VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

PRŮZKUM VLASTNICTVÍ KÓDU VE VELKÉ ORGANIZACI

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. JAN ŠIMONEK
AUTHOR

BRNO 2015

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

PRŮZKUM VLASTNICTVÍ KÓDU VE VELKÉ ORGANIZACI

CODE OWNERSHIP RESEARCH IN LARGE ORGANIZATION

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. JAN ŠIMONEK
AUTHOR

VEDOUCÍ PRÁCE Doc. RNDr. JITKA KRESLÍKOVÁ, CSc.
SUPERVISOR

BRNO 2015

Abstrakt

Tato diplomová práce se zabývá vlastnictvím kódu v softwarových organizacích a tvorbou nástroje, který usnadňuje spolupráci v projektu automatickou identifikací vlastníků. Při zkoumání problému práce vychází z Extrémního Programování, dále jsou popsány modely vlastnictví kódu i dopad vlastnictví kódu na mezi-týmovou spolupráci a dělení práce uvnitř týmu. Teoretický rámec je zasazen do kontextu reálné firmy, kde je identifikován potenciál na zlepšení. Pomoci ke zlepšení by měl dále v práci navržený nástroj, který na základě dat ze systému správy verzí identifikuje vlastníky a experty pro jednotlivé oblasti kódu. Tyto informace zpřístupňuje v přehledné podobě. Také je v práci popsán provedený průzkum vlastnictví kódu v několika projektech, což umožňuje posoudit kvalitu výsledků. V závěru práce je diskutována použitelnost nástroje a jeho přínos na praxi v reálné firmě.

Abstract

This master's thesis is about code ownership in software projects and creating a tool that would improve cooperation by identifying code owners. The theoretical foundation for code ownership starts from Extreme Programming, explaining models of code ownership and the impact of code ownership on cooperation among teams and team members. The concept is demonstrated on a concrete software firm, where a potential for improvement is identified. The potential is exploited by a software tool, which is designed in the this thesis. The tool is capable of identifying code owners and experts for specific areas in the code based on data gathered from a version control system. The resulting information is made easily accessible. The tool is used to conduct a code ownership research in several projects, which allows me to confirm accuracy of the results. Usability and benefits of the tool is discussed in the final chapter.

Klíčová slova

vlastnictví kódu, identifikace vlastníků kódu, extrakce klíčových slov, oblasti kódu, autorství, expertiza, vývoj software, spolupráce, tým, systém správy verzí

Keywords

code ownership, code owners identification, keyword extraction, code areas, authorship, expertise, software development, cooperation, team, version control system

Citace

Jan Šimonek: Průzkum vlastnictví kódu ve velké organizaci, diplomová práce, Brno, FIT VUT v Brně, 2015

Průzkum vlastnictví kódu ve velké organizaci

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením doc. RNDr. Jitky Kreslíkové, CSc.

. . . . . .
Jan Šimonek
25. května 2015

Poděkování

Tímto bych rád poděkoval vedoucí práce doc. RNDr. Jitce Kreslíkové, CSc. za cenné připomínky a odborné rady, kterými přispěla k vypracování této diplomové práce. Dále bych chtěl poděkovat firmě, která byla pod podmínkou anonymity ochotna být pro účely této práce pokusným subjektem. Také děkuji všem respondentům, kteří přispěli k ověření výsledků.

© Jan Šimonek, 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 **3**

2 Vlastnictví kódu a Extrémní programování **4**
2.1 Přístupy k vlastnictví kódu . . . 4
2.1.1 Žádné vlastnictví kódu . . . 4
2.1.2 Silné vlastnictví kódu . . . 5
2.1.3 Slabé vlastnictví kódu . . . 5
2.1.4 Společné vlastnictví kódu . . . 5
2.2 Vlastnictví kódu podle velikosti projektu . . . 6
2.3 Extrémní programování . . . 6
2.3.1 Principy . . . 7
2.3.2 Praktiky . . . 8

3 Vlastnictví kódu v reálné firmě **10**
3.1 Praxe ve Firmě . . . 10
3.2 Vlastnictví kódu ve Firmě . . . 12
3.2.1 Existující snahy o zlepšení situace . . . 14
3.3 Vlastnictví kódu v otevřeném software . . . 15

4 Návrh nástroje pro identifikaci vlastníků **17**
4.1 Požadavky . . . 18
4.2 Uživatelské rozhraní . . . 19
4.3 Hlavní celky systému . . . 20
4.4 Určení vlastníků oblastí ve zdrojovém kódu . . . 21
4.4.1 Logické celky kódu . . . 22
4.4.2 Algoritmus stanovení vlastníků . . . 23
4.4.3 Prezentace výsledků . . . 25
4.5 Extrakce klíčových slov . . . 27
4.5.1 Algoritmus Term Frequency - Inverse Document Frequency . . . 28
4.5.2 Algoritmus HG-Rank . . . 28
4.5.3 Víceúrovňové hodnocení klíčových slov . . . 29
4.6 Vlastnictví na úrovni týmu a jedince . . . 30

5 Implementace **31**
5.1 Použité technologie . . . 31
5.2 Struktura programu . . . 31
5.3 Uživatelské rozhraní . . . 32
5.4 Systém správy verzí . . . 33

5.5 Databáze . . . 35
5.6 Úpravy extrakce klíčových slov . . . 36
5.6.1 Rozdělení na slova . . . 37
5.7 Metodika . . . 37

6 Testování a zhodnocení **39**
6.1 Průzkum vlastníků kódu v ElasticSearch a Guava . . . 39
6.1.1 Rychlost analýzy . . . 40
6.1.2 Ukázka výsledků . . . 40
6.2 Klíčová slova v porovnání oblastí kódu . . . 41
6.2.1 Vyhodnocení . . . 42
6.3 Guava, ElasticSearch – Dotazník . . . 42
6.3.1 Vyhodnocení . . . 42
6.4 Aktuální nasazení ve Firmě . . . 43
6.5 Budoucí uplatnění ve Firmě . . . 44
6.5.1 Náplň práce respondentů . . . 44
6.5.2 Situace vlastnictví kódu . . . 44
6.5.3 Stávající praxe určení vlastníků . . . 45
6.5.4 Uplatnění Who-Owns . . . 45
6.5.5 Odhad dopadu . . . 46

7 Závěr **47**

A Ostatní diagramy **51**

B Snímky obrazovky **53**

C Manuál **55**

D Transkript **57**

E Elasticsearch - dotazník **65**

F Obsah přiloženého CD **67**

Kapitola 1

Úvod

Termínem vlastnictví kódu jsou označovány konvence při vývoji software, které určují, kdo v projektu může upravovat jaké části zdrojových kódu. Tento koncept ovlivňuje každodenní práci zúčastněných programátorů. Může ji zefektivnit případně upřednostnit jiný cíl a pohyb vpřed zpomalit. Vlastnictví kódu nebývá v projektech nařízeno. Členové vývojových týmů postupem času sami dospějí ke konsenzu, který reflektuje pevněji zakotvená pravidla — metodiku vývoje softwaru, dělení práce mezi programátory či týmy, povahu projektu apod.

Ve velkých projektech se vhodnost různých modelů vlastnictví kódu projevuje více. Nevhodný přístup může způsobit chaos, spolupráce mezi týmy může váznout. Vhodně nastavená pravidla mohou situaci výrazně dopomoci, ale pokud jsou příliš přísná, mohou brzdit postup projektu. Optimální je nalézt rovnováhu mezi chaosem a striktním řízení.

V této diplomové práci si pokládám otázku, zdali by nebylo možné hrozící chaos snížit. Pomoci by k tomu měl automatický nástroj, který určí vlastníky oblastí ve zdrojovém kódu. Základní schopností nástroje bude zodpovězení otázky: „Kdo vlastní. . . ?“ Třemi tečkami bude moci být modul, třída, funkce, řádek zdrojového programu nebo klíčové slovo. Využíváním nástroje i integrací do firemních procesů by se měla zlepšit komunikace mezi programátory a spolupráce mezi týmy. To by mělo vliv na zvýšení kvality zdrojových kódů firmy.

V své práci vycházím z rámce metodiky vývoje softwaru, určuji, co softwarového vlastnictví vlastně je, a identifikuji základní modely vlastnictví software (kapitola 2). Následně analyzuji situaci v reálné firmě, kde nacházím prostor pro zlepšení, ke kterému by měl dopomoci právě mnou vytvářený nástroj (kapitola 3). V kapitole 4 nástroj navrhuji a vytvářím nový postup pro stanovení identifikaci vlastníků na základě dat ze systému správy verzí. Také popisuji inovativní použití existujících algoritmů pro extrakci klíčových slov. V kapitole 5 se zabývám implementací. V další části je nástroj použit k provedení průzkumu vlastnictví v několika projektech, ověřuji přesnost výsledků a zabývám se možností použití a především přínosem nástroje (kapitola 6).

Tato práce navazuje na semestrální projekt, ze kterého přebírá kapitoly 2, 3, 4.1 a 4.2.

Kapitola 2

Vlastnictví kódu a Extrémní programování

Na začátku této kapitoly nejdříve vysvětlím, co konkrétně je myšleno termínem *vlastnictví kódu*. Následně se zabývám vlastnictvím kódu v kontextu metodiky vývoje softwaru, přičemž nejvíce pozornosti se dostane extrémnímu programování. Dále vysvětluji základy extrémního programování, abych na ně mohl v následující kapitole navázat.

Pro účely vývoje softwaru může být pojem *vlastnictví zdrojového kódu* (zkráceně jen *vlastnictví kódu*) vnímán poněkud méně striktně, než implikuje slovo vlastnictví. Nejedná se o majetek v právním slova smyslu, ani o práva nakládání se softwarovým kódem, jak je vnímá autorský zákon. Místo toho můžeme vlastnictví kódu definovat následovně:

> Vlastnictví zdrojového kódu v softwarové organizaci jsou zvyklosti dané organizace, které stanovují, kdo spravuje a je zodpovědný za oblasti zdrojového kódu. [Vlastní]

Oblastí kódu může být modul, soubor, třída, metoda nebo jakýkoliv jiný logický celek jazyka. Koncept vlastnictví kódu je užitečný zejména pro členění práce a koordinaci jak uvnitř týmů, tak mezi týmy. Možnosti použití a dopady na softwarovou organizaci jsou vysvětleny v následujících kapitolách.

2.1 Přístupy k vlastnictví kódu

Dle definice v předchozím odstavci je vlastnictví kódu sada zvyklostí. Tyto zvyklosti však mnohdy vycházejí z metodiky vývoje softwaru, která o tomto konceptu může něco určovat, nebo tyto zvyklosti vycházejí ze zaběhnutých zkušeností. Ve většině organizací se tak vlastnictví kódu dá abstrahovat jedním z následujících modelů.

2.1.1 Žádné vlastnictví kódu

Všichni programátoři mohou měnit jakoukoliv oblast kódu [1]. Tento přístup je, spíše než modelem, jeho absencí. Absence jakýchkoliv bariér může být výhodou v malých projektech. Přitom je třeba dávat dobrý pozor na zachovávání účelu modulů a jejich konzistenci. Pro programátory je lákavé překrucovat účel modulů pro řešení momentálních problémů na úkor dlouhodobých. Nikdo se totiž necítí být zodpovědný za kvalitní design.

2.1.2 Silné vlastnictví kódu

Přirozeným řešením problému nastíněného v předchozím odstavci je silné vlastnictví kódu. (Také označováno jako *individuální*.) Při něm je pro každou oblast stanoven jednoznačný vlastník. Vlastníkem je v tomto případě vždy jedinec, nikoliv tým. Programátor má v tomto modelu dovoleno upravovat pouze své vlastní oblasti. Pokud potřebuje provést změnu v cizí oblasti, musí vlastníka o změnu požádat. Vlastník změnu provede sám a o výsledku žadatele informuje. Jedna z variant tohoto modelu umožňuje navrhovateli připravit změnu samostatně, ale následně musí vlastníka požádat o schválení a zanesení změny do systému (například pomocí `.patch` souboru). [8]

Výhodou silného vlastnictví je lepší kontrola prováděných změn a jasně přiřaditelná zodpovědnost. Nevýhod je však celá řada. Při reálném vývoji je změna cizího kódu nutná poměrně často. Například přejmenování veřejné metody (*refactoring*[1]) způsobí změnu ve všech výskytech volání této metody. Přestože je úprava triviální, musí o provedení změny žádat autor každého vlastníka samostatně. Z triviálního úkolu tak vyvstává problém, protože změny několika programátorů se stávají vzájemně závislé. Po několika zkušenostech podobného charakteru ztratí členové týmu motivaci k provádění podobných úprav. Refactoring je přitom pro moderní vývoj software klíčový. Dalším zdrojem potenciálních problémů jsou opravy chyb a obecně všechny změny vyžadující implementaci. V časové tísni mohou být programátoři okolnostmi donuceni okopírovat cizí metodu do svého kódu a tím se vyhnout zdlouhavému požadavku na změnu [8].

Silné vlastnictví kódu má své uplatnění v projektech, kde klady vyváží zmíněné nepříjemnosti, nebo kde k interním požadavkům na změnu kódu dochází jen minimálně. Tento předpoklad splňuje například striktně modulární systém s minimalistickým rozhraním mezi moduly. Přesně kvantifikovat hranici, kdy zvolit silné vlastnictví kódu a kdy jiný model, by bylo velmi náročné. Do hry vstupují nejen výše zmíněné podmínky, ale i vnější požadavky na vývojový proces, povaha úkolu i zažité zvyklosti.

2.1.3 Slabé vlastnictví kódu

V tomto modelu má každá oblast svého vlastníka podobně jako u silného vlastnictví. Na rozdíl od něj není při změně cizího kódu vyžadována bezprostřední participace vlastníka. Vlastníci jsou stále za své oblasti zodpovědní a měli by na ně dohlížet. [8]

Programátoři měnící cizí kód by měli informovat, požádat o asistenci nebo si vyžádat souhlas vlastníka dané oblasti, ale je ponecháno na zhodnocení programátora, kterou úroveň zvolí. Předpokladem je dobrý úmysl autorů a schopnost dobře odhadnout komplexnost změny.

2.1.4 Společné vlastnictví kódu

Tento model byl poprvé popsán Kentem Beckem [1] v metodice s názvem Extrémní programování. Společné (také kolektivní) vlastnictví kódu umožňuje komukoliv měnit jakoukoliv oblast kódu jako při neexistujícím vlastnictví. Navíc nabádá programátory, aby vylepšovali kód, kdykoliv se k tomu naskytne příležitost. Tento rozdíl spolu s ostatními hodnotami a principy extrémního programování by měl stačit k potlačení negativních důsledků chybějícího vlastnictví kódu.

[1] Slovo refactoring nemá žádný přesný český ekvivalent. V této práci jej definuji jako zásah do zdrojového kódu s cílem zvýšit jeho kvalitu (design, čitelnost, pojmenování) bez změny funkčnosti programu.

Podmínkou fungování modelu je porozumění programátorů veškerých zdrojových kódů alespoň na základní úrovni. V opačném případě přestává být společné vlastnictví efektivní. Změny v cizích oblastech totiž mohou jít proti zamýšlenému účelu. Extrémní programování řeší tuto situaci pomocí párového programování a ostatních postupů popsaných v kapitole 2.3.2. Pokud velikost zdrojových kódů přesahuje jistou hranici, není už jednotlivec schopen je obsáhnout celé a je nutné zavést kombinaci výše popsaných přístupů.

2.2 Vlastnictví kódu podle velikosti projektu

Všechny přístupy k vlastnictví kódu fungují do určité velikosti projektu (jednotky měsíců a lidí). S narůstající velikostí jsou rozdíly více patrné. Neexistující vlastnictví kódu je nejméně škálovatelné [4, 5]. Fungující tým si brzy uvědomí zvětšující se chaos a vytvoří ustálená pravidla hry, čímž se situace překlopí do jiného konceptu vlastnictví. U silného modelu se s přibývající velikostí zvětšuje režie při změnách designu, vykonávání refactoring a s nimi spojené žádosti o změnu. Záleží na povaze projektu, zda je situace únosná či nikoliv.

Slabé vlastnictví je dobře škálovatelný model, který ale sám o sobě nijak nepomáhá zvládat výzvy velkých projektů. Extrémní programování doporučuje velikost týmu od 2 do 12 členů. Nad tuto velikost se příliš zvýší nutná komunikace v týmu. Pro velké projekty také nedostačují nástroje extrémního programování na koordinaci — párové programování, plánovací hra a právě společné vlastnictví kódu. Větší projekty jsou tak nutně řešeny více týmy. V takovém případě společné vlastnictví kódu může fungovat uvnitř týmu, ale nemůže fungovat v celém projektu. Kent Beck v takové situaci doporučuje začít řešit problém pouze v malém týmu a po nabrání zkušeností projekt mezi týmy rozdělit po jeho přirozených hranicích. Rozdělením projektu mezi týmy opět řešíme stejný problém vlastnictví kódu. Za vlastníky už nejsou považováni jednotlivci, ale týmy. Problémy spojené s nejasným vlastnictvím kódu jsou za takovéto situace daleko menší. V praxi ale i tak nastávají problémové situace. Možné problémy popisuji na příkladě konkrétní firmy v kapitole 3.2.

2.3 Extrémní programování

Extrémní programování je již zaběhnutá metodika vývoje softwaru. V knize Extreme Programming Explained: Embrace Change z roku 1999 ji popsal Kent Beck. Přestože některé konkrétní postupy se příliš nezažily, extrémní programování jako takové je stále aktuální a používané. Je totiž vystavěno od základních hodnot, které by měli dobří softwaroví inženýři ctít. Hodnoty jsou příliš abstraktní, než aby mohly něco říkat o rutinních činnostech, proto Kent Beck pokračuje principy. Principy zasazují hodnoty do kontextu vývoje software. Na principech už staví konkrétní praktiky. V prvním vydání Extreme Programming Explained je předešlé tvrzení shrnuto takto:

> „When I first articulated XP, I had the mental image of knobs on a control board. Each knob was a practice that from experience I knew worked well. I would turn all the knobs up to 10 and see what happened.“
>
> Když jsem poprvé formuloval XP (extrémní programování), představoval jsem si knoflíky na ovládacím panelu. Každý knoflík představuje činnost, u které jsem ze zkušenosti věděl, že dobře funguje. Otočil jsem všemi knoflíky až na desítku a sledoval, co se stane. [1]

Hodnoty říkají, na čem při vývoji software záleží. Jsou důvodem a hybnou silou činností. Extrémní programování pokládá tyto základní kameny takto:

- Komunikace — Dobrá a otevřená komunikace je základem všeho ostatního.
- Jednoduchost — Při vývoji software bychom měli hledat nejjednodušší možná řešení, která fungují a dostačují. Měli bychom se soustředit na problémy a úkoly dneška, protože ty zítřejší tolik neznáme a navíc se mohou změnit.
- Odezva — Vyvíjený software je často nedokonalé řešení nedokonalého zadání. Proto je třeba zajistit co nejrychlejší a nejkvalitnější zpětnou vazbu, aby se zadání i řešení mohlo postupně vylepšovat bez dodatečných nákladů.
- Odvaha — Udělat zásadní změnu s cílem zjednodušit řešení nebo zahodit nikam nevedoucí výsledek celého dne vyžaduje odvahu. Odvaha jako samostatná hodnota by byla nebezpečná. V symbióze s ostatními hodnotami je ale dobrým pomocníkem.
- Respekt — Vzájemný respekt kolegů i společný zájem na projektu jsou také nutným předpokladem pro fungující projekt.

Hodnoty extrémního programování (*XP*) nejsou nijak překvapivé. Vychází ze zdravého rozumu a dobrých zvyklostí týmové spolupráce. Hodnoty je ale možné vykládat v různých kontextech různě. Na zlepšení odezvy v marketingu může posloužit průzkum, při vývoji software může dopomoci zkrátit vývojový cyklus. Proto XP obsahuje i principy a praktiky, které hodnoty upřesňují.

2.3.1 Principy

Principy XP začínají tam, kde končí jeho hodnoty. Navazují na ně a dávají je do souvislostí s konkrétními činnostmi. K následujícímu převzatému výčtu principů připojuji vlastní komentář:

- Lidskost — Metodika i procesy by měly mít vždy na paměti, že software je vyvíjen lidskými bytostmi s potřebami a touhami. Extrémní programování také vychází z předpokladu, že lidé rádi odvádějí dobrou, kvalitní a smysluplnou práci.
- Ekonomika — Vývoj software je velmi drahý. Je tedy nutné snažit se udržet náklady na nejmenší možné míře a nasadit software co nejdříve.
- Vzájemný prospěch — Extrémní programování se snaží řešit situace ve prospěch všech zúčastněných. Nejjednodušší fungující řešení prospívá jak celému projektu (minimalizace nákladů), zákazníkovi (menší počet závad), vývojářům (srozumitelnější kód, pocit dobře odvedené práce).
- Soběpodobnost — Pokud se nějaké řešení osvědčí, je výhodné ho aplikovat na podobné problémy. Pokud se aplikace soběpodobnosti osvědčila při vývoji software opětovným používáním kódu, můžeme se pokusit aplikovat soběpodobnost na vývojový cyklus (např. programování řízené testy a psaní systémových testů místo specifikace).
- Zlepšování — Žádný proces ani produkt není dokonalý okamžitě. XP radí začít ihned, přestože nedokonale, a postupně stav vylepšovat.

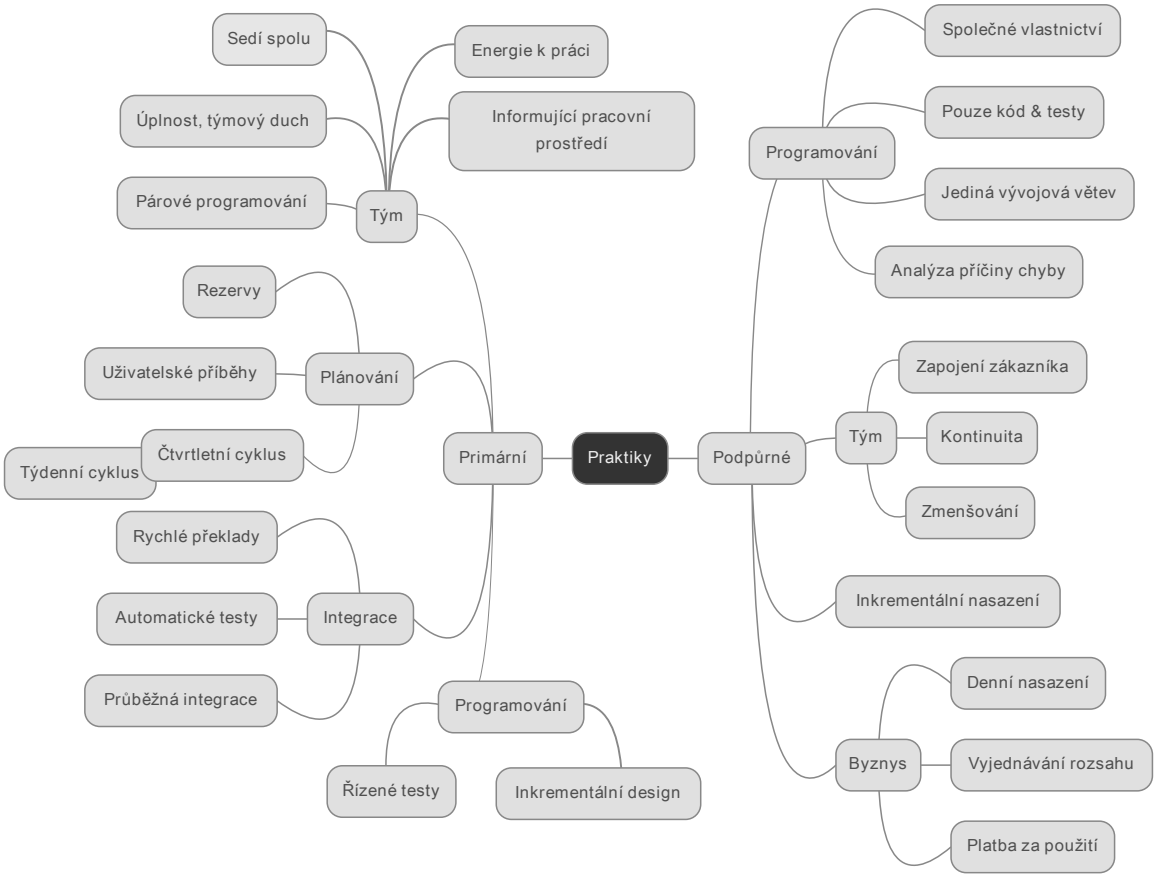
- Různorodost — Tým je nejefektivnější, když jsou jeho členové různorodí. Z rozdílných názorů je možné vybírat nejlepší řešení.
- Reflexe — Občas je třeba se zamyslet nad smyslem a způsobem práce.
- Malé přírůstky — Zajištění stálého proudu malých částí je lepší, než menší počet velkých změn. Výrazně se tím zrychlí zpětná vazba.
- Příležitost — Využít každé příležitosti pro zlepšení.
- Redundance — Vývoj software je natolik komplexní, že praktiky se musí vzájemně podporovat. Například úrovní testů je více a všechny mají svůj smysl.
- Selhání — Je důležité nesnažit se pokračovat po slepých cestách. Mnohdy je lepší zahodit část práce a začít znovu s nabytými zkušenostmi. To se týká nejen malých činností ale i celých projektů. Samozřejmě je v takové situaci potřeba buď si být jist bezvýchodností situace, nebo vědět o existenci jiného přístupu, který by mohl být efektivnější.
- Kvalita — Kvalitu není z dlouhodobého hlediska možné použít pro řízení softwarového projektu. Snížením požadavku na kvalitu nedojde ke zkrácení času nebo nákladů. Ve vyvíjeném programu se časem neúnosně zvýší počet chyb a jejich oprava bude zabírat stále více času. Vysoká kvalita má naopak pozitivní dopad na projekt. Snižuje počet chyb a zvyšuje uspokojení členů týmu z práce. [2]
- Přijetí zodpovědnosti — Z psychologického hlediska by neměla být zodpovědnost přiřazena. Aktivní přijetí zodpovědnosti vzbuzuje větší závazek a pocit možnosti volby.

2.3.2 Praktiky

Z principů extrémního programování vychází již konkrétní praktiky. Ty nejsou povinné, jedná se spíš o doporučení. Jednotlivé činnosti se sice vzájemně podporují, ale je vždy na zvážení daného týmu, zda zavedení praktiky má význam. Následováním praktik je také možné jednoduše odhadnout, nakolik tým následuje ideál extrémního programování.

Seznam praktik XP čítá zhruba dvě desítky položek kategorizované na primární a podpůrné. Zobrazuji je na obrázku 2.1. Pro účely této práce rozebírám pouze ty, které mají přímý vztah k vlastnictví kódu:

- Tým sedí spolu – Usnadňuje komunikaci a zvyšuje vzájemné porozumění.
- Párové programování – Výrazně přispívá pro distribuci znalostí a expertízy v týmu.
- Inkrementální design – Design by měl v ideálním případě přesně odpovídat řešenému problému. Ten se ale mění, proto je třeba průběžně měnit design s ním. Společné vlastnictví kódu umožňuje změny návrhu provádět jednodušeji.



Obrázek 2.1: Praktiky extrémního programování. Inspirováno [2]

Kapitola 3

Vlastnictví kódu v reálné firmě

V této kapitole analyzuji situaci vlastnictví kódu v existující firmě. Nejdříve načrtnu její hrubé rysy, poté přejdu ke stávající praxi softwarového inženýrství, kde se zaměřím na vlastnictví kódu. Následně identifikuji problémová místa a zhodnotím existující možnosti řešení. Tato kapitola poslouží jako základ pro stanovení uživatelských příběhů (suplujících požadavky) pro nástroj vyvíjený v další části této diplomové práce. Dohoda s firmou mi bohužel neumožňuje zveřejnit její název. Pro účely této práce ji tedy budu označovat jako Firma.

Firma se orientuje pouze na vývoj a prodej svého jediného softwarového produktu (dále označován jako *Produkt*), ze kterého jí plynou veškeré zisky. Firma vznikla na konci devadesátých let jako start-up a již od začátku bylo jejím cílem co nejrychlejší vývoj Produktu. Zpočátku se firmě příliš nedařilo, ale v první polovině desetiletí nového milénia prorazila na trh. Firma se začala rychle rozvíjet a stále rozšiřovala svůj Produkt, do kterého byly přidávány nové funkcionality i celé podpůrné systémy. Firma podstoupila globalizaci a otevřela několik poboček po celém světě a začala nabírat mnoho nových pracovních sil. V současnosti se rychlý vývoj začíná stabilizovat. Ve Firmě mimo jiné pracuje zhruba 250 programátorů, 200 testerů a 50 vlastníků produktu (product owner podle Scrum). Zaměstnanci jsou rozděleni do 50 týmů. Z tohoto počtu asi 20% zaměstnanců v 15 týmech plní podpůrnou roli, ostatní pracují přímo na Produktu. Příslušnost zaměstnance k více týmům současně je vzácná.

3.1 Praxe ve Firmě

Historie Produktu má přímý dopad na praxi vývoje software ve Firmě. Kontinuita a postupné změněny Produktu implikují iterativní vývojový cyklus. Firma pro management vývoje používá metodiku Scrum, přestože některé praktiky se odlišují. Nová verze Produktu je uvolňována pravidelně každý půlrok, většinou tak týmy nedodržují několikatýdenní sprinty a soustředí se na pevně dané milníky pro vydání nové verze. Pozice scrum mastera není tak pevně zakotvena, jako to doporučuje Scrum. Scrum master má často i další funkce v týmu, například je běžným vývojářem. Firma se na druhou stranu opírá o velký počet dedikovaných vlastníků produktu (*product owner*), kteří jsou buď přímou součástí týmu, nebo s ním mají velmi těsný vztah. V týmu je také od 2 do 7 programátorů a podobný počet testerů. Kvůli rozsáhlosti produktu a požadavkům na kvalitu jsou testeři a vývojáři specializováni.

Dlouhá historie produktu podstatně ovlivňuje i podobu zdrojových kódu. Postupnými změnami je kód neustále inovován a vylepšován. V některých oblastech se tímto procesem

Praktika	Dodržování	Popis
Tým – sedí spolu	✓	
Tým – úplnost, týmový duch	✓	
Tým – Energie k práci	✓	
Tým – Informující prac. prostředí	✓/ ✗	Některé týmy používají scrum board, jiné softwarovou podporu.
Tým – Párové programování	✗	Místo párového programování musí být každá změna kódu zhodnocena alespoň jedním dalším vývojářem.
Plánování – Rezervy	✓	
Plánování – Uživatelské příběhy	✓	
Plánování – Čtvrtletní cyklus	✓/ ✗	Interval je půlroční. Procesem je Scrum.
Plánování – Týdenní cyklus	✗	
Integrace – Rychlé překlady	✗	Sestavení programu trvá od 20 minut do 2 hodin.
Integrace – Automatické testy	✓	Automatické testování je pomalé, ale plně automatizované. Celá sada testů běží zhruba den. Testy se spouští automaticky po zanesení změny do systému správy verzí.
Integrace – Průběžná integrace	✓	
Programování řízené testy	✗	Unit testy pro většinu tříd neexistují. Systémové testy jsou psány testery, nikoliv programátory.
Inkrementální design	✓	

Tabulka 3.1: Primární praktiky Extrémního programování ve Firmě

daří udržovat přehlednost a dobrý design. V jiných je situace podstatně komplikovanější. Malý tým programátorů například strávil celý rok identifikací a izolováním funkcionality protkané napříč systémem. Musel postupovat po malých logických částech a vždy jeden krok za druhým. Tato úmorná práce měla za cíl snížit provázání nesouvisejících funkcionalit a zlepšit udržovatelnost. S odstupem času hodnotí tým svou práci jako smysluplnou, podařilo se totiž opravit dlouho ignorovanou chybu návrhu.

V jiných případech je ale téměř nemožné podstoupit radikální změnu návrhu. Na několika fundamentálních třídách a modulech je vystavena základní funkcionalita zbytku systému. Návrh těchto tříd ale v mnoha ohledech pokulhává a programátoři tak jsou nuceni při jejich používání vynalézat způsoby, jak obejít chyby v návrhu i ve faktických chybách kódu. Vylepšit návrh by vyžadovalo dokonale koordinované úsilí celé firmy po dobu několika měsíců. K tomuto radikálnímu kroku zatím nebylo přistoupeno. Může se tak stát, že některé nedokonalosti nebudou opraveny nikdy. Problémy způsobují i existující nedokonalá řešení na databázové úrovni, ale i to je způsobeno špatným návrhem. Neexistují totiž database access objekty (DAO) a kód pro práci s databází prostupuje všemi úrovněmi programu. Změna by tedy byla velmi nákladná. Některá inženýrská řešení, která měla svůj význam v dobách začínajícího start-upu, již nedostačují pro potřeby korporace. Firma investuje mnoho prostředků do inkrementálního návrhu a vylepšování kódu.

Praktika	Dodržování	Popis
Společné vlastnictví kódu	✓/ ✗	Uvnitř týmu ano, mezi týmy slabé vlastnictví
Soustředění se pouze na kód a testy	✓/ ✗	Někdy existuje dokumentace v podobě interních wiki stránek.
Jediná hlavní vývojová větev	✓/ ✗	Každý tým má jednu nebo více svých větví, krom toho existuje společná mezi-týmová vývojová větev pro integraci.
Analýza příčiny chyby	✓/ ✗	V některých případech je podrobné hledání příčiny příliš pracné. Zvláště pokud je chyba způsobena starým kódem nebo interakcí starých a nových částí.
Zapojení zákazníka do vývoje	✓/ ✗	Ano, jen roli zákazníka přebírá product owner.
Kontinuita týmu	✓	
Zmenšování týmu	✗	
Inkrementální nasazení	✓	
Denní nasazení	✗	
Smlouvy umožňující změny rozsahu	✓	
Platba za použití produktu	✓	

Tabulka 3.2: Podpůrné praktiky Extrémního programování ve Firmě

Produkt je mezi týmy rozdělen po zvenčí viditelných hranicích – hranicích subsystémů, modulů, skupin funkcionalit apod. Toto rozdělení dobře funguje na vyšších úrovních systémů. Technická podpora je například lépe schopna určit tým s nejlepší znalostí problémového místa pouze na základě uživatelem dodaného popisu. Na nižších úrovních systému toto rozdělení selhává. Pro hlavní části nižších vrstev systému byly seskupeny specializované týmy, jejichž klienty jsou ostatní týmy. Některé týmy se specializují na úkoly přesahující hranice ostatních týmů.

Přestože Firma se aktivně neřídí Extrémním programováním, mnohé principy i praktiky jsou ve Firmě ctěny a dodržovány. Nepodařilo se mi zjistit, zda má tato podobnost nějakou hlubší příčinu nebo se jedná jen o přirozené směřování k tomu, co je při vývoji software dobré a užitečné.

Tabulky 3.1 a 3.2 shrnují dodržování praktik Extrémního programování ve Firmě. Je patrné, že se praxe Firmy a XP v mnoha ohledech shodují. Tam, kde se rozcházejí, buď existuje suplující řešení (párové programování vs. dodatečná kontrola) nebo se praktika neshoduje s obchodním modelem Firmy (denní nasazení). Na pomyslném ovládacím panelu praktik nejsou ovládací páčky na maximálních hodnotách, ale jsou nastaveny poměrně vysoko (viz citát v úvodu kapitoly 2.3).

3.2 Vlastnictví kódu ve Firmě

Jak jsem již naznačil v předchozí kapitole, můžeme ve Firmě sledovat dvě úrovně vlastnictví kódu. První úrovní je vlastnictví uvnitř týmu, druhá je tvořena vlastnictvím mezi týmy.

Na vyšší úrovni pojem vlastnictví kódu úzce souvisí s rozdělením produktu mezi týmy, ale nejedná se o totožnou záležitost.

Situaci nejlépe popisuje kombinovaný model společného a slabého vlastnictví kódu. Uvnitř týmů fungují principy Extrémního programování a komukoliv je dovoleno měnit cokoliv (podmínkou samozřejmě zůstává výsledná změna k lepšímu). Ostatní členové týmu se o změnách kódu nedozvídají skrze párové programování, ale přes povinné vzájemné kontroly kódu (code review). Na inter- týmové úrovni situaci dobře vystihuje slabé vlastnictví. Týmy jsou zodpovědné za jim přiřazenou funkcionalitu, proto si nárokují vlastnictví tříd, které jsou pro ni kritické. Členům ostatních týmů je dovoleno kód měnit zvenčí, ale je slušností prováděnou změnu probrat s vlastnícím týmem. Kontrola změny (code review) je v takovémto případě vyžadována. Tato povinnost sice zatím není nikde dokumentována ani explicitně vyžadována, ale nedodržení tohoto postupu se může brzy projevit jako problém při integrování změn. Vzhledem k rozsáhlosti zdrojových kódů produktu se také u změn v cizím kódu stává, že jdou naprosto proti směru současného vývoje. (Př.: Autor změny při provádění změn vychází ze starého návrhu, zatímco vlastnící tým přetváří kód do nové podoby.)

Netriviálním úkolem a zdrojem omylů je identifikace vlastníka (týmu) na základě souboru nebo řádky ve zdrojovém kódu. Týmy mají obvykle na starost přiřazenou funkcionalitu, nikoliv část zdrojových kódů. Některé třídy ale nemají jasně viditelný vztah k žádné z funkcionalit. U těchto tříd vývojáři postupem času dospěli ke konsenzu o tom, který tým má třídu na starosti. Určit vlastníka je tak schopen pouze vývojář s dlouhou zkušeností ve firmě. Situace se mnohdy komplikuje jemnější granulací vlastnictví, v některých případech i na úrovni použití třídy a tedy na úrovni řádky kódu. Není tedy divu, že méně zkušení zaměstnanci často nejsou schopni určit vlastníka kódu a musí se obracet na zkušenější.

V případě třídy pro kontrolu uživatelských práv nikdy konsenzus vytvořen nebyl. Třída tak nemá vlastníka. Přitom se jedná o základní stavební kámen systému, který většina týmů používá a potřebuje ho měnit. Protože za třídu není nikdo zodpovědný a nikdo ji neudržuje, stala se z ní pod vrstvou jednoúčelových změn odstrašující ukázka špatného vývoje softwaru.

Potřeba dohledat zodpovědný tým podle zdrojového kódu vyvstává ve Firmě poměrně často. Zde uvádím krátký seznam takových případů:

- Vývojář — dotaz na použití metody/třídy (Dokumentací je většinou pouze kód a testy.)
- Vývojář — konzultace návrhu
- Vývojář — průzkum za jakým účelem je metoda/třída používána
- Vývojář — změny cizího kódu, nebo kódu s nejasným vlastnictvím
- Technická podpora / Tester — analýza chyby podle zásobníku volání (stack trace).

Poslední bod si zasluhuje detailnější pozornost. Chyby do systému na základě hlášení od uživatele zadává technická podpora. Ta má za úkol i identifikovat vhodný tým na ověření chyby. K tomu pracovníkům technické podpory slouží pouze oblast produktu, kde se chyba projevila. Technická podpora tedy stanoví tým a dalším krokem v procesu je analýza chyby testerem. Ti ověří, že je chybu možné navodit. Při navozování chyby mají k dispozici zdrojový kód a často i výpis ze zásobníku volání problémového místa. V této chvíli tester ověřuje, zda je vlastníkem kódu s chybou skutečně jeho tým a případně tým mění. Některé týmy ve Firmě u 75% nahlášených chyb zjišťují, že vlastníky chybujícího kódu je někdo jiný.

3.2.1 Existující snahy o zlepšení situace

Management Firmy si je současného stavu vědom. Nejedná se sice o palčivý problém, ale přesto se vyskytla iniciativa současnou situaci zlepšit. Navrhovaná změna nese pojmenování „Code Stewardship“, volný překlad by mohl znít „Správcovství kódu“ a obnáší formální zavedení slabého vlastnictví s možností explicitního nastavení silného vlastnictví. Každý tým by si anotoval své třídy pomocí speciálních značek. Následovala by úprava nástroje pro kontrolu změn kódu, který by tyto značky zohledňoval. Vlastníci by anotacemi určili i reakci nástroje na cizí změnu:

- Upozornit vlastníka na změnu.
- Automaticky přizvat vlastníka do hodnocení kódu. Jeho souhlas by byl pro schválení změny vyžadován, ale mohl by být udělen až dodatečně.
- Automaticky zahrnout vlastníka do hodnocení kódu. Souhlas vlastníka je vyžadován před zveřejněním změny ostatním vývojářům.
- Změnu zakázat, protože třídu je dovoleno měnit pouze vlastníkovi.

S minimálními náklady by tak bylo možno zavést lepší kontrolu nad kódem. Zároveň by donutila zdokumentovat vlastníky kódu a jejich identifikace by tak byla mnohem jednodušší. Dokumentace by vyžadovala nemalé počáteční úsilí a nejspíše by musela být nařízena vedením firmy. Návrh také neřeší sporné oblasti popsané v předchozí kapitole, ani nepodporuje menší granularitu vlastnictví. Nejistou přidanou hodnotu by přinášely i nastavitelné úrovně vlastnictví. Ve Firmě je problémem identifikace vlastníků, nikoliv neochota s vlastníky spolupracovat. Z těchto příčin zůstává iniciativa ve stádiu návrhu.

Nástroj pro kontrolu změn kódu (code review) v sobě již dnes zahrnuje základní podporu pro odhad vlastníků kódu. Vychází při tom z nástroje pro správu verzí kódu. Za vlastníky jsou označeni autoři největšího počtu řádků v souboru. Pro každý řádek v souboru je identifikován poslední vývojář, který řádek upravil. Tento přístup ignoruje starší historii souboru i povahu změn. Navíc nástroj neumí identifikovat vlastníky spolehlivě. Při přenášení změn mezi vývojovými větvemi je za autora všech přenášených změn chybně považován člověk přenos provádějící. Automatická identifikace autorů v tomto nástroji tak je nespolehlivá a ve Firmě téměř není používána.

Dále je ve Firmě manuálně udržován hrubý seznam modulů a některých tříd s přiřazenými vlastníky. Tyto informace používá systém počítající pokrytí zdrojového kódu testy. Ten počítá, kolik procent řádků kódu patřící danému týmu je pokryto testy, a týmy mají za úkol udržovat tuto hodnotu nad stanovenou mez. Obtíží v tomto procesu je celá řada. Seznam vlastníků je udržován ručně, což vyžaduje značné úsilí a je náchylné k chybám. Granularita je příliš vysoká a často neodpovídá skutečnému vlastnictví, navíc některé třídy či moduly nemají vlastníky přiřazeny vůbec.

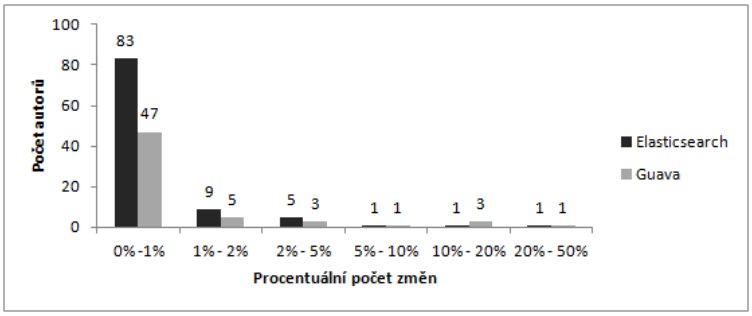
3.3 Vlastnictví kódu v otevřeném software

Situace ve Firmě se dá považovat za typickou pro celou kategorii softwarových projektů – rozsáhlé projekty s dlouhou historií. Naproti tomu existuje velmi početná skupina projektů menších a to jak svým rozsahem tak i trváním. V této kapitole se pokusím nastínit, jak vlastnictví kódu funguje v otevřeném software (anglicky *open-source software*). K tomu mě vede několik důvodů – velmi se oproti Firmě odlišují a zdrojové kódy jsou veřejně přístupné (často i celé systémy správy verzí).

Většinou na otevřeném software dlouhodobě pracuje jen několik málo vývojářů. Tito vývojáři jsou často otci myšlenky projektu a jejich přínos tvoří naprostou většinu odvedené práce. Tyto vývojáře budu označovat jako *správce*. Úspěšný otevřený software si časem vybuduje svou komunitu. Programátoři z komunity pak přispívají do projektu. Zabývají se převážně pouze opravami chyb, případně vývojem malých nebo izolovaných funkcionalit. Výsledek své práce pak zasílají na kontrolu a schválení správcům. Tuto kategorii vývojářů budu nazývat *přispěvatelé*.

Rozdělení na *správce* a *přispěvatele* může být i formálně ustanoven v kodexu projektu. Například platforma Eclipse, pod kterou je vyvíjeno několik desítek projektů, má sepsaná jasná pravidla popisující nejen dělení na správce a přispěvatele ale i proces jak se z přispěvatele může stát správce a naopak [11]. GitHub[1] toto rozdělení zabudoval do principů fungování své služby. Repozitář systému správy verzí je veřejně přístupný pro čtení, zápis je umožněn pouze správcům (Git-Hub je označuje jako *collaborators*). Přispěvatelé mají možnost vytvořit si vlastní duplikát celého repozitáře, v něm dělat změny a následně požádat o začlenění změn zpět do původního repozitáře vytvořením požadavku na správce (v terminologii GitHub *pull request*).

Zajímavé velké projekty se také vyznačují tím, že přispěvatelů je podstatně více než správců. Jasně to dokládá histogram 3.1, zpracovaný pro dva projekty otevřeného software[2]. Pokud se podíváme na třídu v histogramu úplně vlevo, zjistíme, že převážná většina autorů je autorem nejvýše 1% změn v projektu (Elasticsearch – 83 autorů ze 100, Guava – 47 autorů z 60). Tyto autory můžeme s jistotou označit za přispěvatele. V obou projektech je poměr přispěvatelů a správců podobný – cca. 10% správců, kteří jsou autory zhruba 80% provedených změn. Také výzkum provedený na projektech Apache a Mozilla dospěl ke shodným závěrům [?].



Obrázek 3.1: Histogram přínosu autorů v otevřeném software. [Vlastní]

Tato situace se nutně promítá i do možných přístupů k vlastnictví kódu. U přispěvatelů

[1] Webová služba usnadňující spolupráci na projektu. Nabízí repozitář systému správy verzí, systém hlášení chyb, Wiki stránky apod. https://github.com/

[2] Elasticsearch https://github.com/elastic/elasticsearch/, Guava - https://github.com/google/guava

se z principu věci nedá hovořit o vlastnictví. Přínos přispěvatele je příliš malý, nejvýše tedy můžeme hovořit o specializaci. Každá změna přispěvatele navíc musí být schválena správcem, což odpovídá modelu slabého vlastnictví kódu. V tomto modelu je dovoleno měnit cizí kód, ale je nutné o změně vlastníka informovat, případně ji musí vlastník schválit.

Podobně jako u Firmy tak můžeme model vlastnictví kódu rozdělit na dvě úrovně – uvnitř týmu a mezi týmy. Tým je v tomto případě pouze jediný, jeho členové jsou správci. Tým je relativně malý, je tedy v něm možné uplatňovat libovolný model vlastnictví kódu. Přispěvatelé nejsou vlastníky žádného kódu a ani nejsou členy žádného týmu. Veškerý kód je tedy pro přispěvatele cizí a pro jeho úpravu uplatňují mezi-týmový model vlastnictví. V případě otevřeného software jím je slabé vlastnictví.

Kapitola 4

Návrh nástroje pro identifikaci vlastníků

V předcházející kapitole jsem ve vybrané firmě identifikoval potenciál na zlepšení. V další části mé diplomové práce se pokusím zlepšení dosáhnout tvorbou softwarového nástroje. Ten by měl sloužit k lepší identifikaci vlastníků oblastí ve zdrojovém kódu. Nástroji jsem dal pracovní název *Who-Owns*. Název odkazuje na předpokládaný nejtypičtější případ použití — zodpovídání dotazu „Who owns…?“ tedy česky „Kdo vlastní…?“.

V této kapitole identifikuji požadavky na nástroj (4.1), navrhnu uživatelské rozhraní (4.2), identifikuji hlavní komponenty systému (4.3), vytvořím vlastní postup pro určení vlastníků kódu (4.4), vyberu a upravím postup pro extrakci klíčových slov (4.5) a rozhodnu otázku zda-li má nástroj stanovovat vlastnictví na úrovni jedinců nebo celých týmů (4.6). Jednotlivé aspekty pak v navazující kapitole Implementace (5) složím do funkčního celku.

Pro identifikaci vlastníků zdrojového kódu, potažmo osob s největší znalostí dané oblasti, využívá Who-Owns data získaná ze systému správy verzí (anglicky *version control system*). Všechny moderní systémy správy verzí ukládají stav projektu při každé změně, autora změny, poznámku autora ke změně a čas provedení. (Systémy správy verzí jsou častěji implementovány tak, že místo celkového stavu ukládají změny. Stav lze rekonstruovat.) K dispozici je tedy kompletní historie vývoje zdrojového kódu. Z něj nástroj určuje vlastníky oblastí v kódu. Tento proces stojí na několika předpokladech:

1. Změny zdrojového kódu nejsou nahodilé, vyjadřují autorovu znalost o řešeném problému.

2. Řešený problém je lokální. Příbuzné koncepty jsou ve zdrojovém kódu v příbuzných oblastech.

3. Největší znalost dané oblasti kódu má její vlastník. Pokud pro danou oblast neexistuje autor s jasně převažující znalostí, jednoznačný vlastník neexistuje.

Předpoklad č. 1 můžeme označit za realistický. Bez znalosti dané problematiky nelze ve větším projektu dosáhnout požadovaného výsledku. Existující výzkumy toto tvrzení podporují [9]. Předpoklad č. 3 je automaticky platný v modelu silného vlastnictví kódu (změny má dovoleno provádět pouze vlastník). V případě slabého vlastnictví může nastávat situace, kdy jednoznačný vlastník nebude identifikován. Jako náhradu může nástroj v tomto případě nabídnout seznam expertů v dané oblasti kódu.

Nejvíce kontroverzí vzbuzuje předpoklad č. 2. Ten je postaven na jednom ze základních principů objektově orientovaného návrhu - vysoké koheze a nízké provázanosti (GRASP – General Responsibility Assignment Software Patterns). Můžeme tedy předpokládat, že moderní software by se o tento stav měl snažit. Nelze ale obecně říci, do jaké míry je v reálných projektech skutečně aplikován. Pokud by zdrojový kód známky dobrého návrhu nevykazoval, nebude Who-Owns u větší programové jednotky schopen identifikovat vlastníky, protože by obsahovala nesouvisející koncepty, a tím pádem by byla rozdrobena i vlastnická struktura. Kvalitní návrh software je tedy předpokladem pro spolehlivé určení vlastníků, v opačném případě nástroj bude schopen nalézt vlastníky pouze u malých jednotek. Obejít tento nedostatek si dává za cíl extrakce klíčových slov, kterou se podrobně zabývám v kapitole 4.5.

Předpoklady na kterých Who-Owns stojí se tedy zdají být realistické. Jaké informace jsou však ze systému správy verzí dostupné a co vypovídají o znalostech autora změny? Krátký souhrn uvádím v následujícím výčtu, podrobněji problematiku rozvádím v následujících kapitolách.

- Odhad závažnosti změny — O závažnosti změny vypovídá její rozsáhlost a dotčené konstrukce. Například změna deklarace metody nebo třídy je jistě závažnější, než změna její implementace.
- Model zapomínání — Staré změny mají menší váhu než mladé. Velké netriviální změny jsou zapomínány pomaleji, než změny malé. Opakované změny v příbuzné oblasti udržují i staré změny čerstvé. [6, 7]
- Model zastarávání — Staré indikátory znalostí přestávají být relevantní, pokud je příslušné místo změněno někým jiným.
- Následováním souvislostí v kódu — Použití metody značí, že autor věděl jak metodu použít. To implikuje jistou znalost metody samotné.
- Extrakce klíčových slov — Komentáře, identifikátory a poznámky autorů v systému správy verzí mohou obsahovat klíčová slova (v lingvistickém smyslu, nikoliv klíčová slova programovacího jazyka). Ta mohou dopomoci najít další souvislosti v kódu. Navíc by mohlo být možné přiřadit vlastníky i k těm konceptům, které jsou rozprostřeny napříč zdrojovým kódem.

Systém bude mít dvě hlavní části. První z nich bude analyzovat systém správy verzí a výsledky bude ukládat do databáze. Uživatelé budou používat druhou část, která bude databázi prohledávat a výsledky zobrazovat. Jeho podoba je rozebrána v kapitole 4.2, poté co se v následující kapitole věnuji nalezení požadavků na systém.

4.1 Požadavky

V této kapitole uvádím požadavky na systém ve formě uživatelských příběhů. Ty vyjadřují záměr uživatele, s jakým by nástroj použil, i způsob použití. Příběhy záměrně nejsou příliš podrobné, aby neomezovaly rozvíjení návrhu. Před každým příběhem uvádím v závorkách prioritu jako číslo od 1 do 10, přičemž vyšší hodnota značí vyšší prioritu. Při vývoji nástroje se budu nejdříve zaměřovat na realizaci položek s vyšší prioritou.

1. (10) Jako programátor chci vyhledat osobu s nejlepší znalostí modulu, třídy nebo metody.

2. (6) Jako programátor chci vyhledat osobu s nejlepší znalostí pojmu.
3. (7) Jako uživatel chci, aby systém automaticky dokončoval vyhledávaný dotaz.
4. (6) Jako programátor chci zobrazit položky (moduly, třídy, funkce, pojmy, týmy, lidé), které s hledanou položkou souvisí.
5. (4) Jako programátor chci vyhledat tým, jež je vlastníkem modulu, třídy, metody nebo konkrétního řádku v kódu. Osoby z toho týmu budu zvát ke kontrole mé změny, přeji si tedy vidět i aktuální členy týmu seřazené podle jejich znalosti hledané položky.
6. (3) Jako programátor chci vylepšit bod 5 o analýzu vlastníků všech dotčených oblastí mé změny.
7. (1) Jako programátor si přeji vylepšit bod 6 o automatické zaslání pozvánky na hodnocení kódu. Přeji si mít možnost z vlastníků vybrat příjemce pozvánky.
8. (5) Jako tester chci vyhledat osobu s nejlepší znalostí kódu na cestě zásobníku volání.
9. (6) Jako uživatel chci zobrazit položky, které s mou osobou systém spojuje.
10. (1) Jako uživatel chci upravit váhy položek, které se mnou systém spojuje.
11. (2) Jako uživatel chci zakázat nebo opětovně povolit zobrazování mé osoby v systému.
12. (1) Jako uživatel chci přispívat ke zpřesnění výsledků označováním nerelevantních položek a přidáváním chybějících relevantních položek.
13. (10) Jako administrátor chci nastavit parametry pro přístup nástroje do systému správy verzí.
14. (8) Jako administrátor chci analýzu pozastavit a opětovně spustit.
15. (7) Jako správce systému chci upravovat seznam týmů, zaměstnanců i členství v týmu.

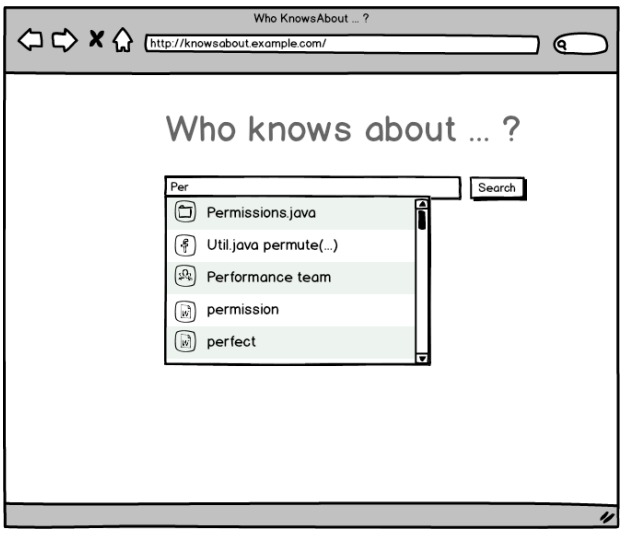
Nefunkcionálními požadavky jsou dostatečná relevance výsledků, intuitivní zobrazení dat a přiměřeně rychlá odezva systému. Konkrétní hodnoty nemá v brzké fázi vývoje tohoto inovativního nástroje smysl stanovovat.

4.2 Uživatelské rozhraní

Hlavním vstupem do systému bude textové vyhledávací pole s funkcí automatického doplňování. Vyhledávat bude možné libovolnou položku (modul, třída, funkce, řádek kódu, klíčové slovo, tým nebo jedinec). Příslušný typ položky bude v celém systému zaznačen ikonou přímo u jejího názvu. Očekávám totiž, že názvy tříd a metod se budou často shodovat s pojmy (klíčovými slovy). Po zadání vyhledávaného dotazu bude uživatel přesunut na stránku s přehledem všech souvisejících prvků. Mezi nejdůležitější bude patřit propojení vyhledávaného dotazu s jedinci (uživatelský příběh 1 a 2) a s týmy (příběh 5). Prototyp rozhraní pro zadání dotazu je načrtnut na obrázku 4.1.

Velice přínosnou funkcí při sestavování dotazu je funkce automatického dokončení. Poskytuje uživateli okamžitou odezvu o dostupných položkách. Očekávám také, že vyhledávané

pojmy mohou být velmi dlouhé. Nejdelšími vyhledávanými položkami budou nejspíše plně kvalifikovaná jména z jazyka Java, které nezřídka dosahují délky desítek znaků. Pro uživatele bude v tomto případě nejsnadnější zadat podřetězec, kterým se vyhledávaný dotaz odlišuje. Zatímco uživatel píše, systém nabízí všechny položky podřetězec obsahující. Požadovanou položku pak bude možné jednoduše vybrat z krátkého seznamu.



Obrázek 4.1: Prototyp uživatelského rozhraní — zadání vyhledávaného dotazu. [Vlastní]

4.3 Hlavní celky systému

Systém se vnitřně dělí na několik hlavních komponent – část zabývající se analýzou změn uložených v systému správy verzí a na část výstupní, která uložené informace zpracovává a zpřístupňuje. Výstupem může být prostý textový soubor nebo interaktivní webové uživatelské rozhraní. Obě části jsou součástí jediného programu, přičemž vykonávané úlohy a výstupy uživatel specifikuje konfiguračním souborem. Hranice systému, hlavní komponenty a tok dat je znázorněn na obrázku 4.2. Konkrétní informace včetně diagramu tříd jsou uvedeny v kapitole zabývající se implementací (5).

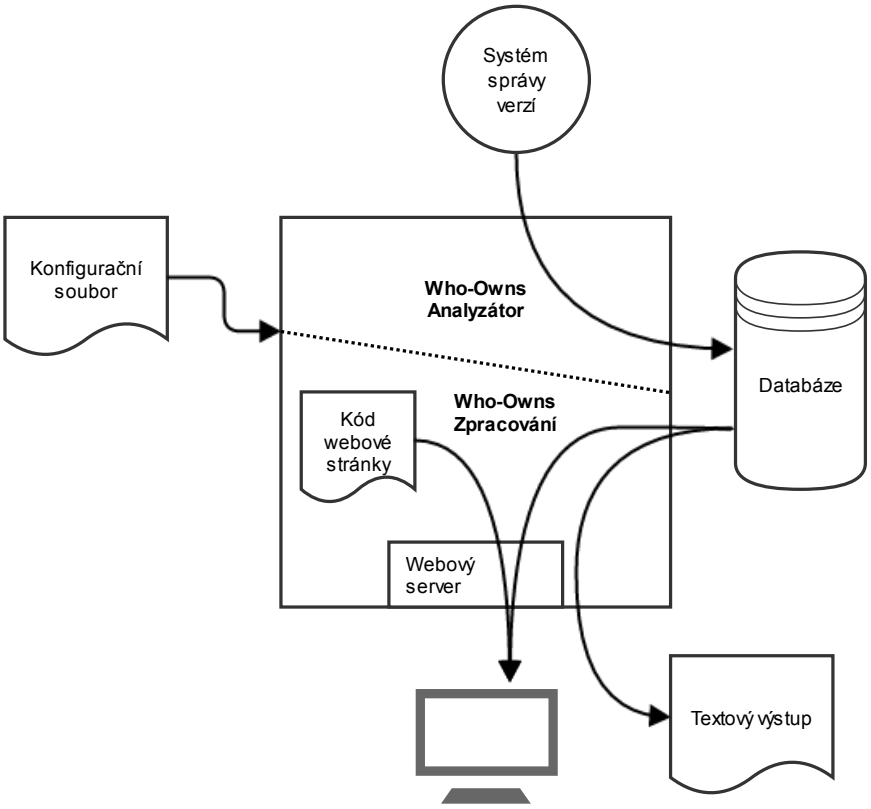
Analyzátor změn prochází systém správy verzí a extrahuje z něj podstatná data, běží na pozadí a administrátor jej může pozastavovat a opětovně spouštět s cílem ulehčit systému správy verzí v časech velkého vytížení. Analyzátor zpracovaná data ukládá do databáze.

Modul uživatelského rozhraní je spuštěn jako webová služba a také se dělí na dvě hlavní části. Na webovém serveru běží modul back-end, jehož úkolem je na požádání vyhledávat data v databázi. Výsledky vrací ve formátu JSON[1]. Uživatelské rozhraní je vytvořeno dynamicky pomocí jazyka JavaScript přímo ve webovém prohlížeči.

Databáze obsahuje nejen potřebné podpůrné údaje pro analýzu ale i výstup analýzy. Tedy klíčová slova, nalezené oblasti kódu se strukturou autorů a všechny podpůrné informace pro model lidské paměti a model zastarávání kódu.

Požadavkem na databázi je zajištění persistence dat pro systém, ale plná podpora modelu *ACID* není požadována (atomičnost, konzistence, izolace, stálost). Zapisovat do systému smí

[1] *JavaScript Object Notation* je formát pro výměnu dat, který vychází ze způsobu zápisu objektů a polí v jazyce JavaScript.



Obrázek 4.2: Obrázek znázorňuje dvě hlavní vnitřní komponenty a hranice systému. Šipky představují směr toku dat. [Vlastní]

pouze jediná aplikace – analyzátor, která přebírá zodpovědnost za atomičnost a konzistenci. Aby databáze nemusela řešit izolovanost transakcí, omezil jsem počet instancí analyzátoru na jednu. Garance stálosti dat taktéž není potřebná, ukládaná data budou kvantifikačního charakteru. Nebude tak vadit, pokud systém několik málo změn zpracuje vícekrát a jiné přeskočí. Přesnost výsledků se tím sice sníží, ovšem při velkém počtu zpracovaných změn budou celkové výsledky ovlivněny minimálně.

Požadavky na databázi umožňují použití databáze i jiné než relační, tedy některé databáze z rodiny *NoSQL*. Výběru databáze věnuji dále samostatnou kapitolu 5.5. Nejdříve se věnuji postupům pro stanovení vlastníků. Z nich pak požadavky na databázový systém vyplynou.

4.4 Určení vlastníků oblastí ve zdrojovém kódu

Analýza zdrojových kódů má za cíl nalézt osoby případně týmy s největší znalostí logického celku zdrojového kódu a tím identifikovat vlastníky. Tento úkol v sobě zahrnuje i otázku, která není na první pohled patrná. *„Co je to logický celek zdrojového kódu?“* Logický celek by měl nést jedinou zodpovědnost, existovat za jediným účelem, případně by měl zajišťovat jedinou funkcionalitu programu. Toto tvrzení je logické a v konzistenci se softwarovým inženýrstvím, nijak ale nepřispívá k řešení praktického problému. Tedy jak logický celek konkrétně definovat, aby ho *Who-Owns* byl schopen automaticky poznat. Touto otázkou

se zabývám v následující podkapitole. Dále se zabývám návrhem vlastního algoritmu pro stanovení vlastníků (4.4.2) a otázkou, v jak co nejlépe prezentovat výsledky (4.4.3).

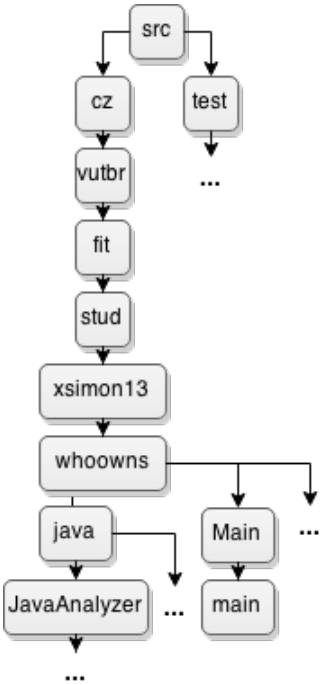
Rozhodl jsem se, že Who-Owns bude analyzovat zdrojové kódy v jazyce Java. Je to moderní a rozšířený jazyk, ve kterém je napsána většina zdrojových kódů ve Firmě. Důležitou vlastností jazyka je i jeho velmi dobrá podpora modulů. Proč je toto výhodou vyplyne z následujících řádků.

4.4.1 Logické celky kódu

Definice logického celku je závislá na programovacím jazyce, kterým je Java. Java je objektově orientovaný, silně typový jazyk. Na třídy klade velký důraz – nepodporuje ani samostatné funkce, ty musí být uzavřeny do tříd jako statické metody. Jednoznačně tak za logické celky můžeme považovat třídy a metody. Třídy jsou v Javě typicky zapisovány do samostatných souborů (vyjma zanořených tříd, syntaxe také umožňuje zapsat více privátních tříd do jednoho souboru, ale nejedná se o doporučený postup). Soubory jsou organizovány do balíčků (také modul, terminologií jazyka `package`), které tvoří stromovou strukturu. Stromová struktura balíčků odpovídá adresářové struktuře, ve které jsou soubory uloženy. Při návrhu software by měla být nemalá část úsilí věnována vhodným rozdělením na moduly, můžeme je tedy považovat za další logický celek.

V jazyce Java jsem tedy identifikoval několik logických celků, které vyplývají přímo z konstrukcí jazyka:

- modul (případně lze použít adresářovou strukturu)
- třída
- metoda



Obrázek 4.3: Příklad stromové struktury programu. [Vlastní]

Tyto logické celky lze uspořádat do stromu, který lze chápat jako rozdělení celé funkcionality programu. Kořen stromu tedy odpovídá celému programu. Postupem na nižší uzly se vyčleňují velké oblasti kódu (např. testy oproti kódu zajišťujícím funkcionalitu), níže leží moduly, pak třídy, zanořené třídy a na nejnižší úrovni se díváme na jednotlivé metody (případně také bloky kódu). Každou oblast ve zdrojovém kódu lze tímto způsobem jednoznačně identifikovat. Ukázka stromové struktury je na obrázku 4.3.

Who-Owns by měl usilovat o identifikaci vlastníků na co nejvyšší úrovni. Mnohem větší hodnotu má informace o vlastnictví celého modulu, než o autorech jednotlivých metod. Pokud ovšem vlastnictví na vyšší úrovni nebude jednoznačné, bude nutné nalézt vlastníky na úrovni nižší. Výhodné bude také identifikovat výjimky z vlastnické struktury vyskytující se na nižší úrovni. (Např. pokud celý modul má jediného vlastníka, ale konkrétní třídu má na starost někdo jiný.)

Emergentní logické celky

Logické celky nemusí přímo vyplynout z konstrukcí jazyka. Jistě může nastat situace, kdy jeden vlastněný koncept bude rozprostřen napříč několika moduly. Tato situace může nastat například při uplatnění návrhového vzoru *strategie*, kdy všechny implementace dané strategie mají sice odlišné chování a typicky se budou nacházet na různých místech zdrojového kódu, ale sdílí společný účel. Právě tento účel může být někým vlastněn.

Stanovit logické celky se tedy můžeme pokusit i analýzou vzájemných závislostí ve zdrojovém kódu. Logické celky tak vyplynou přímo ze zdrojového kódu – odtud název emergentní (z latinského emergo – vyplynout). Dvě oblasti jsou si tím více podobné, čím více:

- jedna oblast vytváří instance / obsahuje proměnné typu / volá metody druhé oblasti.
- spolu sousedí ve zdrojovém kódu.
- obsahují stejná klíčová slova.
- jsou upravovány v jediné změně.
- mají stejného vlastníka.

Identifikace celků by pak mohla fungovat zhruba následovně. Nejdříve zjistíme vzájemné vztahy mezi oblastmi a s patřičnými váhami je zaneseme do čtvercové matice. Z matice sestavíme graf. Oblasti zdrojového kódu budou představovat uzly, maticí vzájemných vztahů ohodnotíme hrany. Provedeme shlukovou analýzu grafu[2]. Výsledné shluky označíme za logické celky a stanovíme pro ně vlastníky.

Postup by byl velmi zajímavý, v této diplomové práci jsem se ale rozhodl jej dále nerozvíjet. Kvalitu výsledků nelze předem nijak odhadnout – předpokládám, že podoba logických celků bude velmi citlivá na nastavení vah. Také výpočetní náročnost i požadavky na ukládaný objem dat jsou velmi vysoké. Stanovuji tedy, že Who-Owns bude v současné verzi považovat za logické celky pouze struktury přímo obsažené v jazyce – moduly, třídy, metody a volitelně bloky kódu.

4.4.2 Algoritmus stanovení vlastníků

V předchozích odstavcích jsem tedy identifikoval celky, pro které Who-Owns stanovuje vlastníky. Nyní se zaměřím na druhou stranu problému – samotné určení vlastníků. Pro tento problém jsem *navrhl a vytvořil vlastní postup*. Jak již několikrát zaznělo, Who-Owns získává data ze systému správy verzí. Ten ukládá každou změnu, která byla ve zdrojovém kódu provedena. Pro každý dotčený soubor umožňuje systém nalézt všechny rozdíly oproti verzi předchozí. Soubor jako takový může být vytvořen, přesunut, modifikován nebo smazán. Ke změně uložené v systému její autor obvykle přikládá krátký popis.

Hlavním nositelem informace jsou upravené a přidané řádky kódu. Tedy přesněji jejich počet a význam. Význam je samozřejmě nemožné přesně kvantifikovat, můžeme ale alespoň přiřadit strukturám jazyka váhy. Samotná změna řádku kódu má váhu 1. Za každou další obsaženou strukturu získává body navíc. Konkrétní hodnoty jsou uvedeny v tabulce 4.1.[3]

[2] Shluková analýza rozdělí graf na podgrafy tak, aby vzájemná provázanost uvnitř podgrafu byla co největší, mezi podgrafy co nejmenší. Jedná se o NP-těžký problém. Moderní algoritmy, které řešení pouze aproximují, dosahují složitosti $O(N)$. Jedním z nich je MCL `http://micans.org/mcl/`

[3] Pozn. vybočující z návrhu – První iterace Who-Owns jsem testoval při váze třídy 500. Neuvědomil jsem si ale, že v v jazyce Java je běžné vytvářet velký počet zanořených tříd. Případně upravovat třídy přímo při vytváření instance. Bylo tedy nutné váhu třídy patřičně snížit.

	Váha	Revidovaná váha
Řádek kódu	1	1
Deklarace proměnné	2	2
Deklarace členské proměnné	50	30
Definice metody	20	15
Definice třídy	500	50

Tabulka 4.1: Váhy struktur jazyka pro stanovení vlastníků. Před a po úpravě.

Pokud je tedy ve změně dotčena i deklarace či definice patřičné konstrukce jazyka, změna obdrží větší váhu. Domnívám se, že tento bonus bude mít klíčový přínos pro správnou identifikaci vlastníků. Má domněnka vychází z přirozeného chování programátorů, kteří při úpravách cizího kódu mají tendenci dělat minimální možnou změnu. Motivací tohoto chování je snížení rizika konfliktů protichůdných změn. Naopak vlastník tuto obavu mít nemusí. Navíc vlastníci typicky mají na starost udržování vysoké úrovně kódu, což obnáší změny názvů, přesuny kódu apod. Pokud vlastník v rámci údržby kódu přejmenuje metodu, obdrží bez většího úsilí bonus o váze 15.

Jak jsem popsal v kapitole 4.4.1, logické celky tvoří stromovou strukturu. Každý uzel uchovává seznam autorů s jejich zjištěným skóre, které rekurzivně zahrnuje i skóre z nižších uzlů. Váha každé změny je tedy připočtena ke všem nadřazeným uzlům. To umožňuje algoritmu efektivně porovnávat vlastnictví velmi obsáhlých oblastí kódu.

Další potřebnou komponentou je stáří změny. Staré změny jsou méně relevantní kvůli nedokonalosti lidské paměti i kvůli změnám v systému. Informace, získané před rokem již nemusí být platné, protože podoba systému se mění. Pro modelování těchto efektů je typicky používána exponenciální funkce [3, 9]. Přesný tvar uvádím ve vzorci 4.1.

$$\mathrm{W_t}(t) = 2^{\frac{t-epocha}{\gamma}} \tag{4.1}$$

- t je čas změny.
- *epocha* je konstantní časový okamžik. Je vhodné jej zvolit tak, aby výsledné $\mathrm{W_t}$ bylo přiměřeně velké.
- γ je doba za kterou se v projektu vytratí polovina hodnoty informací. Výchozí délku jsem stanovil na 1 rok. Rychleji měnící se projekty by měli používat kratší interval.

Who-Owns odlišuje vlastníky od cizích zásahů kontrolou pravidelnosti. Pokud autor alespoň jedenkrát ve všech časových intervalech provedl v oblasti kódu zásah, neobdrží skóre žádnou penalizaci. V opačném případě penalizace lineárně roste s přibývajícím počtem intervalů, kdy kód ležel ladem. Jako délku těchto intervalů jsem zvolil měsíc. Je jasné, že velmi malé oblasti (např. metoda) nebudou každý měsíc měněny. Zbytečná penalizace ale ničemu nevadí, vlastníci nejsou určováni na základě absolutní velikosti svého skóre, ale podle vzájemných poměrů. Tento efekt je naopak velmi výhodný. Díky němu jsou zvýhodňovány oblasti ležící výše ve stromové struktuře. Určení skóre vlastníka pro oblast o je popsáno vzorcem 4.2.

$$\mathrm{W}(v,o) = \left(0,2 + \frac{m_v}{M}\right) \cdot \sum_{z \in Z_v} \mathrm{W_t}(z_t) \cdot \mathrm{W_{zmena}}(z,o) \tag{4.2}$$

- v je vlastník, o oblast, Z_v množina změn vlastníka, z_t je čas změny
- m_v je počet měsíců, kdy vlastník provedl alespoň jednu změnu v oblasti.
- M je celkový analyzovaný počet měsíců.
- $\mathrm{W_{zmena}}(z,o)$ je suma vah všech upravených či přidaných řádků kódu ve změně z zahrnuté v oblasti o. Váhy jsou přiřazeny podle tabulky 4.1.

Postup jsem vytvářel tak, aby jeho vlastnosti usnadňovaly implementaci. Například je důležité, že je možné veškeré potřebné informace získat jediným průchodem historií systému správy verzí. I jednotlivé změny mohou být analyzovány v libovolném pořadí.

4.4.3 Prezentace výsledků

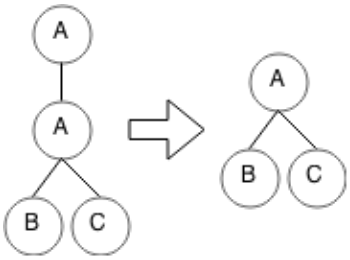
Z požadavků (viz kapitola 4.1) vyplývají dva typy otázek, které bude Who-Owns zodpovídat. Prvním je „Co uživatel vlastní?“, druhým pak „Kdo je vlastníkem…?“. Obě otázky tedy nahlíží na vlastnictví kódu, každá ale z opačného konce. Protože stromy logických celků mohou být velice rozsáhlé, hledám v následujících kapitolách vhodný způsob zpracování dat a následné prezentace odpovědí v co nejpřehlednější formě. Způsoby se odlišují podle typu otázky.

Co uživatel vlastní?

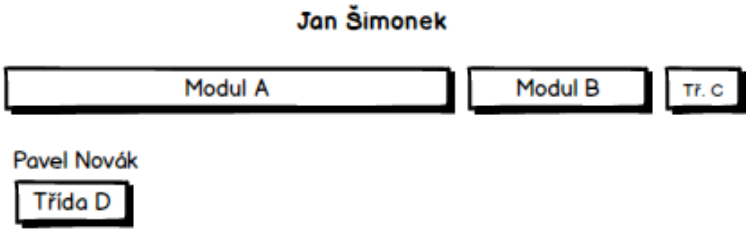
Vstupem pro zodpovězení dotazu je v tomto případě uživatel (případně tým) a požadovaným výstupem je seznam oblastí kódu, které uživatel vlastní. K dipozici je stromová struktura oblastí, kde každý uzel obsahuje seznam autorů s vypočítaným ohodnocením $\mathrm{W}(v,o)$.

Prvním krokem zpracování je identifikace jednoznačného vlastníka pro každou oblast (pokud existuje). Jako nejefektivnější se mi jeví filtrovat vlastníky podmínkou $w_1 > k \cdot w_2$, kde w_1 a w_2 je nejlepší respektive druhé nejlepší ohodnocení v oblasti. Konstanta k je mez o kterou se musí vlastník odlišovat. Výchozí hodnotu meze jsem empiricky stanovil na $1,2$. Pokud podmínka není splněna jednoznačný vlastník neexistuje.

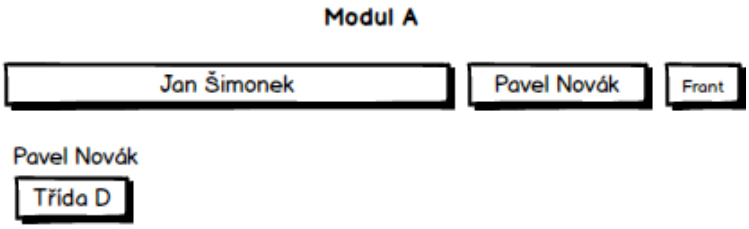
Strom vlastníků je možné zjednodušit odstraněním přebytečných uzlů. Uzel je odstraněn, pokud je jeho vlastník shodný s vlastníkem jeho rodiče. Před odstraněním uzlu je samozřejmě zapotřebí přemístit potomky uzlu do jeho rodiče. (Viz obrázek 4.4.)



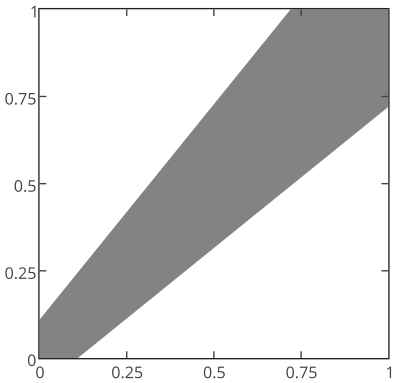
Obrázek 4.4: Zjednodušení stromu vlastníků. [Vlastní]



Obrázek 4.5: Prezentace výsledků pro dotaz „Co vlastní Jan Šimonek?“ [Vlastní]



Obrázek 4.6: Prezentace výsledků pro dotaz „Kdo je vlastníkem modulu A?“ [Vlastní]



Obrázek 4.7: Funkce porovnání vlastníků. [Vlastní]

S takto zpracovaným stromem je již jednoduché se k odpovědi dopracovat. Ze stromu jsou vybrány ty uzly, které uživateli patří. Ty jsou seřazeny podle ohodnocení $\mathrm{W}(v, o)$ a v tomto pořadí prezentovány uživateli, přičemž dostupná plocha obrazovky je mezi oblasti rozdělena podle poměrů nenormalizovaného ohodnocení. Pokud se na zanořených úrovních nacházejí podstatné výjimky, jsou zobrazeny. Nejlépe myšlenku zachycuje obrázek 4.5. Povšimněte si výjimky `Třída D`, která je vlastněna Pavlem Novákem.

Kdo je vlastníkem?

Pro zodpovězení tohoto dotazu jsem navrhl podobný postup jako pro dotaz předchozí. Pro každý uzel ale algoritmus nevybírá pouze jediného vlastníka, ale ponechá seznam vlastníků celý. Ohodnocení vlastníků je vyděleno sumou ohodnocení v daném uzlu, čímž dostává význam procentuálního podílu.

Odstraňování přebytečných uzlů je prováděno obdobně, pouze vzájemné porovnání uzlů musí být jiné. Uzel je odstraněn, pokud si rozdělení vlastníků zhruba odpovídá. Význam slova zhruba definuji takto: Nechť V je množina všech vlastníků. Rozdělení vlastníků je pro oblasti o_1 a o_2 *zhruba* stejné právě tehdy, když platí podmínka ze vzorce 4.3.

$$\forall v \in V: \quad |\mathrm{W}(v, o_1) - \mathrm{W}(v, o_2)| < 0.1 + 0.1 \cdot (\mathrm{W}(v, o_1) + \mathrm{W}(v, o_2)) \tag{4.3}$$

Porovnání je tedy tolerantnější vůči odchylkám od hodnoty 1 a striktnější s poklesem podílu vlastnictví. Průběh funkce je zaznačen na grafu 4.7. Přípustná tolerance je na grafu vyznačena šedě.

Výsledky jsou taktéž zobrazeny obdobně jako u předchozího případu. Na nejvyšší úrovni je dotazovaná oblast, pod ní jsou vlastníci, přičemž velikost kolonky odpovídá vypočtenému ohodnocení. Návrh zachycuje obrázek 4.6.

4.5 Extrakce klíčových slov

Jak jsem již dříve nastínil, důležitou součástí určování vlastnictví je nejen zodpovězení otázky „Kdo je vlastníkem?“, ale i „Co je vlastněno?“ Co se může stát vlastněným objektem vyplývá z rozdělení zodpovědnosti v týmu. Předchozí kapitola stála na předpokladu, že předmětem vlastnictví může být jednotka ve zdrojovém kódu (metoda, třída nebo modul). Naproti tomu extrakce klíčových slov v nástroji Who-Owns je založena na předpokladu, že vlastněné abstraktnější koncepty a zodpovědnosti mohou být vyjádřeny klíčovými slovy či frázemi. Také předpokládá, že tato klíčová slova se vyskytují v kódu i komentářích a je možné je odtud extrahovat. Na rozdíl od vlastnictví zdrojového kódu nejsou klíčová slova závislá no kvalitním softwarovém návrhu (viz úvod kapitoly 4).

Identifikace a extrakce klíčových slov je velmi dobře prozkoumaný obor s dlouhou historií. Existuje řada přístupů s velmi dobrými teoretickými podklady, které jsou i hojně prakticky používány. Některé z nich popisuji níže. Nicméně, vstupní text pro analýzu je v nástroji Who-Owns velmi specifický. Klasické přístupy často pracují s běžným textem a nejlépe pracují na dlouhých dokumentech. Who-Owns analyzuje texty získané ze zdrojových kódu, jejich délka je tedy velmi krátká. Charakter textu je také odlišný, neboť obsažené informace jsou kondenzovanější. Netypický je i požadavek na unikátnost klíčového slova pro daného vlastníka. Musel jsem tedy nalézt algoritmus, který si s těmito specifiky dokáže poradit, a dále jej upravit. V následujících odstavcích velmi krátce popisuji princip několika postupů. Zvláště se zaměřuji na hodnocení možnosti aplikace algoritmu pro toto neobvyklé použití.

4.5.1 Algoritmus Term Frequency - Inverse Document Frequency

Term Frequency - Inverse Document Frequency (*TF-IDF*) [14] je jedním z nejstarších algoritmů pro extrakci klíčových slov. Pracuje se dvěma vstupy – jazykový korpus a samotný analyzovaný dokument. Jazykový korpus je obsáhlý soubor dokumentů a slouží pro naučení se průměrné frekvence výskytu slov v jazyce. Princip samotné analýzy je prostý – dává do souvislosti pravděpodobnost výskytu slova v daném dokumentu s pravděpodobností výskytu slova v korpusu. Čím častěji se slovo v dokumentu vyskytuje (složka TF – *term frequency*) a méně dokumentů v korpusu jej obsahuje (IDF – *inverse document frequency*), tím je váha slova větší (viz vzorce 4.4 a 4.5).

$$\mathrm{TFIDF}(slovo, dokument) = IDF(slovo) \cdot \mathrm{TF}(slovo, dokument) \tag{4.4}$$

$$\mathrm{IDF}(slovo) = -\log\left(\frac{\mathrm{df}(slovo)}{N}\right) \tag{4.5}$$

- $df(slovo)$ je počet dokumentů korpusu, které dané slovo obsahují.
- N je počet dokumentů korpusu.
- $\mathrm{TF}(slovo, dokument) = \frac{\mathrm{tf}(slovo, dokument)}{|dokument|}$
- $\mathrm{tf}(slovo, dokument)$ je počet výskytů slova v dokumentu.
- $|dokument|$ je počet slov v dokumentu.

Korpus máme v našem případě k dispozici. Můžeme za něj považovat identifikátory a text komentářů v celém projektu. Pokud budeme za dokument považovat komentáře nebo identifikátory ve zdrojovém kódu, zjistíme, že složka TF nepřináší žádnou informační hodnotu. Naveed et. al [13] pro překonání nízké statistické významnosti definoval TF rovno jedné. Nejlepší ohodnocení se tak ale dostane těm slovům, které se v celém korpusu objevují pouze jednou – překlepy, příliš unikátní výrazy apod. Přesto IDF složka jistou informační hodnotu přináší, a v kombinaci s dalšími ukazateli je možné ji použít.

Alternativně by bylo možné jako dokument zvolit konkatenaci všech textů sepsaných daným vlastníkem. Potom by obě složky složky měli svůj význam. Dokonaleji je ale tato myšlenka rozvedena v přístupu vyvinutým Mikou Timonenem, který popisuji v kapitole 4.5.3.

4.5.2 Algoritmus HG-Rank

HG-Rank [3] je moderní algoritmus speciálně navržený pro analýzu krátkých dokumentů v sociální síti Twitter. Ve svém jádru algoritmus vytváří hypergraf[4], kde uzly jsou množinou nalezených slov a kde dokument dává vzniknout hraně propojující obsažená slova. Důležitou informaci přináší časový a sociální aspekt dokumentu, který je zakódován jako váha hrany. Algoritmus dává přednost novým a populárním dokumentům. Také využívá TF-IDF pro váhy uzlů. Ohodnocení samotných slov je vypočítáno na základě náhodného průchodu hypergrafem, pro což autoři vytvořili speciální postup.

Nevýhodou tohoto algoritmu je jeho náročnost na velikost ukládaných dat a výpočetní náročnost náhodné procházky grafem. Nejasné je také čím v prostředí analýzy zdrojového

[4] Hypergraf je zobecněním grafu, kdy jedna hrana e může propojovat neomezený počet uzlů. $e \subseteq V$ (V je množina uzlů.)

kódu nahradit sociální aspekt dokumentů ze sociální sítě. Výhodu algoritmu spatřuji ve využívání informace náležitosti slov do jednoho dokumentu. Podivuji se ale nad tím, že autoři algoritmu přesto extrahují pouze jednotlivá klíčová slova. Extrakce celých frází by byla zvláště výhodná při zpracování identifikátorů, kde i běžná slova mohou v unikátní kombinaci být nositelem značné informační hodnoty.

Rozhodl jsem se upřednostnit jiný algoritmus a HG-Rank nepoužít.

4.5.3 Víceúrovňové hodnocení klíčových slov

Autorem dalšího postupu je Miky Timonen [14], který jej ale nijak nepojmenoval. V této práci se něj tedy odkazuji jako na *algoritmus víceúrovňového hodnocení* (*klíčových slov*). Jeho největší odlišností je zavedení kategorií dokumentů, ta by měla sdružovat dokumenty s podobným obsahem. Celkové hodnocení slova sestává ze tří složek.

První složkou je hodnocení slova na úrovni korpusu, které vychází z IDF. Navíc ale zavádí požadovanou optimální frekvenci slova (viz vzorec 4.6). Pokud je slovo až příliš vzácné, složka FW ho penalizuje (viz vzorec 4.7). Zcela unikátní slova jsou totiž často pouze překlepy nebo jsou až příliš specifická.

$$\mathrm{IDF_{FW}}(s) = \mathrm{IDF}(s) - \mathrm{FW}(s) \tag{4.6}$$

$$\mathrm{FW}(s) = \alpha \cdot \left| \log \frac{\mathrm{tf}(s)}{n_0} \right| \tag{4.7}$$

Hodnocení na úrovni kategorie je nepodstatnějším prvkem algoritmu. Je tvořen hned několika komponentami, které uvádím ve vzorcích 5.1, 4.9 a 4.10. Rovnici 4.9 jsem na rozdíl od zbytku algoritmu převzal z [**?**].

$$\mathrm{CAT}(s,k) = \mathrm{BNS}(s,k) \cdot \mathrm{P}(s,k) \cdot \mathrm{IFL}(s) \tag{4.8}$$

$$\mathrm{BNS}(s,k) = \left| \mathrm{F}^{-1}\left(\frac{N_{s,k}}{D_k}\right) - \mathrm{F}^{-1}\left(\frac{N_{s,\neg k}}{D_{\neg k}}\right) \right| \tag{4.9}$$

$$\mathrm{P}(s,k) = \frac{N_{s,k}}{D_s} \tag{4.10}$$

- s označuje slovo, k kategorii, d dokument.
- F^{-1} je inverzní funkce z kumulativního standardního rozdělení pravděpodobnosti.
- $N_{s,k}$ je počet dokumentů v kategorii k, které obsahují slovo s.
- D_k je počet dokumentů v kategorii k.
- $N_{s,\neg k}$ je počet dokumentů obsahující slovo s, které náleží do jiné kategorie než k.
- $D_{\neg k}$ je počet dokumentů v jiných kategoriích než k.
- D_s je počet dokumentů, které obsahují slovo s.
- $\mathrm{IFL}(s)$ je obrácená hodnota průměrné délky vět, ve které se slovo vyskytuje. Pod pojmem věta není myšleno souvětí, pouze samostatná věta.

Výsledná váha slova je spočítána váženým průměrem z $\mathrm{IDF_{FW}}(s)$ a $\mathrm{CAT}(s, k)$ přičemž autor doporučuje váhy určit v závislosti na doméně použití.

Pro potřeby Who-Owns se algoritmus jeví jako velmi vhodný. Za kategorii můžeme v kontextu Who-Owns považovat všechny texty vytvořené jedním člověkem nebo týmem. Složka *BNS* pak koresponduje s intuitivním chápání vlastnictví – objekt může být vlastněn nejvýše jedním vlastníkem. *FW* složka na úrovni korpusu zajistí, že příliš unikátní slova nebudou přespříliš zvýhodňovány.

Oproti výše zmíněné podobě je ovšem nutná úprava funkce *BNS* (Bi-Normal Separation). Ta totiž ve výše uvedeném tvaru upřednostňuje ta slova, u kterých je četnost výskytu v kategorii jiná než ve zbývajících kategoriích – tedy i nižší. Pro určení vlastnictví je ale zajímavá pouze pozitivní korelace. Taková slova, která se častěji vyskytují v dané kategorii než v ostatních. Plánuji tak funkci modifikovat přidáním podmínky $\frac{N_{s,k}}{D_k} > \frac{N_{s,\neg k}}{D_{\neg k}}$.

4.6 Vlastnictví na úrovni týmu a jedince

V případě klasických modelů vlastnictví (viz kapitola 2.1) je vlastníkem vždy programátor. Na základě analýzy situace ve Firmě jsem vytvořil víceúrovňový model. Silné nebo slabé vlastnictví je aplikováno mezi týmy. Uvnitř týmu praxi ve Firmě nejlépe popisuje společné vlastnictví kódu, ale obecně to může být jakýkoli model. Můžeme tak rozlišovat *vlastnictví* na úrovni týmu a *autorství* na úrovni jedinců. Své výhody přináší analýza obojího.

Who-Owns pro určení vlastnictví na úrovni týmů potřebuje znát informace o přiřazení lidí do týmů. To může být zadáno konfiguračním souborem, který obsahuje seznam týmů spolu s jeho členy a počátečním a koncovým datem přiřazení osoby k týmu. Tak je možné vyjádřit i přesuny mezi týmy.

Samozřejmě mohou nastávat i situace, kdy programátor je přiřazen do několika týmů současně. Obsah vytvořený nezařazeným člověkem by pak bylo možné ke správnému týmu přiřadit klasifikací podle již vypočítaného vlastnictví a na základě obsažených klíčových slov. Vodítkem by také mohly být další informace ze systému správy verzí, například jací autoři odeslali své změny v podobný čas. Vícenásobné přiřazení člověka ve Firmě příliš nenastává, z důvodu jednoduchosti tedy tuto eventualitu pominu.

Polemiku předchozích odstavců shrnuji do tohoto rozhodnutí:

> Klíčová slova se vztahují přímo k vlastněným konceptům, jejich extrakce je prováděna pouze na úrovni týmů. Vlastnictví kódu je analyzováno odděleně na úrovni jedinců i na úrovni týmů.

Kapitola 5

Implementace

V kapitole návrh jsem popsal přístupy stojící v jádru nástroje Who-Owns. Nyní se budu zabývat jejich převodem do použitelného nástroje. Některé body jsou vůči tradičnímu pojetí kapitol prohozeny. Například výběr databázové technologie a návrh ukládaných dat uvádím až v této kapitole. Rozdělení je ovšem z odstupu logické. Za hlavní přínos mé práce považuji inovativní myšlenky z předchozí kapitoly. Ty mohou existovat samostatně bez jakéhokoliv software. Program, který jsem v rámci diplomové práce vytvořil, tedy považuji za *implementaci těchto myšlenek* do spustitelné podoby.

Na následujících řádcích popisuji zajímavé body z implementace Who-Owns. Nejdříve věnuji kapitoly přehledu použitých technologií (5.1) a identifikaci hlavních částí programu (5.2). Poté se věnuji vybraným jednotlivým částem programu (5.3, 5.4, 5.5) a jak jsem algoritmus extrakce klíčových slov dále upravoval (5.6). Kapitolu zakončuje metodika vývoje softwaru, kterou jsem se řídil (5.7)

5.1 Použité technologie

Pro implementaci programu jsem zvolil jazyk Java. K tomu mne vedlo několik důvodů. Především je to jazyk s automatickou správou paměti a velice rozsáhlou paletou datových typů. To umožňuje soustředit se na samotný algoritmus. Javu lze považovat za rozumný kompromis mezi nativním kódem s velkou rychlostí spuštění (např. C, C++) a vysokým stupněm abstrakce (např. Python, Smalltalk). Je pro ni také dostupná široká paleta knihoven. K zajímavějším patří syntaktická analýzy Javy a zabudovaný webový server. Přehled všech použitých knihoven a software uvádím v tabulce 5.1.

5.2 Struktura programu

Jádro funkcionality je zajištěno jediným programem. Ten zajišťuje jak analýzu změn a zpracování informací, tak i jejich zpřístupnění textovým výstupem nebo interaktivním rozhraním webového serveru. Chování programu uživatel definuje konfiguračním souborem, kam zadá informace o tom, jaké úlohy se mají vykonat. Druhý konfigurační soubor slouží pro specifikaci personálního složení týmů. Dostupné příkazy i jednoduchou syntaxi souborů popisuji v příloze C. Vykonávání příkazů z konfiguračního souboru zajišťuje modul `Tasks`, složení týmů modul `HR` (viz diagramy tříd 5.1 a A.1).

Na zmíněné moduly navazuje balíček `Versioncontrol`, který umí ze systému správy verzí získat potřebná data (podrobně jej popisuji v kapitole 5.4). Analýzu klíčových slov

Název	URL	Licence	Použití
Inliner	https://github.com/remy/inliner	MIT	Zabalení závislostí webové stránky do jediného souboru.
Redis	http://redis.io/	BSD	Databáze
Redisson	https://github.com/mrniko/redisson	Apache	Klientská knihovna pro Redis
JavaParser	https://github.com/javaparser/javaparser	GNU-LGPL	Syntaktická analýza zdrojových kódů Javy
Statutil	http://jarvis.cs.ucdavis.edu/code_essence/functions/8839925	BSD	Výpočet inverzní funkce z kumulativního standardního rozdělení pravděpodobnosti
NanoHttpd	http://nanohttpd.com/	BSD	Jednoduchý zabudovaný HTTP server
TestNG	http://testng.org/	Apache	Testovací framework
jQuery	http://jquery.com/	MIT	JavaScript knihovna usnadňující tvorbu dynamických webových stránek
typeahead	http://twitter.github.io/typeahead.js/	MIT	

Tabulka 5.1: Použité knihovny a podpůrný software

a zdrojových kódů zajišťují moduly `Text` a `Java`, jejichž chování jsem již podrobně vysvětlil v návrhu. Konkrétní diagramy tříd uvádím v příloze A.

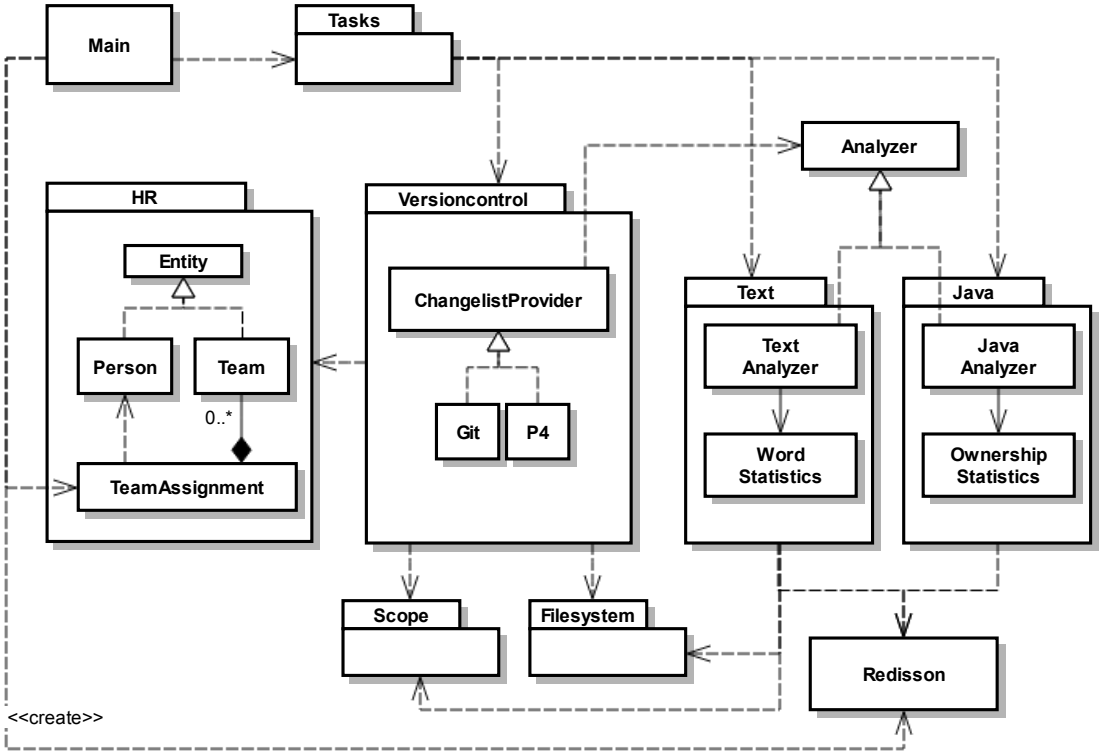
Na nejnižší úrovni poskytují podporu analyzátorům moduly `Filesystem` a `Scope`. První jmenovaný má na starosti přístup k souborům se zdrojovým kódem. Pomocí knihovny *Javaparser* zajišťuje i jejich syntaktickou analýzu. `Scope` nabízí plejádu prostředků potřebných při analýze zdrojových kódu. Klíčovou roli hrají třídy pro identifikaci oblastí kódu (navržená v kapitole 4.4.1) a schopnost průchodu abstraktním syntaktickým stromem. Zde hojně využívám návrhového vzoru *Visitor*.

5.3 Uživatelské rozhraní

Uživatelské rozhraní systému (také *UI* z anglického *user interface*) jsem se rozhodl vytvořit pomocí klasické webové kombinace technologií HTML, CSS, JavaScript a AJAX. Pokud pomineme staré webové prohlížeče, je tato kombinace v současnosti jedním z nejefektivnějších způsobů tvorby UI. Umožnily to nové verze standardů HTML 5, CSS 3 spolu s moderními knihovny pro JavaScript, jakou je například jQuery [1].

Webové technologie samozřejmě ke své činnosti potřebují server. Ten jsem pojal minimalisticky a zabudoval jsem ho přímo do aplikace pomocí knihovny NanoHttpd. Server má na starosti dva úkoly – zpřístupnit zdrojový kód stránky a poskytovat aplikační rozhraní pro asynchronní dotazy. Oba úkoly jsou z pohledu serveru triviální. Kód stránky je zabalen do jediného souboru a server jej pouze načte z disku a odešle. Požadavky směřované na aplikační rozhraní server předává na zpracování nižší vrstvě a výsledek odešle zpět ve formátu

[1] http://jquery.com/



Obrázek 5.1: Diagram tříd – přehled celého systému. [Vlastní]

JSON, což je nativní formát JavaScriptu pro výměnu dat.

Samotnou stránku jsem navrhoval podle zásad tvorby moderního webu s důrazem na přehlednost a uživatelský komfort. Celá webová stránka je načtena pouze jednou (*Single-page application*). Veškeré požadavky na aplikační rozhraní jsou odesílány na pozadí a výsledek je vykreslován dynamicky. Stránka si zachovává dobrou použitelnost při jakékoliv šíři obrazovky a plynule se přizpůsobuje změnám (*Responzivní web design*). Veškeré zobrazované položky je možné použít jako odkaz, což usnadňuje navigaci. Některé méně důležité popisky je často nutné zkrátit, při přejetí myší se však vždy zobrazí celá délka obsahu.

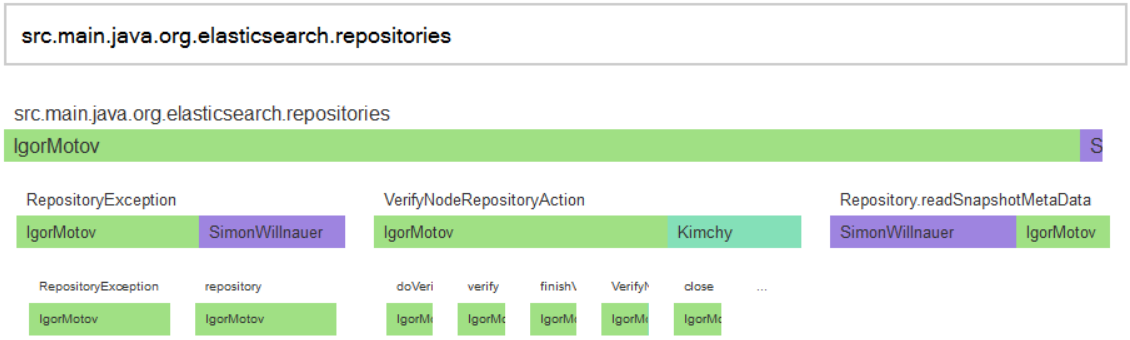
Výsledkem je rychlé, přehledné a informující uživatelské rozhraní. Jeden snímek obrazovky je vložen v textu práce (5.2), další jsou k dispozici v příloze B.

5.4 Systém správy verzí

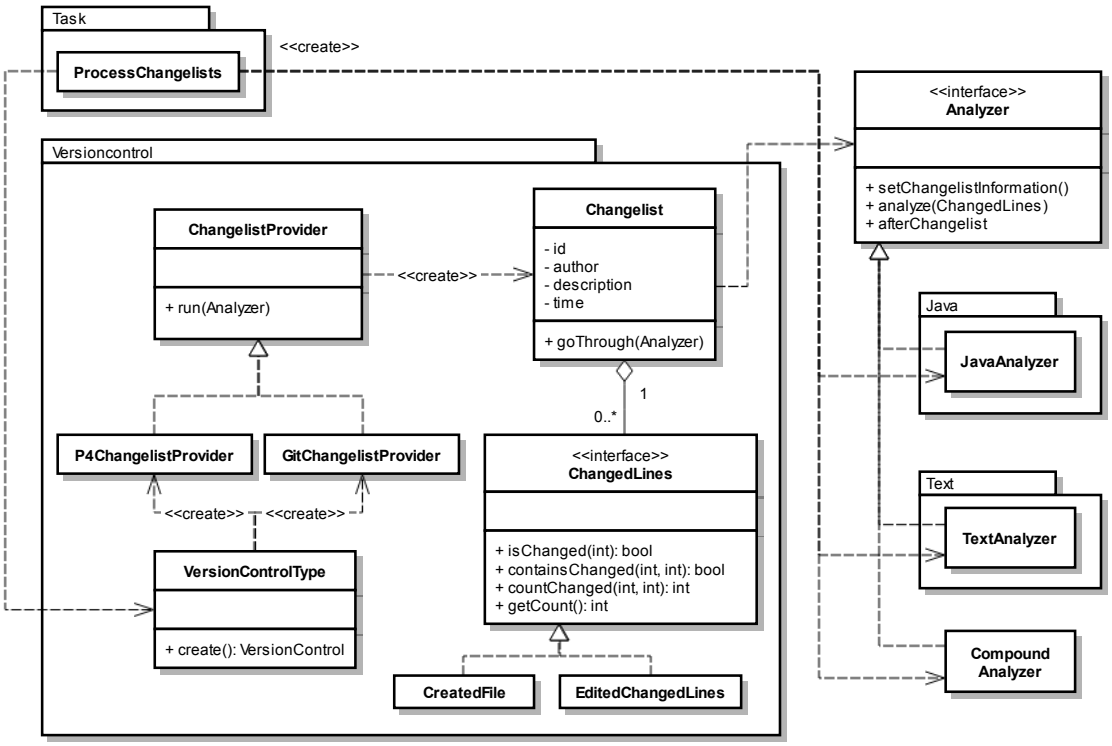
Who-Owns pro přístup do systému správy verzí používá příkazovou řádku, podobně jako by ji mohl použít uživatel. Přesná sekvence příkazů a výstupní formát jsou závislé na konkrétním systému, proto jsem tato specifika zapouzdřil do několika tříd implementujících jednotné rozhraní `ChangelistProvider` (viz diagram tříd 5.3). Dynamické chování systému je možné shrnout takto:

1. Načíst seznam všech změn, které mají být analyzovány.

2. Aktualizovat obsah pracovního adresáře tak, aby odpovídal stavu po provedení změny.



Obrázek 5.2: Uživatelské rozhraní – Oblast kódu s jednoznačným vlastníkem. [Vlastní]



Obrázek 5.3: Diagram tříd – integrace se systémem správy verzí. [Vlastní]

3. Načíst obsah změny – rozdíly oproti předchozí verzi.
4. Projít každý upravený nebo přidaný soubor a analyzovat jej.
5. Pokud je to nutné, počkat na uplynutí zadané prodlevy. Tím je zabráněno přetížení systému správy verzí.
6. Opakovat body 2 až 5 dokud nejsou analyzovány všechny změny.

V současné verzi Who-Owns podporuje Git[2] a Perforce[3]. První z jmenovaných systémů jsem vybral z důvodu jeho velké obliby v komunitě otevřeného software. Systém je navíc decentralizovaný, průchod změnami tak nezatěžuje žádné sdílené zdroje. Přímým protikladem je systém Perforce. Ten je upřednostňován ve velkých organizacích, dokáže pojmout obrovské množství dat (velikost zdrojových kódu může dosahovat desítek GB) a je silně centralizován. Kvůli centralizaci jsem zavedl podporu omezení rychlosti analýzy změn. Tím může administrátor předejít zpomalení serveru Perforce.s

5.5 Databáze

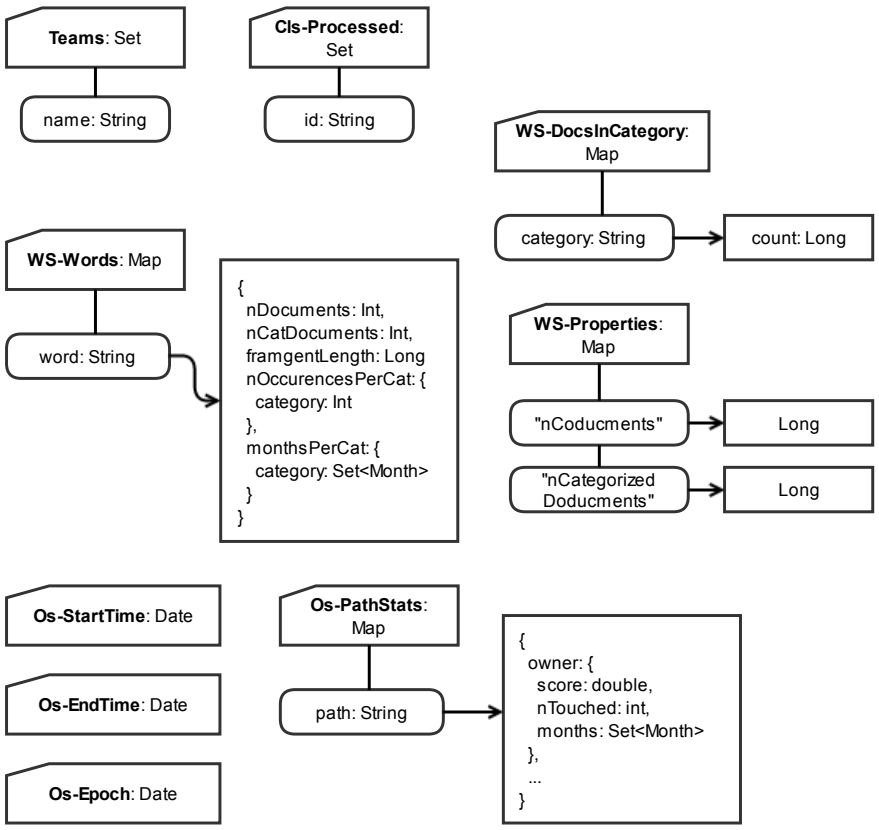
Jak jsem již zmínil v popisu hlavních komponent systému (kapitola 4.3) databáze nemusí plně podporovat vlastnosti *ACID* (atomičnost, konzistence, izolace, stálost). To mi umožňuje výběr z rozmanité rodiny NoSQL databází. Nejdříve však shrnu, jaké požadavky na databázi kladou použité algoritmy:

- Velmi často bude potřeba číst i zapisovat. K oběma akcím bude docházet zhruba stejně často.
- Je nutné indexovat záznamy pomocí řetězce, který bude měněn pouze zřídka.
- Data jsou lokální, potřeba spojovat více záznamů tak není nutná.

Všechny 3 požadavky nejlépe splňují NoSQL databáze, které fungují jako úložiště typu klíč-hodnota (*key-value* store). Jedním z hlavních představitelů v této kategorii je databázový systém Redis[4]. Redis data uchovává v paměti, a je tak extrémně rychlý, škálovatelný, nabízí velmi rozmanitou podporu ukládaných datových struktur a dostupná dokumentace a podpora je na vysoké úrovni. Pro jazyk Java existuje klientská knihovna Redisson[5], která zpřístupňuje datové struktury Redis jako standardní kontejnery. S daty uloženými v databázi je tak možno pracovat téměř jako s nativními.

Strukturu ukládaných dat nemá smysl modelovat klasickým ER diagramem – jak je běžné pro NoSQL databáze, protože data nejsou normalizována a provázanost mezi daty je minimální. Žádný standardní postup pro modelování dat typu klíč-hodnota zatím bohužel neexistuje. Náčrtek 5.4 tedy používá vlastní konvenci. Obdélníky bez rohu představují datové struktury Redis na nejvyšší úrovni. Zaoblené obdélníky jsou klíče a běžnými obdélníky značím hodnoty. Pokud má hodnota nějaké vnitřní schéma, je naznačeno syntaxí vycházející z formátu JSON.

[2] http://git-scm.com/
[3] http://www.perforce.com/
[4] http://redis.io/
[5] https://github.com/mrniko/redisson



Obrázek 5.4: Databázové schéma. [Vlastní]

5.6 Úpravy extrakce klíčových slov

Víceúrovňové hodnocení (viz kapitola 4.5.3) bylo původně stvořeno pro extrakci klíčových slov krátkých dokumentů napsaných běžným jazykem. Já tento postup aplikuji v nové oblasti – zpracovávaným textem jsou komentáře, identifikátory a popisky změn. Povaha textu se velmi odlišuje od běžné řeči. Extrahovaná slova by se také měla vztahovat k vlastnictví, což přináší svá specifika a možné další podklady pro hodnocení slov. Algoritmus popsaný v návrhu jsem tedy upravil, aby v novém prostředí dobře pracoval.

Nejvíce zavádějící se v původním vzorci ukázala složka IFL(s) (převrácená hodnota průměrné délky vět, ve kterých se slovo vyskytuje). Tato složka se zakládá na předpokladu, že popisná slova nesou oproti běžným slovům větší informační hodnotu, a tak nemusí být součástí dlouhých větných celků. Pokud ovšem analyzujeme dvojí typ textu – komentáře a identifikátory, tato funkce nepřiměřeně zvýhodňuje slova, která se vyskytují pouze v identifikátorech. Efekt není příliš velký, proto jej koriguji pouze zmenšením významu celé komponenty.

Protože extrahovaná slova mají vystihovat vlastnictví, je zásadní komponentou ohodnocení BNS(s, k). To vyzdvihuje ta slova, která se vyskytují v jedné kategorii (tj. patřící jednomu vlastníkovi), a která se v jiných kategoriích objevují zřídka. V rovnici tedy na funkci kladu největší důraz.

Podobně jako u vlastnictví kódu využívám i informace o používání slova v čase. Toho jsem docílil obdobným postupem – pokud vlastník použije daný výraz alespoň jedenkrát za

stanovený interval, neobdrží žádnou penalizaci. S klesající pravidelností použití váha slova lineárně klesá. Funkci $\mathrm{M}(s,k)$ definuji jako podíl počtu intervalů, ve kterých bylo slovo s použito v kategorii k, a počtu všech intervalů. Who-Owns používá pro hodnocení slov vzorce 5.1 a 5.2.

$$\mathrm{CAT}(s,k) = \mathrm{BNS}(s,k) \cdot (0.2 + \mathrm{M}(s,k)) \cdot (0.4 + \mathrm{P}(s,k)) \cdot (3 + \mathrm{IFL}(s)) \tag{5.1}$$

$$\mathrm{W_s}(s,k) = \mathrm{IDF_{FW}}(s) + 20 \cdot \mathrm{CAT}(s,k) \tag{5.2}$$

5.6.1 Rozdělení na slova

Zajímavým problémem je též rozdělování řetězce do samostatných slov. V běžném textu je tento problém jednoduchý, ovšem Who-Owns zpracovává i identifikátory. Pro nalezení hranic slov tak používám i podtržítka, pomlčky, číslice, a velká a malá písmena. Identifikátory `dbAccess`, `DbAccess`, `DBAccess` tak jsou rozděleny na dvě slova `db` a `access`. Všechny tyto konvence jsou ustáleny a je možné se na ně spolehnout. Problém tvoří pouze zkratky. Jeden z možných způsobů zápisu je napsat zkratku všemi velkými písmeny, přičemž první písmeno navazujícího slova je také velké (`DBAccess`). Tento způsob je v konfliktu s zápisem `DBaccess`, který se mi ovšem jeví jako méně častý.

Možnost zajímavým způsobem rozšířit schopnosti programu je seskupování po sobě jdoucích slov. Analýze bychom tak mohli podrobit i krátké fráze. Začátek této věty by byl seskupen do frází `začátek, začátekTéto, této, tétoVěty, věty, větyBy`... V tomto příkladu žádné zajímavé fráze nevznikly. Pokud se ale podíváme na celý odstavec, je po sobě jdoucích klíčových slov celá řada – například `schopnostiProgramu` nebo `krátkéFráze`.

5.7 Metodika

Při implementaci Who-Owns jsem se řídil Extrémním Programováním [2]. Přesněji řečeno těmi částmi, které je možné aplikovat na samostatnou práci. Je až překvapivé, že většinu praktik je možné přizpůsobit a vykonávat je samostatně.

Plánování v čtvrtletním a týdenním cyklu se ukázalo být velmi prospěšné. Je důležité vždy vědět, čeho chci v daný okamžik dosáhnout. To mi umožnilo poznat situace, kdy jsem sice pracoval na programu, ale má činnost se neubírala potřebným směrem. V takovém případě jsem si stanovil časový limit, do kterého jsem musel aktivitu dokončit a začít se věnovat něčemu přínosnějšímu. S plánováním také souvisí stanovení uživatelských příběhů, které jsem si určil hned na počátku.

Oproti Extrémnímu Programování jsem neměl k dispozici zákazníka. V krajním případě by se dalo říci, že má vlastní vize byla mým zákazníkem. V průběhu jsem například se svojí vizí musel několikrát vyjednávat o rozsahu, což také je praktika Extrémního Programování. Toto tvrzení není myšleno zcela doslovně, pouze chci naznačit, že řízení rozsahu bylo velmi přínosné. Dospěl jsem tak ke dvěma rozhodnutím – nepokračovat v implementaci emergentních logických celků (kapitola 4.4.1) a prozatím nezkoušet extrakci klíčových frází (nápad v závěru kapitoly 5.6.1). Místo toho jsem upřednostnil zachování vysoké kvality.

Párové programování a průběžnou integraci jsem neměl jak realizovat. Náhradou jsem se pravidelně vracel ke starému kódu a snažil jsem se na něj dívat z nadhledu. Kladl jsem si otázky typu „Proč je to právě takto?“ a „Je možné to udělat jednodušeji?“ Místo průběžné integrace jsem pravidelně spouštěl automatické testy a udržoval jsem program vždy funkční.

Také jsem dodržoval principy programování řízeného testy. Před implementací funkcionality jsem tak nejdříve vytvořil automatický test. To přineslo vysokou kvalitu kódu, spolehlivost a umožnilo včasně odhalit chyby. Výsledné pokrytí kódu testy dosahuje zhruba 60% (63% tříd, 61% metod, 60% řádků). Statistiku kazí především modul `Tasks`, jehož obsah je sice poměrně obsáhlý, ale není komplikovaný. V jeho případě jsem udělal výjimku a automatické testy pro něj neexistují vůbec.

Pro tvorbu testů jsem používal prostředí TestNG, které vychází z populárního JUnit[6]. Oproti němu přináší velmi dobrou podporu parametrických testů. Zvláště při testování funkcí zajišťujících výpočty se tento typ testů velmi hodí.

[6] http://junit.org/

Kapitola 6

Testování a zhodnocení

V další části mé diplomové práce se zabývám nasazením Who-Owns do praxe. Nástroj jsem vyzkoušel na dvou projektech otevřeného software a v omezené míře také ve Firmě. Nejdříve popisuji, jak jsem v těchto projektech provedl průzkum vlastnictví kódu (6.1). Dále hodnotím přesnost výsledků (6.2, 6.3). V závěru kapitoly rozebírám aktuální stav nasazení Who-Owns ve Firmě (6.4), zkoumám možnosti budoucího nasazení a odhaduji možný přínos (6.5).

6.1 Průzkum vlastníků kódu v ElasticSearch a Guava

S využitím Who-Owns jsem pro dva projekty otevřeného software provedl průzkum s cílem určit vlastníky kódu programových jednotek. Vybral jsem organizaci ElasticSearch[1] a projekt od vývojářů Google s názvem Guava - [2]. Oba jsem vybral, protože jsou rozsáhlé co do počtu zdrojových souborů (653 000 a 415 000 řádků kódu) i změn v systému správy verzí (cca. 12 000 a 2 800). Na vývoji také spolupracuje relativně větší počet vývojářů (8 správců a 75 přispěvatelů; 9 správců a 38 přispěvatelů). Počet velikost produktu i počet vývojářů se nedá srovnávat s Firmou, ale v porovnání s ostatním otevřeným software se nejedná o malé projekty.

ElasticSearch je větší než Guava, většinu pozornosti tedy věnuji jemu. Informace o ElasticSearch jsem získal z veřejně dostupných zdrojů na internetu a z komunikace s technickým vedoucím Simonem Willnauerem (viz příloha E). Projekt byl spuštěn v únoru 2010 pod stejnojmennou začínající firmou. Zajímavostí je přístup firmy, která již od začátku zveřejnila zdrojové kódy svého produktu. V současné době se na vývoji aktivně podílí zhruba 25 vývojářů, z nichž 8 můžeme zařadit do kategorie správců, ostatní jsou přispěvatelé. Vývojáři jsou rozřazeni do týmů, ty jsou ovšem organizovány napříč funkcionalitám a každé 3 měsíce jsou střídáni. Střídání členů v týmu má nejspíš zajistit lepší výměnu informací, což je typické pro společné vlastnictví kódu. Tuto domněnku podporuje i Simonova odpověď na otázku, ve které se ho táži na oblasti projektu, kde se považuje za jediného experta:

> „I can't think of any, there is always a second person...“
>
> Žádná mne nenapadá, vždy je tu někdo druhý...

[1] https://github.com/elastic/elasticsearch/
[2] https://github.com/google/guava

Z těchto informací jsem vyvodil několik důsledků na automatické stanovení vlastníků mým nástrojem. Není možné sestavit žádné týmy (každý programátor bude mít svůj vlastní tým), nabízí se pouze možnost vyčlenit nejméně aktivní přispěvatele. Na výsledky analýzy není možné pohlížet jako na *vlastněné* oblasti, nýbrž oblasti *expertýzy* autorů.

6.1.1 Rychlost analýzy

Oba projekty používají pro správu verzí Git. Při svém běhu tak Who-Owns četl historii z lokální kopie repozitáře. Časová náročnost analýzy je lineární v závislosti na počtu a objemu analyzovaných změn. Paměťová náročnost je konstantní pro analýzu kódu a lineární pro klíčová slova. Důvodem je ukládání lokální kopie seznamu všech nalezených slov, které přináší jisté zrychlení. To by samozřejmě nemuselo být nutné.

Systém zvládl zpracovat zhruba jednu změnu za sekundu. Většinu z tohoto času však zabírá práce systému správy verzí, konkrétně aktualizace pracovního adresáře. Nároky na paměť jak pro ElasticSearch tak Guava se držely mírně pod 1 GB.

Pokud jsou zdrojové kódy pod správou Perforce, je největším zdržením právě centrální server. Při svém testování ve Firmě jsem v dobách špičky vytížení serveru omezil rychlost zpracování změn na 3 za minutu. Tato hodnota je velmi konzervativní. Nejvyšší možný dosažitelný výkon je podobný jako při použití Git, tedy zhruba jedna změna za sekundu.

Pro reálné nasazení jsou tyto hodnoty dostatečně dobré. Zvlášť pokud vezmeme do úvahy, že historické záznamy bude nutné analyzovat pouze jedenkrát. Pak bude postačovat jednou za čas spustit systém pro nově vzniklé změny.

6.1.2 Ukázka výsledků

V obou otevřených projektech jsem Who-Owns spustil v módu analýzy změn nad krátkými úseky i celou historií zdrojových kódů. Zvyklosti v ElasticSearch odpovídají společnému vlastnictví kódu, očekával jsem tedy vlastnickou strukturu velmi roztříštěnou. V některých oblastech je tomu skutečně tak a jednoznačný vlastník neexistuje. Například moduly `search.highlight` a `search.sort` jsou téměř stejným dílem tvořeny třemi až čtyřmi vývojáři, přičemž další přispívají. Jednotlivé třídy již jednoznačné vlastníky v drtivé většině případů mají. Naproti tomu existují i celkem velké moduly, kde je vlastník dobře patrný. Příkladem je i modul `common.inject`, který je zcela jasně vlastněn jediným autorem. Přitom obsahuje 180 souborů obsahující 17 000 řádků kódu. Ostatní úpravy v tomto modulu pocházejí pouze od přispěvatelů. Je tedy zřejmé, že vlastnictví tohoto modulu je respektováno.

Algoritmus zobrazující výjimky z vlastnické struktury je dobře patrný v modulu `common.text`. Modul obsahuje 5 tříd, z nichž 4 jsou vlastněny Shayem Banonem (s přihlašovacím jménem Kimchy). Každá třída však obsahuje metodu `compareTo`, kterou vytvořil Martijn Groningen. V tomto konkrétním případě se jedná o velmi specifickou metodu, která nikdy nebyla měněna. Chtěl jsem pouze ukázat, že zobrazení výjimek může být užitečné. Uvažujme kupříkladu situaci, kdy by se v této metodě vyskytla chyba a byla by hledána osoba nejvhodnější pro její opravu. V takovém případě by bylo by velmi užitečné vidět, že Martijn Groningen je sice jejím jediným autorem, o větším kontextu má ale lepší znalost Shay Banon.

Přesné oblasti kódu, klíčová slova ani nalezené vlastníky ve Firmě nemohu z pochopitelných důvodů zveřejnit. V ostatních kapitolách ale výsledky vyhodnocuji.

6.2 Klíčová slova v porovnání oblastí kódu

Zajímavé je také porovnání stanovených vlastníků pro oblasti kódu a identifikovaná klíčová slova. Postup jejich stanovení je diametrálně odlišný a zakládá se na jiných datech. Případné zjištění, že klíčová slova a vlastněné oblasti si odpovídají, vypovídá o dobré kvalitě obou metodik. Srovnání jsem provedl na datech získaných z projektů *ElasticSearch* a *Guava*.

Pro porovnání jsem si stanovil tento postup:

1. Vybrat takové oblasti kódu, které mají stejného jednoznačného vlastníka ve všech svých výskytech se stejným jménem.
2. Pro všechny takové oblasti určit vlastníky slov odpovídající názvům.
3. Pokud jednoznačný vlastník daného slova neexistuje, slovo přeskočit.
4. Porovnat vlastníky kódu a slova.

Pojem	Vlastník identifikovaný v kódu	Vlastník slova
alphabet	cgruber	yetanothernerd
converter	cgdecker	cgdecker
fire	lukes	lukes
invokable	cpovirk	cpovirk
sink	fry	cpovirk
binding	Kimchy	Kimchy
estimator	other	AdrienGrand
mappers	Kimchy	Kimchy
mustache	other	LucaCavanna
o	AdrienGrand	Kimchy
releasables	AdrienGrand	AdrienGrand
repository	IgorMotov	IgorMotov
section	LucaCavanna	LucaCavanna

Tabulka 6.1: Porovnání nalezených vlastníků kódu a klíčových slov.

V tabulce uvádím všechna nalezená slov a stanovené vlastníky v projektu Guava. Pro ElasticSearch uvádím pouze prvních několik příkladů. Drtivá většina oblastí kódu byla přeskočena, protože identifikátory jsou převážně tvořeny více než jedním klíčovým slovem. Pro tuto ukázku to není závadou. V projektu Guava bylo takto porovnáno 6 slov, z toho ve 4 případech došlo ke shodě. Příčinu neshod vysvětluji níže. V projektu ElasticSearch se 49 slov shodovalo a 24 neshodovalo.

`Alphabet` je obecné slovo, které se v identifikátorech téměř nepoužívá. Existují pouze dvě třídy s tímto názvem. Slovo je ovšem používáno napříč celým projektem – vyskytuje se v 11 souborech. V tomto případě tedy klíčové slovo s názvem třídy nesouvisí.

`Sink` je názvem rozhraní, z něhož dědí desítka tříd s názvy jako `CharSink` a `PrimitiveSink`. Můj jednoduchý postup tedy tyto třídy přeskočil, protože neexistuje klíčové slovo `charsink` nýbrž dvě – `char` a `sink`. Zjistil jsem tedy, kdo je vlastníkem těchto tříd. Vlastníků je více, ovšem nejvýrazněji působí programátor s přihlašovacím jménem cpovirk, který byl správně označen za vlastníka klíčového slova. Obdobná situace nastala i u slova `Estimator`.

Zajímavostí je slovo `o`, u kterých se vlastníci neshodují. Vybrání písmene jako klíčového slova je bezpředmětné. V anglickém jazyce ani neexistuje takové slovo, proto je jeho výskyt v podstatě náhodný. V projektu ElasticSearch se ale vyskytuje třída s tímto názvem. Není tak s podivem, že v tomto případě nedošlo ke shodě.

Nedostatek odhaluje neshoda u slova `mustache`. Je to název knihovny, kterou ElasticSearch používá. Zdrojové kódy této knihovny jsou běžnou součástí systému správy verzí a Who-owns tak nijak nedokázal rozlišit, že se nejedná o původní autorskou změnu. Za vlastníka kódu tak byl označen člověk, který knihovnu do projektu zanesl. Za vlastníka slova byl označen ten programátor, který ji nejvíce používal.

6.2.1 Vyhodnocení

V některých případech se ukázalo, že algoritmus extrakce klíčových slov příliš nepočítá s porovnáním vlastníků. Přiřazené váhy jsou totiž přesné, pokud je porovnáváme v rámci jedné kategorie. Pokud ovšem porovnáváme váhy jednoho slova v různých kategoriích vstupují do hry rozdíly mezi kategoriemi. Například pokud některý autor má výrazně méně změn a tím je i délka analyzovaného textu kratší, je pravděpodobnost výskytu daného klíčového slova mnohem vyšší. Velmi zhruba tento efekt normalizuji vynásobení váhy slova logaritmem z počtu dokumentů v dané kategorii.

Pro praktické využití tento problém není přílišnou překážkou. Například ve Firmě, kde jsou programátoři zaměstnáni na plný úvazek je velikost kategorií řádově stejná. Také je mnohem důležitější schopnost porovnat váhy slov v rámci jedné kategorie, která je přesná.

Celkově tento experiment považuji za úspěšný. Ve většině případů si vlastníci klíčového slova a oblasti kódu odpovídají. To vypovídá o dobré kvalitě obou přístupů.

6.3 Guava, ElasticSearch – Dotazník

Nejlépe mohou kvalitu výsledků posoudit ti, jichž se přímo týká – tedy vývojáři. Rozeslal jsem tedy dotazníky devíti členům ElasticSearch a pěti členům Guava. Dotazník se skládal z části, kde se táži na projekt jako celek, na velikost přínosu a specializaci respondenta a jeho přiřazení k případnému týmu. Dále v dotazníku žádám o ohodnocení klíčových slov ze dvou hledisek – do jaké míry se vztahují k aktivitě respondenta v projektu, a jak dobře odlišují respondenta od ostatních programátorů. Pokud v projektu existuje vlastnictví na úrovni jedince, mělo by klíčové slovo vypovídající o vlastnictví dosáhnout dobré skóre na obou stupnicích.

Poslední otázka obsahuje krátký seznam oblastí kódu. Respondent přiřazuje hodnocení oblastem podle vnímané míry odpovědnosti. Zde předpokládám, že mezi vnímanou odpovědností a vlastnictvím existuje jasná korelace. Podobně jako u vlastnictví vnímaná odpovědnost se zmenšuje s počtem jedinců, kteří ji nesou. Jediný vlastník nese plnou odpovědnost. V modelu společného vlastnictví kódu je přesnější tázat se na odpovědnost. To totiž evokuje krizovou situaci, kdy řešení úkolu (v ideálním případě) bude mít na starosti osoba s nejlepšími znalostmi o dané problematice.

6.3.1 Vyhodnocení

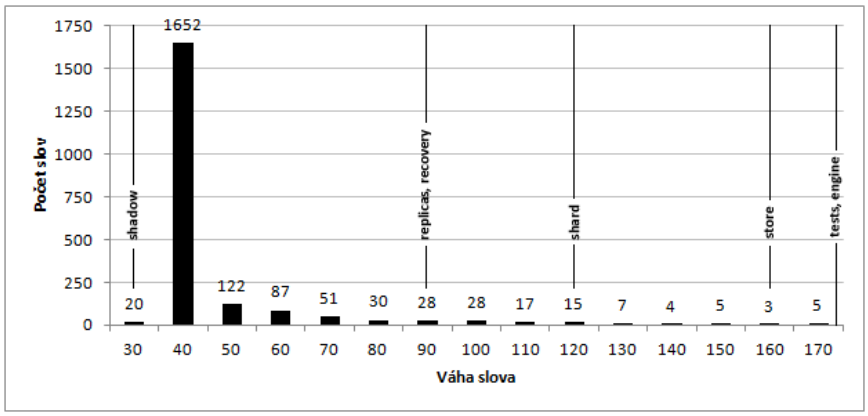
Respondentem dotazníku byl Simon Willnauer, technický vedoucí v ElasticSearch (plné znění je v příloze E). Projektu se věnuje na plný úvazek již 2,5 roku a jeho přínos pro kód projektu je velký. Soudím tedy, že má dobrý přehled a důvěryhodnost odpovědí tomu

bude odpovídat. Simon také vyjádřil živý zájem o Who-Owns. V případě zveřejnění pod svobodnou licencí, by schopnosti systému rád využil v ElasticSearch. Jeho odpovědi na následujících řádcích detailně rozebírám.

Simon takto popsal oblasti, na které se nejvíce specializuje:

> „engine, store, shard, recovery, shadow replicas, tests“

Na tento seznam jsem se podíval jako na klíčová slova a zjistil jsem, jestli mezi ostatními slovy mají dobrou váhu. Všechna slova s jejich váhami pro Simona jsem vynesl do histogramu 6.1. Do grafu jsem též zanesl umístění termínů, které sám Simon použil pro popis své činnosti. Všechny až na jeden mají percentil nad 95%. 4 slova dokonce nad 98%. Slovo `shadow` obdrželo výraznou penalizaci a to z toho důvodu, že uživatel se jménem Kimchy jej používal mnohonásobně více než Simon. V ostatních případech je ale shoda velmi dobrá. Zvláště uvážíme-li, že Simon tato slova použil sám od sebe. (Nejdříve jsem omylem zaslal nesprávný seznam extrahovaných klíčových slov, který žádné z výše uvedených neobsahoval.)



Obrázek 6.1: Histogram váhy klíčových slov Simona Willnauera [Vlastní]

V dalším bodě dotazníku respondent hodnotil klíčová slova. Pro otázku jsem vybral 11 termínů s nejvyšší váhou v dané kategorii. Výsledné ohodnocení není excelentní. Střední hodnota dosáhla 2,86 se směrodatnou odchylkou 1,14. V seznamu se objevují i velmi dobrá klíčová slova s hodnocením 1 (`tests`, `engine`) i většina průměrných a jedno velmi špatné s hodnocením 5 (`ex`). Velký počet průměrných slov je dle mého názoru způsoben jejich nejednoznačností. Zatímco statisticky je použití daného slova pro daného autora význačné, jemu samotnému bez dalšího kontextu nedává smysl. Domnívám se, že tento nedostatek by bylo možné překonat extrakcí krátkých frází.

Poslední otázka směřovala na oblasti kódu. Vybral jsem ty celky, pro které byl respondent označen jako vlastník, a z nich jsem vybral 11 s nejvyšším absolutním skóre. Průměrné ohodnocení respondentem je 2,10 se směrodatnou odchylkou 0,94. Jelikož vlastnictví kódu v ElasticSearch je společné, považuji tento výsledek za velmi dobrý.

6.4 Aktuální nasazení ve Firmě

Who-Owns v době dokončení této práce ve Firmě prochází schvalovacím procesem, který kontroluje, zda-li je jeho nasazení bezpečné. Přesto jsem měl možnost krátkého testování. Při něm bylo analyzováno pouze 200 změn vytvořených třemi týmy. Toto množství je příliš

malé na stanovení vlastníků zdrojových kódů. Pro identifikaci klíčových slov tento počet začíná dostačovat. Přesnost je tedy velmi nízká, na hranici statistické relevance.

Přesnost extrahovaných slov jsem vyhodnotil dotazníkem. Ten obsahoval 40 klíčových slov s nejvyšším hodnocením pro všechny tři týmy. Respondenti znají koncept vlastnictví kódu, proto jsem je požádal přímo o přiřazení bodů na stupnici od 1 do 5 podle toho, jak slovo popisuje vlastnictví týmu.

Výsledná střední hodnota ze všech odpovědí činí 3,08 se středním rozptylem 1,46. Seznam slov v sobě zahrnuje i velmi dobrá klíčová slova (7 slov s průměrným hodnocením do 1,6 a velmi malým rozptylem). Obsaženo je i značné množství nejednoznačných nebo dokonce i obecně používaných termínů. Zvláště početnou je kategorie, která sice se zodpovědností týmu souvisí, ovšem tým není jejich přímým vlastníkem. Tato klíčová slova by Who-Owns dokázal správně přiřadit, pokud by mohlo být analyzováno větší množství dat. Přesnost výsledků je tak podle očekávání nízká. Pro stanovení vlastnictví byl analyzován malý počet změn. Přesto ale seznam zahrnuje výrazy, které velmi dobře vlastnictví týmu popisují. Pouze jsou zastíněny značnou mírou šumu. Ten by vymizel s větším množstvím dat.

6.5 Budoucí uplatnění ve Firmě

Ve Firmě jsem také provedl kvalitativní výzkum, ve kterém jsem se zaměřil na nalezení uplatnění pro Who-Owns. Výzkum jsem prováděl prostřednictvím neformálních rozhovorů se zodpovědnými osobami i strukturovaného interview se dvěma zaměstnanci. Transkript rozhovoru obsahuje příloha D. Výzkum jsem rozčlenil do několika navazujících oblastí, přičemž stejnou strukturu zachovávám i v této kapitole. U každé přebrané informace uvádím její původ. Pokud informace pochází z interview, označuji ji číslem odpovědi v kulatých závorkách. Konkrétní odpověď pak lze dohledat v transkriptu. Bez označení ponechávám své vlastní myšlenky a závěry. Těžiště kapitoly spočívá v jejím závěru (6.5.4), kde na základě předchozích informací vymezím oblast budoucího uplatnění mého nástroje.

6.5.1 Náplň práce respondentů

Strukturovaného interview se zúčastnil vývojář, kterého označuji zkratkou V., a tester, jehož jméno zkracuji na A. Pracovní náplň A. lépe vystihuje celý název pozice – *Quality Assurance Engineer* (zkráceně QA, česky inženýr zajištění kvality). Neobsahuje totiž pouze samotné testování, ale celé řízení kvality. To začíná již při přípravě návrhu, pokračuje vytvořením plánů na testování a je završeno samotným testováním (5). Lidé na pozici QA mají také na starosti prvotní vyšetření chyby (5, 50) a, pokud je to nutné, i nalezení zodpovědné osoby (50).

Vývojář ve Firmě se podílí na přípravě návrhu, do kterého vnáší své technické schopnosti, implementuje novou funkcionalitu a opravuje chyby (4). Týmy jsou ve Firmě složeny jak z vývojářů, tak z QA. Obě specializace tak pracují bok po boku.

Oba respondenti uvedli shodný poměr času věnovaný vývoji nových funkcionalit a opravě chyb. A. odhadl, že 60% času věnuje chybám a 40% funkcionalitám (10). V. věnuje stejný čas obojímu (7).

6.5.2 Situace vlastnictví kódu

Tato kapitola přímo navazuje na kapitolu 3.2, která ovšem stejnou problematiku rozebírá z teoretičtějšího hlediska.

Oba respondenti vnímají zdrojové kódy jako velmi velké (14, 20, 24). Většinu kódu vůbec neznají. Navíc tvrdí, že je možné mentálně obsáhnout jen jeho zlomek (16, 17, 18, 20). Kód dělí na příbuzné oblasti, které se týkají příbuzné domény, a na zcela odlišné oblasti (26, 28). A. odhaduje, že v rámci příbuzné oblasti zná zhruba 10% kódu (30).

Současnou situaci vlastnictví kódu ve firmě si je možné představit jako ostrovy. Ostrovy představují třídy s jasným vlastníkem, který je většinou respektován. Mezi ostrovy leží „šedá zóna," kde vlastnictví kódu buď není jasné vůbec nebo není dodržováno (36 - 40). Dokonce i nově vytvořený kód je často modifikován jinými týmy, které tak narušují jeho vlastnictví (36, 40).

Vlastnictví kódu je ve firmě vnímáno jako neformální koncept – je slušné jej respektovat, ale bohužel toto chování není vždy samozřejmé (36, 46, volná diskuze).

6.5.3 Stávající praxe určení vlastníků

Povědomí o vlastnictví převážně vychází ze zkušeností 46. Může být i zdokumentováno (54, kapitola 3.2.1), ale tato informace je často zastaralá, nepřesná nebo málo známá. Při určování vlastníků tak A. často používá funkci systému správy verzí, která ke každému řádků zdrojového kódu přiřadí osobu, která ji naposledy měnila (46). Tento postup je dobře použitelný pouze v případě, kdy QA zjišťuje odpovědnou osobu za chybu a zná přesné problémové místo. V opačném případě je nutné následně dohledat i povahu změny a zjistit, zda-li se nejedná o jednorázový zásah (46).

Dalším používaným způsobem je analýza oblasti kódu krokováním. Z vykonávané činnosti kódu je často možné odvodit si pravděpodobného vlastníka kódu. Tato činnost je velmi časově náročná (46).

V. zmínil, že v některých případech je možné určit vlastníka na základě zvyklostí pojmenování oblastí v kódu. Jako příklad uvedl dvě po sobě následující klíčová slova (62). Dohledání vlastníka je okamžité v případě, pokud se jedná o známou oblast kódu (46, 68).

Potřeba zjistit vlastníka kódu vyvstává v běžné pracovní činnosti poměrně často. V. i A. věnují této činnosti 2 až 4 hodiny týdně (70, 73). Velkou zátěží pro A. je hledání správného týmu pro chyby, které nespadají pod zodpovědnost jeho týmu. Jako zdroj problému identifikuje technickou podporu, která by tuto činnost měla mít na starosti. Ta ale nemá pro její efektivní vykonání dostatek informací, a tudíž se často mýlí. (50, 54, 70)

V. zmiňuje dva nejčastější případy, kdy vývojáři potřebují určit vlastníka. První z nich je při procesu hodnocení změny. Zodpovědností jejího autora je nechat si úpravu schválit vlastníkem kódu. Druhý vyvstává při specifikaci a návrhu. Funkcionality se totiž často překrývají s oblastmi jiných týmů a je zapotřebí vývoj koordinovat. Druhý případ identifikuje i A. (56, 78)

6.5.4 Uplatnění Who-Owns

Who-Owns by se mohl ve Firmě uplatnit hned v několika oblastech. Jistě by ji mohli využívat vývojáři i QA pro své vlastní použití. Who-Owns by tak mohl bez zásadních změn výrazně urychlit všechny případy použití popsané výše, které zabírají až 4 hodiny týdně. Z počátku by dostačovalo mnou navržené uživatelské rozhraní. Přístupnost by se zlepšila zabudováním do integrovaného vývojového prostředí.

Ještě lepšího zefektivnění by mohlo být dosaženo integrací Who-Owns do procesů a nástrojů Firmy. Prvním oblastí by mohl být nástroj používaný v procesu hodnocení změn. Při něm by k hodnocení mohli být automaticky přizváni i vlastníci daného kódu (75).

Velkou výzvou s ještě větším potenciálem by bylo uplatnění Who-Owns v oddělení technické podpory. Who-Owns by na základě klíčových slov obsažených v popisu chyby nabízel zodpovědný tým (76). Pracovník technické podpory by mohl nabízený tým přijmout nebo stanovit tým dle svého uvážení. Popisy chyb navíc někdy obsahují i výpis ze zásobníku volání (*stack trace*). V takovém případě by bylo vedle klíčových slov možné přímo použít vlastnictví kódu (90).

Manažer ve Firmě B., přišel i na další možnost uplatnění. Ve Firmě existuje databáze chyb, do které se některé typy chyb ukládají. Každý záznam obsahuje i výpis zásobníku volání funkcí. Who-Owns by výpis mohl analyzovat a vlastníkovi zaslat e-mailové upozornění. Tento nápad podpořil i V. (92).

6.5.5 Odhad dopadu

Zvláště přesnější určování týmů již na úrovni technické podpory by mělo velmi pozitivní efekt. Méně by docházelo k přeposílání chyb mezi týmy i vracení chyby zpět na technickou podporu (50, 70, 76). Tím by došlo k úspoře času zaměstnanců firmy i k urychlení opravy chyby, což by mělo přímý pozitivní vliv na vnímanou kvalitu služeb zákazníky.

I efektivita při opravě chyb by byla vyšší. Jak popisuje V.:

> „Kolikrát se mi vyplatí si ten bug fixnout, než abych dohledával, čí to je.“ (73)

Pokud by na opravu chyby byla rovnou přiřazena nejvhodnější osoba, byla by oprava efektivnější, protože by ji prováděl člověk s nejlepší znalostí problematiky.

Další efekt by nebyl tak přímý a měl by spíše dlouhodobější účinek. Domnívám se totiž, že nasazení Who-Owns by mělo příznivý vliv na mezi-týmovou kooperaci a celkově by vedlo ke zvýšení kvality kódu. K tomuto tvrzení mne přivedla následující úvaha.

Možnost jednoduše stanovit vlastníka kódu a automatické přizvání vlastníka k hodnocení změny by vedlo k postupnému vytříbení „šedé zóny“ kódu (58). To by mělo za následek zmenšení obav provádět při provádění refaktoring a drobné opravy kódu, protože se zmenší pravděpodobnost konfliktních změn. Také by se snížilo riziko cizích zásahů, které porušují čistotu nebo účel kódu (40, 56). Díky automatickému nabízení vlastníků při hodnocení kódu budou vývojáři podporováni v navazování kontaktů i se zcela neznámými lidmi. To přispěje k lepšímu přenosu znalostí ve Firmě a ke zkvalitnění komunikace.

V krajním případě se dá stanovení vlastníků použít i pro donucení týmu přebrat odpovědnost za nepopulární oblast kódu. Tím není myšlen postih, ale spíše zabránění snahy vyhnout se opravě složitých chyb. (81, 82, 83)

Kapitola 7

Závěr

V mé diplomové práci jsem se zabýval otázkou softwarového vlastnictví ve velkých softwarových organizacích. Vycházel jsem ze širšího kontextu managementu znalostí a metodiky vývoje softwaru – Extrémního Programování. Identifikoval jsem tři základní modely vlastnictví software a analýzou praxe ve Firmě vytvořil model čtvrtý. Tento model zohledňuje rozdílné zvyklosti k vlastnictví kódu uvnitř týmu a mezi týmy. Dále jsem zjišťoval, co může být předmětem vlastnictví, a jaké informace je možné použít pro automatické určení vlastnictví. Nabyté zkušenosti jsem použil pro tvorbu nástroje s názvem Who-Owns, který je schopen provést průzkum vlastnictví kódu. Nástroj pracuje na dvou rovinách, kromě kódu stanovuje i typická klíčová slova. Ta dále vypovídají o zodpovědnostech vlastníka v projektu. V další části jsem nástroj použil k provedení průzkumu vlastnictví ve dvou projektech otevřeného software a v omezené míře i ve Firmě. Na závěr práce jsem analyzoval přesnost výsledků a diskutoval možnosti použití a přínos pro Firmu.

Za můj největší přínos považuji zejména princip fungování nástroje Who-Owns, který je z poloviny můj vlastní. Druhá polovina (extrakce klíčových slov) aplikuje známé algoritmy, ale ve zcela nové oblasti. Ty jsem navíc značně vylepšil a obohatil jsem je o další vstupy informací. Také jsem vytvořil nový model vlastnictví, kterým je možné popsat situaci ve velké softwarové organizaci.

Dalším mým přínosem je samotná implementace nástroje a jeho nasazení a důsledné otestování v praxi. Ve Firmě jsem identifikoval problémová místa, která lze pomocí Who-Owns účinně řešit. Pokusil jsem se odhadnout i dopad na Firmu a její zákazníky, ten ovšem za současného stavu nelze kvantifikovat. Jak Firma, tak i otevřený software ElasticSearch projevili o můj nástroj živý zájem. Firma chce nástroj integrovat do svých procesů a dát jej k dispozici programátorům a testerům. V současné době prochází Who-Owns bezpečnostním schvalovacím procesem, po kterém bude toto plné nasazení možné.

Oproti existujícím podobným přístupům, které pracují s podobnými daty, se má práce značně odlišuje. Například Xiaozhu Meng a kolektiv ve své práci [12] pouze přesněji určují autora řádku kódu. Já jdu o několik kroků dále – pracuji s vlastnictvím kódu, přičemž vlastníkem může být tým. Respektuji strukturu zdrojových kódu, kterou používám pro určení závažnosti změny, a pro sjednocení přesunů kódu. Dále identifikuji vlastníky větších celků, nikoliv řádků kódu. Zcela navíc je určení vlastnictví s pomocí extrakce klíčových slov.

V oblasti managementu znalostí [10] existují příbuzné systémy. Ty dokáží vyhledat experta dané oblasti, k čemuž využívají analýzu dokumentů. Systémy lze použít například ve farmaceutickém průmyslu pro zkvalitnění komunikace mezi výzkumnými pracovníky. Tyto systémy jsou obecnější – je možné je aplikovat ve více oborech, nejen farmacii. Nelze je ale použít při vývoji software. Ten se od ostatních technických oborů příliš odlišuje. Po-

kud pomineme ostatní překážky, obecné systémy by často neměly k dispozici ani dostatečné množství dat, neboť analyzují dokumenty, nikoliv zdrojový kód.

Přesnost výsledků jsem ověřoval několika způsoby s rozdílnými výsledky. V některých případech byla kvalita výstupů velmi dobrá a někdy horší. Domnívám se, že horší výsledky byly způsobeny aplikací na ne zcela vhodnou oblast (otevřený software, kde je praktikováno společné vlastnictví) nebo malým počtem dat. Experimenty bych tedy shrnul takto – Who-Owns nedokáže zázraky, pokud je ale k dispozici dostatek dat, a pokud je nasazen správným způsobem, je jeho použitelnost velmi dobrá a přínos velký. Domnívám se, že situace ve Firmě tyto podmínky splňuje, a výsledky plného nasazení nástroje to potvrdí.

Některé uživatelské příběhy z návrhu Who-Owns ve své současné verzi nepodporuje. Převážná většina je ale hotová. Vynechané části zahrnují myšlenku emergentních celků kódu. Myslím si, že další rozvoj tímto směrem by mohl přinést velmi zajímavé výsledky. V průběhu návrhu jsem také identifikoval další zajímavou oblast rozšíření – schopnost extrakce krátkých frází. Úprava programu by byla mohla zlepšit informační hodnotu klíčových slov. Nástroj je možné dále rozvíjet – navazujícím vylepšením by se mohlo stát:

- rozšíření schopností aplikačního rozhraní nástroje. To umožní lepší integraci do okolních systémů.
- schopnost rozhodnout vlastnictví na základě výpisu ze zásobníku volání a z textu obsahujícího klíčová slova.
- možnost dodat nástroji další informace o struktuře zdrojových kódu v projektu. To by mohlo sloužit například pro odlišení zdrojových kódů knihoven (viz kapitola 6.2).

Vytvořený nástroj se mi jeví jako velmi perspektivní a budu se jeho rozvoji nadále věnovat. Prvním krokem bude plné uvedení do provozu ve Firmě a budu pokračovat realizací některých rozšíření z předchozího odstavce.

Literatura

[1] Beck, K.: *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 1999, ISBN 02-016-1641-6.

[2] Beck, K.; Andres, C.: *Extreme Programming Explained: Embrace Change, 2nd Edition (The XP Series)*. Addison-Wesley, 2004, ISBN 03-212-7865-8.

[3] Bellaachia, A.; Aldhelaan, M.: HG-RANK A Hypergraph-based Keyphrase Extraction for Short Documents in Dynamic Genre. *Microposts2014*, ročník 1141, 2014 [cit. 2015-04-10], doi:10.1007/s13748-014-0058-1.
URL http://ceur-ws.org/Vol-1141/paper_06.pdf

[4] Bird, C.; aj.: Don't Touch My Code! Examining the Effects of Ownership on Software Quality. *ESEC/FSE*, 2011 [cit. 2015-05-08]: s. 4 – 14, ISSN 978-1-4503-0443-6.
URL http://research.microsoft.com/pubs/150498/bird2011dtm.pdf

[5] Bird, J.: Code Ownership: Who Should Own the Code? [online]. 2013 [cit. 2015-01-07].
URL http://agile.dzone.com/articles/code-ownership-%E2%80%93-who-should

[6] Cherry, K.: Forgetting: How and Why Memory Fails [online]. 2007 [cit. 2015-01-13].
URL http://psychology.about.com/od/cognitivepsychology/p/forgetting.htm

[7] Ebbinghaus, H.: *Memory: A Contribution to Experimental Psychology*. New York: Teachers College, Columbia University, 1885.
URL http://psychclassics.yorku.ca/Ebbinghaus/index.htm

[8] Fowler, M.: Code Ownership [online]. 2006 [cit. 2015-01-08].
URL http://martinfowler.com/bliki/CodeOwnership.html

[9] Fritz, T.; aj.: Does a programmer's activity indicate knowledge of code? *ESEC/FSE*, 2007: s. 341 – 350, ISSN 978-1-59593-811-4.

[10] Kolektiv autorů: *Knowledge Management: Concepts and Best Practices*. Springer, 2010, ISBN 978-3642055980.

[11] Kolektiv autorů: Eclipse Development Process 2014 [online]. 2014 [cit. 2015-04-24].
URL https://eclipse.org/membership/become_a_member/committer.php

[12] Meng, X.; aj.: Mining Software Repositories for Accurate Authorship. *ICSM*, 2013: s. 250 – 259, ISSN 978-0-7695-4981-1.

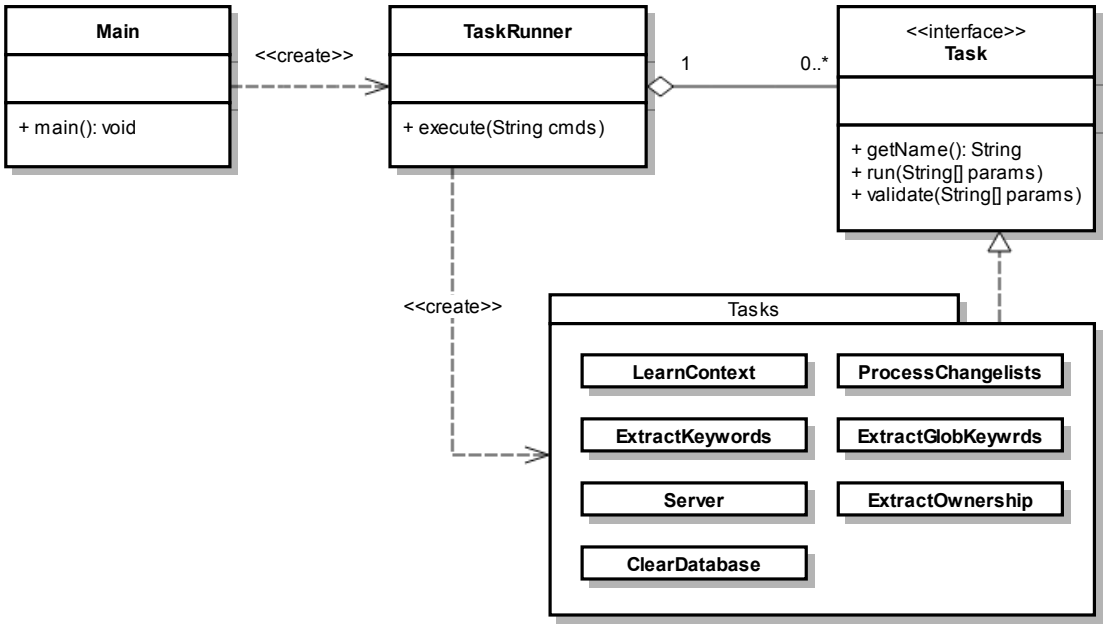
[13] Naveed, N.; aj.: Searching Microblogs: Coping with Sparsity and Document Quality. *CIKM*, 2011 [cit. 2015-04-08]: s. 183 – 188, ISSN 978-1-4503-0717-8.

URL http://userpages.uni-koblenz.de/~kunegis/paper/twitter-sparsity-and-document-quality.pdf

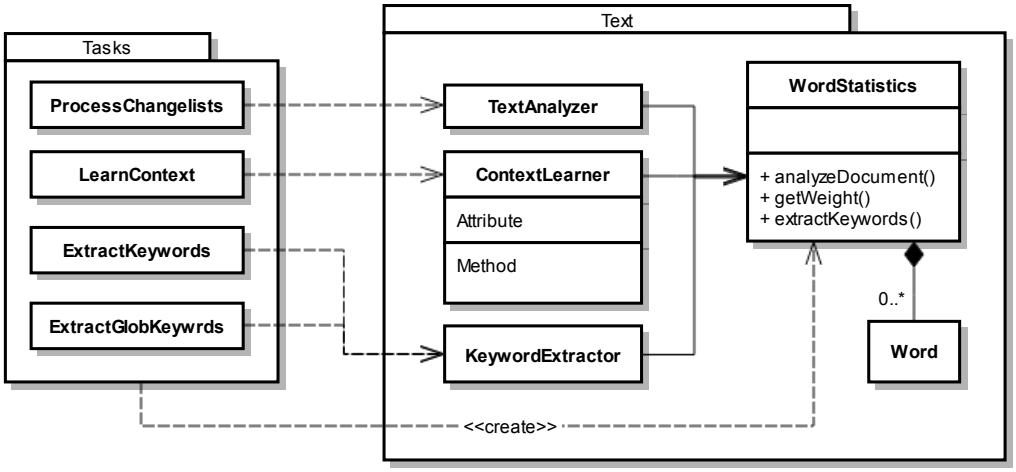
[14] Timonen, M.: *Term Weighting in Short Documents for Document Categorization, Keyword Extraction and Query Expansion*. Dizertační práce, University of Helsinki, 2013 [cit. 2015-05-18].
URL https://helda.helsinki.fi/bitstream/handle/10138/37924/timonen_dissertation.pdf

Příloha A

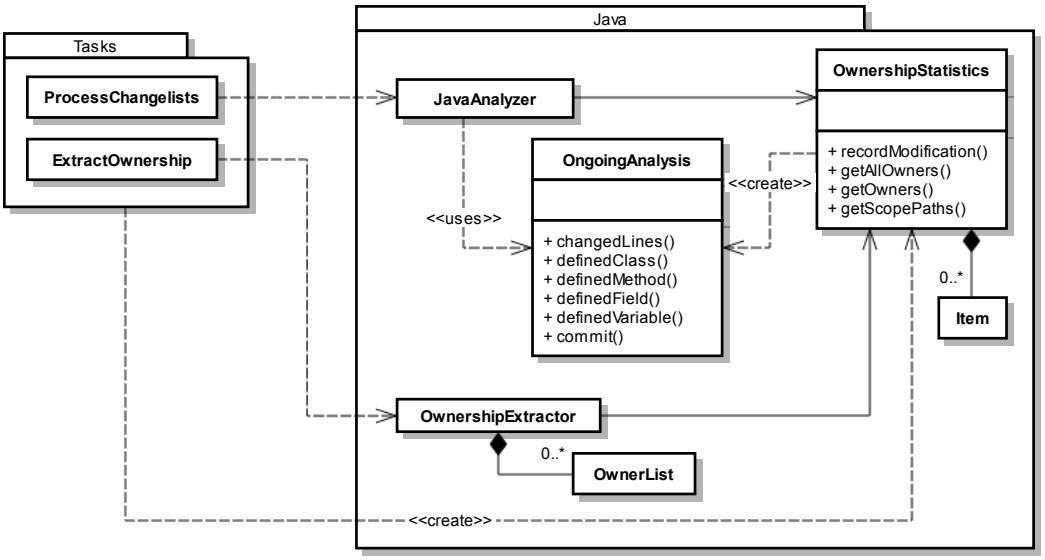
Ostatní diagramy



Obrázek A.1: Diagram tříd – úlohy programu. [Vlastní]



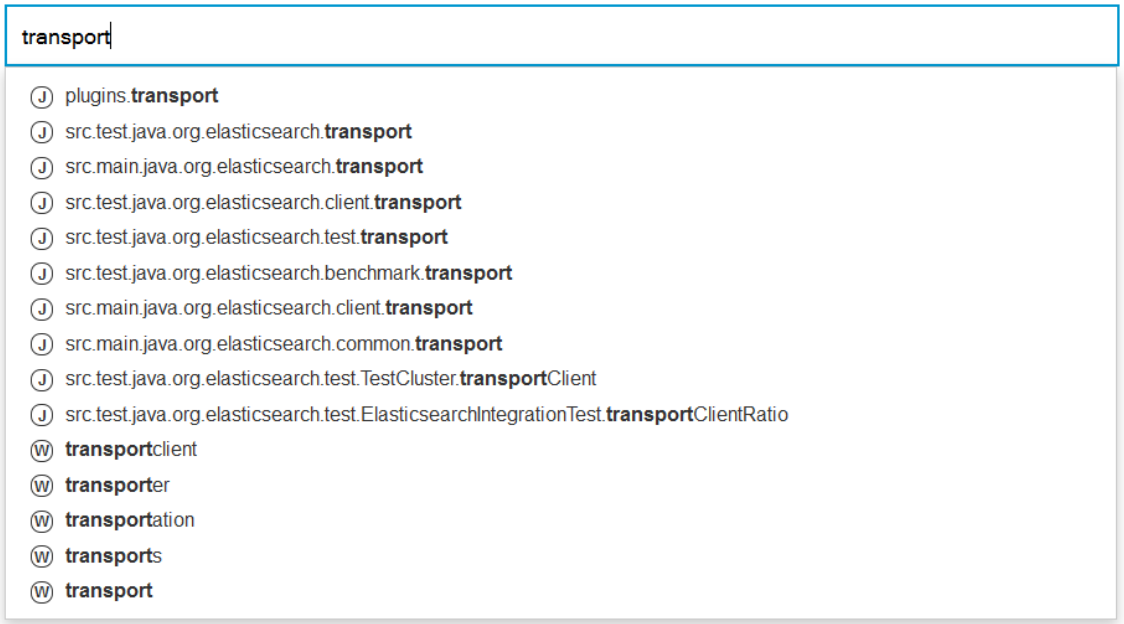
Obrázek A.2: Diagram tříd – analýza textu. [Vlastní]



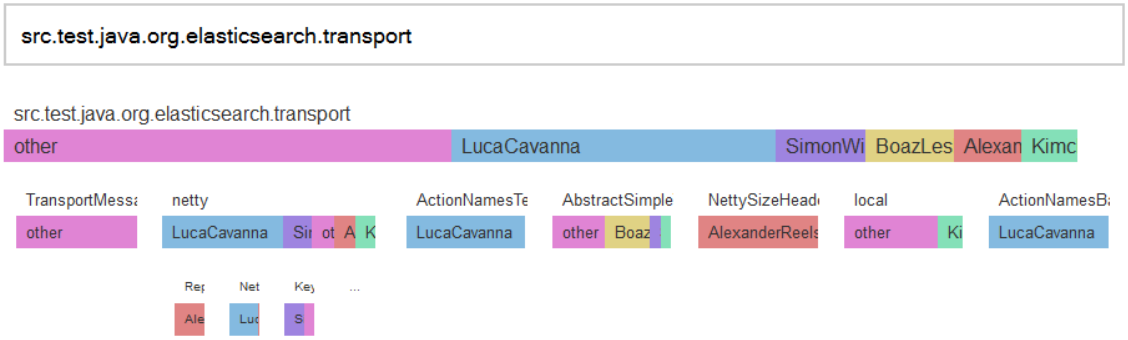
Obrázek A.3: Diagram tříd – analýza zdrojových kódů. [Vlastní]

Příloha B

Snímky obrazovky



Obrázek B.1: Uživatelské rozhraní – Automatické dokončení dotazu [Vlastní]



Obrázek B.2: Uživatelské rozhraní – Oblast kódu kde vlastník jednoznačný není [Vlastní]

Obrázek B.3: Uživatelské rozhraní – Odpověď na dotaz „Co vlastní uživatel rmuir?“ [Vlastní]

Příloha C

Manuál

Who-owns extracts code ownership and authorship information from a version control system. Currently *git* and *p4* are supported.

Installation

Make sure you have your version control environment ready. Who-owns uses command line programs `p4` or `git` to access the changelists. Install Redis and configure it. These settings are recommended: `save 300 1`, `save 120 10000`, `appendonly no` Build Who-owns (Ant build file is included)

Usage

First configure how people are assigned to teams. For the most accurate results also reflect the history of people moving around. Write the information into a text file, with following syntax. Use any string for team names. For people's names use the same user name as in version control system.

```
# Who-owns uses it's own syntax. It is really simple, line based - one entry per line.
# Lines starting with a hash are comments.
# Lines with only whitespace are ignored.
    # Whitespace at beginning and end of the line is ignored.

# Create a team by using a keyword ‘team‘ followed by a space and team's name:
team Lorem

# Specify a team member by writing his name, followed by a starting date, a keyword ‘to‘ and ending
# date:

ipsum.dolor 1942-11-27 to 1970-09-18

# You can skip ending date which means he is still assigned to the team.
# Be sure to leave the word ‘to‘ in place.
sit.amet 1945-11-20 to

# You can also skip the starting date which means he was part of the team since beginning of time,
# or skip both days meaning he was always part of the team:
consectetur

# Current limitation is that a person can be assigned at most to a single team in any given time.
team afterlife
adipiscing.elit 1970-09-20 to

# If a person is not assigned to any team (at all or at some point in time) Who-owns assumes they
# are a team on his own. Or you can specify a default team:
```

```
default team Pellentesque

# You can ignore all changes done by people assigned to as special team:
ignore changes of team Lacus
```

Then create your configuration file, it has similar syntax as team configuration. Who-owns will execute the commands written in this file once it is executed.

```
# There are several available commands:
# clear database
# learn context             (root)
# process changes           (‘p4‘|‘git‘, root directory, branch name, start date, end date)
# extract keywords          (maximal count of keywords)
# extract global keywords   (maximal count of keywords per team)
# extract ownership         (‘team‘|‘author‘, threshold between 1 and 100, recommended = 20)

# Start a command with keyword ‘task‘. This command clears the Redis database.
task clear database
    # this command has no parameters

task learn context
    # One line per parameter.
    path-to-project-dir

task extract global keywords
    # Global keywords are not very useful. They are calculated just from IDF-FW value.
    # However you don’t have to process any CLs to get them.
    100

# The main task - it goes though the CLs and analyzes them.
task process changes
    # Specifies type of version control system
    git
    path-to-project-dir
    # The branch name (for p4 \\\\depot\\\\branchName...)
    master
    # Process CLS between the following start date:
    2014-05-01
    # ... and the following end date:
    2015-05-10

# Extracts keywords most suitable for the team.
task extract keywords
    20

# Extracts ownership / authorship and groups it based on modules, classes, methods,
# blocks... This sequence uniquely identifies any piece of code. We call it ‘ScopePath‘.
task extract ownership
    # ‘author‘ or ‘team‘ to extract individual authorship or team ownership.
   author
   # Threshold - an entity has to be 20\% better than the next in order to be an owner.
   20

# Starts the web server providing UI
task server
    # Port
    80
    # Path to root html file.
    web/inlined.html
```

After you have created the configuration file, start Redis and run Who-owns: `java Main taskfile teamfile localhost:6379`. The last argument is the address and port of Redis.

Who-owns continuously stores the extracted information so you can kill it at any time and it resume where it ended after you start it again. The processing speed is not exactly mind blowing. On medium size test data using git, it analyses about 1 CL per second.

Příloha D

Transkript

V této příloze uvádím doslovný přepis interview se dvěma zaměstnanci *Firmy* – A. a V. Plná jména z důvodu anonymity nemohu uvést. Rozhovoru jsem se účastnil já (Jan Šimonek, JŠ) jako dotazovatel, A. jako Quality Assurance Engineer (tester) a V. jako vývojář. Hlavní bloky otázek jsou označeny podkapitolami. Některé příliš konkrétní názvy jsem musel odstranit, jedná se o názvy týmů a klíčová slova.

Náplň práce

1 JŠ: Na úvod se zeptám, jaká je vaše pozice, náplň práce, velice zhruba. Jeden z vás je QA[1] a jeden je DEV[2] a jak dlouho jste tady ve firmě.

2 V: Jeden a půl roku. Spíš dva, dva roky.

3 A: Nechceš to shrnout celý? Co děláš?

4 V: Takže od začátku, v podstatě co dělám, tak většinu času je to programování a návrhu designu, no, částečně o issue[3] jinak, tak to je asi všechno.

5 A: No já jsem na pozici Quality Assurance Engineer. Mou náplní je vlastně jak, jak podílení se na přípravě specifikace. V podstatě během celého vývoje softwaru bychom se měli podílet na tom, že jsme nějakým způsobem zainteresovaní, to znamená od vývoje specifikace až po poslední testování. To znamená, neměli bychom být jenom testeři nebo klikaři, klikači, klikaři – to by bylo dobrý. To znamená, podílíme se na kvalitě vývoje softwaru respektive měli bysme ji řídit. Pak samozřejmě, pokud je potřeba, tak hledáme chyby v kódu. Pokud přijde ňáký bug, tak bychom měli být schopní identifikovat, co je příčinou toho bugu a programátorům předat informaci typu „v této třídě na této řádce jsi napsal špatně podmínku“.

6 JŠ: Dokázali byste odhadnout, kolik času věnujete vývoji novým funkcionalitám a kolik opravám chyb?

7 V: Z mého pohledu je to tak padesát na padesát.

[1] Quality Assurance Engineer – Podílí se na přípravě návrhu, navrhuje a implementuje testy, testuje, hledá chyby, analyzuje chyby nahlášené zákazníkem.

[2] Developer – Podílí se na návrhu z technického hlediska, funkcionalitu implementuje, opravuje nahlášené chyby.

[3] Nahlášená chyba v systému, také označována výrazem bug.

8 A: Jako nevyvíjím nové features[4].

9 JŠ: Jasně, tak testování nových feature.

10 A: Ale no když to tak vezmu, tak asi víc se určitě věnuje hledání chyb než testování. Já bych řekl tak 60 na 40 to znamená 40% připravit test plánů a testování a vývoji a 60% je pracování na issues to znamená scrubování[5], hledání chyb.

11 JŠ: Jo. Ještě nevím, jestli si to zmiňoval, jak dlouho jsi tady?

12 A: Jo rok a půl, něco málo přes rok a půl, rok a tři čtvrtě.

Vlastněný kód

13 JŠ: Jo. Dobře. Jak moc velkou vnímáte code base[6] ve firmě?

14 V: Z mého pohledu relativně obří oproti projektům, na kterých jsem dělal dřív.

15 JŠ: A ještě možná doplňující otázka, kolik si myslíš, že si z ní si schopen obsáhnout, kolik už jsi viděl a kolik si třeba sám měnil nebo vytvářel.

16 V: To je dobrá otázka. Z mého pohledu se pohybuju jenom na určitejch úrovních toho systému a řekl bych, že z mého pohledu se pohybuji tak na pětadvaceti procentech kódu systému. Kolik jsem schopen obsáhnout tak pět procent?

17 A: Z těch pětadvaceti? [7]

18 V: Abych si pamatoval nějakým způsobem vazby a tak. Nějaký takový ten náhled, tak řekněme pět procent. Kolik jsem sahal, tak to bude asi méně než procento.

19 JŠ: Jo. Jasně. Z pozice QA – ve firmě je zase obří sada testů. Jak je to, jestli bys mohl tuhle otázku rozdělit – jednak kolik si vlastně viděl code base na produkci a kolik si třeba sám upravoval a viděl v testu.

20 A: No já jsem překvapenej tím číslem, co řekl V. Kolik je schopnej pojmout, no nevím mě to připadá jako strašně moc. Spíš bych to uvedl tak, že přestože vlastníme ňákej kód, tak v legacy kódu[8] se absolutně nevyznám, nevím vůbec nic. To spíš z těch novejch featur, na featurách na kterých sem se ňákým způsobem podílel, tak tam jsem schopen relativně rychle najít, kde došlo k chybě nebo kde jsme udělali chybu. Takže na featurách, který jsme vyvíjeli už my, tak tam je to dobrý. Stejně tak je to i s testama, jo, neviděl bych tam žádnej rozdíl, jestli je to produkční kód nebo testovací co se týká vyznat se v něm. To znamená testy, který se dělaly za období, kdy tady jsem, tak v těch se docela vyznám. Vím, k čemu jsou, jsem schopen se v nich zorientovat naproti tomu legacy testy, jo, jsou psaný úplně jiným stylem teda jednak a je to jak s produkčním kódem to prostě jenom podstatně těžší že jo. Je až dvanáct let starej.

21 JŠ: Dobrá. A kdybys měl říct nějaký to procento za sebe?

[4] Nová funkcionalita.

[5] Prvotní analýza nahlášené chyby. Má za cíl přesně identifikovat problé, jak jej navodit, zjistit jeho povahu a nalézt zodpovědný tým nebo osobu, která bude chybu opravovat.

[6] Veškeré zdrojové kódy firmy.

[7] Řečeno žertovně. V. měl na mysli 5% kódu z celkového počtu.

[8] Starší existující kód. Může být použito i pro označení zastaralého kódu.

22 A: To si netroufnu moc, to bych jako střílel úplně.

23 JŠ: Tak o to v podstatě nejde o nějaký přesný číslo. Jde spíš o podíl neznámého kódu oproti tomu známému.

24 A: Výrazně velký.

25 JŠ: Jasně.

26 A: Teď je otázka, jestli tu otázku myslíš vztaženo k oblasti k naší doméně, to znamená, ta na kterou se zaměřujeme, anebo obecně ke všemu kódu.

27 JŠ: Jo.

28 A: Pokud ke všemu, tak to je obrovský diametrální rozdíl mezi tím.

29 JŠ: Jasně. A v doméně týmu?

30 A: Tam bych řekl těch 10% jsme schopni nějakým způsobem. To odpovídá zhruba tomu, kolik jsme toho, za tu dobu co tady jsem, vyvíjeli my.

31 JŠ: Jasně. A kdyby ses měl podívat na nějakou rozšířenější oblast týmu, nějaké příbuzné, tak tam by to bylo zhruba kolik?

32 A: Když se zaměříme asi na to <Tým A> a <Tým B> [9], tak pro mě docela blízkej <Tým C> [10], protože to je předchozí tým, kde jsem pracoval, takže tam bych se asi taky vyznal. Ale na některých featurách jsem pracoval v jinejch týmech tak řeknu, tak třeba dvě tři procenta, víc ne.

33 JŠ: Dobře. Tak teď taková obecná otázka, jsou nějaké oblasti, které vyloženě patří týmu? Jsou to vyloženě oblasti, které byly vytvářeny v rámci nějaké featury, zatímco jste tady byli, jak jsi třeba zmiňoval ty A., anebo jsou to třeba i nějaké zodpovědnosti nebo nějaký kód, který jsi vlastně ten váš tým jako kdyby nese s sebou? Jako kdyby odkaz. Nebo jak byste vůbec charakterizovali kód, který považujete za vlastní?

34 V: Tak z mého pohledu to množství kódu, které si neseme s sebou je jednak to, co jsme vyvinuli, takže jsou to třídy, které jsou ryze v našem vlastnictví a vzhledem k situaci našeho týmu, tak vlastníme třídy, které jsou na rozhraní s dalšími týmy, kvůli tomu, že my zajišťujeme jakoby obecnou podporu, nebo takový ten abstraktní náhled pro ně, pro jejich funkcionalitu.

35 JŠ: Takže jsou to vyloženě nějaké třídy, jako kdyby patří týmu, vyloženě.

36 V: Řekl bych, že to máme tak, no nedovedu si troufnout procenta, ale řekl bych tak padesát procent tříd, které jsme napsali, patří ryze nám. Pak padesát procent těch, které jsme napsali, jsme sice napsali a vlastníme je a opravujeme je, ale rozhodně nejsme jedinými uživateli.

37 JŠ: Jo. Takže by se dalo říct, že je tam jakoby šedá zóna?

38 V: Tak těch padesát procent je ta šedá zóna.

39 JŠ: Takže abych to tak shrnul, by se dalo přirovnat jako kdyby k nějakým ostrůvkům, že

[9] Odstraněno z důvodu zachování anonymity firmy. Jedná se o týmy mající na starost podobnou oblast produktu.

[10] Odstraněno, jedná se tým zabývající se podobnou oblastí v produktu, ale odlišnou funkcionalitou.

vlastně každý ten tým vlastní nějaký svůj ostrov, pak je tam nějaký ten prostor mezi tím.

40 V: O který bojujeme.

41 JŠ: Jo. Dobrá. Jak to vnímáš ty z pozice QA?

42 A: Ještě jednou mi polož tu otázku, co je přesně předmětem té otázky?

43 JŠ: Jak, jasně. Jak vnímáš nebo jak bys definoval kód, který vlastní tvůj tým.

44 A: Jako definoval z pohledu kvality?

Stávající praxe určení vlastníků

45 JŠ: Ne, jak vůbec ten kód vypadá, nebo jak bys poznal z nějaké třídy ano, tento kus kódu máme na starosti my? Nebo jestli se to vůbec dá poznat.

46 A: Rozumím, jasně. Myslím si, že každej, spousta lidí na to má svůj postup, někteří právě používaj anotace [11], hodně často na základě anotací se snaží rozhodnout, čí je to kód. Buď ho poznám tak, že ho znám, že jsem ho viděl, že jsem se na něm podílel, že jsem ho viděl na review, což je jasný. Hned si su jistej, že je náš. Pokud si jistej nejsu, tak velice často se jím snažím prohrabat pomocí debuggingu. Například pokud musím scrubovat nějakou issue, tak se snažím zjistit kudy a kterým kódem vlastně prochází, tak podle toho. Pokud už objevím chybu, tak pak se snažím najít konkrétní changelist[12] a pak samozřejmě pokud najdu konkrétního člověka, který ten changelist měnil, tak dokážu identifikovat tým.

47 JŠ: Jasně. Takže jdeš vyloženě po tom, kdo naposledy šahal do toho kousku kódu.

48 A: Jasně.

49 JŠ: Dobrá. Přejdeme dál. Ty už jsi to trošku nakousl A., tak dám možná prvně prostor tobě. Při kterých příležitostech tedy hledáš vlastníka kódu.

50 A: Jednak teda jak V. avizoval, tak náš tým funguje tím způsobem, že děláme v podstatě podporu pro mnoho jiných týmů, čímž bohužel, bohužel fungujeme pro mnoho jiných týmů jako router[13] pro přehazování problémů. Nad náma stojí technická podpora, která je vlastně interfacem mezi námi a zákazníkem, tak mnoho těch problémů přehazuje přímo na náš tým, protože si nejsou přesně jistí, komu patří. Proto říkám, že my fungujeme jako router. To je první případ, kdy já narazím na to, že potřebuju nějakým způsobem vlastně zjistit, kam ta daná problematika spadá, do kterýho týmu, koho se ten kód týká. Některý problémy jsou samozřejmě mě už zřejmý z pouhýho popisu, kdy vidím, o jakou oblast se jedná, takže já vůbec neřeším kód v tu chvíli, ale z takových padesáti procent musím zjišťovat, kam to spadá. Tim způsobem, co jsem popsal a to je jenom na přehození týmů. Druhá věc je samozřejmě ve chvíli, kdy musím už konkrétně najít, kde je daný problém. Jedna věc je, že zjistím že, když to samozřejmě spadá na nás ten problém, tak snažím se i identifikovat, který člověk to od nás dělal. To je druhý případ, kdy se snažím to issue scrubovat do hloubky, takže potřebuju vědět, že tam zanesl bug například V.

51 JŠ: Jo. Takže jedna věc je teda investigace issue, je ještě nějaká další příležitost, při které to potřebuješ zjistit?

[11] Slovo vychází z názvu funkcionality annotate integrovaného vývojového prostředí. Ke každému řádku kódu vypíše, kdo jej naposledy měnil.

[12] Změna uchovávaná v systému správy verzí.

[13] Směrovač

52 A: Jedna věc je investigace issue, druhá věc je jakoby na té vrchní úrovni – to přehazování.

53 JŠ: Což, předpokládám, by měla řešit technická podpora.

54 A: To by mělo, přesně tak, přesně tak. Ale jelikož k tomu technická podpora buď k tomu nemá vůbec nástroje, nebo nástroje tu jsou formou wiki stránek, ale o těch často neví ani leadové týmů[14], což sem si mnohokrát ověřil. Třeba já jsem to za technickou podporu někomu přehodil a oni mě psali, proč jim to hážu a já jsem jim odepsal, že z toho a tohohle důvodu, že tady na téhle stránce je napsaný, že vy jste vlastníci téhle části kódu. Oni o tom vůbec třeba ani nevěděli, i to se párkrát stalo. A přiznali, že vůbec tu stránku neznaj. Problém je samozřejmě u téhleté stránky, že absolutně nepracuje s žádnými verzovacími systémy. Je to vlastně statickej text a tím pádem samozřejmě není updatovanej, není aktuální. Třetí věc, kdy si myslím, že pro nás je dobrý pracovat s podobným systémem, který jsem vymyslel, je při specifikaci, kdy my potřebujeme investigovat, jak zapracujeme danou featuru a potřebujeme zjistit, který kód budeme měnit a s jakými týmy ten kód souvisí, že jo. Protože ve chvíli, kdy je to hodně složitý a potřebujeme s ostatními týmy komunikovat, tak potřebujeme zjistit, s kterými týmy budeme komunikovat.

55 JŠ: Jasně. Kdy ty potřebuješ jako DEV zjistit a určit vlastníka kódu?

56 V: Tak u nás jsou to dvě strany mince, ta první je v případě review[15], kdy se dodělávají ať už nové změny anebo se opravují bugy, tak to je první případ. Je poměrně důležitý, protože většinou původní vlastník měl často nějaký úmysl, proč danou věc napsal, rozhodně důležitý bod. A druhý věc je, jak už zmiňoval A., při specifikaci a designu. Pokud se na to podíváme z výšky, tak některé třídy, některé funkcionality jsou vlastněny jinými týmy, takže tam je důležitá kooperace, v podstatě i schválení designu.

57 JŠ: Ty jsi zmínil opravu issue. Stává se ti, tobě jako programátorovi, že vlastně opravuješ kód, který ti nepatří. I když vlastně tady tím rozdělením by to mělo přistát na správném místě, abych tak řekl.

58 V: To se stává, stává se to zrovna u nás v týmu relativně často, protože vlastníme hodně té šedé zóny. Nevlastníme, ale operujeme poměrně hodně v šedé zóně, nicméně nejsou nástroje na to, aby se to vytříbilo rychleji.

59 JŠ: Dobrá.

60 A: Jestli k tomu můžu ještě něco dodat, mě napadla jedna věc. Docela často se využívá fakt, že my třeba fixujeme náš bug, ale je to právě to, co zmínil V. v oblasti té šedé zóny, kdy není úplně jasný, kdo by měl být vlastníkem toho kódu nebo to nelze tak jednoduše rozhodnout a ve chvíli, kdy my jsme schopni najít tým, se kterým to blízce souvisí, tak často žádáme jejich QA o pomoc při verifikaci[16]. Není to vůbec výjimečný, kdy vlastně spolupracujeme zejména s <Tým A> a <Tým B> na spolupráci při verifikaci.

61 JŠ: Ještě něco vás k tomu napadá? Super. Takže další otázka, ty už jsi to taky zase nakousl. Jak vlastně postupujete při určování vlastníka, ty jsi zmínil anotace v Perforce. Případně se ještě díváš, kdo vlastně tu konkrétní změnu už udělal, tedy konkrétního člověka. Ještě předpokládám, že jsou to vlastně vaše znalosti a to, co jste už viděli. Je ještě něco co vás napadá, co se dá použít na určení vlastníka kódu, nebo používáte?

[14] Team leader – vedoucí týmu
[15] Proces hodnocení kontroly a hodnocení změny kódu ostatními vývojáři.
[16] Proces ověření, zda nahlášená chyba byla skutečně opravena.

62 V: Patterny[17] ve jménech. Protože určité týmy mají jednoznačně určené někeré třídy, který k nim patří.

63 JŠ: Promiň, teď jsem nerozumněl.

64 V: Patterny ve jménech tříd. V podstatě, dalo by se říci, že takový pattern je například <Klíčové slovo><Klíčové slovo>[18].

65 JŠ: Ještě něco tě napadá?

66 V: Nic dalšího.

67 JŠ: A tebe A.?

68 A: No tak některej kód poznám, že jo. Když každá proměnná začíná final[19] tak vím že, (smích) se stačí otočit a zavolat V.

69 JŠ: Jasně. Dokázali byste odhadnout, kolik vám určování vlastníka zabere času? V průběhu dne, nebo týdne dejme tomu? Samozřejmě to záleží na vykonávané činnosti, klidně to pojměte víc ze široka, než nějaké číslo jenom.

70 A: No z mýho pohledu je největším žroutem času právě fakt, že fungujeme jako router a ve chvíli, kdy já musím rozhodovat, kam to issue padá, nemám dostatek informací, tak samozřejmě kolikrát vracím to issue na technickou podporu, abych si vyžádal dost informací, což samozřejmě, jako když se to ve finále nasčítá a je toho hodně, tak já si troufnu říct, že za týden strávím čtyři hodiny jenom tímto. Co se týká investigace samotné, tak tam samozřejmě záleží na tom, jak to issue je složitý, ale pokud vím, jaká je to oblast, tak se o tom nemá cenu zmiňovat vůbec, protože tam je to jasný, ale pokud je to věc, se kterou jsem se nesetkal a musím to scrubovat do hloubky, tak na tom strávím klidně den dva. Plus samozřejmě jsou případy, kdy jsme strávili nad identifikací kódu týden ve dvou lidech. Ale to jsou samozřejmě extrémy. Jsou taky extrémy, že člověk klikne a za dvě minuty ví, kam to patří, a kde je chyba.

71 JŠ: To je pak spíš asi debugging...

72 A: Jasně, jasně.

73 V: U mě jsou to, když bych to vzal hodně na hrubo, tak za týden tak dvě až čtyři hodiny. A když to tak řeknu, tak kolikrát se mi vyplatí si ten bug fixnout[20], než abych dohledával, čí to je.

Uplatnění Who-Owns

74 JŠ: Jo. Dobrá, tak asi přejdeme dál. Zase ty už jsi A. trošku předběhl. Jak byste do těchto problémů, co jste zmínili, jak byste dokázali použít ten můj nástroj.

75 V: Já klidně začnu. Že bych nefixoval issue, které nejsou naše, tak to určitě je první věc. Automatické napovídání na review, další věc. Tam by byla třeba zajímavá v budoucnu ňáká rozšíření tohoto toolu[21]. Takže to za mě.

[17] vzorce, šablony
[18] Odstraněno pro zachování anonymity firmy.
[19] Klíčové slovo jazyka Java.
[20] Opravit
[21] Nástroje

76 A: Já bych na té úrovni při komunikaci s technickým supportem vůbec tvůj nástroj nepoužíval, já bych ho ideálně přesunul rovnou na technický support, tedy aby ten byl schopen na základě toho rozhodnout, ke kterému týmu to patří, protože to je prakticky to, co by oni měli dělat. A na té úrovni samozřejmě, kterou zmínil V., to se shodnem no, pokud by mě to pomohlo identifikovat problém, že zjistím, že to nepatří našemu týmu, tak já nebudu ve scrubování pokračovat, k devům[22] se to vůbec nedostane a poputuje to do relevantního týmu.

77 JŠ: Jasně. Ty jsi říkal, že by se to mělo přesunout na technickou podporu a ty jsi zmínil review, napadá vás ještě nějaký firemní proces, do kterého by se to dalo integrovat?

78 A: Myslím si, že i během meetingů jsme narazili na problém, kdy jsme řešili novou featuru, narazili jsme na problém, že vlastně nevíme, kdo vlastní danou část kódu a pak samozřejmě musel být někdo dedikovanej[23] na to, kdo musel ten problém rozlousknout. To znamená, že musel zjistit, kdo to vlastní a s kým je potřeba komunikovat, abychom dostali nějaké relevantní informace.

79 V: Možná by to bylo zajímavé i s pohledu plánování.

80 JŠ: Mohl bys to nějak rozvést?

81 V: Když některé části systému patří jednoznačně některému týmu, tak možná by bylo vhodné, aby ten tým dál dělal dané modifikace nad tím kódem. Dalo by se tím zajímavě ušetřit zbytečné zpomalování při vývoji, kdy se člověk musí učit úplně nový kód, který vlastně v životě neviděl.

82 JŠ: Jasně. Takže v podstatě je to situace, kdy nějaký jiný tým vytvořil kus kódu a pak za něho nechtějí nést odpovědnost.

83 V: Přesně tak.

84 JŠ: Dobrá. Myslím, že už jsme tady z těch hlavních bodů probrali zhruba všechno, takže bych to možná dál bral jako volnou diskuzi. Jestli vás napadá ještě něco k tomuto tématu, co by se dalo zmínit, co nepadlo, klidně to můžeme pobrat jako takový brain storming.

85 V: Možná v rámci rozšíření by bylo zajímavý vytvořit plug-in do Cruciblu[24] a automaticky přidávat hinty na reviewery[25].

86 A: Já si pamatuju na takovej fakt, jenom jsem si vzpomněl. Když jsi to otevřel poprvé nebo když jsem to viděl poprvé, tak jsem si vzpomněl na fakt z předchozí firmy. Majitel firmy hodlal ohodnocovat zaměstnance na základě počtu řádek. A dokonce i vznikla metrika, teď nevím, jestli to bylo 300 řádků za den, že by měl programátor vyprodukovat, ale pak obrovským překvapením samozřejmě bylo, když statistiky, který vypadly z SVN[26] nám říkaly, že programátoři napíšou pouze 10% kódu a že 90% kódu napíšou naši technici, což bylo hrozně divný. Technici samozřejmě nasazovali ten produkt na servery, že jo, což se dá prostě přirovnat k naší integraci[27]. To by asi s tvým řešením ty statistiky vypadaly jinak, že jo. Rozhodně blíž skutečnosti.

87 JŠ: Jasně.

88 A: Teď jsem to teda řekl tak jako. Já si samozřejmě myslím, že je to úplnej nesmysl, oceňovat někoho za počet řádku kódu, jen abych to uvedl na pravou míru.

[22] Vývojářům
[23] vyhrazen
[24] Crucible, nástroj pro proces kontroly kódy ostatními vývojáři.
[25] hint – nápověda, reviewer – vývojář, který byl vybrán autorem změny, aby ji zkontroloval
[26] Jeden ze systémů správy verzí
[27] Přesun uložených změn do jiné vývojové větve v systému správy verzí.

89 V: Na tom se shodnem. Jasně.

90 V: Možná by bylo zajímavý, další řekněme plug-in, kterým by se prohnal stack trace a určil by na základě toho tým, který by měl issue dostat.

91 JŠ: Jasně. To mě už taky napadlo. Podobnou věc vlastně zmínil i B.[28], který říkal, že databáze chyb, která ukládá některé chyby nebo třeba pomalé dotazy do databáze, vlastně produkuje jako vedlejší produkt stack trace, tak to by se dal jako kdyby odhadnout vlastník toho problémového místa a mohl by mu systém automaticky poslat nějaké e-mailové upozornění: „pozor pomalé SQL“, „pozor chyba“. Uvítali byste to, nebo by vás to spíš obtěžovalo? I když to je spíš otázka na V. jako deva.

92 V: Já bych to rozhodně uvítal. Z mého pohledu každá issue, která mi přistane s pomalým SQL, tak téměř okamžitě opravuju.

93 JŠ: Dobrá. Ještě něco vás napadá? Super, v tom případě je to všechno. Děkuju moc.

[28] Manažer týmu do kterého patří A. i V.

Příloha E

Elasticsearch - dotazník

V této příloze uvádím přepis komunikace se Simonem Willnauerem, který je technickým vedoucím v projektu ElasticSearch. Mé dotazy jsou vyznačeny kurzívou.

1) When did the project start?
The project started in Feb. 2010

2) How long have you been contributing to the project?
I joined elasticsearch development in late 2012

3) Roughly how many developers actively contribute to the project?
I think at this point there are about 20 to 25 active developers

4) Are developers organized into teams in the project? If yes, could you list your team mates?
Yes it's organized in teams but they are cross feature etc. and they shuffle every 3 month. I am the tech-lead on Elasticsearch so I am on top of the teams if you will. In practice I am working with all of the guys depending on the task and focus.

5) Please estimate the percentage of the codebase that you have:

- created yourself: 10%
- modified: 50%
- seen: 85%

6) Please write a few areas of the codebase in which you are mostly specialized. (packages / modules / classes / typical methods. . .)
engine, store, shard, recovery, shadow replicas, tests

7) Please write a few areas of the codebase in which you would consider yourself the only expert. (packages/modules/classes/typical methods. . .)
I can't think of any there is always a second person..

8) The list below contains words extracted from the code. How much do they relate to your activity in the project? Please assign a score to each word on scale from 1 to 5 (1 - Absolutely relevant; 1 - Not relevant at all)

- `ex`: 5
- `directory`: 3
- `engine`: 1
- `random`: 2

- `files`: 3
- `tests`: 1
- `commit`: 3
- `env`: 4
- `store`: 2
- `lock`: 2
- `dir`: 3

8) And how well do they set you apart from other developers? Please assign a score to each word on scale from 1 to 5 (1 - Very descriptive; 5 - Commonly used)

- `ex`: 5
- `directory`: 4
- `engine`: 1
- `random`: 3
- `files`: 4
- `tests`: 2
- `commit`: 3
- `env`: 3
- `store`: 2
- `lock`: 3
- `dir`: 4

9) The list below contains a few code areas (package name optionally followed by a class and a method). How responsible do you feel for them? Please assign a score to each item on scale from 1 to 5 (1 - Full responsibility. If something breaks there, you would be probably the best person to fix it; 5 - No responsibility, probably because you have never seen the code;)

- `test.java.org.elasticsearch.test`: 2
- `main.java.org.elasticsearch.index.engine`: 1
- `main.java.org.elasticsearch.index.store`: 1
- `main.java.org.elasticsearch.gateway`: 3
- `main.java.org.elasticsearch.cluster.routing`: 3
- `test.java.org.elasticsearch.cluster.routing`: 3
- `test.java.org.elasticsearch.gateway`: 3
- `test.java.org.elasticsearch.index.store`: 1
- `main.java.org.elasticsearch.index.shard`: 1
- `test.java.org.elasticsearch.search.basic`: 3

Příloha F

Obsah přiloženého CD

- `report` - Tato práce ve formátu PDF.
- `report/src` - Zdrojové kódy práce pro systém LaTeX.
- `whoowns/src` - Zdrojové kódy programu Who-Owns.
- `whoowns/bin` - Spustitelná podoba Who-Owns.
- `whoowns/3rdparty` - Potřebný software třetích stran
- `whoowns/readme.md` - Krátký popis používání programu v angličtině.
- `sample/elasticsearch/dump.rdb` - Databáze obsahující ukázková data z projektu ElasticSearch.
- `sample/guava/dump.rdb` - Databáze obsahující ukázková data z projektu Guava.