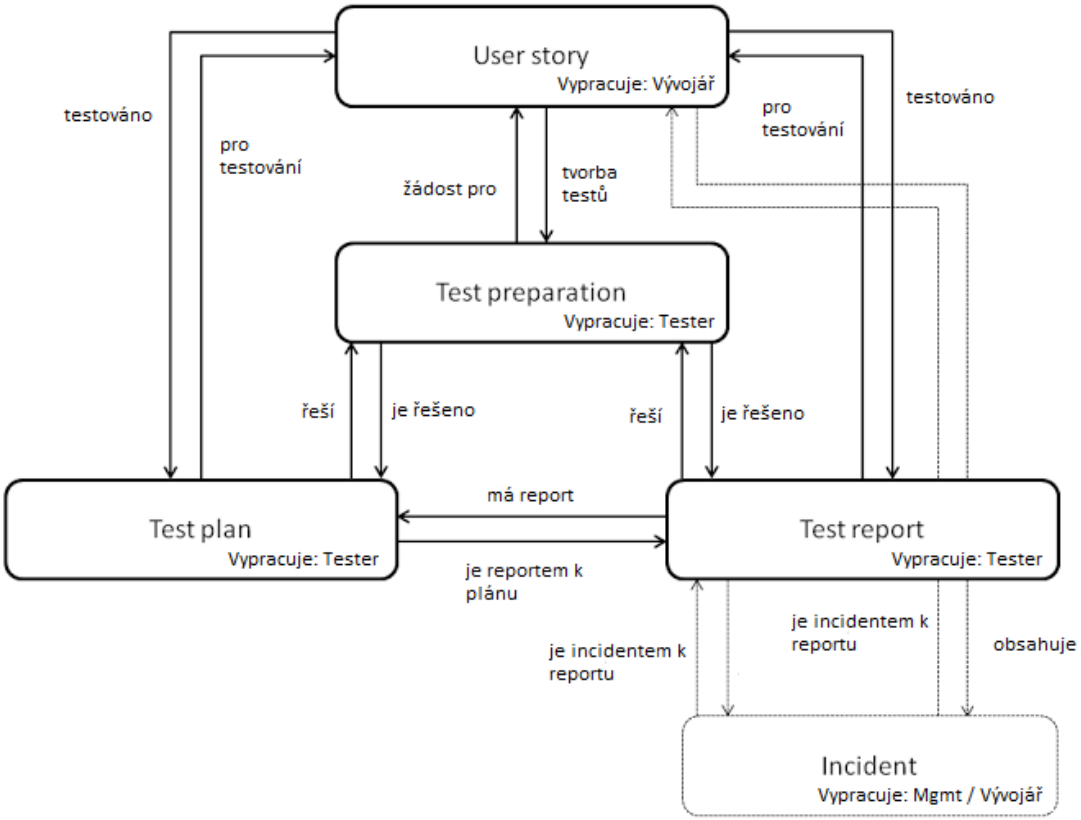
Pro naprosté pochopení jednotlivých vazeb na bázi testování Siemens používá následující terminologii a pracovní postup (angl. *workflow*):

- **Uživatelský příběh (angl. *user story*) –** v podstatě se jedná o funkcionalitu požadovanou zákazníkem na základě analýzy požadavků (*requirements analysis*). User Story má vytvářet obraz či představu a popisovat nějaký příběh. Lidský mozek vnímá obrázky a příběhy daleko snadněji než technický popis v bodech, a nejen proto se tato technika při realizaci agilních řešení ukázala jako vhodnější. Uživatelské příběhy implementuje vývojář.
- **Příprava k testování (angl. *test preparation*) –** v momentě, kdy vývojář zahájí práci na přiděleném příběhu, vzniká paralelně v backlog souboru (viz kapitola pojednávající o Scrum metodice) záznam typu „*task*“, který ve svém názvu obsahuje klíčové slovo „*test preparation“*

[4] Zajímavým blízkým konceptem je rovněž metodika ACRUM, která obsahuje řadu podobných rysů. Více o této metodice uvádí zdroj [8].

a označuje přípravu k danému příběhu ze strany testera. Jedná se o obecný koncept, který pod sebou zahrnuje komponenty, které budou popsány v následujícím textu.

- **Testovací plán (angl. *test plan*)** – uvádí detail k provedení testovací strategie. Testerům a vývojářům říká, co má daný test dělat (jaké funkce či funkčnost testovat), za jakých podmínek, kdy a jak se má provádět a co je potřeba k jeho splnění. Jinými slovy nastiňuje týmu záběr (angl. termín *scope*) daného testování. Rovněž popisuje umístění testu v časovém plánu, rozpis potřebných zdrojů (lidé / software / nastavení) a podrobnosti ohledně testovacího cyklu, reporting a sledování procesu testování. Testovací plán, ať už je aplikován v různých metodikách, představuje dobrý způsob komunikace mezi týmem testerů, týmem vývojářů a managementem. Celý koncept pomáhá tomu, aby byly týmy z pohledu testování synchronizované.
- **Report z testu (angl. *test report*)** – jedná se o dokument podobné struktury jako testovací plán. Report hlásí výsledky testu v reakcích na zadané vstupy a porovnává, jak moc se výstupy liší od předpokládaných výsledků. Na základě reportu se vývojář rozhoduje, co považovat za chybu nutnou k opravení a v takovém případě (znovu)otevřít incident.
- **Incident** – obecný název pro chybu, která vzešla na povrch na základě testování. Seznam incidentů přebírá z výsledků práce testera vývojář, nebo popřípadě manažer projektu a re-evaluuje závažnost těchto incidentů v orientaci na další postup.



Obrázek 4: Propojení užívaných pojmů v procesu testování (převzato z fy Siemens)

3.1.3 Popis užívaných nástrojů

Brněnská divize firmy Siemens pracuje s poměrně úzce vymezeným specifikem nástrojů, které jsou, podobně jako vývojové procesy, nastaveny značně individuálně. Pro pochopení celého konceptu testování tak bylo nutné seznámit se s aplikacemi, který v této kapitole blíže popíši.

JIRA – softwarový nástroj pro podporu a usnadnění procesního řízení projektů vyvíjený firmou Atlassian. V případě brněnské divize firmy Siemens CZ je tento nástroj využíván ke koordinaci a synchronizaci vývojářských a testovacích procesů v rámci běžného projektu. Prostředí softwaru JIRA klade důraz na naprostou jednoduchost a zároveň možnost používat nástroj nejen jako podporu projektového řízení, ale i jako nástroj pro tzv. *workflow management* či nástroj pro sledování statistik a reportů. Vzhledem k tomu, že se jedná o nástroj, který se bude v drtivé míře využívat v praktické části této práce, uvedu níže několik pojmů, které se k tomuto prostředí vážou:

- problém (angl. *issue*)[5] – základní jednotka projektu. Definování jednoznačné funkce se liší od uživatele, může sloužit jako záznamní pro chyby, úkoly či připomínky.
- typ problému (angl. *issue type*) – parametr instance problému. Nabízí čtyři předdefinované hodnoty – chyba (angl. *bug*), úkol (angl. *task*), nová vlastnost (angl. *new feature*), vylepšení (angl. *improvement*) – a možnost vytvořit si typ vlastní.
- priorita – určuje důležitost řešeného problému. Předdefinovaných hodnot je pět, ale aktivně se používají zejména hodnoty pro vysokou prioritu (angl. *major*) a nízkou prioritu (angl. *minor*).
- status – určuje pozici v životním cyklu problému. Obsahuje pět předdefinovaných hodnot pro standardní JIRA projekt a tři pro agilní přístup („otevřený“, „ke zpracování“ a „dokončeno“).
- projektová obrazovka (angl. *dashboard*) – hlavní stránka daného projektu, která zobrazuje různé typy statistik

JIRA Agile – (do srpna 2013 nabízen jako produkt GreenHopper) je nejčastěji nasazovaným přídavným modulem v prostředí JIRA. Zajišťuje funkčnost nezbytnou pro agilní metody – zejména SCRUM a KANBAN. Samozřejmostí je pokrytí typů s názvy *epic*, *user story* a *task*. Dále podporuje agilní testování a nasazení systému JIRA jako agilní řešení systému podpory (angl. *helpdesk*).

[5] Více informací dostupných na oficiální stránce JIRA: https://confluence.atlassian.com/display/JIRA043/What+is+an+Issue

FishEye – hojně využívaný zásuvný modul pro software JIRA, jež umožňuje ke každému problému, chybě či uživatelskému příběhu přiřazovat seznam úprav na kódu (angl. tzv. *commit*), které se k danému typu vztahují. Ve stručnosti tedy tento model zajišťuje synchronizační mechanismy mezi systémem JIRA a používaným verzovacím systémem.

Enterprise Architect – je softwarový balík sloužící v prvé řadě pro modelování UML a podnikových procesů, jejich vizualizace a řízení testování. Použitelnost tohoto produktu sahá od individuálních analýz požadavků až po modelování strategických byznys procesů. Svými komponentami je schopný pokrýt celý vývojový cyklus software. Vývojářský tým firmy Siemens ho používá zejména v procesu analýzy požadavků pro tvorbu jednotlivých uživatelských příběhů.

Atlassian Git – firma Atlassian rovněž vyvinula uživatelsky příjemné rozhraní pro systém správy verzí (Git a Mercurial), jehož použití je vzhledem k distribuované práci týmu firmy Siemens nevyhnutelné.

Jenkins – představuje relativně snadno použitelný kontinuální integrovaný systém (CIS) a funguje jako extrémně dobrý pomocník zejména při metodách extrémního programování a testování. Dokáže výrazně usnadnit proces integrace změn v projektu a tester může díky tomuto přístupu získat nejčerstvější verzi programu.

Selenium – aplikace Selenium (konkrétně verze s přídomkem RC) umožňuje tvorbu automatických testů ve velké škále programovacích jazyků. Základním stavebním kamenem tohoto nástroje je server, který vytváří virtuální proxy server pro instance internetového prohlížeče, a ty sám spouští a vypíná. Pro každý z podporovaných jazyků je připravena knihovna funkcí a dá se říci, že dnes je Selenium aplikace v podstatě jazykově nezávislá.

3.2 Evaluace existujícího projektu a stanovení metrik

Jedním ze stěžejních bodů této práce byla evaluace a zhodnocení reálného projektu pomocí výše stanovených testovacích metrik. Nejprve byla potřeba projektová data získat, na základě nich provést vhodný výběr metrik, vypočítat tyto metriky a zhodnotit je.

3.2.1 Získání a zpracování dat

Pro potřeby této práce byly konzultantem firmy Siemens CZ vybrány historické obrazy (angl. tzv. *snapshoty*) jednoho z uzavřených projektů. Dolování a užití projektových dat ze systému JIRA

proběhlo pod záštitou smlouvy o mlčenlivosti. Zdrojová data popisující jednotlivé sprinty na daném projektu byla skriptem v jazyku Python zparsována z PDF do databáze a veškeré výstupy byly podrobeny datové analýze.

3.2.2 Stanovení metrik

Vzhledem k jisté generičnosti dat (ve smyslu chodu projektu bez větších problémů) a jejich nepřílišné podrobnosti (dodané typy dat – Story a Task, tudíž bez rozdělení do konkrétní chyby), bylo nutné zorientovat se při výběru metrik na některé, které přímo vycházejí z konceptu systému JIRA, nebo se mu alespoň přibližují. V mnoha případech se tak nejedná o metodiku dříve definovanou, ale spíše vytvořenou vzhledem k povaze dodaných dat. Před samotným rozborem bylo nutné si ujasnit jednotlivé typy záznamů (úkolů), které se v extraktu vyskytovaly:

- *Closed* – úkol byl ze strany testerů ukončen a potvrzen ze strany vývoje (tzn. v případě úkolů trvajících právě jeden sprint se záznam uvádí rovnou jako *Closed*)
- *Open* – úkol se v daném sprintu otevřel a nebyl ukončen (tzn. úkol bude trvat minimálně dva sprinty)
- *In Progress* – úkol je v daném sprint rozpracován (tzn. úkol trvá minimálně tři sprinty a ve druhém z těchto tří je uveden jako *In Progress*)
- *Resolved* – úkol by ze strany tester ukončen a čeká na potvrzení ze strany vývoje (záznamy s touto signaturou se objevovaly pouze v prvních dvou sprintech, a to patrně proto, že v této fázi se jednalo především o samotnou přípravu testů, nikoli o jejich následné využití)
- *Reopened* – úkol byl z důvodu nenalezeného defektu znovu otevřen

Po důkladném rozboru byly vybrány následující metriky a shrnující analýzy:

- Pracovní zátěž (angl. *workload*)
- Graf Burndown
- Analýza priorit
- Analýza životnosti
- Korelace testů s vývojem

3.2.3 Výsledky měření

V následující kapitole uvádím jednotlivé části předem popsané datové analýzy. Celkově došlo k analýze 450 záznamů, z nichž 220 mělo povahu přípravy testu či nějak jinak souviselo s testováním.

Těchto 220 záznamů představovalo celkem 120 individuálních úkolů vykonaných v průběhu 12 sprintů, kdy každý sprint trval dva pracovní týdny.

3.2.3.1 Pracovní zátěž

Metrika pracovní zátěže může být vyhodnocena z různých úhlů v závislosti na dodaném typu dat. V tomto případě jsem zátěž vyhodnotil z hlediska zpracovávaných úkolů v daném sprintu. Z níže uvedené tabulky je zřetelné, že první čtyři sprinty zpracovával tým testerů nárazové přípravy testů, které vyplývaly z vývojové stránky projektu. Jednalo se povětšinou o úkoly, jež byly bez problému splněny v průběhu jednoho sprintu. Tyto úkoly neměly ve zprávách zanesenou výši hodnocení Story Points a proto byly ohodnoceny nejnižším možným počtem bodů (1). Naopak nejvytíženějším sprintem z hlediska komplexity byl sprint č. 6, tzn. polovina zpracovávaného projektu. Zde se jednalo o téměř pětinový podíl na celkovém projektu.

	Uzavřené	Probíhají	Otevřené	Znovu-otevřené	Vyřešené	Celkem	Story Points	Zátěž (Celkem)
Sprint 1	3	-	-	-	5	**8**	**8**	**4%**
Sprint 2	2	4	1	3	5	**15**	**15**	**7%**
Sprint 3	14	2	-	1	-	**17**	**17**	**8%**
Sprint 4	13	-	-	-	-	**13**	**13**	**6%**
Sprint 5	-	8	19	-	-	**27**	**392**	**12%**
Sprint 6	20	13	1	4	-	**38**	**530**	**17%**
Sprint 7	8	10	1	-	-	**19**	**212**	**9%**
Sprint 8	-	11	-	-	-	**11**	**120**	**5%**
Sprint 9	13	1	-	-	-	**14**	**165**	**6%**
Sprint 10	7	6	-	3	-	**16**	**227**	**7%**
Sprint 11	13	10	1	-	-	**24**	**395**	**11%**
Sprint 12	18	-	-	-	-	**18**	**327**	**8%**

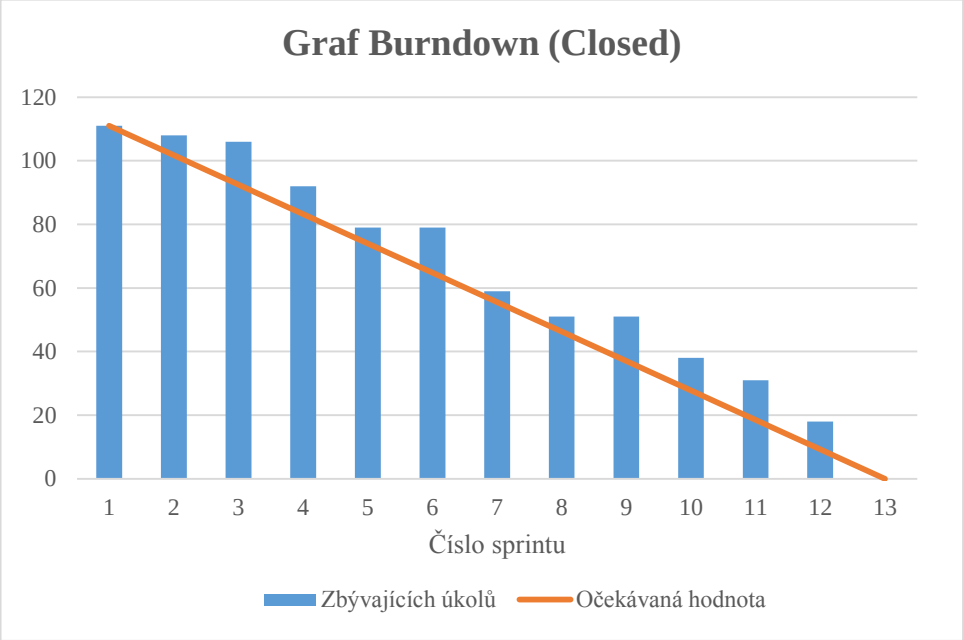
Tabulka 1: Shrnutí aktivity a pracovní zátěže v jednotlivých sprintech

3.2.3.2 Graf Burndown

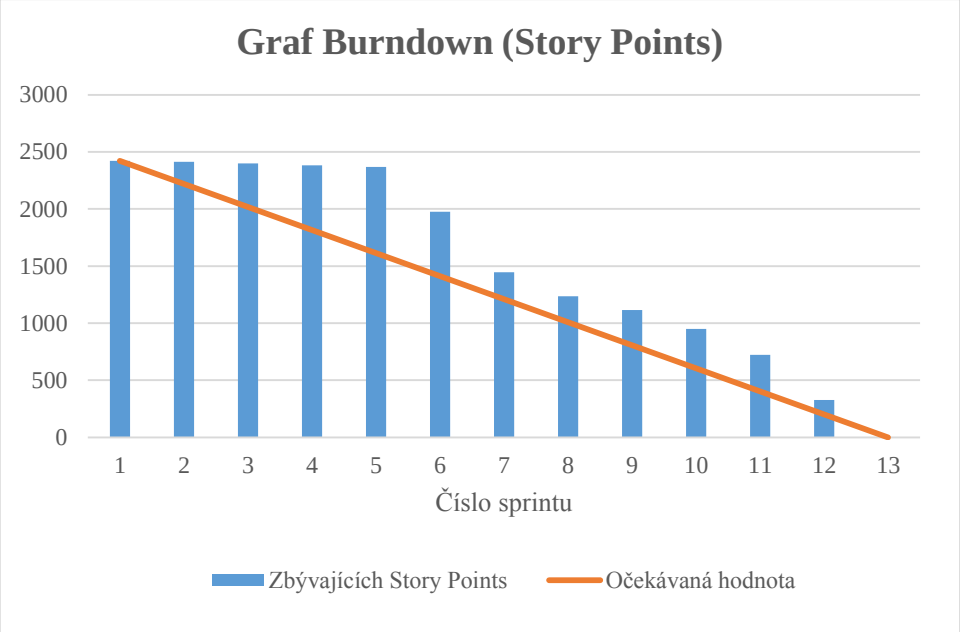
Burndown křivka představuje zcela nový přístup ke grafickému znázornění časové náročnosti projektu. Lineární křivka ukazuje hypotetický ideální průběh prací pro splnění příslušného termínu. Sloupcový graf zobrazuje skutečný počet úkolů zbývajících do konce projektu vázaný k daném sprintu. Tento počet se přitom může zvyšovat, když dojde ke špatnému odhadu pracnosti některého z úkolů. Pokud se hodnota nachází pod ideální křivkou (tzv. ideální burndown), je projekt v předstihu, pokud nad ní, má projekt zpoždění. V případě, kdy tak burndown graf indikuje několik dní (popř. sprintů) po

sobě zpoždění, má projektový manažer možnost analyzovat nejkritičtější (tj. nejdéle trvající) úkoly a přidělit k nim další členy týmu, aby se zpoždění začalo včas dohánět.

Níže uvádím dva grafy, které se liší v hodnotách vertikální osy – v prvním případě se jedná o počet zbývajících úkolů, v druhém případě o počet zbývajících Story Points (z celkových 120 úkolů se jako standardních ukázalo 111, ostatní záznamy byly většinou okamžitě smazány, ale v systému přesto uchovány). Na prvním grafu je zřetelně vidět, že projekt byl po celou dobu veden s větším (3., 6. a 9. sprint) či menším (5., 7. a 8. sprint) zpožděním. Druhý graf předesílá ještě větší rozdíly oproti ideálnímu stavu, tentokrát z pohledu úsilí, jež je třeba vykonat k dokončení projektu. V tomto případě odchylka vzniká patrně kvůli nevhodnému nastavení Story Points (tedy komplexity) jednotlivých úkolu v čase, popř. jejich nesprávnou prioritizaci. Problém mohl také nastat v úplném zobecnění nastavení – v plánování kapacit. Nicméně očekávané hodnoty byly v posledních dvou sprintech výrazně „dohnány“ a projekt skončil s očekáváním (žádné úkoly s indikátorem *Open* či *In Progress*).



Graf 1: Burndown v závislosti na měřítku zbývajících úkolů



Graf 2: Burndown v závislosti na měřítku zbývajících Story Points

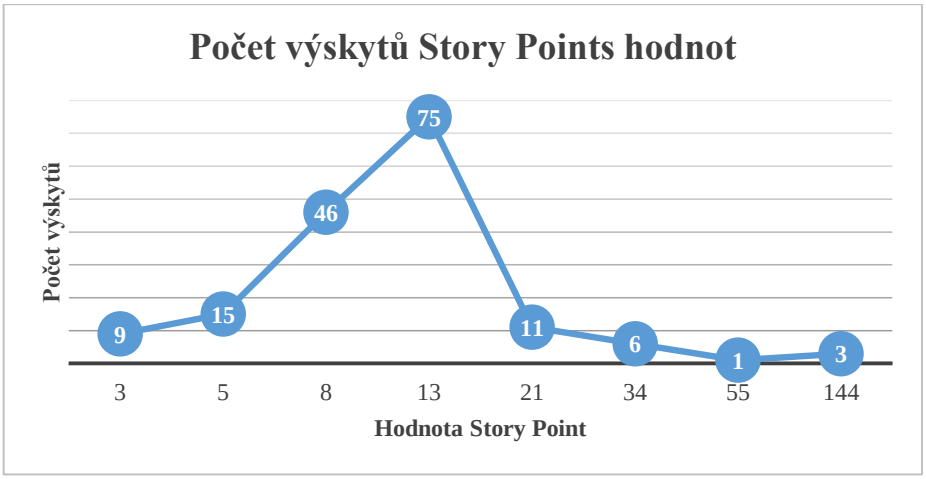
3.2.3.3 Analýza priorit a úsilí

Priority jsou v případě zkoumaného projektu určeny jednak hodnotami „*Minor*“/“*Major*“ ve sloupci s názvem „*Priority*“. Na druhou stranu úsilí je hodnoceno konceptem Story Points blíže představeným v předchozích kapitolách. V této podkapitole rozebírám problém obou těchto měřítek, protože spolu úzce souvisejí.

Celých 97% úkolů bylo ohodnoceno prioritou s hodnotou „*Major*“. Takovéto absolutní rozdělení může být například pro nové členy týmu značně zavádějící, obecně snad i zbytečné.

Třetina záznamů měla hodnotu Story Points (dále jen SP) menší nebo rovnu 8 (tzn. nízkou komplexitu), třetina hodnotu střední (13 SP) a šestina hodnotu vyšší. Až čtvrtina záznamů neměla hodnotu SP vůbec vyplněnou, což lze považovat za závažnou chybu při rozdělování činností a chápání komplexity projektových úkolů.

Zajímavým faktem však zůstává, že v některých rozsáhlejších úkolech (2-5 sprintů) došlo v jejich průběhu k manuálnímu navýšení příslušných Story Points, a to s průměrnou hodnotou 11,8 bodu/záznam. Tento fakt si lze vysvětlit pozdním zjištěním, že problém bude náročnější vyřešit, než se předtím zdálo. Obecně každá změna komplexity značí, že se něco v projektu nepovedlo podle plánů.

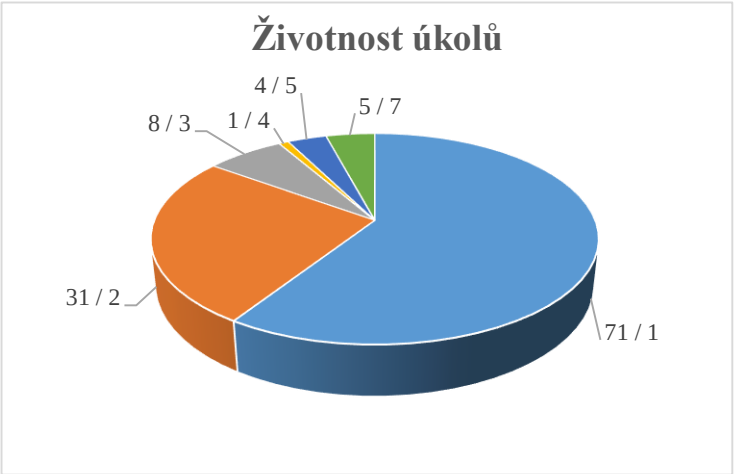


Graf 3: Počet výskytů Story Points hodnot

3.2.3.4 Analýza životnosti

Průměrná doba životnosti jednoho úkolu testerů byla 1,83 sprintu. Při výpočtu této metriky se uvažovala životnost jako přítomnost úkolu v jakémkoli stavu v daném sprintu. To znamená, že pokud byl úkol uzavřen a později znovu otevřen, nebylo tato doba „čekání“ do metriky započtena. Nejvyšší naměřenou hodnotu délky životnosti (7 sprintů) obsahovalo 5 úkolů. Až 16% úkolů bylo delších nebo rovných 3 sprintům, to znamená, že jedna šestina úkolů byla poměrně problémová, což se ve srovnání s praxí dá považovat za hodnotu nízkou (normální rozdělení počítá s hodnotou 33%). Je však zajímavé sledovat, že s přibývající délkou řešení se zvyšuje i komplexita problémů (viz Tabulka 2). Pokud jsou tedy problémy náročné časově, budou náročné i úsilím (a to nikoli pouze celkovým úsilím, ale i úsilím za daný sprint). Vznikají tak dva extrémy názorně viditelné v řádku 1 a 6 přiložené tabulky. Normalizaci hodnot by mohlo prospět jemnější dělení větších problémů na menší.

Počet úkolů	Délka trvání [sprint]	Průměrný počet SP
71	1	8.54
31	2	8.06
8	3	28.5
1	4	13
4	5	10.5
5	7	10.4



Tabulka 2 a Graf 4: Analýza životnosti úkolů

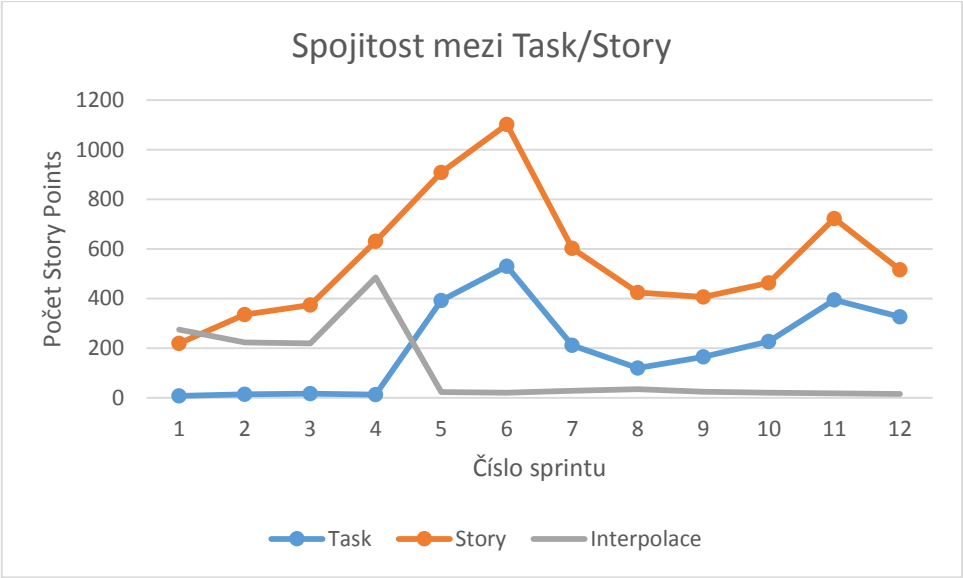
3.2.3.5 Korelace testů s vývojem

Vývojářský tým je v systému JIRA zastoupen zpracováváním jednotlivých uživatelských příběhů. Vzhledem k povaze dat je tak možné provést korelační analýzu na základě srovnání záznamů testovacího týmu (angl. *task*) a vývojářského týmu (angl. *story*). Tabulka 3 značí, že přístup k vývoji a k testování je v některých ohledech odlišný. Pokud je při testu nalezena chyba, promítne se znovuotevřením úkolu typu „Story“, což je zřetelné ve čtvrtém řádku. Nicméně vzhledem k tomu, že znovuotevření může v některých případech proběhnout i z jiných důvodů, nemohl jsem v dokumentu zpracovávat metriky týkající se chyb. Při potvrzení výše uvedeného je však bude možné do analýzy případně zařadit. Ostatní naměřené hodnoty víceméně korelují.

Typ záznamu	Korelace	Počet záznamů (Task)	Počet záznamů (Story)
Closed	96%	111	115
In Progress	77%	65	8
Open	96%	23	8
Reopened	39%	11	92
Resolved	67%	10	6

Tabulka 3: Korelace Task/Story záznamů na základě typu záznamu

Níže uvedený graf uvádí spojitost mezi komplexitami jednotlivých sprintů dvou oddělených týmů. Z vysoké hodnoty korelace (83,7% v případě Story Points a dokonce 99% v případě počtu zpracovávaných úkolů) můžeme vyčíst úzkou spolupráci a prakticky i synchronizaci úkolů, která mezi týmy vládne. Odchylka v prvních čtyřech sprintech je dána neohodnocením úkolů pro přípravu testů. Mimo tuto nedokonalost ale křivka neznačí žádnou obtíž ve spolupráci (zpoždění apod.). Na druhou stranu by si konkrétnější analýza žádala rozdělení na jednotlivé související úkoly, což by ale vzhledem k rozsahu a konceptu dokumentu bylo nevhodné.



Graf 5: Spojitost mezi Task/Story záznamů na základě počtu Story Points

3.2.4 Shrnutí výsledků

Analyzovaný projekt je z velké části tvořen úkoly, které lze vyřešit během jednoho, maximálně dvou sprintů. Jen výjimečně se objevují úkoly složitější. I to má za následek velmi dobrou synchronizaci týmů (zpravidla dochází k tvorbě i zanikání úkolu pro vývojáře i testery ve stejném sprintu, maximálně se zpožděním jednoho sprintu). Otázkou zůstává, jak by naměřené metriky vypadaly v případě projektu složitějšího a kolik času týmy investují do editování záznamů systému JIRA a delegování. Zřídka se objevuje zjištění, že problém bylo náročnější vyřešit, že se původně zdálo.

Hodnocení priorit (angl. zkr. *major/minor*) se ukazuje jako poměrně zbytečné a přesto uváděné, což může mít za následek nesrozumitelnost, jaký skutečný impakt při vývoji/přípravě testů na daný úkol nasadit. Čtvrtina záznamů neměla hodnotu Story Points vůbec vyplněnou, což lze považovat za závažnou chybu při rozdělování činností a chápání komplexity projektových úkolů. Nutnou nápravou by měla být podmínka, která zaručí, že bude vždy hodnota SP vyplněna. Pokud jsou problémy náročné časově, budou náročné i úsilím, což má za následek velké množství dvou extrémních typů záznamů. Normalizaci hodnot by mohlo prospět jemnější dělení větších problémů na menší.

Automatické zhodnocení a návrhu dalších řešení výše uvedených nesourodostí a souvisejících problémových aspektů bude předmětem následující kapitoly této práce.

4 Analyzátor projektu v prostředí JIRA

V této kapitole představím proces návrhu a implementace analyzátoru projektu dle metrik uvedených v předchozích kapitolách.

S ohledem na chronologický odstup zadání práce od aktuálních potřeb firmy se okolnosti poupravily a výsledný zásuvný modul není logicky postaven na bázi automatizace testovacího procesu jako spíše na udržení přehledu o kvalitě testování a sledování odchylek od naplánovaného průběhu projektu. Jelikož má firma systém JIRA nastavený tak, aby mohly být testovací a vývojové týmy sledovány odděleně, lze prohlásit, že snaha optimalizace průběhu projektu v systému JIRA je vlastně optimalizací testovacího procesu.

4.1 Motivace a prvotní přípravy

Podkapitola popisuje prvotní fázi vývojového cyklu software, včetně specifikace a ujasnění požadavků, analýzy alternativních produktů postavených na podobné bázi a shrnutí metodiky, která byla pro další postup použita.

4.1.1 Požadavky a cíle

Brněnská divize firmy Siemens pracuje s aktualizovanou verzí webového prostředí JIRA a má zájem o zobrazování dodatečných statistik svých projektů ve formě zásuvného modulu ke zmiňovanému prostředí.

Při zadání kladla důraz na využití aktuálních vykreslovacích technologií, jednoduchost, invenci a modularitu (možnost jednoduše přidávat další analýzy). Rozšíření by mělo pracovat dynamicky nad jakýmikoliv vstupními daty formátu projektu JIRA. Mělo by být volně dostupné a z hlediska dokumentace udržované pro případ přidávání dalších funkcionalit.

4.1.2 Analýza alternativ

Při hledání alternativních a již dostupných řešení jsem dospěl k produktům, jejichž specifikaci, výhody a nevýhody zobrazuje tabulka níže. V případě, že nebylo možné zásuvný modul otestovat, byla tato informace daném řádku explicitně uvedena:

Název	Specifikace	Verze	Výhody	Nevýhody
JIRA Charting Plugin[6]	Volně dostupný zásuvný modul pro vykreslování grafů, nefunguje ale na bázi OpenSource	Od JIRA 3.4 a výše	Dostupné zdarma, možnost tvorby grafů (limitovaná)	Malá interakce a dynamičnost, chybí hodnotící prvky, nepracuje s agilním prostředím
eazyBI JIRA Reports And Charts[7]	Nástroj firmy eazyOne, který disponuje paletou funkcionalit pro úplný reporting projektu	Od JIRA 4.3 a výše	Možnost tvorby vlastních grafů, vysoká interaktivita	Placená verze (100$ ročně za každých deset uživatelů)
Integrovaný reportovací systém	JIRA v základní verzi obsahuje reportovací systém, který obsahuje osm typů grafů (Velocity, Standard Workload atp.)	Od JIRA 4 a výše	Dostupné zdarma, možnost exportu do PDF	Nulová interakce, neobsahuje možnost přidávání analýz, neobsahuje hodnocení projektu, nepracuje s agilním prostředím

Tabulka 4: Dostupné alternativy analyzátoru projektu na trhu

Po rozboru a konzultacích s druhou stranou jsem dospěl k následujícím závěrům o možnostech analyzátorů projektu v systému JIRA:

- některé analýzy jsou těžkopádné a složité
- neodpovídají aktuálním potřebám, některé analýzy chybí
- nejsou jednoduše modifikovatelné
- nepřinášejí shrnující hledisko o daném projektu (chybí faktor *see-and-learn*)

[6] Oficiální stránka zásuvného modulu je dostupná na: https://marketplace.atlassian.com/plugins/com.atlassian.jira.ext.charting
[7] Oficiální stránka zásuvného modulu je dostupná na: https://marketplace.atlassian.com/plugins/com.eazybi.jira.plugins.eazybi-jira. Pro aktuální (k 17.5.2015) nekompatibilitu s nejnovější verzí JIRA (6.4.3) nebylo možné modul otestovat.

4.2 Analýza technologií a návrh zásuvného modulu

Tato kapitola volně přechází z analýzy požadavků a alternativ k analýze technologií, které jsou pro tento typ projektu dostupné. Následně přednáším se jmenovaným tématem úzce spjatý návrh zásuvného modulu.

4.2.1 Analýza technologií

Jelikož bude UpAnalyzer fungovat na principu zásuvného modulu, je třeba se víceméně podřídit z hlediska návrhu prostředí, do kterého bude nástroj instalován. V následujících kapitolách uvádím stručné shrnutí jednotlivých technologií, které jsou systémem JIRA nějakým způsobem využívány. Vycházím z faktu, že jediný, aktuálně plně podporovaný jazyk pro aplikační rozhraní systému JIRA, je jazyk Java [17]. Systém zásuvných modulů v prostředí JIRA může pro neznalce být značně komplexní, a proto bylo v rámci návrhu aplikace nutné strávit podstatnou porci času jeho studiem. Pochopení fungování tohoto systému a inteligentní návrh považuji za stejně důležité jako samotné psaní kódu. Pokud má rozšiřující zásuvný modul řešit nějaký nedostatek či zkrátka přinášet přidanou hodnotu, musí vývojář pochopit, co JIRA umí a co nikoli [22]. Prvky návrhu, které v následujících podkapitolách nejsou zmíněny, nepovažuji pro návrh aplikace za naprosto zásadní a budou okrajově zmíněny v kapitole o samotné implementaci.

4.2.1.1 Atlassian JIRA API

Základní stavební kámen pro vývoj aplikací pod systémem JIRA reprezentuje aplikační rozhraní výrobce, uváděné pod názvem Atlassian JIRA (Java) API. Objektový model obsahuje téměř tisíc tříd a rozhraní, které nabízí kompletní správu systému JIRA přes programový kód. Prvotní krok ve fázi samotné implementace bude představovat právě přístup k aplikačním možnostem systému JIRA. Ze systému potřebujeme získávat vstupní data do analýz. Ze zdrojů [6,7] vyplývá, že v podstatě lze přistupovat k informacím o jednotlivých dílčích objektech několika způsoby:

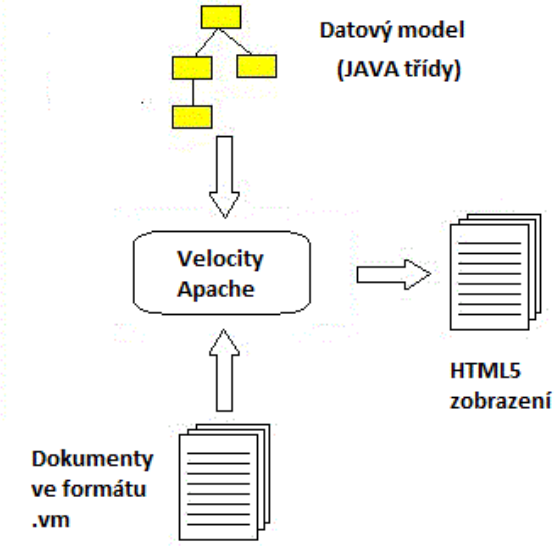
- JQL dotazy na databázový server
- Strukturovanými dotazy přes REST API
- Využíváním přímo implementovaných Java tříd a metod rozhraní JIRA

Pro tuto práci jsem zvolil třetí možnost, jež se na první pohled zdála metodou přímočarou a dobře dokumentovanou. V procesu sběru informací a snahy o kusé testování aplikačního rozhraní jsem se však setkal se značnými problémy, zejména v nemožnosti kód svižně ladit a faktu, že rozsáhlá

dokumentace fungovala pouze jako generický prvek, který se nezaobíral reálnými možnostmi problematického chování[8]. Bližší informace uvádím v kapitole o implementačních zajímavostech.

4.2.1.2 Velocity Apache

Framework Velocity představuje šablonovací systém spravovaný firmou Apache. Z vlastníka tohoto produktu lze již rychlým úsudkem vytušit, že se jedná o systém běžící na straně serverové. Transformační systémy na bázi šablon jsou v oblasti vývoje softwaru prakticky všudypřítomné. Široká řada nástrojů používá šablony pro transformaci dat z jednoho formátu do druhého. V případě Velocity se jedná o převod kódu Javy do jazyků webových aplikací (angl. *Java servlet-based application development*). Obrázek 5 ukazuje složky podílející se na procesu transformace založeném na šablonování:



Obrázek 5: Schéma znázorňující princip systému Velocity Apache

- Datový model obsahuje strukturovaná data, která mají být transformována.
- Šablona (dokumenty ve formátu .vm) naformátuje datový model do výstupního kódu. Obsahuje odkazy na subjekty, které patří do datového modelu.
- Jádro šablony (Velocity Apache) představuje aplikaci, která provádí transformaci, tzn. produkuje výstup nahrazením šablony s vnitřními odkazy reálnými daty přicházejícími z modelu.

[8] Uživatelská fóra byla na odpovědi ohledně čistého Java JIRA API rovněž velmi skoupá, ale překvapivě nabízela tucty řešení pro REST API. Pokud bych měl návrh a implementaci zásuvného modulu v tomto prostředí provádět znovu, zřejmě bych se rozhodl právě pro tuto možnost.

Velocity může přijatá data pouze prezentovat a manipulovat (editace, replikování, atp.) s nimi, nikoli je vytvářet. Tento fakt prakticky odrazuje od programování v rámci Velocity šablony [19]. Při složitějších implementacích se nicméně alespoň triviálnímu programování šablon vyhnout nedá.

Podstatou rozdělení projektu mezi Java třídy a Velocity je striktní udržování operačního kódu v Javě a závěrečných zobrazení ve Velocity. Výjimku představuje výchozí sada JIRA API proměnných a funkcí pro Velocity – mezi tyto patří objekty přístupné z jakéhokoli umístění zásuvného modulu a objekty přístupné všem uživatelům systému (např. objekt `projectManager` zprostředkovávající přístup ke stručným informacím o jednotlivých JIRA projektech). Pro správu a přehlednost není rozhodně vhodné zejména vytvářet a rušit instance projektových problémů (*issues*) nebo sprintů.

4.2.1.3 Apache Maven

Maven v základu popisuje projekt (v tomto případě zásuvný modul) pomocí systému Project Object Model (zkr. POM). Tento model popisuje softwarový projekt nejen z pohledu jeho zdrojového kódu, ale včetně závislostí na externích knihovnách, popisu procesu sestavení a různých funkcí s tím spojených (jako je spouštění testů, sbírání informací o zdrojových kódech a podobně). Definující XML dokument se nachází v kořenovém adresáři projektu a je pojmenován *pom.xml*. Pokud je projekt složen z více dílčích projektů nebo modulů, každý z nich má pak svůj vlastní pom.xml soubor, který dědí vlastnosti od nadřazeného souboru a může přidávat další položky. Díky této struktuře je pak možné sestavit celý projekt jediným příkazem [21].

Maven sám je postaven na modulární architektuře a funguje na principu volání jednotlivých zásuvných modulů. Jeho program pouze obstarává dodání a spuštění nadefinovaných modulů. Maven nemá žádné vlastní grafické uživatelské rozhraní a běží pouze na příkazové řádce a pluginy tak mohou využívat všechny nástroje, které dokáží komunikovat pomocí standardních vstupů.

4.2.1.4 Flot (knihovna pro jQuery)

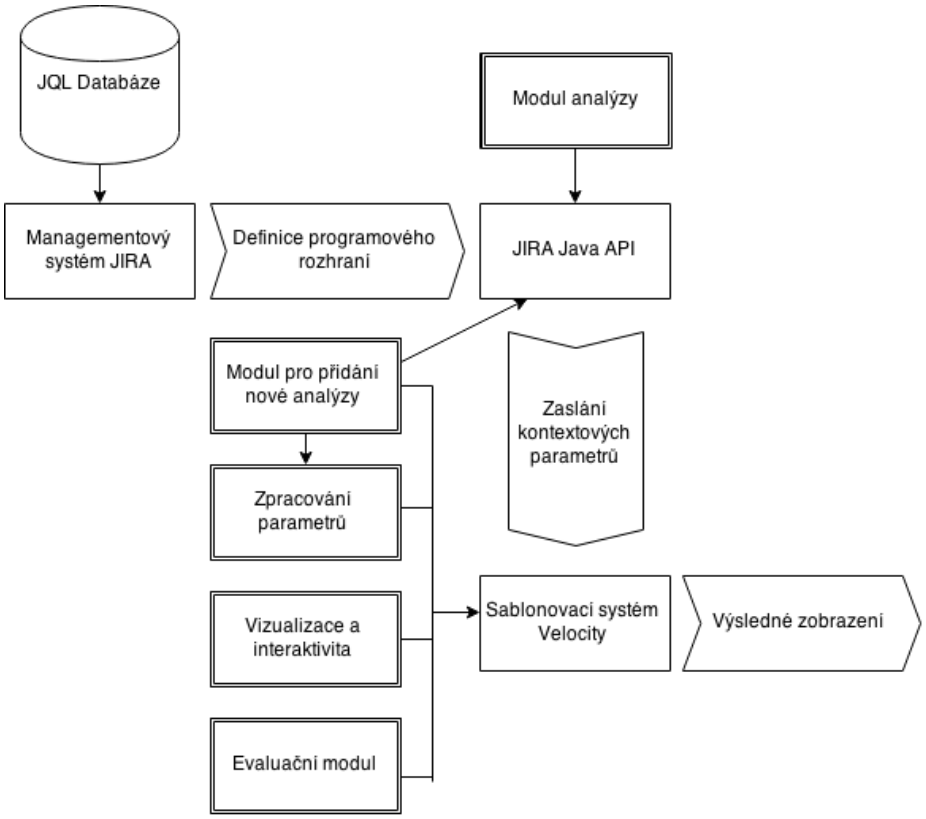
Flot nabízí řešení pro rychlé a uživatelsky příjemné vykreslování grafů a diagramů v jQuery na bázi jazyka JavaScript. Knihovna klade důraz zejména na možnost interaktivity uživatele s vykresleným prvkem. Samotné jádro knihovny Flot je poměrně strohé, většinu nadstandardních funkcionalit nabízí zásuvné moduly (soubory s koncovkou .js obsahující nové funkce) tvořené silnou uživatelskou komunitou [18].

V tomto projektu aktivně využívám rozšíření pro automatickou změnu velikosti grafu při změně velikosti okna prohlížeče (`flot.resize.js`), možnost využívat koláčové grafy (`flot.pie.min.js`), různé typy symbolů pro liniové grafy (`flot.symbol.js`), zobrazování

řetězcových kategorií na x-ové ose grafu (`flot.categories.js`) a možnost pracovat s vrstvenými sloupcovými grafy (`flot.stack.js`).

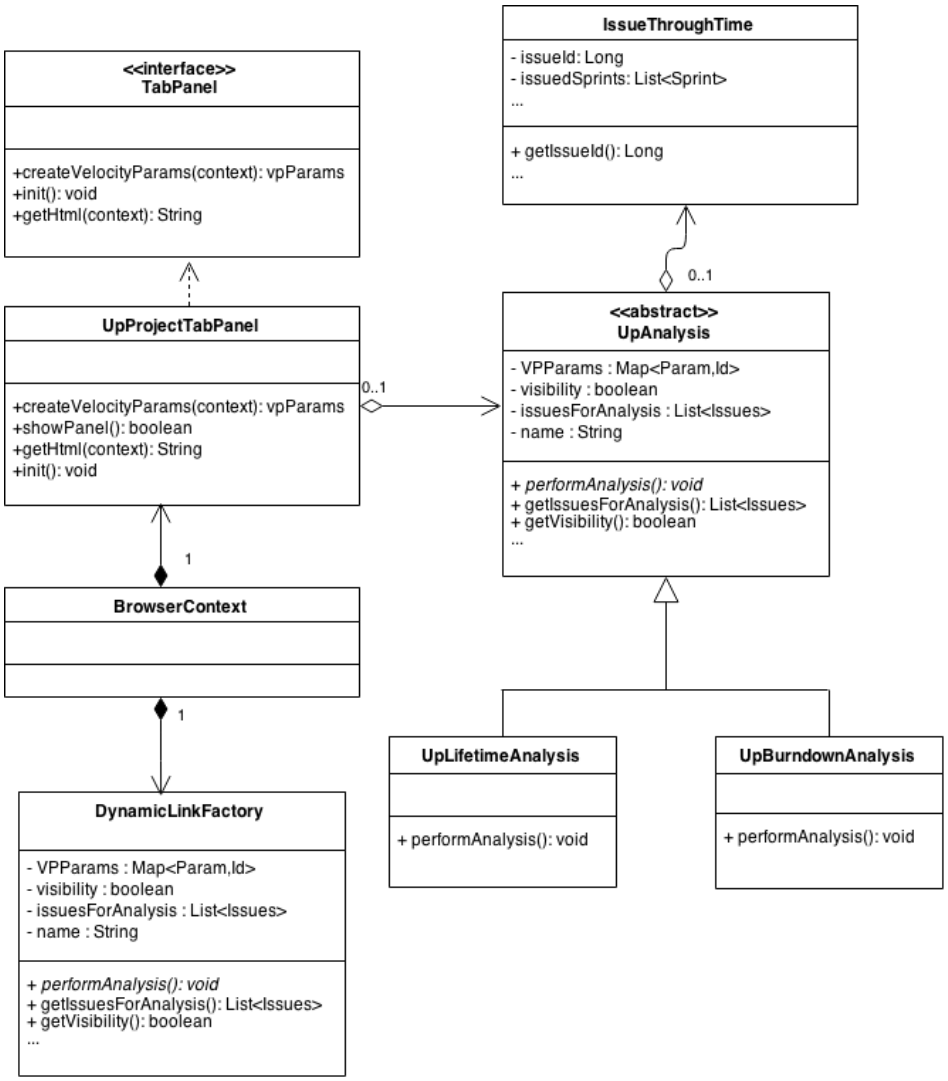
4.2.2 Návrh zásuvného modulu

Po získaných znalostech ohledně komponent, které systém JIRA využívá nebo alespoň podporuje, bylo možné navrhnout konceptuální schéma (viz Obrázek 6). Součásti modulu UpAnalyzer označuje objekt obdélníku s dvojitým rámečkem. Nástroj se bude skládat z pěti hlavních součástí – modulu aktuálních analýz, modulu pro přidání nových analýz, zpracování kontextových parametrů, vizualizace a interaktivity a evaluačního modulu.



Obrázek 6: Konceptuální schéma zasazení nástroje UpAnalyzer do procesu zpracování dat

Již na první pohled je zřetelné, že části modulu analýzy a modulu přidání nové analýzy musí nutně vstupovat do procesu zpracování dat v JIRA Java API, neboť dále (resp. „později“) se již s projektovými daty nedá pracovat. Analýza tak bude muset být stavěna na úrovni objektově orientovaného jazyka Java. Pro přiblížení návrhu uvádím diagram tříd (Obrázek 7), podle kterého byl nakonec implementován modul analýz. Třídu `BrowserContext`, která nepatří do vlastní implementace zásuvného modulu, uvádím pouze orientačně. Generalizace v rámci různých typů analýz je rovněž omezena ukázkou dvou tříd.



Obrázek 7: Diagram tříd pro implementaci v jazyce Java

4.3 Implementace zásuvného modulu

Tato kapitola se věnuje implementaci zásuvného modulu UpAnalyzer. Rozebírám zde důvody k vybrání konkrétních prostředí, standardní postup při implementaci zásuvného modulu v prostředí JIRA a uvádím zajímavé části programu z hlediska struktury, efektivity či inteligentního řešení.

4.3.1 Implementační prostředí

Po vyzkoušení nástrojů obecně doporučovaných vývojářskou komunitou – Eclipse[9] a Netbeans[10] – se nakonec se jako nejpříhodnější ukázalo svižné prostředí rozšířeného poznámkového bloku – programu Notepad++, jež disponuje zvýrazňováním téměř všech myslitelných programovacích jazyků, inteligentním napovídáním (podtržení neznámých znaků, automatické doplňování známých proměnných, metod a syntaktických znaků daného jazyka) a v neposlední řadě i dodržování nastavené štábní kultury (zalamováním řádků, odsazování, apod.). Usoudil jsem, že doba seznamování se s výše zmíněnými nástroji, přednostně určených k vývoji podobných aplikací, nebude odpovídat míře efektivního použití.

Ke kompilaci zdrojových souborů jsem použil Apache Maven Compiler Plugin (dále jen AMCP). Počínaje verzí 3.0, používá tento zásuvný modul firmy Apache jádrový kompilátor `javax.tools.JavaCompiler`. Rodina nástrojů Apache Maven představuje efektivní přístup k řízení a automatizaci jednotlivých sestavení (angl. *build*) Java aplikací, ačkoliv jej lze využít i pro některé další populární jazyky (nejvyšší zastoupení ve využití má po Javě jazyk C#). Hlavním impulzem pro vznik byla snaha o standardizaci a znovupoužitelnost sestavovacích skriptů. Zdroje ostatních typů, jako jsou knihovny jazyka JavaScript, kaskádové styly, a podobně, jsou hierarchicky zpracovány právě za pomoci AMCP a přidruženy k výslednému projektu.

4.3.2 Postup tvorby zásuvného modulu pro Atlassian JIRA

Oficiální zdroj [18], který vyhledávač nabídne hned při prvním pokusu se o problému něco dozvědět, dokumentuje postup tvorby zásuvného modulu do systému JIRA v rozsahu projektu typu "Ahoj Světe". V zásadě tedy poradí jen při počátečních přípravách, mezi které se řadí nainstalování balíčků Java Development Kit a Atlassian SDK (zkratka pro angl. programátorský ustálený výraz Software Development Kit), jejich správné nastavení v rámci proměnné PATH (systém Windows) a další systémová nastavení.

Atlassian SDK nabízí vývojáři k použití řadu užitečných příkazů, které fungují na principu standardních dávkových spustitelných souborů s funkcemi pro veškerý software rodiny Atlassian (Confluence, JIRA, GitHub a další) [17]. Kompletní seznam příkazů lze po instalaci nalézt ve složce `bin`. Celý "kit" stojí na základech rámce Apache Maven, který byl blíže popsán v předchozích kapitolách této práce. Konkrétní podoba nastavení systému Maven se v případě produktů firmy

[9] Volně distribuované integrované vývojové prostředí s modulární architekturou, nejhojněji využívané Java programátory
[10] Druhé nejvyužívanější prostředí pro programování aplikací v jazyku Java, lišící se od NetBeans zejména svou silnou závislostí na komunitně vyvíjených zásuvných modulech. Eclipse v základní verzi obsahuje pouze nejnutnější prostředky jako kompilátor a debugger, ale ostatní součásti je nutné instalovat zvlášť.

Atlassian jmenuje Atlassian Maven Plugin Suite (angl. zkratka AMPS). Mezi nejčastější příkazy užívané při vývoji zásuvného modulu či jiné nadstavby patří:

Název příkazu	**Popis funkce**
`atlas-run-standalone (--product jira)`	Příkaz stáhne nejnovější verzi systému JIRA do počítače, včetně veškerých závislých prvků (angl. *dependencies*), vytvoří Tomcat kontejner a vloží systém JIRA do tohoto kontejneru.
`atlas-create-jira-plugin`	Provede vývojáře nastavením základních parametrů nového zásuvného modulu.
`atlas-package`	Zkompiluje a zabalí výsledný zásuvný modul do .JAR archivu.
`atlas-compile`	Zkompiluje aktuální verzi zásuvného modulu.
`atlas-install-plugin`	Nainstaluje daný zásuvný modul do vybraného systému firmy Atlassian.

Tabulka 5: Příkazy knihovny Atassian SDK využívané modulem UpAnalyzer

V případě vytváření nového zásuvného modulu je potřeba vytvořit tři prvky, které jednoznačně tento software definují:

- `<groupId>` - identifikátor skupiny prvků, do které modul patří (v tomto případě např. `com.atlassian.development`)
- `<artifactId>` - identifikátor modulu (`upanalyzer`)
- `<version>` - aktuální verze produktu

4.3.3 Vnitřní komunikace a rozhraní

Jazyk Java byl při implementaci zásuvného modulu *UpAnalyzer* využit pro jádrovou komunikaci se systémem JIRA (JIRA API je momentálně dostupné pouze v Javě) a získávání informací o projektu a jeho součástech. Po celou dobu návrhu i vývoje byl hlavní důraz kladen na modularitu řešení. Přístup přes k prvkům systému JIRA přes Java API vypadá například takto:

```
Collection<Long> allProjectIssueIDs =
ComponentAccessor.getIssueManager().getIssueIdsForProject(projectId);
List<CustomField> projectCustomFields =
ComponentAccessor.getCustomFieldManager().getCustomFieldObjects(projectId,
ComponentAccessor.getConstantsManager().ALL_ISSUE_TYPES);
```

Zdrojový kód 1: Získání základních informací o analyzovaném projektu

V následujícím textu obecně shrnuji deset implementovaných Java tříd. Funkčně se dělí na ty, které vytvářejí či připravují analýzy, komunikují se systémem JIRA nebo provádějí vedlejší přípravy pro vykreslení.

Třída UpProjectCustomTab

Tato třída zaštiťuje veškerou vnitřní komunikaci se systémem JIRA. Při vytvoření instance tohoto objektu v direktivě Project Object Manageru (soubor pom.xml, viz samostatná kapitola) se volá přetížená metoda `createVelocityParams(BrowseContext ctx)`, jež dědí základní funkčnost z abstraktní třídy JIRA API `AbstractProjectTabPanel`. V aktuálním kontextu systému JIRA (kontext v jednoduchosti představuje balíček různých nastavení – filtrování, řazení, výběr určitých prvků –, které jsou v momentálním zobrazení k dispozici), se připravují instance objektů jednotlivých analýz. Před tímto úkonem se však provedou kroky společné pro vstupy všech analýz, tj. následující:

- získání aktuálního projektu a nastavení (přes veřejnou instanci tzv. `ComponentAccessor` třídy)
- získání přístupu k modulu JIRA Agile (v dřívějších verzích nazýván GreenHopper)
- stažení informací z modulu JIRA a JIRA Agile
- vytvoření sady adekvátních problémů k analýze (viz níže třída `IssueThroughTime`)

Seznam požadovaných informací z JIRA Agile čítá data týkajících se sprintů (identifikátor, název, datum začátku a datum konce). Jelikož systém JIRA Agile pracuje na principu zasílání JSON odpovědí na daný dotaz, bylo potřeba data korektně rozparsovat pomocí regulárních výrazů. V případě neúspěchu se vrací chybová hláška, jež je na klientské straně vypsána na obrazovku, v případě úspěchu tvorby analýz se vrací parametry pro Velocity Apache, který s nimi dále pracuje. Parametry prostředí Velocity z hlediska organizace dat představují mapu klíč-hodnota typů řetězec-objekt (`Map<String, Object>`). Definice projektové lišty (angl. *project tab panel*) probíhá v objektovém XML modelu následovně:

```
<project-tabpanel key="UpAnalyzer" name="UpAnalyzerProjectTab"
        class="com.atlassian.development.upanalyzer.UpProjectTabPanel">
        <label key="UpAnalyzer Project Tab Panel"/>
        <order>200</order>
        <resource type="velocity" name="view"
location="templates/UpProjectTabPanel.vm"/>
    </project-tabpanel>
```

Zdrojový kód 2: Přidání navigačních prvků pro zásuvný modul (pom.xml)

Samotné vložení parametrů do kontextu systému Velocity nastává konsekutivně až v přetížené metodě `getHtml()` pomocí následující konstrukce (volá se tedy totožná funkce nadřazené třídy, tj. deskriptoru):

```
public String getHtml(BrowseContext browseContext) {
        Map<String, Object> params = createVelocityParams(browseContext);
        return super.getHtml("view", params);
}
```

Zdrojový kód 3: Zaslání parametrů z aktuálního kontextu do Velocity

Třída IssueThroughTime

Podstata rozšíření JIRA Agile tkví ve faktu, že původní smysl celého základního systému pro projektový management nebyl v tom, uchovávat informace o *průběhu* jednotlivých problémů, ale o jejich aktuálním stavu. Vzhledem k nově nabytému smyslu sprintu a obecnému zasazení metodiky Scrum do projektového managementu je naprosto nutné přistupovat k objektu problému (*issue*), jako ke chronologickému objektu. Vlastní třída `IssueThroughTime` proto kombinuje prvky *issue* z klasické instance JIRA (třídní proměnné `issueType`, `issueStatus` apod.) uchovává atributy o průběhu jednotlivými sprinty (`Long issueId`, `List<Double> issueStoryPoints`, `String issueType`, `String issueStatus` a další).

Třída DynamicLinkFactory

Tato třída je podtřídou `SimpleLinkFactory`, jež v zásadě programátorovi zprostředkovává přístup k hlavnímu navigačnímu menu v systému JIRA. Modul UpAnalyzer zde získává přístup k tomuto prvku a přidává na něj odkazy na analýzy k veškerým projektům v systému. Děje se tak pomocí následující konstrukce:

```
public List<SimpleLink> getLinks(User user, Map<String, Object> map) {
      List<SimpleLink> links = new ArrayList<SimpleLink>();
      // Získej cestu ke kořenu instance JIRA
      String jiraBaseUrl = ComponentAccessor.getApplicationProperties()
      .getString(APKeys.JIRA_BASEURL);

      // Projdi všechny existující projekty
      ProjectManager pM = ComponentAccessor.getProjectManager();
      for (int i = 0; i < pM.getProjectObjects().size(); i++) {
            Project project = pM.getProjectObjects().get(i);
            // Přidej odkazy na navigační panel
            links.add(new SimpleLinkImpl(project.getName(),
            project.getKey(), null, null, null,
            jiraBaseUrl+"/projects/"+project.getKey()+
```

```
            "?selectedItem=com.atlassian.development.upanalyzer:custom-
            project-tab-panel", null));
        }

        return links;
}
```

Zdrojový kód 4: Přidání navigačních prvků pro zásuvný modul (Java třída)

Třída UpAnalysis

Abstraktní třída, která figuruje jako prostředník v konstruktorech svých podtříd (viz dále). Obsahuje čtyři třídní proměnné, jednoznačně identifikující danou analýzu:

- `Map<String, Object> velocityParameters` – parametry pro vykreslovací systém Velocity; přítomnost této proměnné v každé analýze v podstatě znamená možnost zobrazení analýzy prakticky na jakémkoli místě (kontextu) systému JIRA
- `List<IssueThroughTime> issuesForAnalysis` – seznam úkolů a příběhů, které jsou vstupem pro danou analýzu; znamená to, že každá analýza může fungovat nad libovolnou populací či vzorkem dat
- `String name` – název analýzy, který se pak bude zobrazovat v reportu
- `boolean visibility` – proměnná s binární hodnotou pro určení, zda je analýza viditelná v reportu

Třída UpLifetimeAnalysis

Třída přetěžuje metodu své nadtřídy `UpAnalysis performAnalysis()`, která navrací parametry pro systém Velocity, reprezentující výsledky analýzy životnosti. Ty se dále dělí na výsledky pro tabulární a pro grafové zobrazení. Vzhledem k fungování knihovny pro vykreslení grafů Flot (viz další kapitoly) a struktury standardní HTML tabulky, se tyto dva přístupy nepodařilo sloučit.

Třída UpWorkloadAnalysis

Třída přetěžuje totožnou metodu své nadtřídy `UpAnalysis`, která navrací parametry pro systém Velocity, reprezentující výsledky analýzy pracovní zátěže. Ty se vracejí pouze pro tabulární zobrazeuní. Grafické zobrazení bylo řešeno, ale vzhledem k množství parametrů analýzy se ukázalo být spíše matoucí a nepřehledné.

Třída UpPEAnalysis

Třída přetěžuje totožnou metodu své nadtřídy `UpAnalysis`, která navrací parametry pro systém Velocity, reprezentující výsledky analýzy priorit a úsilí. Ty se vracejí pouze pro grafové zobrazení. Zde by naopak tabulární zobrazení bylo příliš triviální a diagramové zobrazení naprosto dostačující.

Třída UpCorrelationAnalysis

Třída přetěžuje totožnou metodu své nadtřídy, která navrací parametry pro systém Velocity, reprezentující výsledky analýzy životnosti. Ty se dále dělí na výsledky pro tabulární a pro grafové zobrazení.

Třída UpBurndownAnalysis

Třída přetěžuje totožnou metodu své nadtřídy `UpAnalysis`, která navrací parametry pro systém Velocity, reprezentující výsledky analýzy životnosti. Ty se dále dělí na výsledky pro tabulární a pro grafové zobrazení.

Třída UpTable

Třída poskytuje rozhraní pro tvorbu výchozího naformátovaného typu tabulek, které se následně posílá jako jeden parametr do produkce Velocity. Její metoda `generateTable(String [] tableHeader, Object[][] tableArray)` tento parametr ukládá. Výhoda tkví v možnosti vkládat do tabulky jakákoli data vzhledem ke generičnosti Java typu `Object`.

4.3.4 Transformace parametrů a vizuální zpracování

Transformace parametrů probíhá na pomezí systému Velocity a jazyku JavaScript. Jedná se vlastně o převod různých syntaxí. Tento úkol, který není z popisného hlediska tak zajímavý jako z hlediska implementačního, dokumentuje ukázka zdrojového kódu v kapitole 4.3.7.3. Získané hodnoty parametrů přenáší konstrukce jazyku JavaScript prostřednictvím různých funkcí do závěrečných podob. Funkce na straně klienta (prohlížeče) implementované speciálně pro modul UpAnalyzer lze rozdělit do několika skupin (funkce nelze vzhledem k početnosti a rozsahu jednotlivě komentovat):

- Funkce spojené s tvorbou PDF verze reportu – funkce přidávající HTML5 objekt typu `canvas`, generování dokumentu
- Funkce pro správu prostředí – např. funkce pro navigační tlačítka, zobrazení sumáře a výpisů
- Funkce pro vyhodnocení projektu – funkce pro nastavení uživatelských tolerancí pro jednotlivé analýzy, hodnotící algoritmy, základní statistické algoritmy
- Funkce pro vykreslování – nastavení událostí při přejezdu myší nad různými částmi grafu, přepínání typů grafu a podobně

Závěrečným krokem z pohledu implementace rozšíření UpAnalyzer byla otázka pozicování vygenerovaného HTML dokumentu, kde jsem využil principů tabulkových rozložení (angl. *table-cell layout*) tak, aby jednotlivé analýzy vyplňovaly vždy celou plochu (a jejich části, nezávisle na počtu, vždy ideálně velkou relativní část plochy).

4.3.5 Externí knihovny a závislostí

Využívané závislosti na externích knihovnách pro potřeby modulu UpAnalyzer jsou následující:

Závislost (angl. dependency)	**Popis funkce**
`jira.web.resources:jquery`	Verze knihovny jQuery udržovaná v aktuální instalaci systému JIRA. Obecně se doporučuje držet se při programování vlastností této verze, ač nemusí být nejnovější. Při snaze o přidání novější verze dochází k nečekaným kolizím.
`jira.webresources:global-static`	Balíček datových prvků typických pro systém JIRA (správa tzv. *issues*, projektů apod.)
`com.atlassian.auiplugin:ajs`	AUI je knihovna pro tvorbu uživatelského rozhraní dle přesných designových výměr firmy Atlassian (panely, formuláře apod.)
`com.atlassian.auiplugin:jquery-ui-other`	Řada doplňujících vizuálních funkcí pro knihovnu AUI. Napsáno v jQuery.
`com.atlassian.auiplugin:aui-experimental-table-sortable`	Speciální tabulkové prvky knihovny JIRA AUI. Blíže popsané v kapitole 4.3.1.3.
`jquery.flot`	Grafická knihovna pro tvorbu a správu interaktivních diagramů.
`BlobBuilder, FileSaver`	Knihovny sloužící ke správě a vytváření souborů.
`jspdf`	Knihovna pro vytváření PDF dokumentů z HTML dokumentů a přidružených kaskádových stylů.

Tabulka 6: Využívané knihovny v projektu UpAnalyzer

Jednotlivá rozšíření byla v několika bodech jemně upravena pro potřeby tohoto konkrétního projektu. Například pro názornost jedné z analýz bylo vhodné přidat pro každý bod liniového grafu symbol šipky, přičemž úprava spočívala v definici dodatečné funkce přímo v knihovně Flot:

```
// Vykreslení šipky v závislosti na sklonu čáry grafu
triangle: function (ctx, x, y, radius, shadow) {
      var size = radius * Math.sqrt(2*Math.PI/Math.sin(Math.PI/3));
      var height = size * Math.sin(Math.PI / 3);
      ctx.moveTo(x - size/2, y + height/2);
      ctx.lineTo(x + size/2, y + height/2);
      if (!shadow) {
          ctx.lineTo(x, y - height/2);
          ctx.lineTo(x - size/2, y + height/2);
      }
}
```

Zdrojový kód 5: Ukázka ad-hoc změny v knihovně Flot

4.3.6 Modul pro přidání další analýzy

Pro možnost rozšíření sady analýz byl vytvořen speciální modul ve formě spustitelného souboru (formát .exe). Modul pracuje na úrovni příkazové řádky, kde uživateli podá 9 dotazů týkajících se typu analýzy, zobrazení, typu grafu, popisu apod. Program, který byl vytvořený v jazyce Python, pak uživatelské vstupy zpracuje a na jejich základě doplní do zdrojové struktury kódu příslušné nastavení. Jazyk Python byl k implementaci zvolen proto, že v současnosti představuje jeden z nejlepších a nejrychlejších nástrojů pro modifikaci textových souborů. Automatická modifikace kódu probíhá na následujících úrovních:

- Vytvoření nové Java třídy pro analýzu a popis základních metod
- Přidání volání do hlavní třídy, která ovládá systém analýz
- Zpracování parametrů
- Vykreslení analýzy (diagram, tabulka nebo obojí)

Z výše uvedeného popisu je zjevné, že modul neprovede samotnou implementaci třídy pro analýzu, což by při principu programových dotazů na uživatele nebylo triviální úlohou (tuto skutečnost diskutuji v kapitole o možných rozšířeních). Uživatel je nicméně formou komentářů a příkladů informován o tom, jak přistupovat k jednotlivým prvkům rozhraní. Se znalostí několik příkazu lze jednoduchou analýzu vytvořit během několika minut.

4.3.7 Problémy a zajímavosti při implementaci

Následující podkapitola shrnuje výčet několika nejzásadnějších problémů při implementaci zásuvného modulu *UpAnalyzer*. Snažil jsem se pro tento stručný seznam vybrat problémy, jejichž řešení

bylo nějakým způsobem zajímavé. Problémy jsou seřazeny dle chronologické posloupnosti implementace.

4.3.7.1 Využívání funkcionalit cizích modulů systému JIRA

V průběhu zkoumání aplikačního rozhraní systému JIRA jsem došel k závěru, že vývojáři ostatních rozšíření nejsou vzhledem k finanční politice produktu příliš ochotni sdílet přístupy k vlastním třídám a instancím. Jelikož *UpAnalyzer* ale potřebuje pro svůj chod získávat data, která úzce souvisejí konkrétně s rozšířením JIRA Agile (z diskusí na fórech věnujících se produktům firmy Atlassian bylo možné dohledat četné konverzace s podobnými neúspěchy vývojářů[11]), bylo nutné tento problém nějakým způsobem eliminovat. V podstatě se řešení našlo ve dvou poměrně odlišných dimenzích. Obě byly zaneseny do zdrojového kódu programu vzhledem k možnosti, že se vývojářský tým JIRA Agile rozhodne minimálně jednomu z těmto přístupů nějakým jiným způsobem zamezit. Dokumentační stránky rozšíření samozřejmě existují[12], ale limitace ze strany vývojářů se zvětšují. Následující dva odstavce popisují obě řešení problému.

Přístup k objektům přes REST protokol

Representational State Transfer (zkráceně REST) představuje nestavový protokol, resp. celou architekturu nad tímto protokolem, který se vyznačuje vysokou použitelností v distribuovaných systémech. Na rozdíl od SOAP (Simple Object Application Protocol) se REST orientuje na data, nikoli na procedury. JIRA definuje pro operaci s některými datovými prvky aplikační rozhraní JIRA REST API[13]. O tomto přístupu jsem se zmiňoval již na začátku kapitoly 4 v souvislosti rozhodování o celkové povaze implementace zásuvného modulu.

Dále je potřeba zjistit, jak REST API funguje v zásuvných modulech, jejichž služby chce vývojář použít. Pro JIRA Agile například vypadá posloupnost dotazů pro získání informací o sprintu s identifikátorem 1 následovně:

- `/rest/greenhopper/1.0/rapidview` – vrací seznam tzv. *rapid-views* (JIRA Agile pohledů na data)
- `/rest/greenhopper/1.0/sprints/{rapidViewId}` – vrací seznam sprintů aktuálního pohledu

[11] "*From my experience most of the Service objects from GreenHopper are not exposed to other plugins. You'd be best attempting to do it via the REST interface. Not that GH has a published REST interface but you can discover it fine using the Developer Tools plugin and REST browser in that."* – úryvek z interentové diskuze na portálu stackoverflow.com
[12] Dostupné z: https://confluence.atlassian.com/display/AGILE/JIRA+Agile+Documentation
[13] Dokumentace k tomuto rozhraní je dostupná z: https://docs.atlassian.com/jira/REST/latest/

- `/rest/greenhopper/1.0/rapid/charts/sprintreport?rapidViewId={rapidViewId}&sprintId={sprintID}}` – vrací seznam úkolů či příběhů, které souvisejí s aktuálním sprintem

Přístup přes REST API staví na hierarchičnosti a logice, nicméně je z pohledu vývojáře injektovaného modulu poměrně prosté jej změnit a narušit tak funkčnost aplikací, které z něho data čerpají.

Přístup k objektům přes OSGi platformu

OSGi Service Platform (dále jen OSGi) specifikuje standard modulárního systému pro jazyk Java. Tato platforma umožňuje instalaci a odebírání Java modulů za běhu a nabízí infrastrukturu pro spolupráci modulů skrze služby. Jednotkou modularity v OSGi je takzvaný tzv. *bundle* (neboli modul). Každý modul má za běhu speciálně vytvořený zavaděč tříd (angl. *classloader*), který vidí pouze balíčky a zdroje (obrázky, konfigurační soubory a další) definované v modulu samotném, relevantní části standardní knihovny a balíčky z ostatních modulů, které explicitně importuje. Platforma navíc nabízí vrstvu služeb, které tvoří dynamické závislosti. To znamená, že po čas běhu programu mohou vstupovat (registrovat se) do procesu nové služby s tím, jak se do kontejneru instalují nové moduly. Existující služby mohou být také odregistrovány. Toto dynamické chování klade specifické nároky na klientský kód, který služeb využívá. Toto řešení bylo nakonec pro implementaci zásuvného modelu vybráno jako nejoptimálnější (z pohledu zachování třídního modelu):

```
// Funkce getGreenHopperAppCtx získává přes
// rozhraní OSGi tzv. bundle (balíček) vnitřní
// implementace modulu JIRA Agile (GreenHopper)
private Object getGreenHopperAppCtx() throws InvalidSyntaxException {
      OsgiContainerManager osgi = ComponentAccessor.
      getComponentOfType(OsgiContainerManager.class);

Bundle[] bundles = osgi.getBundles();
for (int i = 0; i < bundles.length; i++) {
     Bundle bundle = bundles[i];

     if ("com.pyxis.greenhopper.jira"
          .equals(bundle.getSymbolicName())) {
      // Získání balíčku (bundle) pro JIRA Agile
      BundleContext bctx = bundle.getBundleContext();
      ServiceReference[] rfs =
      bctx.getAllServiceReferences(null,null);
      if (refs != null) {
         for (int j = 0; j < rfs.length; j++) {
            Object prop = rfs[j]
            .getProperty("org.springframework.context.service.name");
            if ("com.pyxis.greenhopper.jira".equals(prop)) {
                  // Vrácení kontextové služby balíčku
                  return bctx.getService(refs[j]);
             }
          }
```

```
            }
          }
        }
        return null;
    }
```

Zdrojový kód 6: Ukázka využití modulu OSGi pro získání balíčku JIRA Agile

Aplikaci této metody uvádí kód pod tímto textem. Aktuální aplikační kontext JIRA lze s využitím metody `getGreenHopperAppCtx()` jednoduše přetypovat na kontext JIRA Agile a pomocí děděné metody `getBean()` získat libovolný objekt v daném kontextu (zde `SprintManagerImpl`):

```
      ApplicationContext appCtx = (ApplicationContext)
getGreenHopperAppCtx();
      if (appCtx != null) {
            return (SprintManager) appCtx.getBean("sprintManagerImpl");
      }
```

Zdrojový kód 7: Získání cizího objektu třídy sprintManagerImpl

4.3.7.2 Extrahování historických záznamů

Další problém, který bylo nutné řešit, souvisí rovněž s prostředím zásuvného modulu JIRA Agile. Status jednotlivého problému (angl. *issue*) či úkolů se logicky během sprintů mění a některé analýzy potřebují s takovou informací pracovat. Samotná JIRA však tuto informaci sama o sobě neobsahovala a jedinou možností bylo získat přístup do databáze a přečíst aktualizovanou zálohu po ukončení každého sprintu, což se u mnoha projektů na klientské straně nedělo. JIRA od verze 6.3 přináší tuto informaci ve vlastnostech třídy `ChangeHistoryManager`. Vzhledem k nepřílišnému stáří implementace nebyl ale systém ještě dokonale zdokumentován, a vlastní řešení tak bylo nutné hledat metodou pokus-omyl. Jelikož se jedná o poměrně zajímavou vlastnost, která dle mého názoru bude potřebná pro další vývojářské projekty v tomto prostředí, uvádím níže krátkou ukázku:

```
// Získání objektu pro správu historie
ChangeHistoryManager dchhm = ComponentAccessor.
getChangeHistoryManager();
// Přístup k historii konkrétního úkolu
chhI = dchhm.getChangeHistories(theIssue);
if (chhI.get(j).getChangeItemBeans().
   get(i).getField().equals("status")) {
   // Údaj o změně statusu
   statusId = chhI.get(j).getChangeItemBeans().get(i).getTo();
   statusName = ComponentAccessor.getConstantsManager().
   getStatusObject(statusId).getSimpleStatus().getName();
   // ... další práce s objektem historie
}
```

Zdrojový kód 8: Získání historických informací o jednotlivých problémech

4.3.7.3 Kolize syntaxí jazyků Velocity a JavaScript

Vzhledem k relativně vysokému počtu užitých programovacích jazyků v práci na projektu bylo v některých případech nutné řešit problém kolize syntaxí v jednom zdrojovém souboru. Tento problém se nejsilněji projevil při zpracování parametrů v systému Velocity a jejich převod do datových struktur jazyka JavaScript, popř. knihovny jQuery. Velocity jednak užívá znak dolaru pro identifikaci svých proměnných, zatímco knihovna jQuery používá tento znak pro identifikaci vlastních funkcí. Vyřešit se tato kolize dá nahrazením identifikátoru jQuery knihovny (znaku dolaru) za řetězec „*jQuery*“.

Větším problém pro programátora však představuje neschopnost systému Velocity pracovat se některými datovými strukturami, které Java přes parametry může k vykreslování zasílat. Typickým příkladem je dvourozměrné pole (matice), využívané při vykreslování tabulek s výsledky některých analýz. Verze systému Velocity, kterou používá JIRA, neumí k takovým objektům efektivně přistupovat a řešit problém lze přes zasílání *n* parametrů formátu jednorozměrných polí (kde *n* = počet řádků matice). Jelikož se ale tabulka vytváří v jazyku Java dynamicky a předem neznáme počet řádků, musí se vytvářené parametry dynamicky pojmenovávat. To přináší do jazyka Velocity nepříjemný problém, který je nutné „obejít“ následujícím způsobem (znázorněné na případu analýzy pracovní zátěže):

```
var WL_val = [[]];
var WL_val_splitted = [[]];
## Cyklus vykreslovacího systému Velocity
#foreach($i in [1..$WL_col_length])
      ## přiřazení dynamického názvu proměnné
      #set($WL_v = "$WL_values_$i")
      ## předběžné vyhodnocení proměnné pomocí makra evaluate
      #set($WL_v_notescaped = "#evaluate($WL_v)")
      WL_val[$i] = $WL_v_notescaped + '';
      WL_val_splitted[$i] = WL_val[$i].split(',');
#end
```

Zdrojový kód 9: Ukázka převodu dvojrozměrného pole z Velocity do jazyka JavaScript

4.3.7.4 Propojení knihoven js2pdf a Flot

Jednou z nesourodostí plynoucí z přílišně jednotného zaměření JavaScript knihoven bylo uvedení v chod grafů knihovny `Flot` společně s vykreslením do PDF, zprostředkovaného knihovnou `js2pdf`. Problém tkví v tom, že knihovna vytváří graf nikoli jako statický obrázek, ale ukládá ho do (s příchodem konceptu HTML5) velmi populárního formátu *canvas* (volně přeložitelné jako plátno), jež v podstatě znamená vykreslování rozličných vizualizací na klientské straně pomocí JavaScriptu. Neděje se tak ale s dodatečnými prvky grafu jako jsou popisky os a legenda, jež jsou pro správné pochopení grafu uživatelem poměrně esenciální. Ty jsou patrně díky nejednotnému vývoji knihovny přidány pouze do kontejnerů `div`, které ve výchozím nastavení exportovatelné nejsou.

Řešení se v tomto případě rovněž nabízí několik:

- Využití volně distribuovaného skriptu `html2canvas`[14]. Tento skript umožňuje vytvářet záznamy aktuálního zobrazení na displeji (angl. *screenshot*) z webových stránek. Výsledek je ale stejně založen na užívaném datovém objektovém modelu a jako takový nemusí být 100% přesný.
- Vložení dodatečných informací (popisky os, legenda) přímo do grafu. Tento přístup vyžaduje poměrně značný zásah do skriptů knihovny `Flot`.

4.4 Dokumentace a testování

Patrně jedním z nejnáročnějších úkolů, který se při vývoji zásuvného modulu pro systém JIRA naskýtá, je testování, což lze chápat jako malý paradox vzhledem k povaze této práce. Produkt nebyl testován pouze závěrečně a celistvě, ale jeho rozsah a až výjimečná odlišnost jednotlivých sub-modulů nutila k postupnému postupu implementace a testování, leckdy ve smyslu metody pokus-omyl.

Testování jednotlivých implementačních prvků zabralo značnou porci času, a to zejména proto, že protože zásuvný modul je vždy při novém běhu nutné zkompilovat a posléze pomocí dialogového okna systému JIRA nahrát, přičemž celý proces zabere (v závislosti na procentu využívané paměti, kterou lokální JIRA instance zabere) bezmála několik desítek sekund.

V tak citlivých změnách, jako je například pozicování jednotlivých prvků vizuálního prostředí bylo možné alespoň pro některé jednodušší případy využít populární nástroj `jsFiddle`[15]. Tento software představuje jakési hřiště (v angličtině aktuálně často používaný název pro tento typ nástrojů – *playground*) pro webové vývojáře, nástroj, který může být použit v mnoha ohledech. Nejznámější se stal ale jednoznačně pro své extrémně svižné on-line editování a instantní zobrazování výsledků kódu jazyků HTML, CSS a JavaScript, potažmo jQuery. Pomocí tohoto přístupu se může vývojář JavaScriptu velmi snadno vyvarovat chyb. Jelikož ale zásuvný modul vnořuje svou vizuální stránku do existujícího rozmístění okolích navigačních prvků systému JIRA, nebylo možné využít výše zmiňovaný nástroj pro každý případ (deformace původních kaskádových stylů a podobně).

Co se týče části datového modelu (Java třídy), Atlassian definuje ve své dokumentaci typy testů [19] [21], které jsou používány pro testování vývojářských rozšíření. Jejich přehled s dodatečnou informací o tom, zda byly využity v případě tohoto projektu a proč, popisuje následující tabulka:

[14] Skript je dostupný z adresy: http://html2canvas.hertzen.com/
[15] Oficiální verze je umístěná na adrese http://www.jsfiddle.com

Typ testu	Popis	Aplikováno
Jednotkové testy	Kontrolují jedinečnou funkci nebo třídní metodu	ANO
Integrační testy	Testují integraci mezi zásuvným modulem a systémem JIRA nebo jiným externím modulem	ANO
Funkční testy	Testují funkčnost vlastností a funkcí aplikace	ANO, manuálně
Zátěžové testy	Testují chování systému pod tlakem velkého počtu přístupů a užívání v daném období	NE, nepředpokládá se vyšší než jednotkový počet konkurenčních uživatelů
Akceptační testy	Testují, jak dobře aplikace splňuje požadavky zákazníka	ANO, na straně zákazníka

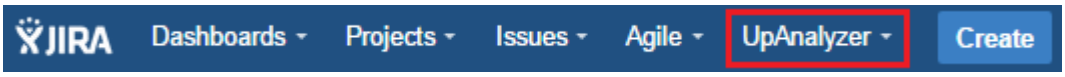
Tabulka 7: Seznam testů standardně využívaných při vývoji zásuvného modulu systému JIRA

Testování na reálných datech přímo na instanci sytému JIRA mělo patrně největší podíl na identifikaci chyb a nedostatků, zejména proto, že mnou vytvořená instance zdaleka neobsahovala tak rozsáhlá projektová data.

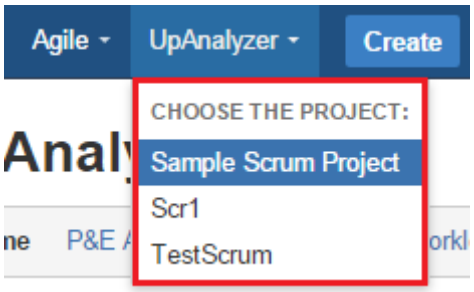
Pro potřeby větších zásahů do kódu (případná změna organizace JIRA API apod.) byla vytvořena dokumentace zdrojového kódu za pomoci softwaru Doxygen. Obsahem generované dokumentace je pak popis jednotlivých skriptů, funkcí, tříd a proměnných, spolu s jejich vlastnostmi a parametry. Dokumentace na úrovni funkcí jazyka JavaScript a HTML je přítomná pouze ve zdrojových souborech kódu. Veškeré doprovodné informace jsou pro možnost obsluhy a instalace zásuvného modulu v zahraničních pobočkách firmy uváděny v anglickém jazyce.

4.5 Popis prostředí zásuvného modulu UpAnalyzer

V podstatě se rozšíření skládá z modulů jednotlivých analýz, jež jsou vykresleny do projektového panelu, ke kterému se dá přistoupit buď ze správy projektu v levém menu a dále sub-menu s názvem „Add-ons“ (testováno v současnosti na poslední verzi JIRA 6.4, která je zejména z designového hlediska trochu odlišná), nebo jednodušeji v horním panelu kliknutím na položku UpAnalyzer s vertikálním vysunovacím menu (obrázky 8 a 9 pod tímto textem), kde lze vybrat zkoumaný projekt.

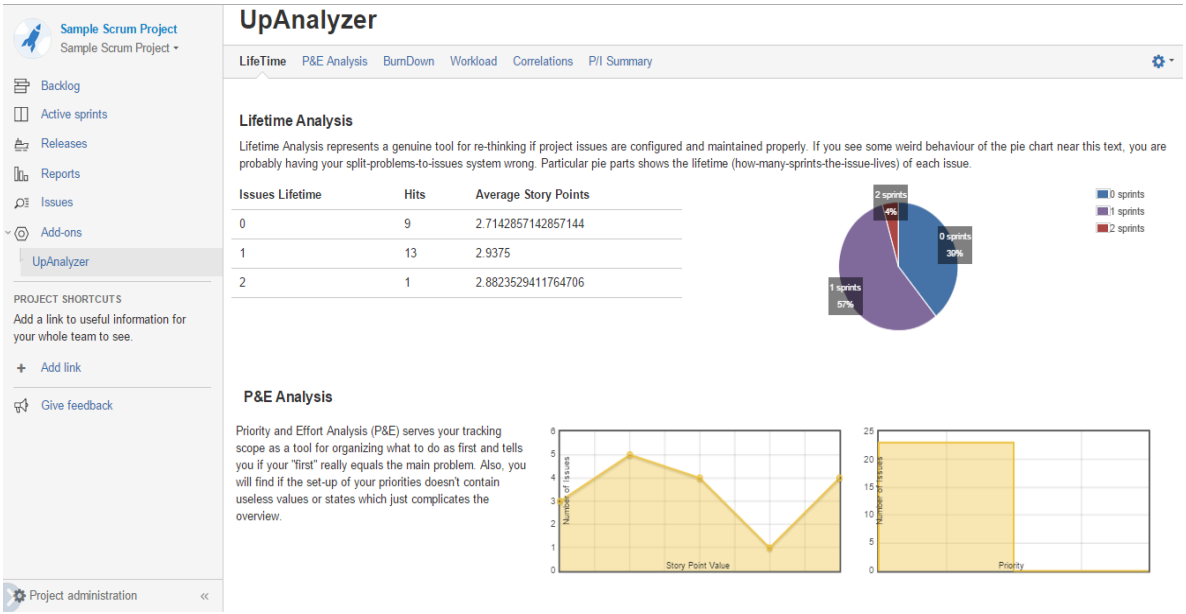


Obrázek 8: Zobrazení možností zásuvného modulu v navigačním menu (nerozbalené)



Obrázek 9: Zobrazení možností zásuvného modulu v navigačním menu (rozbalené)

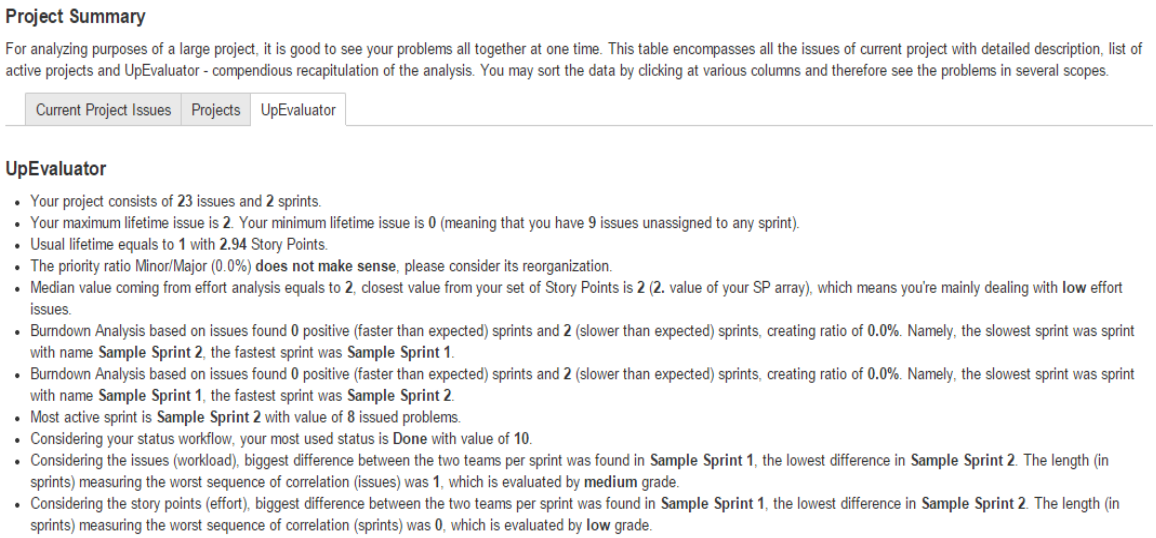
Samotné rozhraní obsahuje navigační menu s jednotlivými moduly analýz (Obrázek 10 Obrázek 12), které se skládají buď z tabulkového, nebo diagramového zobrazení analýzy k danému projektu, včetně vysvětlujícího textového popisu. U některých analýz lze přepínat pohledy grafů. Analýzy získávají vstupní data dynamicky, to znamená, že pro aktualizovaný pohled netřeba nic jiného než přidání úkolu či příběhu do systému, změna parametru či jakákoli jiná klasická rutina v JIRA a následná aktualizace stránky se zásuvným modulem. Celý proces aktualizace zabere maximálně jednotky sekund, nejvážnějším konzumentem paměti je samotné nahrání vizuálních prvků systému JIRA.



Obrázek 10: Horní část vizuálního rozhraní zásuvného modulu

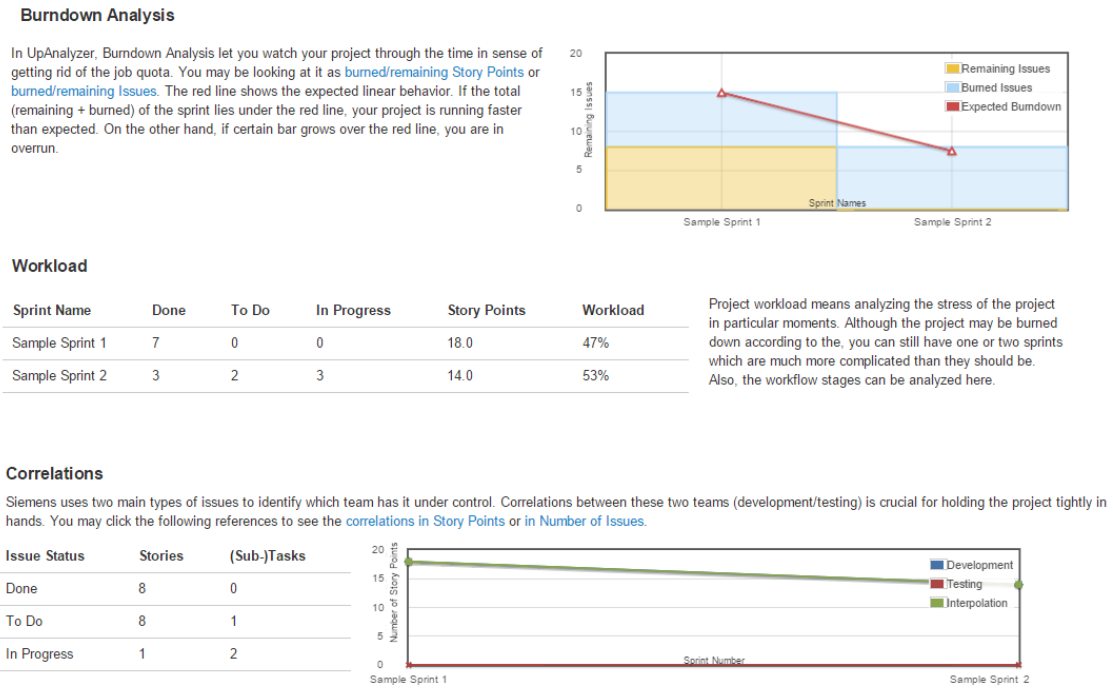
Po sekcích jednotlivých analýz (v základní verzi zásuvný modul nabízí pět typů analýz podrobněji rozebraných na konci třetí kapitoly této práce) obsahuje zásuvný modul shrnující sekci s

názvem *UpEvaluator* (viz Obrázek 11), kde oznamuje nejzajímavější statistiky v textové formě s občasnými návrhy, co je na projektu z hlediska číselné evaluace dostupných dat dobře, a co je špatně.



Obrázek 11: Zobrazení evaluace projektu

Dále tato část obsahuje podsekce s výčtem úkolů a příběhů, které se k projektu vážou a možností řadit si je podle různých priorit. Po kliknutí na tlačítko v pravém horním rohu na navigačním panelu je možné si aktuální stav projektu uložit do jednoduchého PDF, které by mělo sloužit jako záznam stavu projektu, protože se samozřejmě s jakoukoli změnou celá analýza přepočítá. PDF obsahuje shrnující text z evaluátoru daného projektu a naformátované výsledky jednotlivých analýz. Vzhledem k možnému navýšení počtu možných uživatelů bylo celé rozhraní navrženo pro anglicky hovořící osoby.

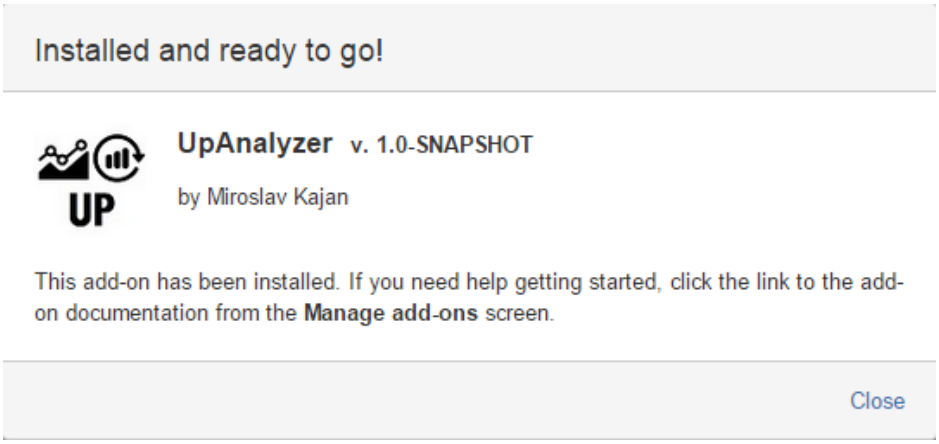


Obrázek 12: Spodní část vizuálního rozhraní zásuvného modulu

4.6 Konfigurace a omezení

Pokud si uživatel přeje projekt sestavit, stačí mu již dříve popisované balíčky JDK a Atlassian SDK. Po zavolání příkazu `atlas-package` v kořenové složce projektu vzniká kompilát ve formátu .jar, který se ukládá do projektové složky *target*.

Zásuvný modul UpAnalyzer lze pak do systému JIRA nahrát několika způsoby, uživatelsky nejpříjemnější se ale jeví možnost využít tzv. Universal Plugin Manager (zkr. UPM), jež JIRA obsahuje jako vlastní modul. Nástroj lze nalézt na adrese servername/jira/plugins/servlet/upm#manage, kde řetězec *servername* odpovídá adrese serveru, na kterém je instance JIRA nainstalována. Po kliknutí na tlačítko „Upload Add-on“ uživatel vybere soubor .jar se zkompilovaným zásuvným modulem. Úspěšné nahrání modulu UpAnalyzer do systému by mělo znamenat zobrazení následujícího dialogového okna:



Obrázek 13: Dialog s oznámením o úspěšné instalaci do systému JIRA

Zásuvný modul byl vyvíjen a testován pod posledními verzemi prohlížečů Chrome (stub 42) a Firefox (Portable 35). Internet Explorer se obecně drží odlišných přístupů v oblasti pozicování, a proto se někdy text či diagram nemusí ideálně zobrazovat. Zejména je potřeba dávat na tento fakt pozor v případě přidávání dalších analýz.

Z hlediska konfigurace se jako administrátor JIRA ujistěte, že máte nainstalovanou verzi systému vyšší nebo rovnou 4.x, protože vnitřní logika programu používá některé konstrukce, které byly přidány až právě pro zmiňovanou verzi. Podobný případ nastává pro rozšíření JIRA Agile (vyšší nebo rovná 2.x). Uživatelská pole vyžadovaná zásuvným modulem musí být ve vašem projektu viditelná (jedná se o následující: priorita, status, „*story points*“ a sprint). V jiném případě bude program na hlavní liště hlásit výjimku.

4.7 Další možná rozšíření

V následujících odstavcích uvádím návrhy na možná vylepšení řešeného projektu, která by mohla být v dalších verzích při dostatku zdrojů či znalostí implementována.

4.7.1 Aktualizování stavu dle výsledku v prostředí Selenium

Jedním z možných rozšíření analyzátoru by bylo dialogové okno přístupné přes tlačítko v nastavení. V tomto okně by uživatel nastavil konexi k prostředí Selenium (blíže představované v kapitole 3), a to nejpravděpodobněji definováním cesty k logovacím záznamům ze zmiňovaného nástroje. Tyto záznamy se uchovávají ve formátu tabulkového editoru Excel. Stáhnutím dat z dostupných listů a jejich jednoduchou analýzou (kdy každý test má unikátní identifikátor a hodnotu výsledku testu formátu „*PASS/FAIL*“), by se aktualizovaly tyto stavy testů v systému JIRA. Každý záznam stavu testu by se dohledal pomocí typu problému (angl. *bug*) a identifikátoru záznamu. Rozšíření by přispělo k automatizaci testovacího procesu.

4.7.2 Interpretované zpracování prvku pro přidávání analýz

V současné době stojí modul pro jednoduché přidávání vlastních analýz na spustitelném souboru (ve formátu .exe), který v podstatě modifikuje kód modulu UpAnalyzer a je tedy nutné kód vždy zkompilovat a nahrát do systému JIRA. Nabízí se navrhnout interaktivní řešení přímo v systému JIRA postaveném logicky na podobném systému. Implementace takového řešení by ale nebyla úplně triviální, a proto nebyla do projektu zařazena. Bylo by nutné zohlednit minimálně následující:

- Vytvoření vizuálních prvků pro přidání analýzy
- Míra dostupných nastavení
- Uchování kontextu v dalších instancích systému JIRA (kam aplikační logiku analýzy efektivně uložit?)

5 Závěr

Předmětem teoretické části této práce bylo nabídnout konzistentní a aktuální shrnutí metod a typů organizace softwarového testování s ukázkou jejich možností a doporučení s orientací na agilní metodiky a postupy. V další části dokument obsahuje definici metodik a práci týmu testerů brněnské divize firmy Siemens CZ. Ukázalo se, že užívané metodiky nelze jednoznačně zařadit do předem definovaných typů. Cílem firem s komplexnější organizací není výběr jedné metodiky, ale snaha o kombinaci nejpoužitelnějších vlastností z každé z nich. V rámci získaných znalostí provádí experimentální zhodnocení reálného projektu zejména z hlediska testovacích metrik se snahou navrhnout vylepšení firemních testovacích procesů jako celku.

Z provedených analýz vyplynulo několik skutečností, které by mohly posloužit k lepšímu návrhu testovacích procesů a spolupráce týmu testerů s týmem vývojářů.

Tyto výsledky jsou v diplomové práci dále rozpracovány a prakticky formují základ k návrhu a implementaci zásuvného modulu pro nástroj k podpoře projektového řízení – Atlassian JIRA –, který modulární sadou analýz zjednodušuje testerovi orientaci při práci na aktuálních projektech a nabízí možnost změnit neúspěšné praktiky procházením analýz projektů předchozích. Vytvořený modul lze v zásadě využívat ke dvojí činnosti – sledování a optimalizaci projektu při jeho průběhu a vyhodnocení a zjištění nedostatků po skončení projektu.

Přednosti implementovaného modulu spočívají v dynamickém a interaktivním prostředí, možnosti jednoduchého přidávání vlastních analýz a obšírné části věnující se kompletnímu ohodnocení projektu (včetně možnosti uživatelského nastavení striktnosti evaluace).

Skutečné vyhodnocení vlivu používání zásuvného modulu UpAnalyzer v systému JIRA firmy Siemens CZ se vzhledem k délce trvání jednotlivých projektů objeví až postupem času, nicméně již při zpětné analýze dokončených projektů se testovací tým divize rozhodl pro několik drobných změn ve vedení týmu. Konkrétní hodnocení produktu konzultantem firmy Siemens Ing. Jakubem Řezáčem obsahuje příloha A na konci tohoto dokumentu.

Dohoda ohledně dalších možných rozšířeních projektu zní tak, že po několika budoucích projektech a soužití s aktuální verzí zásuvného modulu klient dodá podněty k nadstavbám, které se mohou, ale nemusí nést v duchu rozšíření navrhnutých v této práci.

Použitá literatura

Monografie a sborníky

[1] VINAY, Peter Farrell. *Manage Software Testing*. Auerbach Publications, 2008, ISBN 978-0-849-39383-9.

[2] CRISPIN, Lisa a Janet GREGORY. *Agile Testing: A Practical Guide for Testers and Agile Teams*. Addison-Wesley Professional, 2009, ISBN 978-0321534460.

[3] MAYERS, Glenford, SANDLER, Corey a Tom BADGETT. *The Art of Software Testing*. Wiley, 2011. ISBN 978-1118031964.

[4] WEYNS, Danny. Architecture-Based Design of Multi-Agent Systems. Springers, 2010. ISBN 978-3-642-01064-4.

[5] BACHMANN, Felix a Len BASS. *Introduction to the Attribute Driven Design Method*. Proceedings of the 23rd International Conference on Software Engineering (ICSE'01). 2001. ISBN 0270-5257/01. [online] Dostupné z: <http://www.computer.org/csdl/proceedings/icse/2001/1050/00/10500745.pdf>

[6] DOAR, Matthew. *Practical JIRA Administration – Using JIRA Effectively: Beyond the Documentation.* O'Reilly Media, 2011. ISBN 978-1-4493-0541-3.

[7] DOAR, Matthew. *Practical JIRA Plugins – Using JIRA Effectively: Custom Development.* O'Reailly Media, 2011. ISBN 978-1-4493-0827-8.

[8] KURUVILLA, Jobin. *JIRA 5.x Development Cookbook.* Packt Publishing, 2013. ISBN 978-1849681803

[9] FORD, Neal. *Art of Java Web Development: Struts, Tapestry, Commons, Velocity, JUnit, Axis, Cocoon, InternetBeans, WebWork.* Manning Publications, 2003. ISBN 978-1932394061

[10] PEIRIS, Brian. *Instant jQuery Flot Visual Data Analysis*. Packt Publishing, 2013. ISBN 9781783280650

[11] LI, Patrick. *JIRA Essentials 5.2*. Packt Publishing, 2013. ISBN 978-1782179993

[12] RAASCH, Jon, MURRAY, Graham, OGIEVETSKY, Vadim a Joseph LOWERY. *JavaScript and jQuery for Data Analysis and Visualization.* ISBN 978-1118847060

Články a elektronické zdroje

[13] ADZIC, Gojko. *Let's Break The Agile Testing Quadrants* [online]. Poslední aktualizace: 23.10. 2013. Dostupné z: <http://gojko.net/2013/10/21/lets-break-the-agile-testing-quadrants/

[14] ZENDULKA, Jaroslav. Architektura informačních systémů. Brno, 2008.

[15] COUTO, C.; SILVA, Ch.; VALENTE, M.T. *Uncovering Causal Relationships between Software Metrics and Bugs* [online]. [Cit. 3. 4. 2013]. Dostupné z: <http://homepages.dcc.ufmg.br/~mtov/pub/2012_csmr.pdf>

[16] SANGHOON, Jeon et al. *Quality Attribute Driven Agile Development* [online]. Ninth International Conference on Software Engineering Research, Management and Applications, 2011. Dostupné z: <http://www.researchgate.net/publication/221541918_Quality_Attribute_Driven_Agile_Development>

[17] DOAR, Matthew. *Practical JIRA Development* [online]. [Cit. 22. 8. 2014]. Dostupné z: <http://jiradev.blogspot.cz/>

[18] SCHNUR, David. *Flot Chart: Attractive JavaScript plotting for jQuery* [online]. [Cit. 2. 3. 2015]. Dostupné z: <https://github.com/flot/flot/blob/master/API.md>

[19] MAGNUSSON, Geir. *Start up the Velocity Template Engine* [online]. [Cit. 28. 1. 2001]. Dostupné z: <http://www.javaworld.com/article/2075966/core-java/start-up-the-velocity-template-engine.html>

[20] *Atlassian Development Page: Getting Started* [online]. [Cit. 28. 3. 2015]. Dostupné z: <https://developer.atlassian.com/docs/getting-started>

[21] *Atlassian Development Page: Writing And Running Plugin Tests* [online]. [Cit. 28. 3. 2015]. Dostupné z: <https://developer.atlassian.com/docs/getting-started/writing-and-running-plugin-tests/>

[22] LI, Patrick. *How Can I Become An Expert Of JIRA Plugin Development* [online]. [Cit. 5. 8. 2014]. Dostupné z: <http://www.quora.com/How-can-I-become-an-expert-of-JIRA-plugin-development>

Seznam příloh

Hlavní přílohy

Seznam obrázků

Seznam tabulek

Seznam grafů

Seznam rovnic

Seznam zdrojových kódů

Struktura přiloženého CD

/docs/ - složka obsahující dokument semestrálního projektu v editovatelné podobě
/img/ - složka obsahující obrázkové soubory příloh dokumentu
/upanalyzer/ - složka obsahující veškerá data vázající se k modulu UpAnalyzer
/upanalyzer/build/ - složka obsahující zdrojový kód zásuvného modulu projektu

/upanalyzer/docs/ - složka obsahující dokumentaci programu vygenerovanou nástrojem Doxygen

/upanalyzer/src/ - složka obsahující verzi projektu ve formátu .jar

/upanalyzer/video/ - složka obsahující video ukázku prostředí UpAnalyzer na klientském projektu

/print/ - složka obsahující verzi dokumentu určenou k tisku

Příloha A: Hodnocení analyzátoru firmou Siemens

Firma Siemens dlouhodobě využívá systém JIRA pro celkové řízení organizace včetně všech fází projektového vývoje. Proto jsou data tohoto systému cenným zdrojem informací při rozhodování o řízení organizace, investic do podpůrných nástrojů, zlepšování interních procesů a nastavování nových postupů. Systém ovšem sám o sobě nenabízí analytické nástroje, které by jasně poskytly vstupy pro rozhodování na úrovní managementu, kde jasná argumentace prostřednictvím vizualizací dlouhodobého stavu představuje správnou argumentaci, oproti technickým diskuzím, pro které nemusí být vedení organizace dostatečně připraveno.

Diplomová práce tak přináší nástroj UpAnalyzer, který dokáže přehlednou formou v dlouhodobém měřítku pojmenovat a vyčíslit status projektových fází agilního vývoje a postavit před management jasně definované problémy, které lze pak snadněji ošetřit, argumentovat další postup a nalézt odpovídající množství nutných investic.

Nástroj UpAnalyzer je postaven na vyhodnocování naměřených projektových dat, u nichž je možné nastavit kvalitativní mezní hodnoty. Nástroj tedy dokáže nejen analyzovat a vizualizovat reporty, ale také upozornit na vznikající problém a napomoci nalezení jeho podstaty.

Lifetime Analysis report přináší vizualizaci nedokončených úkolů v rámci jednotlivých sprintů k průměrnému počtu story pointů. Tato korelace byla často opomíjená, protože za hlavní přínos agilní estimace je považovaná sama tato aktivita, kdy dochází napříč týmem ke sdílení znalostí a poskytování zpětné vazby. UpAnalyzer tak do organizace vnesl zcela novou formu hodnocení, která umožňuje prokázat na jasných číslech, že slabé investice do analýzy specifických požadavků v naší organizaci přímo negativně ovlivňují pravidelnost přislíbených dodávek zákazníkovi.

P&E Analysis naopak přinesla poměrně pozitivní výsledky, které ukazují, že valná většina zákaznických požadavků dekomponována je, což je prokázáno převážným počtem user story s nízkým ohodnocením story pointy. Analýza ovšem také ukazuje, že je ve velkém počtu opomíjena potřeba prioritizace, což vede k nejasnému rozhodování o pořadí dekompozice hlavně kvalitativních požadavků. Analýza ukazuje na potřebu více investovat do fáze konsolidace požadavků, což povede k přiblížení se přáním zákazníků.

Burndown Analysis rozšiřuje možnosti známé křivky s důrazem na přetíženost vývojového týmu. Jedná se tedy o okamžitý ukazatel trendu, který poukázal včas na nedostatek členů vývojového a testovacího týmu. Analýza dále v detailu vizualizuje jednotlivé sklady (Kanban), kde lze přímo

identifikovat konkrétní tým s největším nedostatkem. Díky závěrům této analýzy byl ještě zvýšen tlak na nábor těchto, na trhu práce, poměrně nedostatkových odborníků.

Velkou výhodou nástroje UpAnalyzer je závěrečný přehled, který vyzdvihuje nejpodstatnější faktory a upozorňuje na jejich nedostatky. Takový přehled nejen nabízí rychlou kontrolu kondice všech projektů, ale navíc značně zjednodušuje pravidelný reporting tzv. „*quality gates*“. Hodnoty z tohoto výčtu jsou dokonalým vstupem pro celou řadu hodnotících workshopů.

Závěry plynoucí z této diplomové práce jsou cenným vstupem pro společnost Siemens a nástroj UpAnalyzer zůstává trvalou součástí systému JIRA, kde bude i nadále využíván pro kontrolu průběhu prací na projektech, ale především pro identifikaci a rozhodování budoucích investicích do zvyšování interní kvality, jež má přímý dopad na kvalitu externí, a tedy na uspokojování zákaznických požadavků**.**

Příloha B: Statistiky zdrojového kódu

Jazyk	Implementovaná část	Počet funkcí/tříd	Počet řádků
Java	Komunikace s JIRA, Pomocné třídy	4 třídy	816
Java	Analýzy	6 tříd	525
JavaScript, jQuery	Vizuální reprezentace dat	14 funkcí	1040
HTML, CSS	Rozložení a vykreslení panelu s nástrojem	-	552
Velocity	Transformace dat	-	402
Python	Přidání nové analýzy	3 funkce	262

Tabulka statistik zdrojového kódu v projektu UpAnalyzer