Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológií

FIIT-5220-46122

Bc. Lukáš Markovič

Refaktorovanie softvérových systémov pomocou XML technológií

Diplomová práca

Študijný program: Softvérové inžinierstvo

Študijný odbor: 9.2.5 Softvérové inžinierstvo

Miesto vypracovania: Ústav informatiky a softvérového inžinierstva, FIIT STU, Bratislava

Vedúci práce: Ing. Ivan Polášek PhD.

máj 2016

Slovenská technická univerzita v Bratislave
FAKULTA INFORMATIKY A INFORMAČNÝCH TECHNOLÓGIÍ

Refaktorovanie softvérových systémov pomocou XML technológií

Študijný program: Softvérové inžinierstvo

Autor: Lukáš Markovič

Vedúci diplomovej práce: Ing. Ivan Polášek PhD.

Máj 2016

Diplomová práca je zameraná na oblasť automatizovaného refaktorovania softvérových systémov. Práca analyzuje aktuálne používané prístupy k refaktorovaniu softvérových systémov a reprezentácie z ktorých vychádzajú. Pozornosť je venovaná možnostiam refaktorovania, pokiaľ sa vychádza z čistého zdrojového kódu v textovej podobe, zjednodušenej reprezentácie pomocou abstraktného syntaktického stromu, alebo z reprezentácií pomocou dodatočných technológií, ako napríklad JSON, alebo XML. Práca sa ďalej zameriava na reprezentáciu pomocou XML, pričom je analyzované vyhľadávanie a refaktorovanie anti-vzorov a pachov kódu pomocou XML technológií XQuery a XQuery Update. Súčasťou práce je taktiež návrh nástroja, ktorý by umožňoval automatizované vyhľadávanie, ale aj refaktorovanie jednotlivých anti-vzorov a pachov v kóde softvérového projektu pomocou používateľom definovaných pravidiel. Ako súčasť nástroja je vytváraný expertný systém, ktorý je postavený s cieľom rozpoznávať vzťahy medzi jednotlivými anti-vzormi a pachmi a rozhodovať o ideálnom postupe aplikovania refaktorovacích pravidiel.

Slovak University of Technology Bratislava
FACULTY OF INFORMATICS AND INFORMATION TECHNOLOGIES

Refactoring of software systems using XML technologies

Degree Course: Software engineering

Author: Lukáš Markovič

Supervisor: Ing. Ivan Polášek PhD.

May 2016

This thesis is aimed at area of software systems refactoring automatization. The work analyses the currently used approaches to refactoring of software systems and representation underlying. Attention is given to refactoring options when starting from a clean source code in text form, simplified representation using abstract syntax tree or representations using additional technologies such as JSON or XML. The work furthermore aims at representation using XML, and there are analysed search and refactoring anti-patterns and code smells using XML technologies XQuery and XQuery Update. Part of the work is design of a tool which would allow automated search, as well as refactoring of individual anti-patterns and code smells in a software project using user-defined rules. As part of the tool is designed expert system that is built with the aim to recognize the relationship between anti-patterns and smells and decide the way, how to apply refactoring rules.

Obsah

Zoznam obrázkov

1. Úvod

Informačné technológie výraznou mierou ovplyvňujú každodenný život dnešnej spoločnosti. Ešte donedávna fungovali mnohé bežné zariadenia, ako napríklad telefón alebo hodinky na pomerne jednoduchom princípe, no v dnešnej dobe sa už aj o chod takýchto zariadení starajú komplikované a náročné programy, postavené na rôznorodých softvérových technológiách. Vývoj softvérových systémov, či už na zákazku pre konkrétneho zákazníka, alebo ako komerčný produkt, je preto veľmi dôležitou činnosťou v dnešnom technologickom svete.

Jedným z hlavných vnútorných problémov s tvorbou softvérových systémov, s ktorým sa stretávali vývojári už od počiatkov softvérového vývoja, bola zložitosť vytváraných systémov. Vyvíjaný systém bol vo väčšine prípadov príliš zložitý, preto sa na vývoji podieľali väčšie, či menšie skupiny vývojárov spoločne. Aj v tomto prípade však vznikali problémy s komunikáciou v tíme, porozumení si, správnom rozdelení úloh, či kvôli rôznemu štýlu písania zdrojového kódu alebo taktiež kvôli rôznej úrovni skúseností programátorov. Práve posledný dôvod častokrát spôsoboval a stále spôsobuje zlyhávanie softvérových projektov. Tento a taktiež ďalšie problémy, ako napríklad neviditeľnosť softvérových systémov, vyústili v šesťdesiatych rokoch minulého storočia do takzvanej softvérovej krízy, ktorá bola jedným z podnetov pre vznik novej inžinierskej disciplíny. Využitie princípov softvérového inžinierstva, ktoré sa zaoberá špecifikáciou, návrhom a vývojom softvérových systémov, je dnes neoddeliteľnou súčasťou pri vývoji každého softvérového systému.

Popri klasických prístupoch, ako napríklad vodopádový, alebo iteratívny vývoj sa v poslednom čase presadzujú aj moderné, agilné metódy vývoja softvéru. Tieto metódy kladú častokrát dôraz na odstraňovanie problémov v skorých štádiách. Jednou z takýchto činností je aj refaktorovanie – vnášanie zmien do zdrojového kódu s cieľom zlepšenia jeho vnútornej reprezentácie, bez ovplyvnenia vonkajšej funkcionality. Refaktorovanie sa prvýkrát rozšírilo medzi programátormi pracujúcimi v jazyku Smalltalk, pričom v súčasnosti sa uplatňuje ako priama súčasť agilnej metódy extreme programming. Za pomoci týchto programátorov preskúmal Martin Fowler základné princípy refaktorovania softvérových systémov, pričom výsledkom bola jeho publikácia, ktorá je dodnes považovaná za základnú príručku pre refaktorovanie softvérových systémov. V danej publikácií opísal prvých a doteraz stále najdôležitejších dvadsať dva pachov kódu – problémov, ktoré

nespôsobujú chyby, no častokrát poukazujú práve na hlbší problém, či už v samotnom návrhu, alebo implementácii systému [1]. Popri pojme pach kódu sa rozšíril aj pojem anti-vzor, po prvýkrát použitý Andrewom Koenigom [2].

Medzi pojmom pach kódu a anti-vzor sa neraz vôbec nerozlišuje a sú považované za synonymá. V prípade hlbšieho štúdia danej problematiky je však možné nájsť drobné odlišnosti. Zatiaľ čo pach kódu predstavuje metaforu pre indikáciu hlbšieho problému systému, no nepredstavuje problém sám o sebe, anti-vzor vystupuje ako opak vzoru – ide teda o mnohokrát používanú techniku pre vyriešenie stanoveného problému, ktorá sa však ukázala ako nesprávna a nežiadúca. Do oblasti anti-vzorov teda v istom ponímaní môžeme zaradiť aj problémy, ktoré môžu spôsobiť dokonca aj chyby systému, pričom cieľom je tieto problémy odstrániť. Pri takejto analýze slova anti-vzor však vzniká rozpor s cieľom refaktorovania – neovplyvniť vonkajšie správanie systému. Za cieľ refaktorovania sa preto vo všeobecnosti označujú pachy zdrojového kódu. Je však možné taktiež upustiť od prísneho dodržania princípu refaktorovania - neovplyvniť vonkajšie správanie zdrojového kódu, pokiaľ je zmena v pozitívnom zmysle, vďaka čomu je možné refaktorovať aj anti-vzory.

Hlavným cieľom tejto diplomovej práce je návrh prístupov k riešeniu jednotlivých problémov automatizovaného refaktorovania zdrojového kódu. Pri automatizovaní refaktorovania je nutné uvažovať mnohé aspekty, ktoré nie sú dostatočne vyriešené ani v súčasnej dobe. Základ práce tak tvorí návrh a rozpracovanie nových, respektíve málo používaných metód pre vyhľadávanie a refaktorovanie pachov v kóde. Práca sa však zaoberá aj zložitejšími problémami, na ktoré automatizované refaktorovacie nástroje neraz nehľadia. Podstatným problémom pri automatizovaní refaktorovania je situácia, kedy sa v kóde nachádza viacero pachov. Takéto pachy sa neraz môžu nachádzať aj na jednom mieste, z čoho vyplýva problém ovplyvňovania sa jednotlivých pachov. Dobrý automatizovaný refaktorovací nástroj musí takéto vzťahy rozoznávať preto, aby bolo samotné refaktorovanie efektívne, ale aj zasahovalo do zdrojového kódu podľa možností čo najmenej.

V rámci práce je poskytnutý taktiež návrh a čiastočná implementácia jednoducho prispôsobiteľného nástroja na automatizované vyhľadávanie anti-vzorov a pachov kódu, ktorý využíva XML reprezentáciu zdrojového kódu a jazyk XQuery na vyhľadávanie a prevádzanie zmien v XML súboroch, teda metódy, ktoré samotná práca navrhuje. Súčasťou práce je taktiež analýza zvolených metód a prípadných ďalších dostupných možností. Diplomová práca vychádza z návrhov popísaných v predchádzajúcej, bakalárskej práci [3].

1.1. Prehľad práce

Kapitola 2 – Analýza konceptov automatizovaného refaktorovania

- kapitola sa zameriava na úvod do problematiky automatizovania refaktorovania softvérových systémov. Ponúka taktiež prehľad východiskových reprezentácií pre jednotlivé refaktorovacie prístupy. V podkapitolách je bližšie rozoberané samotné refaktorovanie spolu s vyhľadávaním nad XML reprezentáciou zdrojového kódu.

Kapitola 3 – Existujúce riešenia

- kapitola poskytuje prehľad a popis existujúcich podobných riešení daného problému. Cieľom kapitoly je popísať dostatočné množstvo existujúcich riešení tak, aby bolo možné urobiť si predstavu o ich kvalite, respektíve nedostatkoch a taktiež minulých a aktuálnych trendoch.

Kapitola 4 – Návrh automatizovaného refaktorovacieho systému

- kapitola popisuje návrh základného procesu refaktorovania, ktorý je použitý pre vytvorenie refaktorovacieho nástroja. Popísané sú výhody XML reprezentácie, ďalej sa venuje efektívnej práci s XML reprezentáciou projektu – XML databázam, a návrhu vyhľadávacích a refaktorovacích postupov pomocou jazykov XPath, XQuery a XQuery Update. Ako súčasť návrhu je taktiež prezentovaný expertný systém Jess a jeho možnosti.

Kapitola 5 – Závislosti pachov kódu

- samostatná kapitola je venovaná identifikácií závislostí medzi pachmi kódu. Rozoberaných je základných 22 pachov kódu a vzťahov medzi nimi pomocou viacerých pohľadov. Táto kapitola tvorí východisko pre tvorbu pravidiel expertného systému Jess.

Kapitola 6 – Optimalizácia procesu refaktorovania

- kapitola je venovaná návrhu metód pre vyhľadávanie optimálneho postupu aplikovania refaktorovacích pravidiel. V kapitole je pozornosť venovaná hlavne metódam, ktoré vychádzajú z grafovej reprezentácie vzťahov medzi pachmi kódu. Navrhované sú rôzne spôsoby prehľadávania, ako pomerne jednoduché prioritizované prehľadávanie, alebo využitie biologicky inšpirovaných algoritmov.

Kapitola 7 – Implementácia automatizovaného refaktorovacieho systému

- kapitola sa zameriava na samotnú implementáciu nástroja na automatizované refaktorovanie. Implementácia vychádza hlavne z konceptov navrhovaných v kapitolách 4, 5 a 6.

Kapitola 8 – Evaluácia riešenia a vyhodnotenie navrhovanej metódy

- kapitola stručne popisuje metódy testovania, ktoré boli použité pre overenie funkcionality implementovaného nástroja. Taktiež sú zhodnotené zistené nedostatky a možnosti ich nápravy. Systém je porovnaný voči iným podobným systémom.

Kapitola 9 – Zhodnotenie a ďalšie smerovanie

- prezentuje záver práce a ďalšie činnosti, ktoré môžu predstavovať ďalšie smerovanie výskumnej činnosti v tejto oblasti.

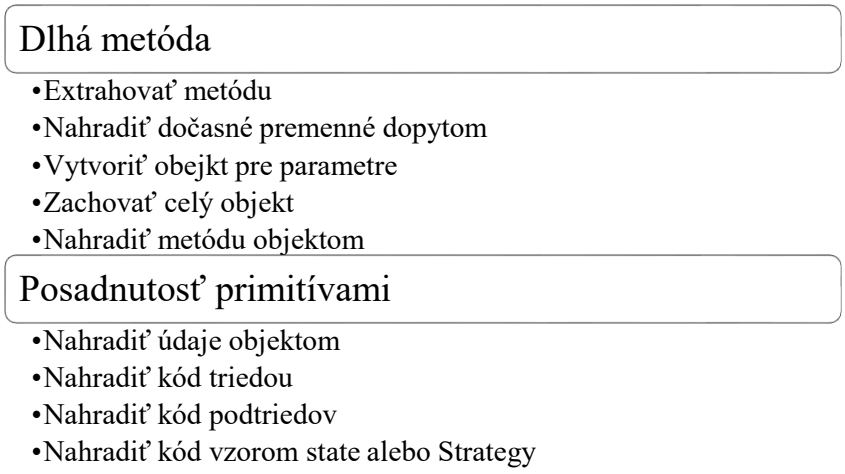
2. Analýza konceptov automatizovaného refaktorovania

Táto kapitola je zameraná na popis problémovej oblasti. Rozoberá aktuálny stav danej problematiky a zameranie tejto práce v celkovom kontexte prebiehajúceho výskumu týkajúceho sa automatizovaného refaktorovania. Popísané sú bežne používané spôsoby reprezentácie zdrojového kódu a vyhľadávania anti-vzorov a pachov kódu. Kapitola sa ďalej zameriava na samotné refaktorovanie anti-vzorov a pachov kódu.

2.1. Výzvy pri procese refaktorovania

Refaktorovanie – úprava zdrojového kódu do efektívnejšej podoby, bez ovplyvnenia vonkajšieho správania, je proces extrémne náročný na automatizovanie. Neraz je pre vyhľadávanie a refaktorovanie aj jednoduchších problémov nutné poznať širší kontext softvérového systému, lebo informácie pre vykonanie korektnej opravy sa mnohokrát nachádzajú na viacerých miestach v zdrojovom kóde a je ich nutné samostatne vyhľadať a rozpoznať. Podobný problém sa vyskytuje pri automatizovanom vyhodnocovaní kvality zdrojového kódu, meraní systému, či navigácii v zdrojovom kóde využívanej vo vývojových prostrediach. Rovnako, ako v tomto prípade, aj pri automatizovanom vyhľadávaní anti-vzorov a pachov kódu a ich refaktorovaní, je riešením dôsledná statická analýza.

Ďalšou výraznou komplikáciou je to, že konkrétne anti-vzory a pachy kódu nemajú vo väčšine prípadov jednoznačné riešenie. Niekedy môže byť ideálnym spôsobom opravy extrakcia metódy, inokedy však uvedenie novej triedy, pričom sa vždy jedná o rovnaký pach kódu, iba v inom kontexte. *Obr. 1* ukazuje typické pachy a spôsoby ich refaktorovania [1].



Obr. 1 Typické pachy kódu a spôsoby ich refaktorovania

Do procesu refaktorovania sa preto v praxi častokrát zapájajú skúsení experti, ktorí dokážu určiť vhodný spôsob refaktorovania v danej situácii. Pri automatizácii však vzniká problém s neprítomnosťou ľudského prvku. Výber spôsobu refaktorovania nájdených problémov je však možné ponechať na expertný systém, ktorý sa na základe dostupných informácii zo statickej analýzy a databázy znalostí rozhodne pre ideálny spôsob riešenia. Pomocou takéhoto systému je taktiež možné riešiť problém závislostí jednotlivých pachov kódu [3].

Refaktorovanie častokrát vyžaduje generovanie, respektíve odstraňovanie častí zdrojového kódu. Pri tejto časti procesu vznikajú taktiež mnohé problémy a komplikácie. Pri vkladaní nových častí kódu je potrebné zachovať funkcionalitu už existujúcich častí, napríklad vyhýbaním sa existujúcich názvov tried, metód a podobne. Potrebné je taktiež zachovať formátovanie vložených, aj pôvodných častí kódu. Niektoré reprezentácie však fungujú na spôsobe prevádzania zdrojového kódu do inej reprezentácie, pričom sa touto, ale aj spätnou transformáciu môže ovplyvniť formátovanie v celom projekte. Riešenie týchto problémov závisí hlavne od spôsobu reprezentácie, ale aj povahy nástroja – je ich napríklad možné ignorovať.

Jednou z najpodstatnejších komplikácií je však rôznorodosť programovacích jazykov. Nástroj, ktorý umožňuje refaktorovanie nad jazykom Java bude bez prispôsobenia pravidiel v konečnom dôsledku úplne nepoužiteľný pre jazyk C# a podobne. Poskytnúť komplexný nástroj pre širokú paletu programovacích jazykov je veľmi náročné, keďže program musí byť prispôsobený pre každý jazyk samostatne. Väčšina existujúcich nástrojov sa preto zameriava iba na jeden konkrétny, prípadne skupinu najpoužívanejších jazykov.

Tieto a mnohé ďalšie problémy robia oblasť refaktorovania veľmi zaujímavou a hodnou ďalšieho výskumu. V posledných desaťročiach vzniklo mnoho nástrojov umožňujúcich automatizované refaktorovanie, no prakticky žiadny z nich však nie je dostatočne komplexný a vyhovujúci pre široké potreby softvérového inžinierstva. Táto diplomová práca sa zameriava hlavne na preskúmanie možností využitia expertného systému pre proces refaktorovania. Vďaka systému, ktorý bude na rozhodovanie využívať definované pravidlá bude jednoduchšie vykonávať optimálne refaktorovanie bez zásahu ľudského experta. Práca sa taktiež dotýka problémov s vkladaním a odstraňovaním častí zdrojového kódu, no vzhľadom na náročnosť celého procesu sa zameriava iba na refaktorovanie programov napísaných v programovacom jazyku Java.

2.2. Reprezentácia zdrojového kódu

Proces automatizovaného refaktorovania možno rozdeliť do troch hlavných častí. Základným východiskovým bodom je vhodná reprezentácia zdrojového kódu. Práve na tejto fáze závisí spôsob vyhľadávania a opravovania anti-vzorov a pachov v kóde. Jednotlivé problémy sú častokrát veľmi špecifické. Zatiaľ čo pri jednom spôsobe reprezentácie kódu môže byť odhalenie veľmi náročné, iný spôsob reprezentácie môže umožňovať vyhľadať konkrétny problém oveľa jednoduchšie.

Prístupy k reprezentácii zdrojového kódu pri procese refaktorovania možno rozdeliť do dvoch skupín, ktoré je možne ďalej bližšie špecifikovať a to:

- *Zdrojový kód bez pridaných elementov* – vyhľadávať a prevádzať refaktorovanie je možné nad *čistým zdrojovým kódom*, ktorý nie je reprezentovaný špeciálnym spôsobom. Alternatívou je forma, kedy sa abstrahuje od nepotrebných elementov nazývaná *abstraktný syntaktický strom*.
- *Zdrojový kód reprezentovaný pomocou dodatočnej technológie* – efektívny spôsob refaktorovania je možné do istej miery vykonávať taktiež nad zdrojovým kódom reprezentovaným pomocou rôznych dodatočných technológií. Takto upravený zdrojový kód je doplnený o rôzne metadáta, ktoré umožňujú efektívnu navigáciu a vyhľadávanie v dokumente pomocou špecializovaných jazykov, asociovaných s technológiou využitou na reprezentáciu zdrojového kódu. Takýmito technológiami sú napríklad *JavaScript Object Notation* (*JSON*), alebo jazyk *XML.*

	Získanie reprezentácie	*Vyhľadávanie*	*Refaktorovanie*
Priamy kód	bez úsilia	metriky	textové nástroje
AST	jednoduché	AST knižnice, metriky	AST knižnice
JSON	veľmi náročné	JavaScript (a iné)	JavaScript (a iné)
XML	náročné	XPath, XQuery	XSLT, XQuery

Tabuľka 1 Základné porovnanie reprezentácií kódu pri procese refaktorovania

Tabuľka. 1 zobrazuje základné rozdiely medzi jednotlivými prístupmi. Tabuľka popisuje náročnosť daného prístupu a technológie, ktoré je možné využiť pre vyhľadávanie a refaktorovanie. Konkrétne možnosti jednotlivých prístupov sú bližšie popísané v nasledujúcich podkapitolách.

2.2.1. Zdrojový kód v textovej podobe

Vyhľadávanie a refaktorovanie možno robiť priamo nad „čistým“ zdrojovým kódom. Tento spôsob je pomerne obmedzujúci, keďže navigácia v dokumente je ťažkopádna a málo efektívna.

Zdrojový kód v textovej forme je však vhodným miestom pre využitie softvérových metrík. S pomocou nich možno efektívne odhaliť jednoduchšie problémy, ako napríklad *podmienková zložitosť*, alebo aj zložitejšie pachy kódu, ako napríklad *reťazec zodpovednosti* (spočítať množstvo volaní metód pre vykonanie určitej činnosti). Takéto využitie metrík pre refaktorovanie kódu v čistej textovej forme vyžaduje zložité prechádzanie a vyhľadávanie v texte, ktoré môže byť časovo náročné. Samotné refaktorovanie je však v tomto prípade extrémne náročné a viac-menej takmer nerealizovateľné. Metriky nad kódom v čistej textovej podobe možno preto efektívne využiť len na vyhľadávanie a identifikáciu anti-vzorov a pachov kódu. Túto techniku je ale možné rozšíriť spôsobom, ktorý namiesto hľadania anomálii v kóde bude vyhľadávať miesta pre využitie konkrétnych refaktorovacích pravidiel definovaných Martinom Fowlerom [4]. Nižšie sú uvedené typické metriky, ktoré môžu byť využité pre vyhľadávanie cieľov refaktorovania v zdrojovom kóde [5].

- *DIT (depth of inheritance tree)* – hĺbka dedičnosti
- *WMC (weighted methods per class)* – vážené metódy na triedu
- *CBO (coupling between objects)* – závislosti medzi objektami
- *LOCCLASS (lines of code in a class)* – počet riadkov v triede
- *LOCMETHOD (lines of code in a method)* – počet riadkov v metóde
- *NAD (number of atributes in a class)* – počet atribútov v triede
- *NMD (number of methods)* – počet metód
- *LCOM (lack of cohesion in methods)* - nedostatok súdržnosti v metódach
- *NACC (number of accessors)* – počet potomkov triedy
- *NPRIVFIELD (number of private fields)* – počet privátnych polí
- *atď.*

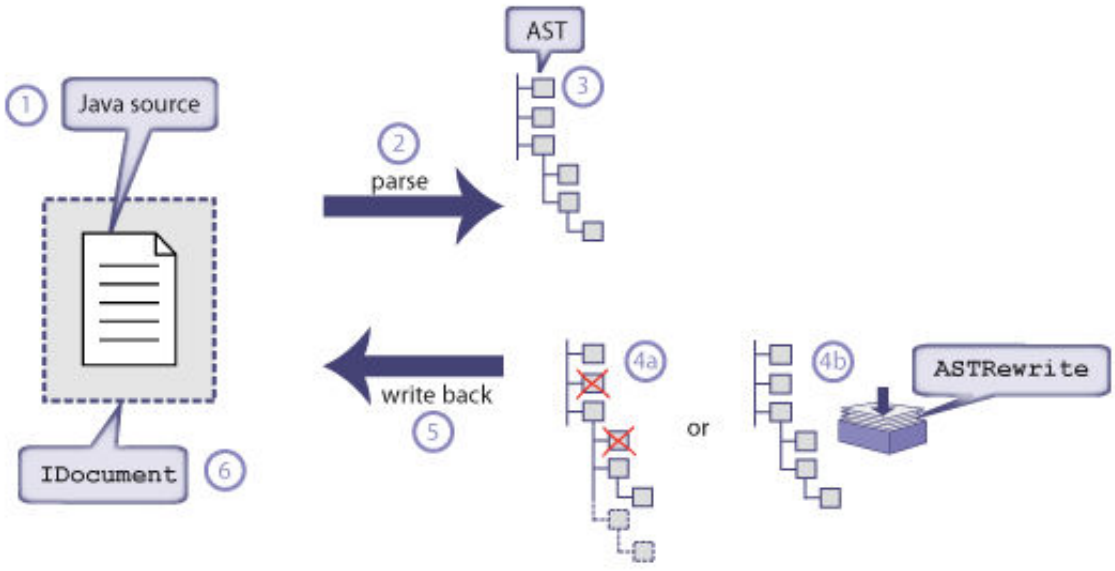
2.2.2. Abstraktný syntaktický strom

Syntaktický strom je spôsobom reprezentácie zdrojového kódy pomocou stromovej štruktúry. Ak sa v takejto reprezentácii upúšťa od syntakticky nedôležitých prvkov, ako zátvorky, medzery a podobne, nazýva sa abstraktný. Syntaktický strom sa využíva vo vývojových prostrediach alebo prekladačoch, pri preklade zdrojového kódu do nižšieho programovacieho jazyka.

Keďže mnohé vývojové prostredia majú priamo implementované nástroje a knižnice pre prácu s AST, je možné využiť AST taktiež na refaktorovanie. Výhodou oproti zdrojovému kódu v čistej podobe je pohodlnejšia manipulácia a navigácia v kóde pomocou stromovej štruktúry. Typickým príkladom vývojového prostredia využívajúceho AST je Eclipse, ktoré využíva AST pre navigáciu v kóde, automatické generovanie kódu, či základné refaktorovanie. Eclipse taktiež priamo obsahuje knižnice pre prácu s AST zdrojového kódu. Samotné refaktorovanie zdrojového kódu je teda možné vykonávať dvoma spôsobmi [6].

- *Priama úprava AST* – zmeny zapísané do existujúceho AST.
- *Využitie triedy ASTRewrite* – vytvára sa nová inštancia AST, v ktorej sa nachádzajú zmeny.

Na *Obr. 2* sa nachádza typický spôsob práce s kódom reprezentovaným pomocou AST v prostredí Eclipse, ktorý môže byť taktiež využitý pri refaktorovaní.



Obr. 2 Spôsob práce s AST v Eclipse *[6]*

2.2.3. Reprezentácia zdrojového kódu pomocou JSON

JavaScript Object Notation je notácia, ktorá vychádza z jazyka JavaScript, no v súčasnosti je úplne nezávislá a podporovaná mnohými ďalšími technológiami. Rovnako, ako pri XML ide o platformovo nezávislý spôsob zápisu údajov. Výhodou oproti jazyku XML je oveľa jednoduchšia a pre človeka čitateľnejšia syntax, ktorej základ tvorí spôsob zápisu dát vo formáte *kľúč – hodnota*. JSON formát si v poslednom čase získava veľkú obľubu, hlavne preto, lebo je priamo podporovaný mnohými modernými technológiami, ako napríklad JavaScript a zároveň postačujúci pre široké potreby súčasných softvérových systémov.

Na základe vyššie spomenutých dôvodov je možné predpokladať, že automatizované refaktorovanie by bolo zaujímavé vykonávať práve nad zdrojovým kódom zapísaným pomocou JSON notácie. V takýchto JSON súboroch by bolo jednoduché vyhľadávať a vykonávať transformácie pomocou akéhokoľvek jazyka, ktorý podporuje JSON notáciu. Ďalšou možnosťou je využitie dokumentových databáz, ktoré uchovávajú údaje, ako JSON objekty. Nad projektom, uloženým v jednej kolekcii takejto databázy je možné pomocou JavaScript jazyka vykonávať dopyty pre vyhľadávanie a update pre zmenu súborov v kolekcii. Príkladom takejto databázy je napríklad voľne dostupná dokumentová databáza MongoDB.

Problémom využitia JSON notácie pre vyhľadávanie v zdrojovom kóde a jeho transformáciu je však nedostatok nástrojov na vytvorenie takejto reprezentácie. V súčasnosti neexistuje prakticky žiadny voľne dostupný nástroj, ktorý by toto v dostatočnej kvalite umožňoval. V *prílohe A* je však uvedený JSON objekt, ktorý by mohol predstavovať reprezentáciu zdrojového kódu uvedeného na *Obr. 3*. Ako možno vidieť, obrovským problémom takejto notácie je práve zápis v tvare kľúč - hodnota. V prípade, ak sa v jednom uzle nachádza viac uzlov s rovnakým názvom, tak na rozdiel od XML nie je možné zachovať poradie. Vytvára sa zoznam dát, ktorý nezachováva informáciu o pôvodnom umiestnení prvku. JSON notácie je preto pravdepodobne nevhodná pre automatizované refaktorovanie, pokiaľ by nebol tento problém odstránený.

```
public int nsd(int a, int b) {
    if(a == 0) {
       return b;
    } else {
       return nsd(a, b % a);
}
```

Obr. 3 Testovací Java kód

2.2.4. Reprezentácia zdrojového kódu v jazyku XML

Jazyk XML – *eXtensible Markup Language*, ktorý bol prvý krát uvedený v roku 1996 slúži na platformovo nezávislý prenos údajov medzi aplikáciami, pričom v súčasnosti tvorí jeden zo štandardov. Jednoduché princípy, ktoré boli zadefinované W3C konzorciom spôsobili v minulosti rýchle a široké rozšírenie jazyka XML. Vďaka svojim vlastnostiam nachádza jazyk uplatnenie aj v dnešnej dobe, aj keď v poslednej čase sa do popredia dostávajú nové, jednoduchšie notácie, ako JSON alebo YAML.

XML dokument je tvorený súborom údajov, ktoré sú štruktúrované pomocou značiek, ktoré si používateľ definuje sám. XML dokument musí spĺňať tri základné kritéria:

- *Dokument musí mať jeden koreňový uzol*
- *Každý uzol musí mať začiatočnú a koncovú značku*
- *Uzly musia byť korektne vnárané*

Oproti JSON notácii nemá XML podporu v bežných programovacích jazykoch priamo, ale vo väčšine prípadov iba prostredníctvom špeciálnych knižníc a API. Ďalšou nevýhodou je v porovnaní s vyššie prezentovanou JSON notáciou oveľa horšia čitateľnosť XML dokumentu, ako možno vidieť na *Obr. 4*, ktorý prezentuje úplne rovnaké údaje, ako sa nachádzajú v *Prílohe A*.

Oproti JSON, prípadne iným nástrojom, je pri použití jazyka XML obrovskou výhodou široká dostupnosť rôznych pridružených, špeciálne zameraných jazykov pre prácu s XML dokumentom. Tieto špecializované jazyky sú prispôsobené tak, aby umožňovali jednoduchú prácu s XML dokumentom podľa povahy úlohy. W3C konzorcium zadefinovalo napríklad doplnkové jazyky pre vyhľadávanie, transformáciu, dopytovanie, či modifikáciu XML dokumentov. *Tabuľka 2* ukazuje prehľad najznámejších jazykov pre prácu s XML dokumentom a ich popis. K dispozícii sú však aj mnohé značkovacie jazyky, ktoré z XML priamo vychádzajú. Či už ide o jazyk XMI na popis modelov, alebo jazyk XHTML, možno vidieť, že základ našlo v XML mnoho iných technológií.

Jazyk – doplnky pre XML	*Popis*
XPath	Vyhľadávanie a navigácia v dokumentoch
XSLT	Transformácia dokumentov do iných dokumentov
XSL-FO	Formátovanie XML dokumentov
XQuery	Dopytovanie nad dokumentami
XSL	Rodina predchádzajúcich jazykov
XPointer	Adresovanie komponentov
XProc	Spracovanie pomocou dátovodov
XLink	Väzby medzi XML dokumentami
XInclude	Spájanie dokumentov
XForms	Vstupy z webových formulárov
XML Namespaces	Jedinečné názvy pre elementy
XML Schema	Popis elementov v dokumente

Tabuľka 2 Doplnky jazyka XML

Vzhľadom na prezentované vlastnosti XML a širokú paletu dostupných nástrojov pre manipuláciu s dokumentami je vhodné využiť jazyk XML aj na refaktorovanie. Ako vhodná a jednoduchá možnosť na vyhľadávanie pachov a anti-vzorov sa javí jazyk XPath [3]. Refaktorovanie, teda vlastne transformáciu XML dokumentov je teoreticky možné riešiť pomocou jazyka XSLT, táto možnosť však nie je veľmi ideálna, no naopak je náročná a ťažkopádna [3]. Ako vhodné riešenie sa však ponúka jazyk XQuery, ktorý sa dokáže postarať o samotné vyhľadávanie, ale aj úpravy viacerých XML dokumentov [7].

Táto diplomová práca skúma práve prístup k automatizovanému refaktorovaniu na základe XML reprezentácie zdrojového kódu pomocou jazyka XQuery.

```
<?xml version="1.0" encoding="UTF-8"?>
<unit>
        <function><type><specifier>public</specifier><name>int</name>
        </type><name>nsd</name><parameter_list>
         (
            <param><decl><type><name>int</name></type><name>a</name></decl></param>
         ,
            <param><decl><type><name>int</name></type><name>b</name></decl></param>
         )
        </parameter_list><block>
         {
            <if>
            if
                <condition>
               (
                    <expr><name>a</name>
                  == 0
                    </expr>
               )
                </condition><then><block>
                  {
                        <return>
                     return
                            <expr><name>b</name></expr>
                     ;
                        </return>
                  }
                    </block></then><else>
               else
                    <block>
                  {
                        <return>
                     return
                            <expr><call><name>nsd</name><argument_list>
                              (
                                        <argument><expr><name>a</name></expr></argument>
                              ,
                                        <argument><expr><name>b</name>
                                    %
                                                <name>a</name></expr></argument>
                              )
                                    </argument_list></call></expr>
                     ;
                        </return>
                  }
                    </block></else></if></block></function>
</unit>
```

Obr. 4 XML reprezentácia kódu na *Obr. 3*

Rovnako, ako aj pri JSON notácii predstavuje problém samotná transformácia zdrojového kódu do XML reprezentácie. Zatiaľ čo pre JSON notáciu neexistujú prakticky žiadne nástroje, pri XML je vzhľadom na jeho širokú podporu k dispozícii niekoľko voľne dostupných nástrojov. Medzi tieto nástroje patria [7] [3] [8]:

- *JavaML*
- *OOML*
- *CppML*

Tieto nástroje využívali konverziu *zdola nahor* – mapovanie AST stromovej štruktúry do XML dokumentu [8]. V súčasnosti nie je ani jeden z daných nástrojov ďalej vyvíjaný, či podporovaný.

Druhý z možných prístupov – prístup *zhora nadol* funguje na základe jednoduchého označovania kľúčových konštrukcií jazyka [8]. Vzniká tak XML dokument, ktorý na rozdiel od predchádzajúceho prístupu obsahuje nadbytočné časti, ktoré nie sú pre proces refaktorovania potrebné, ako napríklad zátvorky a podobne. Tento prístup však podporuje stále aktívne vyvíjaný nástroj *scrML.*

Source Code Markup Language[1] je nástroj, ktorý zabezpečuje prevod zdrojového kódu rôznych jazykov do XML reprezentácie. Transformácia je vykonávaná prechádzaním zdrojového súboru zhora nadol a postupným označovaním konštrukcií jazyka. Spätná konverzia predstavuje jednoduché odstránenie vložených značiek [9]. Práve kvôli svojej jednoduchosti, ale taktiež aj nedostatku iných nástrojov, bol tento nástroj zvolený pre vykonávanie konverzie zdrojového kódu do XML reprezentácie a späť.

Problémom nástroja je však nedôsledné označovanie častí kódu, keď okrem textovej časti obsahuje uzol aj ďalšie vnorené uzly. Častokrát tak vzniká dokument, v ktorom treba špeciálne dbať na zachovanie textovej časti uzla, čo spôsobuje mnohé problémy pri dopytovaní a transformácii dokumentov. Vhodným riešením je návrh nástroja na doplnenie dodatočnej značky, ktorá by reprezentovala textovú časť. V dokumente by teda bolo zabezpečené, že uzol obsahuje jedine zdrojový kód – textová značka, alebo vnútorné uzly.

[1] http://www.srcml.org/

2.3. Vyhľadávanie v XML súboroch

Pred samotným refaktorovaním tvorí kritickú činnosť samotné určenie miesta, kde sa nachádzajú problémy v kóde – pachy kódu a anti-vzory. Táto diplomová práca si kladie za cieľ do hĺbky preskúmať možnosti automatizovaného refaktorovania nad XML reprezentáciou zdrojového kódu. Vzhľadom na tento fakt vychádza táto kapitola z reprezentácie popísanej v kapitole 2.2.4, teda reprezentácie pomocou jazyka XML získanej s využitím nástroja *srcML*.

Ako ukazuje *tabuľka 2*, na vyhľadávanie v XML dokumentoch možno v podstate využiť dva doplnkové jazyky a to:

- *XPath* – jednoduchý jazyk pre vyhľadávanie a navigáciu v XML dokumentoch. Výsledkom dopytu v tomto jazyku sú uzly z XML dokumentu. Jazyk umožňuje taktiež spracovanie výrazov, reťazcov, vyhodnocovanie booleovských hodnôt a mnoho ďalšieho. Napriek svojej jednoduchej syntaxi a celkovej jednoduchosti je jazyk XPath veľmi silný nástroj pre vyhľadávanie v XML dokumentoch. Pomocou tohto jazyka možno teda pohodlne vyhľadávať anti-vzory a pachy v zdrojovom kóde.
- *XQuery* – podobne, ako jazyk XPath slúži tento jazyk na dopytovanie v XML dokumentoch. Okrem dopytovacích možností však zároveň ponúka možnosť funkcionálneho programovania nad XML dokumentami.

Možnostiam jazyka XPath sa venovala predchádzajúca práca [3], z ktorej táto práca vychádza. Výsledkom práce bolo, že jazyk XPath je veľmi vhodný nástroj na vyhľadávanie anti-vzorov a pachov kódu nad XML reprezentáciu zdrojového kódu. Jazyk XPath však nie je dostatočný pre samotné refaktorovanie nájdených problémov. Na refaktorovanie je nutné využiť iné technológie, napríklad jazyk XSLT, ktorý bol v danej práci [3] taktiež analyzovaný.

Na druhej strane jazyk XQuery, ktorý okrem samotného dopytovania umožňuje aj programovanie nad XML dokumentom, predstavuje oveľa flexibilnejší a jednoduchší nástroj, ktorý okrem fázy vyhľadávania dokáže zároveň pokryť aj samotné refaktorovanie vyhľadaných problémov. Jazyk XQuery, ktorý je podrobne popísaný v nasledujúcej podkapitole bol teda zvolený ako hlavný nástroj pre vyhľadávanie a refaktorovanie anti-vzorov a pachov kódu v tejto práci.

2.3.1. Jazyk XQuery

Jazyk XQuery je jedným z doplnkových jazykov pre XML. XQuery je aktívne vyvíjaný *W3C (World Wide Web Consortium),* pričom jeho prvá verzia bola uvedená v roku 2007. Aktuálne je k dispozícii špecifikácia 3.0, ktorá sa stala doporučenou špecifikáciou 8. apríla 2014 [10]. Cieľom W3C pri vývoji jazyka XQuery bolo poskytnúť nástroj pre dopytovanie nad XML dokumentami, ktorý by okrem samotného vyhľadávania umožňoval aj funkcionálne programovanie nad dokumentom pre následné transformácie štruktúrovaných aj neštruktúrovaných dát. Jednotlivé implementácie jazyka sa však neraz neobmedzujú iba na XML dokumenty, ale častokrát poskytujú rozšírenia aj pre iné formáty pre reprezentáciu údajov.

Samotný jazyk vychádza z jazyka XPath, pričom ten je jeho priamou súčasťou. Každý výraz, ktorý je korektným výrazom v jazyku XPath, je taktiež korektným dopytom jazyka XQuery a zároveň je zaručené, že výsledok bude rovnaký v oboch jazykoch. Ako súčasť kódu napísaného v jazyku XQuery môžu byť použité akékoľvek XML značky, HTML značky, a iné. XQuery ako taký je vysokoúrovňový, deklaratívny jazyk, ktorý je silne typový [10]. Jadro jazyka XQuery tvoria takzvané FLWOR výrazy. Ide o konštrukcie, ktoré predstavujú doplnok jazyka XPath a slúžia na prechádzanie výsledkov, priradzovanie premenných, či určovanie návratovej hodnoty z funkcie. FLWOR predstavuje skratku pre 5 výrazov *For, Let, Where, Order by, Return*, ktoré sú bližšie popísané v *tabuľke 3.* V danej tabuľke sú popísané aj ďalšie výrazy patriace do tejto skupiny – *Window, Count, Group by*.

Výraz	*Popis*
FOR	Iterácia cez sekvenciu uzlov
LET	Priradenie hodnoty premennej
WHERE	Filtrovanie výsledkov na základe splnenia podmienky
ORDER BY	Usporiadanie podľa definovaných pravidiel
RETURN	Výraz sa vykoná sa pre každý spracovávaný uzol
WINDOW	Iterácia cez naviazané premenné
COUNT	Definuje pozíciu v spracovávanom zozname
GROUP BY	Vytvára výstup na základe zlúčenia častí vstupu

Tabuľka 3 FLOWR výrazy *[10]*

Jazyk XQuery je pomerne jednoduchý a rýchlo zvládnuteľný. *Obr. 5* zobrazuje XML dokument, ktorý predstavuje zjednodušený záznam z emailovej komunikácie fiktívnej firmy. Na *Obr. 6* je následne XQuery, ktorý demonštruje základné možnosti jazyka.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<emails>
   <email>
      <to>john123@exampleSoft.com</to>
      <from>eva@exampleSoft.com</from>
      <subject>Meeting</subject>
      <body>Don't forget, meeting is at 14:00 AM!</body>
   </email>
   <email>
      <to>eva@exampleSoft.com</to>
      <from>john123@exampleSoft.com</from>
      <subject>RE: Meeting</subject>
      <body>Thank you.</body>
   </email>
   <email>
      <to>john123@exampleSoft.com</to>
      <from>eva@exampleSoft.com</from>
      <subject>RE: RE: Meeting</subject>
      <body>You're welcome</body>
   </email>
   <email>
      <to>smith555@exampleSoft.com</to>
      <from>eva@exampleSoft.com</from>
      <subject>Project</subject>
      <body>Can you send me project status please?</body>
   </email>
   <email>
      <to>eva@exampleSoft.com</to>
      <from>smith555@exampleSoft.com</from>
      <subject>RE: Project</subject>
      <body>Sorry, I have no time right now</body>
   </email>
   <email>
      <to>smith555@exampleSoft.com</to>
      <from>john123@exampleSoft.com</from>
      <subject>Project</subject>
      <body>Can you send me project status please?</body>
   </email>
   <email>
      <to>john123@exampleSoft.com</to>
      <from>smith555@exampleSoft.com</from>
      <subject>RE: Project</subject>
      <body>No</body>
   </email>
</emails>
```

Obr. 5 Príklad XML dokumentu

```
distinct-values(
  for $email in //emails/email
    let $subject := $email//subject//text()
    where $subject[contains(., 'RE: ')]
    order by $email/from
  return $email/from
)
```

Obr. 6 Príklad XQuery

Vyššie uvedený XQuery skript funguje nasledovne:

1. *Prechádza sa všetkými uzlami **email**, ktoré sa nachádzajú v koreňovom uzle **emails***
2. *Do premennej **subject** sa uloží textová časť uzla **subject** z aktuálne spracovávaného uzla*
3. *Skontroluje sa, či sa v premennej nachádza podslovo „RE: “*
4. *Výsledok sa usporadúva abecedne na základe hodnoty **from** z uzla*
5. *Výsledok cyklu je vyfiltrovaný od duplicitných hodnôt pomocou funkcie **distinct-values***

Ako možno vidieť z príkladu, jazyk má veľmi jednoduchú syntax a je pomerne ľahko čitateľný. Jazyk XQuery je však mnohokrát zamieňaný s jazykom XSLT, pretože oba jazyky slúžia na transformáciu XML dokumentov, pričom taktiež využívajú XPath na dopytovanie v XML dokumente. Na rozdiel od XSLT, ktorý slúži na definovanie šablóny pre transformáciu XML dokumentu do nového dokumentu, XQuery slúži na vytváranie komplikovaných štruktúrovaných dopytov do skupiny XML dokumentov – ide v podstate o SQL pre XML dokumenty. Vzhľadom na svoje vlastnosti – funkcionálne programovanie, jednoduché dopytovanie, flexibilná prácu s dátami, typovosť a jednoduchosť na pochopenie, je jazyk XQuery vhodnejším kandidátom na refaktorovanie softvérových systémov [3], vzhľadom na čo je zvolený ako základná technológia využitá na refaktorovanie v tejto práci.

2.4. Transformácie skupín XML súborov pomocou jazyka XQuery

Pri refaktorovaní softvérových systémov je neraz potrebné pracovať naraz nad viacerými triedami, respektíve XML súbormi po konverzii. Pri refaktorovaní pomocou jazyka XSLT bolo prepojenie súborov podstatne zložité, keďže XSLT súbor predstavuje šablónu pre určitý XML súbor. Dopytovanie nad viacerými súbormi je oveľa jednoduchšie pomocou jazyka XQuery. Cestu k súboru je možné jednoducho určiť pomocou funkcie *fn:doc(„uri“).* Táto funkcia dokáže pracovať s dokumentami v súborovom systéme počítača, ale aj priamo na webe, ako možno vidieť nižšie:

```
for $node in doc('http://www.w3schools.com/xml/note.xml')
return $node
```

Obr. 7 Ukážka funkcie fn:doc pre webový dokument[2]

```
for $node in doc('C://emails.xml')
return $node
```

Obr. 8 Ukážka funkcie fn:doc pre lokálny dokument

Výhodnejšie, ako pristupovať ku každému súboru samostatne, je možnosť využiť nástroje, ktoré dokážu pracovať so skupinou XML dokumentov ako s jednou kolekciou. Nad takto vytvorenou kolekciou je možné vytvárať dopyty ako nad jediným dokumentom. Odpadá teda potreba prístupu k jednotlivým dokumentom pomocou funkcie *fn:doc*.

Technológia, ktorá ponúka takéto riešenie sú dokumentové databázy. Takéto databázy pracujú s dokumentami v rôznych formátoch, ako napríklad JSON, XML, či čistá textová podoba. Práve XML databázy sú tie, ktoré je ideálne využiť pre riešenia komplikovaného dopytovania a transformácií nad XML dokumentami. Kapitola 2.4.1 poskytuje prehľad a úvod do XML dokumentových databáz.

[2] http://www.w3schools.com/xml/

2.4.1. XML dokumentové databázy

XML databázy predstavujú nástroj pre perzistentné ukladanie údajov, pričom tie sú popísané pomocou jazyka XML. Na základe vlastností sa tento druh databáz dá opísať ako *NoSQL dokumentová databáza*. V skutočnosti však existujú 2 typy XML databáz.

- *Databázy s povoleným XML* – ide o tradičné relačné databázy, ktoré však dokážu pracovať aj s XML dokumentami.
- *Natívne XML databázy* - NoSQL dokumentové databázy, ktorých primárny spôsob práce s údajmi predstavuje jazyk XML a jeho doplnky ako dopytovacie a modifikačné jazyky. Práve týmito databázami sa okrajovo zaoberá táto práca.

Dokumentové databázy, ktoré predstavujú jednu skupinu z NoSQL databáz, slúžia na uchovávanie informácii v dokumentovej forme, teda ako semi-štruktúrované dáta. Na rozdiel od relačných databáz, ktoré sú silne typové a uchovávajú údaje v oddelených tabuľkách, dokumentové databázy neudržiavajú informácie o typoch. Najčastejšie formáty, ktoré využívajú dokumentové databázy na ukladanie údajov sú JSON a XML.

XML databázy, ktoré tvoria významnú časť dokumentových databáz, uchovávajú údaje v XML formáte. Každá takáto databáza poskytuje jazyk, ktorý slúži na dopytovanie, manipuláciu a aktualizáciu údajov v databáze. Práve na tento účel je väčšinou využitý jazyk XQuery, no pre aktualizáciu údajov v databáze však XQuery neposkytuje dostatočné možnosti. XML databáza však vo väčšine prípadov poskytuje nadstavbu nad jazykom XQuery – *XQuery Update*, ktorý slúži na aktualizáciu uzlov XML dokumentu v databáze. Možnosti jazyka XQuery Update sú bližšie popísané v *kapitole 2.4.2*.

V súčasnosti existuje viacero XML databáz, ktoré sú bližšie popísané v *tabuľke 4*. Každá z databáz väčšinou poskytuje nástroje na pripojenie priamo z vybraných programovacích jazykov. Existuje však aj všeobecné programové Java API, ktoré rieši prístup ku väčšine z týchto XML databáz priamo z jazyka. *XQJ*[3] *– XQuery API for Java* predstavuje spoločné rozhranie pre vykonávanie XQuery nad XML databázami podobne ako JDBC API.

[3] http://xqj.net/

XML Databáza	*Licencia*	*Implementácia*	*XQJ*
BaseX	BSD	Java	Áno
eXist	LGPL	Java	Áno
MarkLogic Server	Komerčné	C++	Áno
Sedna	Apache License 2.0	C++	Áno
QizX	Komerčné	Java	Nie

Tabuľka 4 Najznámejšie XML dokumentové databázy

XML dokumentové databázy sú postavené na niekoľkých všeobecných konceptoch. Mnohé z nich platia pre všetky XML databázy, no nižšie sú zhrnuté hlavné koncepty, ktoré boli zostavené na základe dokumentácie k *BaseX* databáze[4], ktorá je vo všeobecnosti jednou z najvyspelejších.

Databáza – databáza je jednoduchý koncept, ktorý pozostáva z určitého počtu zdrojov, ktoré sú odlíšené unikátnymi cestami. XML databáza pracuje najčastejšie so zdrojmi v XML formáte, no dokáže uchovávať aj binárne dáta (BaseX) a mnohé dokážu taktiež pracovať aj s inými formátmi.

Kolekcia – kolekcia predstavuje databázu, ktorá pozostáva z dvoch a viacerých dokumentov (BaseX). Niektoré databázy chápu kolekciu ako nástroj na ďalšie delenie databázy, podobne ako tabuľky v SQL. Databáza teda pozostáva z viacerých kolekcií.

Dopytovanie – dopytovacie jazyky slúžia na získavanie výstupov z databázy a ich transformáciu. V súčasnosti je najpoužívanejším jazykom na tento účel jazyk XQuery. Keďže jazyk XPath je súčasťou jazyka XQuery, má taktiež podporu v dokumentových databázach. Navyše je však častokrát poskytovaná podpora pre XSLT, ktoré slúži na definovanie transformačnej šablóny pre výstup z dopytu.

Manipulácia – keďže XQuery nemá dostatočné prostriedky pre manipuláciu dát priamo v databáze, existuje nástroj XQuery Update, ktorý navyše poskytuje „aktualizačné“ výrazy, ktoré dokážu meniť stav existujúceho uzla [11].

[4] http://docs.basex.org

2.4.2. XQuery Update

XQuery Update predstavuje špecifikáciu W3C konzorcia pre aktualizáciu existujúcich uzlov v zdroji XML dát. Na rozdiel od XQuery výrazov, ktoré dokážu taktiež prevádzať transformácie, no pri každej zmene sa vytvára nová inštancia, XQuery Update slúži na priamu zmenu pôvodných uzlov [11].

XQuery Update nadstavba bola známa aj využívaná pomerne dlho [12], no finálnou špecifikáciou W3C sa stala až v roku 2011. Neexistuje však všeobecná implementácia, ale špecifikácia má mnohé implementácie, ktoré vo väčšine prípadov slúžia ako doplnok pre konkrétny nástroj pracujúci s XML súbormi a XQuery ako dopytovacím jazykom. Príkladom takýchto nástrojov sú napríklad XML dokumentové databázy popísané v kapitole 2.4.1.

S príchodom tejto nadstavby nad XQuery možno výrazy jazyka XQuery rozdeliť do štyroch skupín [11]:

- *Základné aktualizačné výrazy*
- *Aktualizačné výrazy*
- *Jednoduché výrazy*
- *Pomocné výrazy*

Práve prvá skupina výrazov predstavuje výrazy, ktoré definuje XQuery Update. Druhá skupina je následne rekurzívne vyskladaná zo štandardných XQuery transformácií a XQuery Update výrazov. Tretia skupina predstavuje všetky XQuery, ktoré nie sú transformačné a posledná skupina predstavuje špeciálne, samostatne definované výrazy, ktoré nemožno zaradiť do žiadnej predchádzajúcej skupiny.

S výrazmi, ktoré sú súčasťou špecifikácie XQuery Update možno nájsť podobnosť pri klasických relačných databázach. Ide konkrétne o päť výrazov, ktoré sú bližšie popísané v *tabuľke 5* uvedenej nižšie. Vzhľadom na to, že každý nástroj má vlastnú implementáciu XQuery Update, sú ukážky syntaxe uvádzané z dokumentácie k databáze BaseX.

Výraz	*Popis [11]*	*Syntax*[5]
Insert	Vloží kópiu uzla / uzlov do požadovaného uzlu	`insert node (attribute { 'a'} {5}, 'text', <e/>) into /n`
Delete	Odstráni uzly z inštancie konkrétneho uzla	`delete node //node`
Replace	Nahradí uzol, prípadne hodnotu uzla novým uzlom, alebo hodnotou	`replace node /n with <a/>`
Rename	Nahradí názov uzla iným názvom	`for $n in //node` `return rename node $n as 'renamedNode'`

Transform	Vytvorenie modifikovanej kópie existujúceho uzla	`copy $c := doc('example.xml')//node[@id = 1]`
		`modify rename node $c as 'copyOfNode'`
		`return $c`

Tabuľka 5 XQuery Update výrazy

Nadstavbu XQuery Update je taktiež nutné využiť pri refaktorovaní na základe vyššie predkladaného postupu. Vzhľadom na bezproblémovú spoluprácu s jazykom XQuery nad XML databázou, je jej využitie však veľmi jednoduché a bezproblémové. Vďaka faktu, že najdôležitejšiu časť tejto nadstavby tvorí päť výrazov nie je potrebné venovať skoro žiadne dodatočné úsilie získavaniu poznatkov o nadstavbe.

[5] http://docs.basex.org

2.5. Využitie expertného systému pri procese refaktorovania

Predchádzajúca práca poukazovala na mnohé problémy, ktoré vznikajú pri automatizovaní refaktorovania softvérových systémov. Ako hlavné prezentované komplikácie pri automatizovaní refaktorovania boli v práci určené [3]:

- *Ovplyvňovanie sa jednotlivých anti-vzorov a pachov kódu* – odstránením jedného problému môže vzniknúť iný problém. Refaktorovanie nejedného pachu kódu predstavuje vytváranie nových pachov. Typickým príkladom je, že po odstránení pachu *Middle Man* vo väčšine prípadov vzniká pach *Message Chains.* Ďalším príkladom je refaktorovanie pachu *Long Parameter List*, kde v istých prípadoch môže vzniknúť pach *Data Class.* Pri aplikovaní refaktorovacích postupov preto treba dbať, aby celková kvalita softvérového systému nebola ešte zhoršená. Je potrebné správne rozhodovať, ktorý pach je v danom softvérovom kontexte prijateľnejší. Keďže pri automatizovaní je potrebné tento problém riešiť, musí nástroj poznať jednotlivé závislosti medzi jednotlivými pachmi a anti-vzormi.
- *Optimálna postupnosť opráv* – podobne, ako predchádzajúci problém, aj pri tejto zaujímavej téme je potrebné vychádzať zo závislostí medzi jednotlivými pachmi. Na rozdiel od témy vyššie, tu však ide o dosiahnutie optimálneho výsledku pomocou najmenšieho počtu krokov. V praxi možno pozorovať, že odstránením jedného anti-vzoru alebo pachu možno častokrát docieliť odstránenie niekoľkých ďalších. Napríklad pri identifikácií pachu *Long Method* a zároveň *Long Parameter List* možno najskôr vyriešiť problém s dlhým zoznamom parametro*v.* No v prípade, ak by bola najskôr rozdelená metóda na dve rôzne metódy, pričom každá by si ponechala iba potrebné parametre, pravdepodobne by zanikol aj pach hovoriaci o dlhom zozname parametrov. Vzhľadom na efektivitu, ale aj minimalizáciu zásahov do zdrojového kódu musí nástroj na automatizované refaktorovanie softvérových systémov poznať aj takéto závislosti.

Napriek nedostupnosti ľudského experta, ktorý dokáže takéto závislosti rozpoznať a pracovať s nimi, je pri automatizovaní refaktorovania stále možné využiť softvérovú náhradu. Ide konkrétne o takzvané expertné systémy, ktoré sú vyvíjané za účelom nahradenia ľudského prvku v softvérových systémoch.

Expertné systémy predstavujú softvérové programy, ktoré sú zamerané na rozhodovanie v istej problémovej oblasti. Rozhodovanie prebieha na základe definovaných pravidiel, ktoré sú poväčšine vytvorené expertami v danej oblasti. Expertné systémy neraz nerozhodujú priamo, ale poskytujú iba rady, na základe ktorých sa rozhodujú jednotlivci, alebo tímy v danej problémovej oblasti. Typickými miestami využitia sú oblasti ako bankovníctvo, poisťovníctvo, medicína, vojenský priemysel a podobne.

Každý expertný systém je zostavený z troch hlavných častí. Prvá časť – *báza znalostí* predstavuje zachytené znalosti expertov v danej problémovej oblasti, s ktorými dokáže systém pracovať. Bázu znalostí expertný systém taktiež mnohokrát sám rozširuje takzvanou generalizáciou vstupných faktov na pravidlá. V prípade refaktorovacieho systému by bola práve báza znalostí tvorená informáciami o vzťahoch jednotlivých anti-vzorov a pachov kódu. Tieto fakty by boli poskytnuté na začiatku, no v istom prípade by bolo možné pravidlá aj automaticky rozširovať na základe získaných výsledkov.

Druhá časť – *odvodzovací stroj* predstavuje mechanizmus, ktorý na základe databázy znalostí produkuje výstupy, teda v prípade automatizovania refaktorovania návrhy ideálnych refaktorovacích pravidiel, ktoré je vhodné aplikovať.

Tretia časti – predstavuje *bázu faktov*, teda vstup pre pravidlový systém. V prípade nástroja pre automatizovanie refaktorovania by to bol samotný zdrojový kód a odhalené pachy v ňom. Báza faktov sa postupne rozširuje, ako prebieha spracovanie aktuálneho stavu údajov nachádzajúcich sa v báze faktov.

V súčasnosti existuje mnoho expertných systémov, ktoré majú široké využitie. Vytvoriť jednoduchý expertný systém nie je veľmi komplikované a možné aj v štandardných jazykoch, ako je Java, C#, alebo C++. Oveľa jednoduchšie je však využitie špecializovaných pravidlových systémov, ktoré môžu byť v istom zmysle považované za logické, respektíve deklaratívne programovacie jazyky. Tieto jazyky sú špeciálne zamerané na vytváranie expertných systémov, pomocou definovania bázy znalostí – pravidiel, ktoré zjednodušene hovoria o tom, ako sa zachovať pri detekcii určitého faktu. Pravdepodobne najznámejšie takéto nástroje sú *Clips* a nástoj *Jess*, ktorý z Clips vychádza.

Na základe typu definovaných problémových oblastí na začiatku tejto podkapitoly možno povedať, že oblasť automatizovaného refaktorovania softvérových systémov je ideálnym miestom pre využitie expertných systémov a nástrojov pre vytvorenie potrebného systému.

3. Existujúce riešenia

Táto kapitola súhrne popisuje softvérové riešenia automatizovaného refaktorovania. Nižšie sú uvedené nástroje, ktoré umožňujú vyhľadávať, opravovať, alebo taktiež automatizovať celý proces refaktorovania. V súčasnosti možno nájsť iba málo aktívne vyvíjaných a dostatočne funkčných nástrojov v dostatočnej kvalite, no mnoho riešení bolo vyvinutých ako súčasť výskumných prác. Pomerne dostatočné množstvo nástrojov rieši vyhľadávanie pachov v kóde, no len minimum aj samotné refaktorovanie.

3.1. RefaX

Prvý z uvádzaných nástrojov predstavuje nástroj RefaX – nástroj, ktorý využíval transformáciu zdrojového kódu do XML reprezentácie [12]. Práve z konceptu tohto nástroja vychádza táto diplomová práca.

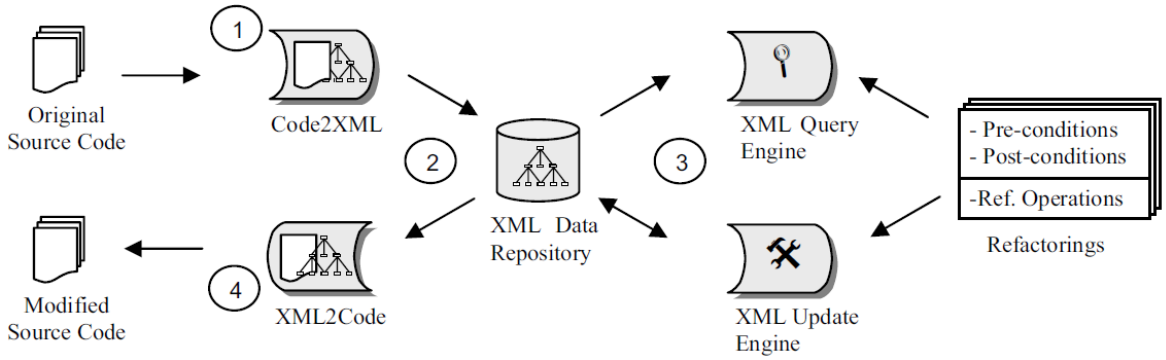
Autori riešenia navrhli nástroj na základe práce [8], pričom pre konverziu zdrojového kódu do XML reprezentácie využili nástroj *JavaML*, ktorý pracoval na princípe zdola nahor – mapovaní abstraktného syntaktického stromu do XML reprezentácie [13]. XML súbory boli ukladané v databáze a následné vyhľadávanie bolo riešené pomocou jazyka XQuery. Pre aktualizáciu uzlov bola využitá vtedy neoficiálna nadstavba XQuery, XUpdate.

Nástroj v skratke umožňoval nasledujúce činnosti:

- *Konverziu zdrojových súborov do XML reprezentácie*
- *Ukladanie XML reprezentácie v XML databáze*
- *Kontrolu a vykonávanie refaktorovacích operácii*
- *Konverziu XML súborov do zdrojového kódu*

Činnosť nástroja je zobrazená na *Obr. 9*, ktorý uviedli autori vo svojej práci. Ako možno vidieť, proces implementovaný v nástroji možno rozdeliť na 2 časti, s následným delením na konkrétne činnosti, ktoré je nutné vykonať a to:

- *Konverzia*
    - *Code2XML (1)* - pre konverziu do XML reprezentácie
    - *XML2Code (4)* – pre spätnú konverziu do zdrojového kódu
- *Refaktorovanie*
    - *XQuery (3)* – pre vyhľadávanie problémov v *XML repozitári (2)*
    - *XUpdate (3)* – pre aktualizáciu uzlov v *XML repozitári (2)*



Obr. 9 RefaX proces *[12]*

Nástroj, ktorý je výsledkom tejto diplomovej práce vychádza práve z tohto návrhu, ale pôvodný prístup rozširuje o expertný systém, pričom dôvody na jeho využitie sú popísané v *kapitole 2.5*.

3.2. JDeodorant[6]

Tento nástroj predstavuje zásuvný modul pre prostredie Eclipse. Nástroj dokáže automaticky vyhľadať 4 pachy v Java kóde. Napriek malému počtu vyhľadávaných pachov je však veľkou výhodou nástroja to, že ich dokáže aj opravovať. Dané pachy sú:

- *God class, Type checking, Long Method, Feature Envy*

30 augusta 2014 sa stal nástroj open-surce a jeho zdrojové kódu sú teraz voľne prístupné v GitHub úložisku[7].

[6] http://users.encs.concordia.ca/~nikolaos/jdeodorant/
[7] https://github.com/tsantalis/JDeodorant

3.3. PMD[8]

PMD je nástroj na statickú analýzu zdrojového kódu. Nástoj vyhľadáva základné problémy zanesené do kódu, ako napríklad nepoužité premenné, prázdne bloky kódu, zbytočné objekty a podobne. Nástroj je zameraný hlavne na programovací jazyk Java, no podporuje aj niektoré ďalšie jazyky, napríklad JavaScript, XML, alebo XSLT. Ako súčasť taktiež obsahuje *CPD – copy paste detector*, ktorý má podporu pre mnoho ďalších jazykov. PMD sa poskytuje ako otvorený softvér vo forme doplnkov pre mnohé vývojové prostredia.

Funkcionalita PMD je postavená na práci s abstraktným syntaktickým stromom, ale taktiež aj využívaní jazyka XML. Nástroj však umožňuje iba vyhľadávanie jednoduchých porušení pravidiel programovania, ale samotné refaktorovanie, ako aj vyhľadávanie pachov nie sú pokryté. Pravidlá pre vyhľadávanie sú zapísané pomocou XPath výrazov, ktoré si môže používateľ doplniť podľa potreby.

Ako príklad použitia poskytuje nástroj možnosť priamo ho vyskúšať na niektorom voľne dostupnom projekte na stránke *sourceforge.net*[9]. Nižšie, na *Obr. 10* je preto uvedený príklad výstupu nástroja po vyhľadávaní nad známym objektovo-relačným mapovačom *Hibernate*[10].

PMD report

Problems found

#	File	Line	Problem
1	hibernate/Environment.java	246	Avoid unused private fields such as 'jvmSupportsProxies'
2	hibernate/eg/Edge.java	59	Avoid unused private methods such as 'setKey(long)'
3	hibernate/eg/Edge.java	67	Avoid unused private methods such as 'setCreationDate(Date)'
4	hibernate/eg/Vertex.java	73	Avoid unused private methods such as 'setKey(long)'
5	hibernate/eg/Vertex.java	81	Avoid unused private methods such as 'setVersion(int)'
6	hibernate/eg/Vertex.java	89	Avoid unused private methods such as 'setCreationDate(Date)'
7	hibernate/engine/Cascades.java	206	Avoid unused private methods such as 'cascade(SessionImplementor,Object,Type,CascadingAction,int)'
8	hibernate/engine/Versioning.java	49	Avoid unused formal parameters such as 'versionType'
9	hibernate/engine/Versioning.java	53	Avoid unused formal parameters such as 'versionType'
10	hibernate/helpers/IdentityMap.java	68	Avoid unused formal parameters such as 'k'
11	hibernate/id/UUIDStringGenerator.java	55	Avoid unused private methods such as 'toString(int)'
12	hibernate/impl/CollectionPersister.java	271	Avoid unused private methods such as 'getSQLSelectString()'
13	hibernate/impl/QueryImpl.java	283	Avoid unused private methods such as 'guessType(Object)'
14	hibernate/impl/ScrollableResultsImpl.java	25	Avoid unused private fields such as 'single'
15	hibernate/impl/SessionFactoryImpl.java	90	Avoid unused private fields such as 'properties'

Obr. 10 Výstup nástroja PMD

[8] http://pmd.sourceforge.net/
[9] http://sourceforge.net/
[10] http://hibernate.org/orm/

Pravidlá definované v PMD však častokrát využívajú aj iné nástroje, ktorý celý proces vyhľadávania anti-vzorov a porušení pravidiel viac automatizujú a spohodlňujú. Príkladom takéhoto nástroja je *Sonar*, popísaný v nasledujúcej kapitole.

3.4. SonarQube[11]

Sonar predstavuje nástroj na celkovú správu kvality softvérového systému. Podľa dokumentácie pokrýva 7 oblastí kvality zdrojového kódu a to:

- *Architektúra a dizajn, Unit testy, Duplikácie, Zložitosť, Potenciálne chyby, Pravidlá programovania, Komentáre*

Sonar je veľmi vyspelý nástroj, ktorý dokáže pracovať s viac ako 25 programovacími jazykmi, pričom medzi najpoužívanejšie patria Java, C#, C++, C a podobne. Pre Sonar je dostupných taktiež množstvo doplnkov, ktoré ešte viac rozširujú jeho funkcionalitu. Sonar je dostupný ako webová aplikácia, ale taktiež aj ako doplnok pre známe vývojové prostredia.

Sonar vykonáva statickú analýzu zdrojového kódu hlavne pomocou metrík. Výsledku poskytuje používateľovi v prehľadnom zobrazení. Ako plugin môže byť použitý nástroj PMD, ktorý rozširuje funkcionalitu Sonaru o možnosti PMD.

3.5. Stench Blossom

Zaujímavý nástroj prestavuje detektor pachov nazvaný Stench Blossom, ktorý dokáže odhaliť 8 pachov v jazyku Java [14]:

- *Data Clupms, Feature Envy, Massage Chain, Switch Statement, Type Cast, Instance of, Long Method, Large Class*

Cieľom nástroja nie je poskytovať informácie o kvalite zdrojového kódu, ale iba vizualizovať prítomnosť daných pachov [15], aby programátor získal rýchly pohľad na problémy vo svojom kóde.

[11] http://www.sonarqube.org/

3.6. iPlasma

Nástoj na vyhľadávanie pachov v kóde vznikol ako výstup výskumnej práce. Cieľom nástroja je vykonávať základné časti statickej analýzy zdrojových kódov. Nástroj okrem toho však vyhľadáva aj niekoľko málo pachov kódu nazývaných disharmónie [14].

Nástroj slúži na analýzu kvality kódu na všetkých úrovniach – od nízkoúrovňového modelu až po analýzu založenú na metrikách [16]. Fungovanie nástroja je postavené na 4 princípoch a to:

- *Metriky* – vyhodnocovanie kvality na základe výsledkov
- *SAIL (Static Analysis Interrogative Language)* – autori definovali jazyk na štrukturálnu analýzu postavený na MEMORIA metamodeli, ktorý dokáže dostatočne reprezentovať C++ a Java systémy [16]
- *Pravidlá pre detekciu* – pravidlá, na základe ktorých možno odhaliť prítomnosť pachu k kóde
- *DUDE (Detection of Code Duplication)* – nástroj, ktorý umožňuje na základe porovnávania podobnosti textu nájsť duplicitný kód

Vďaka svojim vlastnostiam možno nástroj iPlasma považovať za veľmi dobrú vedeckú prácu v oblasti vyhľadávania anti-vzorov a pachov v kóde. Ďalším prínosom je celkový pohľad na statické vlastnosti kódu poskytnutý na základe viacerých prístupov k analýze.

Nedostatkom nástroja je však chýbajúca možnosť refaktorovania, ktoré je nutné spraviť už mimo daného nástroja.

3.7. InFusion

Posledným, zjavne veľmi vyspelým nástrojom, ktorý však nie je voľne dostupný je program InFusion. Podľa dokumentácie[12] dokáže odhaľovať veľké množstvo pachov a anti-vzorov, medzi ktoré patria:

- *Blob Class, Blob Module, Blob Operation*
- *Duplicated Code*
- *Data Class, Data Clumps, Data Module*
- *Distored Hierarchy*
- *Feature Envy*
- *God Class, God Module*
- *Intensive Coupling*
- *Message Chains*
- *Refused Bequest*
- *Schizophrenic Class, Schizophrenic Module*
- *Shotgun Surgery*
- *Tradition Breaker*
- *Underutilizer Interface*
- *Cyclic Dependency, Unstable Dependency, Stable Abstraction Breaker*

Systém okrem toho podporuje statickú analýzu a diagnostiku na úrovni zdrojového kódu, aj architektonickej úrovni. Nástroj vychádzal z nástroja *iPlasma*, popísaného v kapitole 3.6

[12] http://www.intooitus.com/products/infusion/detected-flaws

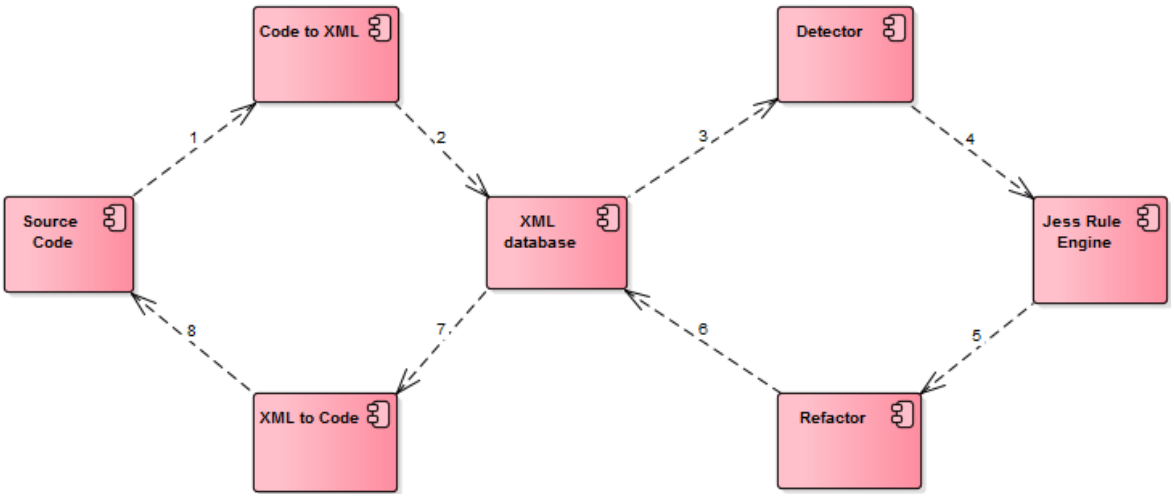
4. Návrh automatizovaného refaktorovacieho systému

Kapitola bližšie popisuje návrh samotného nástroja, ktorý predstavuje výstup tejto práce. Systém vychádza z nástroja, ktorý bol vytvorený ako súčasť predchádzajúcej práce [3], no niektoré základné koncepty sú vzhľadom na predchádzajúcu analýzu úplne prepracované. V rámci kapitoly je rozpracovaný návrh využitia jednotlivých konceptov, ktoré predstavujú súčasť automatizovaného refaktorovacieho nástroja. Kapitoly 5 a 6 následne rozoberajú ďalšie zaujímavé koncepty, ktorým je venovaná väčšia pozornosť. Krátka kapitola 7 sa následne zaoberá reálnou implementáciou daných konceptov do navrhovaného automatizovaného nástroja.

Po zohľadnení kapitol 2 a 3 je zjavné, že existuje veľa prístupov k automatizovaniu vyhľadávania, ale aj opravovaniu anti-vzorov a pachov v kóde. O použitom prístupe sa rozhoduje hneď v prvom bode procesu – reprezentácii zdrojového kódu. Vzhľadom na to, že táto práca skúma možnosti refaktorovania pomocou jazyka XML, nástroj je postavený práve na tejto reprezentácii. Táto časť zostala prakticky kompletne zachovaná z predchádzajúcej práce [3], preto nebude podrobne popisovaná.

Podstatnú zmenu však tvorí nástroj na vyhľadávanie a transformáciu XML dokumentov. Pre tento účel boli využité jazyky XPath, XQuery a XQuery Update pre aktualizáciu uzlov v databáze. XML databáza predstavuje ďalší koncept, ktorý bol využitý v navrhovanom nástroji. Databáza slúži ako úložisko projektových kódov v XML podobe. Vďaka jazyku XQuery je následná práca nad nimi veľmi jednoduchá. Vzhľadom na dobrú podporu jazyka Java pre dokumentové databázy ja nástroj na automatizovanie refaktorovania vytváraný práve v tomto jazyku. Na *Obr. 11* je zobrazený konceptuálny návrh nástroja, ktorý predstavuje výstup tejto práce. Základný princíp vychádza z práce [12], no celkovo je prístup obohatený o mnoho prvkov.

Dôležitým prvkom návrhu je expertný systém. Pomocou takéhoto systému môže používateľ ľahko definovať, akým spôsobom má automatizované refaktorovanie prebiehať. Používateľ môže takto určovať, akým spôsobom budú dané pachy opravované, či kedy sa má oprava vykonať a kedy nie. Tejto téme je bližšie venovaná kapitola 5. Expertný systém by taktiež mal uvažovať jednotlivé závislosti pachov kódu tak, aby bolo možné bez ďalších zásahov určiť vhodnú postupnosť aplikovania jednotlivých refaktorovacích operácií. Táto náročná činnosť nebola v prototype nástroja úplne implementovaná, no návrhu riešenia daného problému sa venuje kapitola 6.



Obr. 11 Komponenty navrhovaného nástroja

Ako možno vidieť na *Obr. 11*, činnosť systému je postavená na ôsmich základných krokoch a to:

1 – Konverzia zdrojového kódu do XML reprezentácie

2 – Import XML reprezentácie projektu do XML databázy

3 – Vyhľadanie pachov pomocou jazyka XPath a XQuery

4 – Príprava informácií o nájdených problémoch pre expertný systém

5 – Rozhodnutie o spôsobe a poradí opráv pomocou expertného systému

6 – Postupná aplikácia opráv pomocou XQuery a XQuery Upadate

7 – Export XML súborov z databázy

8 – Konverzia XML súborov späť do zdrojového kódu

V nasledujúcich podkapitolách sú postupne rozoberané dôležité body uvedeného procesu. Konkrétne je bližšie popísaný návrh využitia XML databázy v kapitole 4.1, vyhľadávanie pomocou jazyka XPath a návrh samotných XPath výrazov v kapitole 4.2, oprava pomocou jazyka XQuery a XQuery Update v kapitole 4.3 a návrh využitia samotného expertného systému v kapitole 4.4.

4.1. XML databáza, ako súčasť navrhovaného nástroja

XML dokumentové databázy, popísané v kapitole 2.4.1 tvoria výhodný spôsob pre prácu nad projektom v XML podobe. Nad takouto databázou možno vykonávať XQuery dopyty a aktualizácie pomocou jazyka XQuery Update. Práca nad celým projektom je jednoduchá, keďže jednotlivé súbory sú uložené v jednej kolekcii, nad ktorou sa vykonávajú XQuery dopyty ako nad jedným celkom.

V súčasnosti je pravdepodobne najvyspelejšou, voľne dostupnou natívnou XML databázou nástroj *BaseX*. Vzhľadom na svoje vlastnosti, aktívny vývoj (aktuálna verzia 8.4.4 bola vydaná 03. mája 2016) a širokú podporu XML technológií, ktorá je zhrnutá v tabuľke nižšie, bola práve táto databáza využitá ako jadro systému, ktorý predstavuje výstup tejto práce. Okrem XML technológií, ktoré sú dôležité pre túto prácu však databáza podporuje aj mnoho ďalších, ako napríklad indexovanie, HTTP protokol, JSON formát, kryptovanie SQL a ďalšie[13].

XML Technológia	*Verzia*	*Popis*
XPath	3.1	Dopytovací jazyk
XQuery	3.1	Dopytovanie a funkcionálne programovanie
XQuery Update	1.0	Aktualizácia uzlov v databáza
XQuery Full Text	1.0	Full-text vyhľadávanie v XML dokumentoch
XSLT	2.0	Šablóny pre výstup dopytu

Tabuľka 6 Podpora XML technológií zo strany BaseX

Práve podpora najnovšej verzie jazykov XPath, XQuery a XQuery Update je hlavný dôvod, prečo je pri návrhu architektúry systému dobré zvoliť práve BaseX XML databázu. BaseX poskytuje ako jediná voľne dostupná XML databáza širokú podporu najnovších verzií XML technológií, ako XPath 3, či XQuery 3.

Ďalšou podstatnou výhodou použitia BaseX XML dokumentovej databázy je podpora zo strany rôznych API pre prístup k databáze. BaseX podporuje všetky používané

[13] http://basex.org/products/xquery/

formy prístupu, ktoré XML databázy bežne implementujú. V podstate existujú 4 používané rozhrania BaseX databázy:

- *XML:DB API*[14] – toto API predstavovalo snahu o štandardizáciu technológie pre prístup a ovládanie XML databáz. Vývoj XML:DB bol aktívny v čase, keď sa o XML databázach uvažovalo ako o novej, progresívnej technológii s veľkým potenciálom do budúcnosti. Keďže XML databázy sa nepresadili tak ako sa očakávalo, vývoj XML:DB bol pozastavený a postupne úplne zrušený. V súčasnosti je toto API nepodporované a nemalo by sa ďalej používať.
- *Client API*[15] – Predstavuje základné API, o ktorého vývoj sa starajú vývojári BaseX. Toto API poskytuje implementácie rozhrania databázy pre rôzne jazyky, napríklad Java, C++, C#, Python, Ryby, atď.
- *XQJ API*[16] – XQJ predstavuje Java implementáciu rozhrania na rôzne XML databázy. XQJ je dostupné iba pre jazyk Java, no snaží sa čo najviac zovšeobecniť prístup k jednotlivým XML databázam, preto je ho dobré použiť vždy pri potrebe prístupu k niektorej XML databáze z jazyka Java.
- *REST API*[17] – BaseX databáza umožňuje taktiež prístup pomocou restfull rozhrania. K databáze sa tak pristupuje pomocou URL požiadaviek protokolu HTTP, teda sú použité klasické HTTP metódy - GET, POST, či PUT.

Vzhľadom na snahu o čo najväčšie zjednodušenie a štandardizované použitie je v navrhovanom nástroji, ktorý je implementovaný v programovacom jazyku Java vhodné využiť práve XQJ rozhranie. Pri takomto API by v prípade potreby výmeny databázy bolo nutné vykonať iba minimálne zásahy do zdrojového kódu.

BaseX databáza pozostáva z 3 rôznych komponentov a to:

- *BaseXServer* – databázový server
- *BaseXClient* – klient, ktorý sa pripája na databázový server
- *BaseXGUI* – grafické rozhranie databázy

Pri využití BaseX, ako nástroja na organizáciu projektu, je tak nutné myslieť na to, že na príslušnej stanici musí byť spustený databázový server.

[14] http://xmldb-org.sourceforge.net/
[15] http://docs.basex.org/wiki/Clients
[16] http://xqj.net/
[17] http://docs.basex.org/wiki/REST_API

4.2. Vyhľadávanie pachov kódu pomocou jazyka XPath a XQuery

Jazyk XQuery predstavuje pravdepodobne najvhodnejší jazyk pre vyhľadávanie a súčasne opravovanie pachov zdrojového kódu nad XML reprezentáciou. Na samotné vyhľadávanie plne postačuje jazyk XPath [3], keďže je ale XPath súčasť jazyka XQuery, tak pri vyhľadávaní pomocou jazyka XQuery ide vlastne o nepriame použitie jazyka XPath.

Koncept navrhovaného nástroja je postavený na oddelených krokoch vyhľadávania a opravovania. Nástroj preto musí udržiavať oddelené skripty pre vyhľadávanie a opravovanie. Pre samotné vyhľadávanie pachov a anti-vzorov v kóde tak možno vytvoriť množinu XPath výrazov, ktoré dokážu jednotlivé problémy identifikovať. *Tabuľka 7* uvádza niektoré XPath výrazy, ktoré boli navrhnuté v rámci tejto, ale aj predchádzajúcej práce, pre identifikáciu pachov v kóde [3] [17].

Pach kódu / anti-vzor	**XPath identifikačný výraz**
Long Method	//function[count(block/child::node())>15]
Large Class	//class[count(//function)>20]
Long Parameter List	//function[count(parameter_list/parameter)>5]
Feature Envy	//function[count(block//expr_stmt[count(expr/call/name/name)=2 and expr/call/name/name/text()!='this'])>count(block//expr_stmt[count(expr/call/name/name)=0 or expr/call/name/name/text()='this'])]
Switch Statements	//switch
Lazy Class	//class[count(//function/name/text()[not(starts-with(., 'get')) and not(starts-with(., 'set'))]) <= 1]
Message Chains	//expr_stmt[count(expr/call)>4]
Data Class	//class[count(//function/name/text()[not(starts-with(., 'get')) and not(starts-with(., 'set'))]) = 0 and count(block/decl_stmt) > 0]
Catch and Rethrow	//try[catch/parameter_list/parameter/decl/name/text()=catch/block/throw/expr/name/text()]
Empty Catch Clausule	//catch[count(block/child::node()) <= 1]

Tabuľka 7 XPath výrazy pre vyhľadávanie pachov a anti-vzorov v kóde

V uvedenej tabuľke možno vidieť, že pre samotnú identifikáciu pachov možno použiť pomerne jednoduché XPath výrazy. Je však nutné poznamenať, že samotné vyhľadanie pachu nie je dostatočné.

Vyhľadávanie musí byť realizované tak, že miesto výskytu pachu kódu je označené tak, že pri oprave ho je možné jednoznačne identifikovať. Keďže ide o zdrojový kód, ktorý je poskytovaný v XML formáte, označenie konkrétneho miesta v kóde je možné realizovať veľmi jednoduchým spôsobom tak, že daná časť kódu prestavujúca pach v kóde je vložená do ďalšieho uzla s jednoznačným označením, ktoré zároveň musí identifikovať o aký pach ide. Pre označenie konkrétneho miesta v kóde tak možno uviesť kódy jednotlivých pachov, napríklad *LPL – long parameter list*. Kód tak následne možno označovať pomocou XML značiek, napríklad LPL1, čo identifikuje prvú nájdenú inštanciu pachu *long parameter list*.

Ďalšou dôležitou vlastnosťou, ktorú musí vyhľadávací skript zabezpečovať, je poskytnutie informácií o kontexte nájdeného problému, to znamená dodatočné informácie, napríklad kde presne sa daný problém nachádza, v akej stromovej štruktúre sa nachádza a podobne. Toto tvorí pomerne výrazný technický problém, ktorý je možné riešiť využitím zápisu informácii priamo z XQuery skriptu do externého súboru, z ktorého budú informácie nasledovne spracované pred rozhodovaním expertného systému Jess.

Práve pre jednoznačnú identifikáciu časti kódu, ale aj pre vizualizáciu pachov v kóde pre používateľa, či zápis informácií, potrebných pre rozhodovanie expertného systému Jess je zjavné, že samotný jazyk XPath je nedostatočný, keďže je nutné manipulovať aj so samotným XML dokumentom. Práve z tohto dôvodu je dobré využiť jazyk XQuery aj pri procese vyhľadávania pachov v kóde. Pomocou XQuery sa tak do dokumentu zanesú nové značky, ktoré identifikujú pach v kóde, prípadne komentáre, slúžiace pre vizualizáciu pachu používateľovi. XQuery, ktoré umožňuje taktiež jednoduchú manipuláciu a zápis do súborov v súborovom systéme počítača je možné využiť aj pre zaznamenanie kontextu o danom pachu, teda napríklad zápis, aké iné pachy sa nachádzajú na danom mieste, či súvisiace triede, alebo veľkosť problému, od ktorej sa môže odvíjať následné refaktorovanie.

XPath výrazy poskytnuté v *tabuľke 7* tvoria základ procesu vyhľadávania pachov. Napriek tomu však reálne skripty pre vyhľadávanie musia obsahovať aj veľa dodatočnej funkcionality. Keďže ide o implementačné záležitosti, bližšie podrobnosti možno nájsť v kapitole 7.

4.2.1. Vizualizácia identifikovaných pachov

Identifikácia pachov predstavuje jednu zo základných častí procesu refaktorovania. Po identifikácií je však potrebné dané pachy určitým spôsobom vizualizovať. Táto vizualizácia je dôležitá preto, aby používateľ nástroja dokázal identifikovať nájdené pachy v kóde. O aký pach išlo, je možné odhadnúť aj z vykonanej opravy, no táto metóda neposkytuje dostatočný priestor pre identifikáciu kvality opravy, keďže dôvod vykonanej opravy nie je vždy možné dostatočne určiť.

Navrhovaný nástroj musí dokázať nájdené pachy dostatočne dobre vizualizovať. Najjednoduchší spôsob vizualizácie predstavuje zachovanie pôvodného kódu, pričom do takéhoto kódu budú pridané komentáre identifikujúce pachy kódu. Tento spôsob je pomerne jednoduchý a ľahko dosiahnuteľný pomocou jazyka XQuery (kapitola 4.2). Zaujímavejšou sa javí možnosť špeciálne zvýrazňovať časti kódu, v ktorých boli identifikované pachy kódu. Existuje niekoľko nástrojov, ktoré takéto zvýrazňovanie dokážu vykonávať. Príkladom je nástroj *Code Review*[18] vyvíjaný n Fakulte Informatiky a Informačných Technológií STU.

Nástroj zvýrazňuje časti zdrojového kódu na základe vložených značiek. Tieto značky môžu byť jednoducho vkladané v tvare formátovaného komentáru. Takéto značky možno vkladať v pároch, teda začiatočná a koncová značka, alebo iba vo forme začiatočnej značky, pričom je označený celý nasledujúci blok kódu. Nástroj umožňuje zvýrazňovať rôzne typy značiek, napríklad – CODEREVIEW, TODO, FIXME, REFACTOR a podobne. Práve poslednú značku REFACTOR možno použiť na označenie pachov v kóde, ktoré boli odhalené a je nutné ich refaktorovať. Vložené značky tak môžu vyzerať nasledovne:

//REFACTOR: Long Parameter List

Automatizovaný refaktorovací nástroj bude práve preto pachy označovať pomocou značiek, ktorých formát je definovaný nástrojom *Code Review* tak, aby bolo možné nájdené pachy vizualizovať pomocou tohto nástroja.

[18] http://147.175.149.150:8080/CodeReview

4.3. Refaktorovanie pomocou jazyka XQuery a XQuery Update

XQuery a XQuery Update tvoria vhodnú kombináciu technológií pre vykonávanie automatizovaného refaktorovania nad XML reprezentáciou. Pomocou jazyka XQuery možno pachy v kóde vyhľadávať a pomocou kombinácie XQuery a XQuery Update je ich možné opravovať.

Oproti jazyku XSLT, ktorý bol použitý ako nástroj na automatizované refaktorovanie v predchádzajúcej práci [3], je písanie refaktorovacích pravidiel v XQuery oveľa intuitívnejšie, keďže viac pripomína klasické programovacie jazyky, zatiaľ čo XSLT je postavené na princípe transformačných šablón. Jednoduchšie pachy kódu a anti-vzory možno častokrát opraviť pomocou krátkeho skriptu, no náročnejšie pachy kódu, hlavne z výberu Martina Fowlera vyžadujú neraz zložitý prístup a kreativitu.

Práve z tohto dôvodu je vhodné nepísať skripty úzko zamerané na jednotlivé pachy a anti-vzory, ale vytvoriť súbor jednoduchších refaktorovacích pravidiel, so základom z práce Martina Fowlera, ktorý okrem samotných pachov definoval aj veľké množstvo refaktorovacích operácií [1] . Tieto operácie, definované ako *„introduce parameter object“*, či *„move method“* je následne možné kombinovať pri oprave určitého pachu.

Riešením určitého pachu preto nemusí byť aplikácia práve jedného refaktorovacieho skriptu, ale opravu môže predstavovať aj postupná aplikácia dvoch, prípadne viacerých skriptov.

Typickým príkladom je porovnanie niektorých z riešení problémov *„Long parameter list“* a *„Feature envy“*

Long parameter list - jedným z možných riešení je vytvoriť objekt pre parametre:

Krok 1: Introduce parameter object

Feature envy – jednou možnou opravou je vyňať časť metódy a tú presunúť do potrebnej triedy

Krok 1. Extract method

Krok 2: Move method

Keďže jeden pach môže mať viacero riešení a tie môžu pozostávať z jedného alebo viacerých krokov, je práve rozhodovanie o spôsobe a postupe opravy vhodnou úlohou pre expertný systém. V takomto prípade budú pre každý nájdených pach výstupom usporiadané refaktorovacie pravidlá, ktoré budú v stanovenom poradí aplikované, aby sa dosiahla oprava daného pachu.

Nižšie sú postupne rozobraté tri refaktorovacie postupy pre anti-vzory a pachy kódu, pre ktoré boli uvedené vyhľadávacie prístupy v predchádzajúcej kapitole. Keďže vyhľadávanie označuje v XML súboroch pach do vlastných značiek, je potrebné sa pri oprave dopytovať priamo na konkrétnu značku, ktorá sa dosadzuje pri volaní skriptu.

Remove Exception Throw

Remove Exception Throw predstavuje autorom definovanú refaktorovaciu operáciu, ktorá odstraňuje vyhodenie výnimky. Túto jednoduchú operáciu je vhodné použiť, keď je zachytená výnimka opätovne vyhadzovaná, teda napríklad pri anti-vzore *„Catch and Rethrow“*. Tento anti-vzor je opravený tak, že zbytočný výraz, ktorý je zodpovedný za opätovné vyhodenie výnimky je odstránený (*Obr. 12*)

```
declare variable $tag external := "CR1";

for $node in xquery:eval(fn:concat("//", $tag))
return
   delete node $node//throw
```

Obr. 12 Refaktorovacia operácia *Remove Exception Throw*

Ako možno vidieť, refaktorovacia operácia je pomerne jednoduchá. Začiatok tvorí deklarovanie externej premennej, ktorá identifikuje konkrétne miesto v XML súbore, na ktorom sa nachádza pach. Pre vyhýbanie sa chybám môže mať premenná prednastavenú určitú hodnotu, v tomto prípade kód *„CR1“*, čo predstavuje prvý nájdený anti-vzor *„catch and rethrow“*. Následne sa pre konkrétne miesto, identifikované na základe poskytnutého kódu vykonáva samotný refaktoring – odstránenie *„throw“* výrazu.

Aj takto jednoduchý anti-vzor môže mať viacero riešení. Podobne by bolo možné použiť operáciu, ktorá neodstraňuje vyhodenie výnimky, ale naopak, odstraňuje zachytenie výnimky na danom mieste. Pre ilustráciu funkcionality refaktorovacích skriptov je však rozhodovanie o použitom refaktoringu vynechané a je možné si ho pozrieť v kapitole 4.4.

Log Exception

Táto refaktorovacia operácia zabezpečuje zaznamenanie výnimky na danom mieste. Operáciu je možné s výhodou použiť na jednoduchý anti-vzor *„Catch and Ignore“*, stručne popísaný v predchádzajúcej kapitole. Daný anti-vzor je možné opraviť tak, že do *„catch“* bloku je pridaný výraz, ktorý danú výnimku zaznamená (*Obr. 13* nižšie)

```
declare variable $tag external := "ECC1";

for $node in xquery:eval(fn:concat("//", fn:concat($tag, "//catch")))
let $exceptionName := $node/parameter_list/parameter/decl/name/text()
return
  replace node $node/block with
<block>&#123;
  <expr_stmt><expr><call>
     <name>{$exceptionName}.<name>printStackTrace</name></name><argument_
list>()</argument_list></call></expr>;
  </expr_stmt>&#125;</block>
```

Obr. 13 Refaktorovacia operácia *Log Exception*

Oproti predchádzajúcemu príkladu ide o mierne náročnejšiu operáciu. Refaktorovacia operácia na *Obr. 12* predstavovala iba jednoduché odstránenie uzla, pričom refaktorovacia operácia na *Obr. 13* nahrádza prázdny „*block“* novým uzlom, ktorý obsahuje predstavuje zaznamenanie výnimky.

Problémom tohto refaktorovania je to, že pridáva nový kód a tak pri snahe aspoň do istej miery zachovať formátovanie zdrojového kódu sa skript stáva mierne neprehľadným. Toto však nie je možné riešiť iným spôsobom, keďže biele znaky uvedené pri zápise skriptu do XML časti sa reálne prenesú do výsledného zdrojového kódu.

Podobne, ako pri predchádzajúcom príklade by bolo možné použiť viacero refaktorovacích operácií, napríklad aj odstránenie *„try-catch“* bloku, prípadne vyhodenie výnimky, čo by ale v tomto prípade práve spôsobilo anti-vzor „*Catch and Rethrow“*.

Introduce Parameter Object

Táto operácia patrí do skupiny refaktorovacích operácií definovaných Martinom Fowlerom. Je často používaná pri viacerých pachoch (bližšie v kapitole 5.4), no typickým príkladom jej využitia je oprava pachu „*Long Parameter List*“.

Refaktorovanie tohoto pachu kódu je pomerne náročné na automatizovanie. Asi najjednoduchší spôsob opravy predstavuje vytvorenie novej triedy, ktorá bude predstavovať dátový objekt, ktorého atribúty budú jednotlivé prvky zo zoznamu parametrov (týmto však pravdepodobne vznikne pach *dátová trieda*). Daný zoznam je potom potrebné nahradiť práve takýmto objektom. Následne je nutné upraviť volania tak, aby sa odkazovali na túto triedu. Na *Obr. 14* je táto refaktorovacia operácia znázornená v zjednodušenej podobe.

```
declare variable $tag external := "LPL1";

(for $node in xquery:eval(fn:concat("//", $tag, "//parameter_list"))
    for $callNode in $node/../block//name[text() = $node/parameter/decl/name
]
      return
        insert node <name><name>parameterObject</name>.</name>
        before $callNode),

for $node in xquery:eval(fn:concat("//", $tag, "//parameter_list"))
  return
    insert node
      <class>
        <specifier>private </specifier>class <name>ParameterObject </name>
        <block>&#123;
          {for $node2 in $node/parameter
          return $node2};
        &#125;</block>
      </class>
      into $node/../../../..,

for $node in xquery:eval(fn:concat("//", $tag, "//parameter_list"))
  return
    replace node $node with
       <parameter_list>(<parameter>
            <decl>
              <type>
                <name>ParameterObject</name>
              </type> <name>parameterObject</name>
            </decl>
          </parameter>
        )</parameter_list>
```

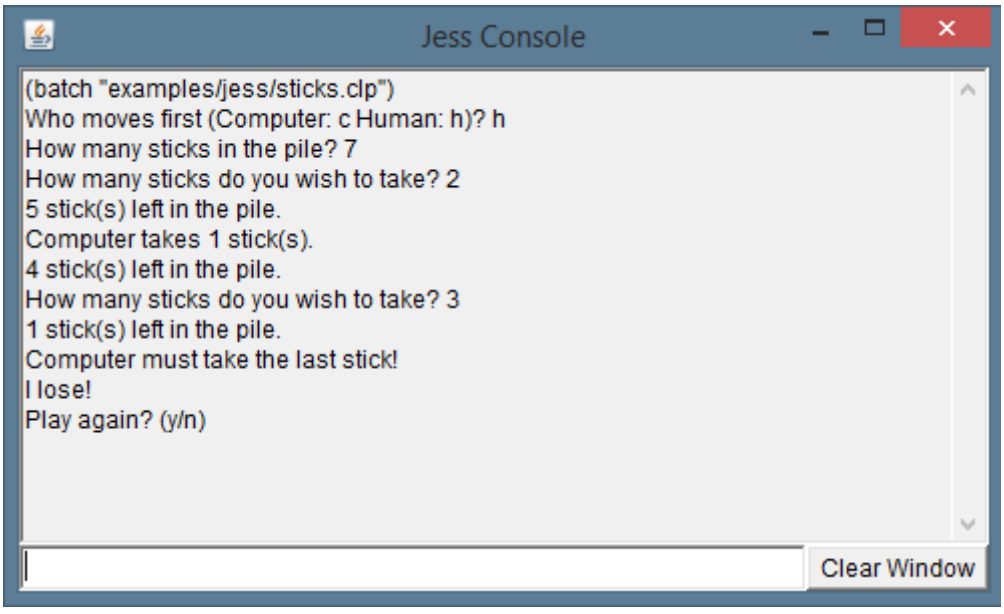
Obr. 14 Refaktorovacia operácia *Introduce Parameter Object*

Ako možno vidieť, už pri mierne náročnejšej operácii, zložitosť skriptu rapídne narastá. Je to spôsobené tým, že operácia sa skladá z viacerých krokov a to – nahradenie volaní parametrov v metóde, vytvorenie triedy pre objekt parametrov a úprava parametrov metódy.

4.4. Expertný systém - náhrada experta pri refaktorovaní

Okrem samotných vyhľadávacích a refaktorovacích skriptov je potrebné riadiť proces ich aplikovania. Na základe kapitoly 2.5 možno povedať, že najvhodnejším spôsobom, ako zabezpečiť prispôsobivé riadenie automatizovaného spôsobu refaktorovania softvérových systémov je expertný systém. Takýto systém bude obsahovať bázu znalostí definovanú „expertom“ na refaktorovanie, ktorý dokáže rozoznať vzťahy medzi jednotlivými pachmi kódu. Systém bude následne rozhodovať o tom, kedy aplikovať aké refaktorovanie a podobne.

Vzhľadom na aktívnu podporu, stály vývoj a implementáciu v jazyku Java je pravdepodobne najvhodnejšou voľbou postaviť expertný systém pomocou nástroja *Jess*. Jess je pravidlový systém pre jazyk Java, ktorý vychádza z jazyka *CLIPS*. Nástroj umožňuje vytvárať expertné systémy pomocou logického programovania. Keďže však nejde o voľne dostupný nástroj, aktuálne je použitá trial verzia produktu vo verzii 7.1p2. Napriek tomuto nedostatku je však nástroj, vzhľadom na iné výhody, použitý aj pri implementácii.



Obr. 15 Jess konzola

Nástroj je možné použiť dvomi spôsobmi. Ako možno vidieť na *Obr. 15*, systém poskytuje rozhranie na prácu s pravidlovými systémami napísanými v jazyku Jess. Súčasťou distribúcie sú však aj knižnice, ktoré možno využiť priamo v Java programe pre prácu s expertným systémom [18]. Jess tak môže byť priamo vnorený v akejkoľvek Java aplikácií, napríklad v systéme, ktorý tvorí výstup tejto práce. Takáto integrácia do aplikácie napísanej

v jazyku Java funguje na základné vytvárania pravidlového stroja pomocou Java Jess API. Inštancii pravidlového stroja je následne poskytnutý zoznam faktov a potom sú na základe definovaných pravidiel odvodené výstupy.

Rozhodovanie pomocou nástroja Jess možno po každom vyhľadaní a označení pachov v kóde. Na základe týchto označení budú vytvorené objekty typu *„JessInput“,* ktoré slúžia ako vstup pre expertné rozhodovanie. Takýto objekt má nastavené všetky potrebné atribúty pre rozhodovanie o konkrétnom type pachu. Napríklad poskytuje záznam o hierarchii, kde sa daný pach nachádza, či o dĺžke, pokiaľ sa jedná o pach závislý na dĺžke parametrov, metódy, či triedy.

Výstupom rozhodovania je následne usporiadaný zoznam refaktorovacích operácií, ktoré sa vykonajú nad systémom v XML reprezentácií uloženej v databáze. Tento výstup predstavuje usporiadaný zoznam objektov type *„JessOutput“*, ktoré obsahujú informáciu o konkrétnom mieste, kde je potrebné opravu vykonať, a o jednej refaktorovacej operácií, ktorú je potrebné vykonať.

Samotné pravidlá sa skladajú z niekoľkých častí. Prvú časť tvorí spracovanie pravidla na základe podmienok, pričom prvotná podmienka je príslušnosť pravidla ku pachu podľa kódu. Následne sa vyhodnocujú dodatočné podmienky, ako napríklad, či sa na danom mieste nenachádza iný pach kódu, ktorý by mal byť riešený prioritne. Poslednou časťou je vytvorenie „*JessOutput“* objektu, prípadne viacerých objektov, ktoré predstavujú stanovené refaktorovacie metódy pre riešenie daného problému. Príklady niekoľkých Jess pravidiel možno vidieť na nasledujúcich obrázkoch.

```
(defrule empty-catch-clausule
    "Decision about refactoring for Empty Catch Clausule anti-pattern"
    ?o <-(JessInput {refCode == "ECC"})
    =>
    (add (new JessOutput ?o.code "LE")))
```

Obr. 16 Jednoduché pravidlo - pach -> jedna refaktorovacia operácia

Pravidlo na *Obr. 16* reprezentuje najjednoduchší typ pravidla. Ide o pravidlo, ktorého „pravá strana“ sa vykoná pri každom identifikovanom probléme s daným kódom – v tomto prípade kódom ECC, čo reprezentuje anti-vzor *Empty Catch Clausule*. Inak nepodmieneným výstupom je objekt *JessOutput*, ktorý stanovuje jedinú refaktorovaciu operáciu – *Log Exception*, ako riešenie problému.

```
(defrule inapropriate-intimacy
    "Decision about refactoring for Inapropriate Intimacy code smell"
    ?o <-(JessInput {refCode == "II"})
    =>
    (add (new JessOutput ?o.code "MM"))
    (add (new JessOutput ?o.code "MF")))
```

Obr. 17 Náročnejšie pravidlo - pach -> usporiadaná postupnosť refaktorovacích operácií

Kód na *Obr. 17* predstavuje pravidlo, ktoré nestanovuje jednu, ale dve refaktorovacie operácie, ako riešenie problému. Operácie sa pridávajú usporiadané, čo znamená, že sa vykonajú v stanovenom poradí, čo je veľmi dôležité pri vykonávaní refaktorovania konkrétneho pachu.

```
(defrule catch-and-rethrow1
    "Catch and Rethrow refactoring decision with context checking"
    ?o <-(JessInput {refCode == "CR"})
    (JessInput {size > 1})
    =>
    (add (new JessOutput ?o.code "RET")))

(defrule catch-and-rethrow2
    "Catch and Rethrow refactoring decision with context checking"
    ?o <-(JessInput {refCode == "CR"})
    (JessInput {size == 1})
    =>
    (add (new JessOutput ?o.code "RTC")))
```

Obr. 18 Pravidlá sledujúce kontext problému

Obr. 18 reprezentuje združené pravidlá, ktoré berú do úvahy taktiež kontext problému, tedači je možné odstrániť vyhodenie výnimky – *Remowe Exception Throw (RET)*, alebo je vhodné odstrániť celkové zachytávanie výnimky *(RTC)* tak, aby nevznikol ďalší anti-vzor *Empty Catch Clausule.* Vyhodenie výnimky je v tomto prípade použité iba vtedy, ak catch blok obsahuje okrem samotného vyhodenia výnimky aj iné operácie, teda jeho veľkosť je väčšia, ako 1. Naopak, ak catch blok obsahuje iba samotné vyhodenie výnimky, použije sa rafaktorovacia operácia pre odstránenie celého try-catch bloku.

Dôvody, ciele a funkcionalita expertného systému Jess predstavujú najväčší prínos tejto práce. Nasledujúca kapitola dôsledne identifikuje dôvody, prečo je expertné rozhodovanie dôležité, identifikuje pozitívne aj negatívne závislosti medzi pachmi kódu a vzťahy pachov a refaktorovacích operácií, na základe čoho sa odvíja ďalšia práca.

Informácie o integrácii a využití systému v automatizovanom refaktorovacom nástroji možno nájsť v kapitole 7, ktorá sa bližšie venuje samotným aspektom implementácie systému.

5. Závislosti pachov kódu

Navrhovaný systém, ktorý dokáže automatizovane rozoznávať a opravovať jednotlivé pachy v kóde je potrebné postaviť tak, aby umožňoval uvažovať taktiež závislosti jednotlivých pachov, čo je dôležité z pohľadu výkonu, ale aj optimálnosti výsledku.

Pokiaľ ide o *pozitívne* závislosti – teda, že opravením jedného pachu môže zaniknúť aj viacero ďalších pachov na danom, prípadne aj inom mieste, je potrebné uvažovať, že najvýhodnejšiu opravu je ideálne vykonávať čo najskôr, aby sa predišlo zbytočným, viacnásobným zásahom do kódu, či zmenám, ktoré v konečnom dôsledku neboli potrebné.

Negatívne závislosti, kedy pri oprave pachu môže vzniknúť iný pach, prípadne niekoľko ďalších je taktiež potrebné zahrnúť do rozhodovania o oprave. Niekedy, kedy pri aplikovaní opravy na nejaký pach vzniká iný pach, je vhodnejšie akceptovať skôr pôvodný pach než vznikajúci, prípadne uvažovať o inom spôsobe opravy. Ak existuje pri refaktorovaní situácia, kedy je pravdepodobné, že opravou jedného pachu bude vytvorený iný pach, je potrebné na základe celkového kontextu systému rozhodnúť o tom, či oprava bude vôbec vykonaná, alebo kód ostane zachovaný bez zmeny.

Tieto závislosti - pozitívne aj negatívne, je potrebné identifikovať tak, aby na základe definovaných vzťahov mohla byť plne využitá sila expertného systému pri rozhodovaní o optimálnom poradí aplikovania refaktorovacích operácií. Napriek tomu, že mnohokrát dokáže o dobrom prístupe rozhodnúť iba ľudský expert, predstavuje identifikácia vzťahov medzi pachmi a vytvorenie príslušných pravidiel dobrý základ pre systém umožňujúci pokročilý, automatizovaný refaktoring celého, komplexného systému.

Táto kapitola identifikuje vzťahy medzi jednotlivými pachmi, ktoré definoval Martin Fowler. Napriek tomu, že v konečnom dôsledku existuje oveľa väčšie množstvo anti-vzorov, či pachov, tato práca analyzuje podrobnejšie vzťahy iba medzi základnými, najznámejšími 22 pachmi kódu. Nasledujúce podkapitoly najskôr rozdeľujú pachy do viacerých tried pachov, následne sú analyzované negatívne a pozitívne závislosti jednotlivých pachov. Rafaktorovacím metódam jednotlivých pachov a ich dôsledkom sa následne venuje ďalšia podkapitola. V závere je naznačený prechod od závislostí pachov na inú, podrobnejšiu úroveň a to úroveň zachytávajúcu vzťahy medzi pachmi rozložené podľa ich refaktorovacích operácií.

5.1. Triedy pachov

Pri identifikácií vzťahov sa táto práca zaoberá iba 22 základnými pachmi kódu, ktoré patria medzi najznámejšie. Pokiaľ je však nutné identifikovať jednotlivé závislosti medzi nimi, vzniká pomerne náročná sieť vzťahov, ktorá je málo prehľadná a náchylná na nezrovnalosti.

Prvotné zlepšenie prehľadu identifikovaných vzťahov medzi pachmi kódu je možno dosiahnuť tak, že vzťahy budú striktne rozdeľované podľa toho, či ide o negatívny vzťah – teda opravou daného pachu môže vzniknúť iný pach, alebo pozitívny vzťah, kedy opravou pachu môže byť opravený aj iný pach. Práve pre takéto rozdelenie boli definované stereotypy vzťahov a to:

- Stereotyp *„cause“* – pre negatívne vzťahy
- Stereotyp *„solve“* – pre pozitívne vzťahy

Napriek takémuto rozdeleniu však stále vzniká množstvo neprehľadných vzťahov, pokiaľ chceme poskytnúť všetky pozitívne, alebo negatívne vzťahy v jedinom pohľade.

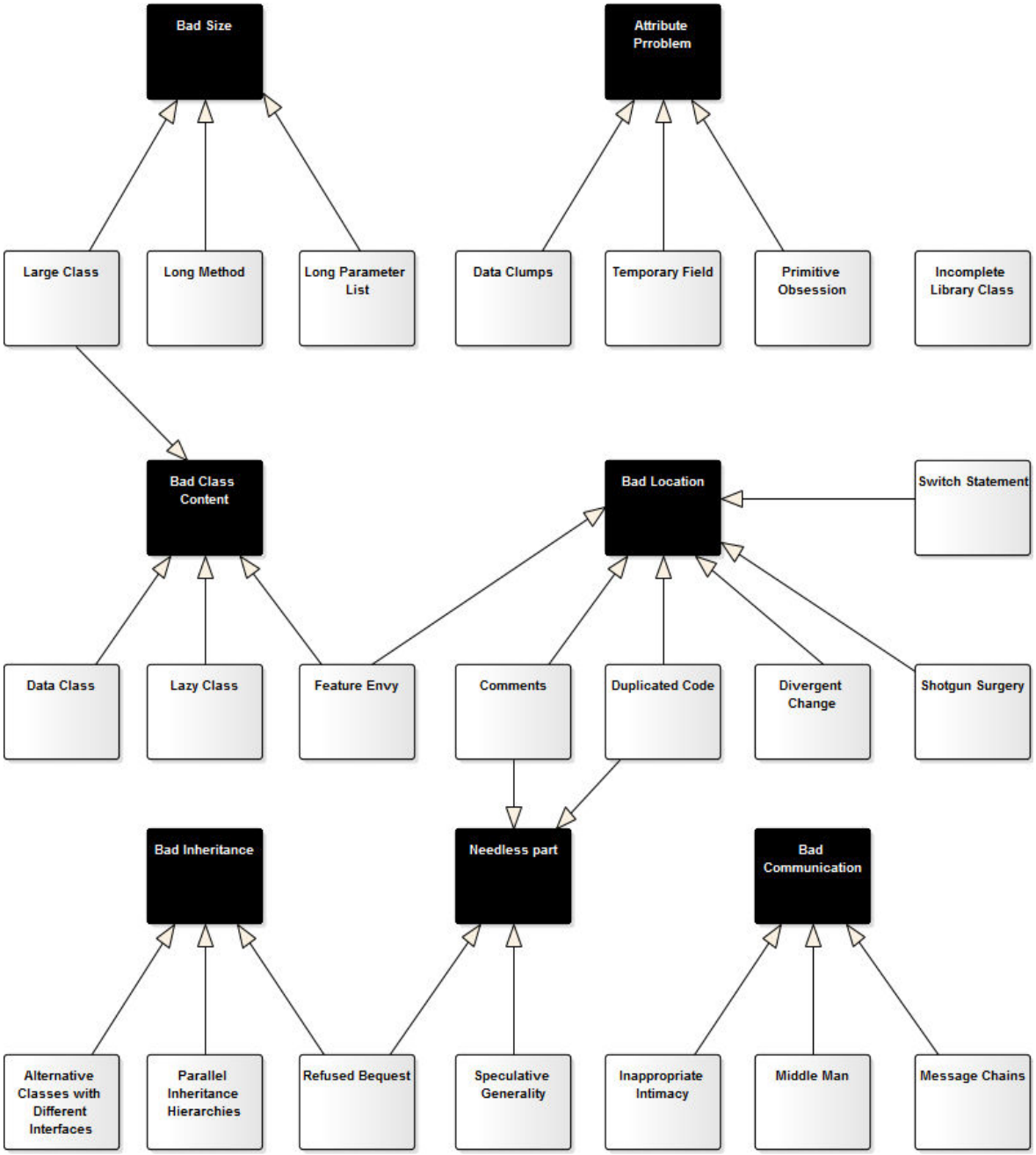
Čiastočné riešenie predstavuje vytvorenie akýchsi nadtried pachov. Mnohé pachy majú podobné črty, kedy ide o zlé umiestnenie časti kódu, nie najlepší spôsob komunikácie medzi triedami, alebo problém predstavuje dĺžka kódu. Takto identifikované spoločné vlastnosti je možné použiť pri vytváraní komplexných, no stále prehľadných, pohľadov na vzťahy medzi jednotlivými pachmi. V tejto práci bolo identifikovaných 7 tried pachov, ktoré sú popísané v *tabuľke 8*.

Keďže však jednotlivé pachy mnohokrát vykazujú črty viacerých skupín pachov, je potrebné ich zaradiť do viacerých tried, teda povoliť viacnásobné dedenie. Toto v konečnom dôsledku však nepredstavuje problém, práve naopak.

Vzniknuté nadtriedy následne slúžia na zjednodušovanie vzťahov medzi pachmi, či už pozitívnych, alebo negatívnych. Vzťah, ktorý platí pre celú triedu pachov sa tak môže viazať na príslušnú nadtriedu a naopak. Pomocou modelovacieho jazyka UML je možné vytvoriť náhľad tried pachov. Takýto diagram tried pachov (vytvorený ako UML Class diagram) je zobrazený na nasledujúcom obrázku uvedenom nižšie.

Názov	*Meno nadtriedy*	*popis*
Zlá veľkosť	Bad Size	Problém predstavuje veľkosť metódy, triedy a podobne
Zlé umiestnenie	Bad Location	Daná časť kódu by sa nemala nachádzať tam, kde sa aktuálne nachádza
Zlý obsah triedy	Bad Class Content	Trieda neobsahuje funkcionalitu, ktorú by mala obsahovať, Ide o problém, kedy triedam chýbajú metódy a podobne
Nesprávne dedenie	Bad Inheritance	Rôzne problémy s dedením. Koncept dedenia je nesprávne použitý, dedenie je nezmyselné a podobne
Nepotrebná časť	Needless Part	V kóde sa nachádzajú rôzne časti, ktoré je možné odstrániť. Takýmto úmyselným či neúmyselným zanášaním vzniká zbytočná neprehľadnosť kódu
Problém s atribútom	Attribute Problem	Problém so spôsobom práce s atribútmi triedy. Atribúty sú využívané neefektívnym spôsobom, alebo používané na činnosti, ktoré možno riešiť iným, lepším spôsobom
Problémová komunikácia	Bad Communication	Zlá, prípadne komplikovaná komunikácia medzi triedami. Dlhá, alebo zbytočne sprostredkovaná komunikácia

Tabuľka 8 Identifikované nadtriedy pachov kódu



Obr. 19 Nadtriedy pachov

Okrem vyššie uvedenej klasifikácie, ktorá bola vytvorená ako súčasť tejto práce existujú aj iné klasifikácie pachov kódu. Jednou z nich je napríklad klasifikácia uvádzaná na stránke *sourcemaking.com*. Táto klasifikácia približne zodpovedá klasifikácií uvedenej vyššie, no rozdeľuje pachy kódu striktne iba do jednej triedy. Daná klasifikácia uvádza iba 5 tried pachov a to[19]:

[19] https://sourcemaking.com/refactoring

- *Bloaters*

  Časti kódu – metódy, triedy a podobne, ktoré majú veľké rozmery

- *Object-Orientation Abusers*

  Nedostatočne aplikované princípy objektovo orientovaného programovania

- *Change Preventers*

  Problémy, ktoré zabraňujú jednoduchým zmenám v zdrojovom kóde. Pri zmenách je tak potrebné vykonávať zmeny na viacerých miestach v zdrojovom kóde.

- *Dispensables*

  Zbytočné časti kódu, ktoré možno za určitých okolností odstrániť pre dosiahnutie väčšej čistoty zdrojového kódu.

- *Couplers*

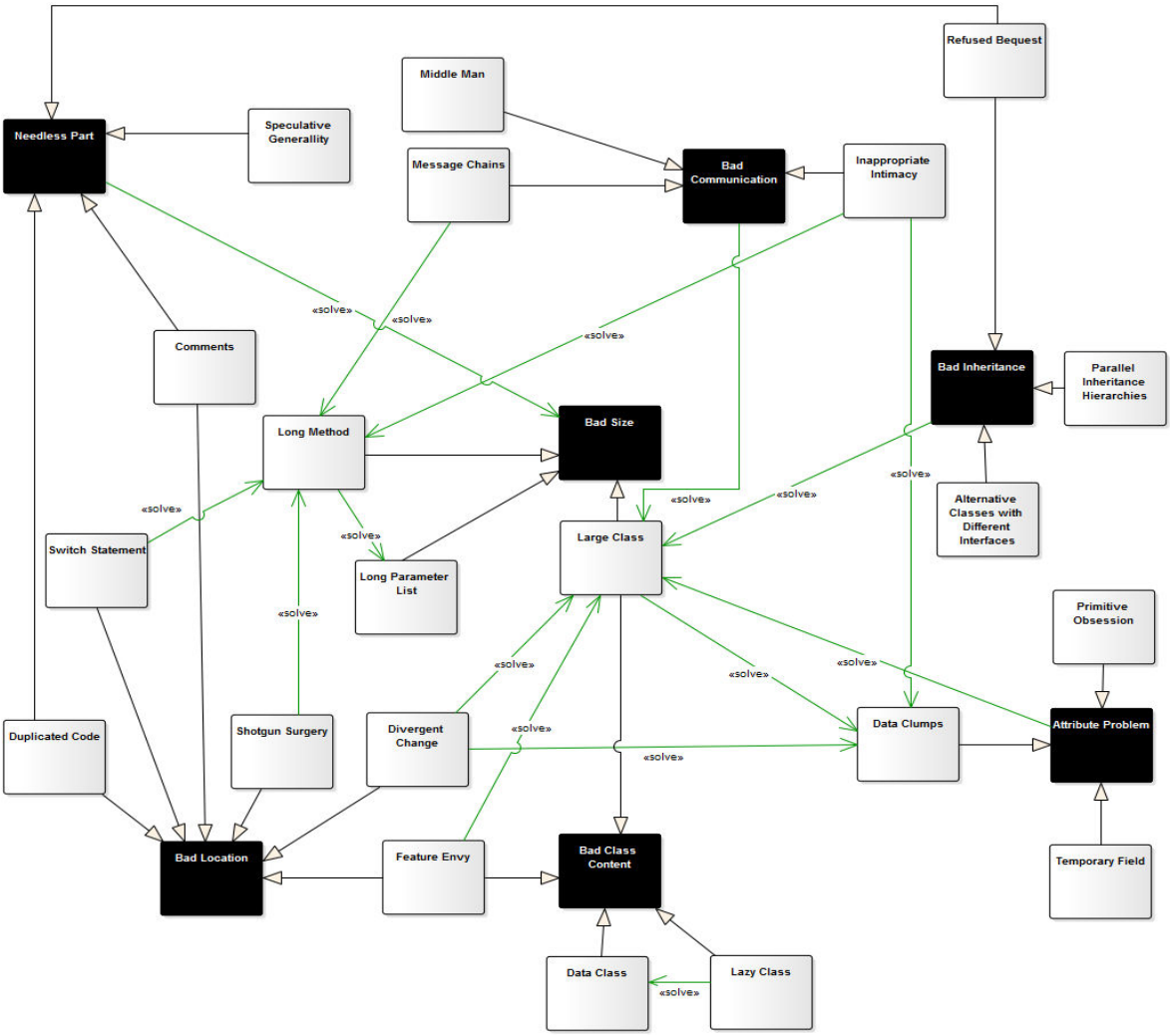
  Pachy, ktoré poukazujú na prílišné viazanie tried. Ide o problémy s komunikáciou medzi jednotlivými triedami, teda napríklad prílišnou previazanosťou, alebo nutnosťou komunikácie cez viacerých sprostredkovateľov.

Tieto, ale aj ďalšie možné kategorizácie sú pomerne dobé a logicky organizované, no vzhľadom na potreby refaktorovacieho systému je výhodnejšie použiť kategorizáciu, ktorá bola vytvorená ako súčasť tejto práce. Napriek tomu, že táto kategorizácia zodpovedá iným kategorizáciám, napríklad vyššie uvedenej, bude považovaná za hlavný podklad pre uvažovanie vzťahov pri tvorbe pravidiel expertného systému.

5.2. Pozitívne závislosti pachov

Pojem pozitívna závislosť je v tejto práci použitý vo význame, že oprava daného pachu môže za určitých okolností predstavovať aj opravu iného pachu kódu. Pri podrobnom preskúmaní vzťahom medzi základnými pachmi kódu je možné odhaliť veľké množstvo prípadných pozitívnych závislostí. Zovšeobecnenie pachov do tried, zadefinované v kapitole 5.1 napomáha lepšiemu pochopeniu a znázorneniu týchto vzťahov, keďže neraz je možné zadefinovať vzťah pre celú nadtriedu pachov, miesto rovnakého vzťahu pre každý jeden pach.

Na nasledujúcom obrázku možno vidieť diagram, ktorý znázorňuje pozitívne vzťahy jednotlivých pachov. Pre pozitívny vzťah bol použitý stereotyp „*solve*“ a zelená, plná šípka. Diagram bol vytvorený pomocou UML diagramu tried, no v tomto prípade nemožno hovoriť, že ide o diagram tried, ale skôr o špeciálny diagram vzťahov medzi pachmi kódu.



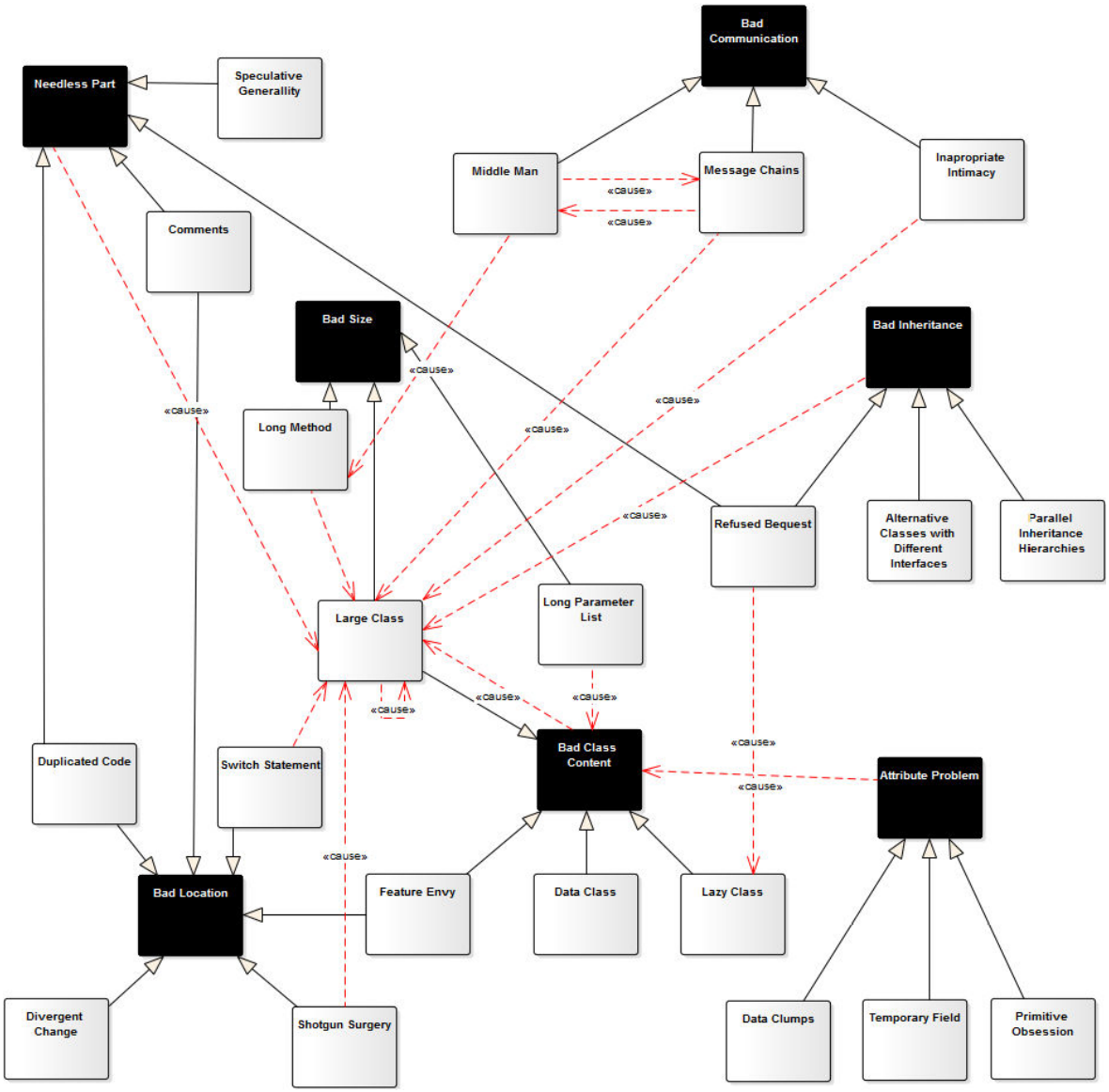
Obr. 20 Pozitívne závislosti pachov kódu

5.3. Negatívne závislosti pachov

V kontexte tejto diplomovej práce je pojem negatívna závislosť chápaný ako vzťah medzi pachmi kódu, kedy oprava pachu môže spôsobiť iný pach na danom, alebo inom mieste.

Pre vyjadrenie negatívneho vzťahu medzi pachmi kódu bol použitý stereotyp „*cause*“ a červená, prerušovaná šípka. Podobne, ako pri pozitívnych závislostiach, sú v maximálnej možnej miere využívané nadtriedy pachov. Napriek tomu, je sieť vzťahov

stále pomerne komplikovaná, čo dobre ilustruje povahu a náročnosť problému. Jednotlivé negatívne vzťahy znázorňuje diagram nižšie.



Obr. 21 Negatívne závislosti

Pri určovaní závislostí medzi jednotlivými pachmi je potrebné spomenúť, že vzhľadom na to, že jeden pach kódu môže byť vo väčšine prípadov riešený viacerými spôsobmi, či už častejšie, alebo menej využívanými, nie je vždy isté, že daný vzťah bude vždy naplnený. V konečnom dôsledku tak možno hovoriť o pravdepodobnosti vzniku závislostí na základe aplikovanej opravy. Ideálnejšie je však rozšíriť diagramy závislostí tak, aby hrana závislosti, okrem začiatočného a koncového uzla, definovala aj konkrétnu refaktorovaciu operáciu, ktorá daný pach môže spôsobiť, čo je bližšie rozoberané v kapitole 5.5.

5.4. Závislosti pachov kódu a refaktorovacích operácií

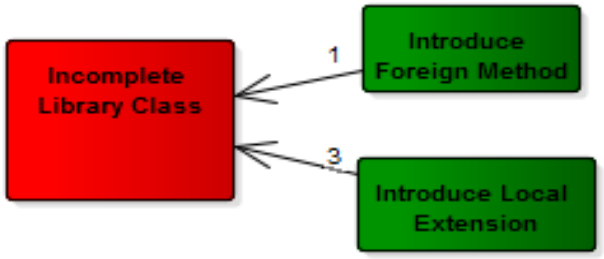
Refaktorovacie operácie, ktoré *Martin Fowler* zadefinoval a uvádzal ako spôsob riešenia pre jednotlivé pachy, predstavujú všeobecné metódy, pomocou ktorých možno odstrániť daný pach. Viaceré refaktorovacie operácie nie sú priradené konkrétnym pachom, ale predstavujú samostatnú, nezávislú metódu, ktorá môže byť aplikovaná na viacero problémov.

Medzi refaktorovacie operácie patria [1]:

Add Parameter, Change Bidirectional Association to Unidirectional, Change Reference to Value, Change Unidirectional Association to Bidirectional, Change Value to Reference, Collapse Hierarchy, Consolidate Conditional Expression, Consolidate Duplicate Conditional Fragments, Decompose Conditional, Duplicate Observed Data, Dynamic Method Definition, Eagerly Initialized Attribute, Encapsulate Collection, Encapsulate Downcast, Encapsulate Field, Extract Class, Extract Interface, Extract Method, Extract Module, Extract Subclass, Extract Superclass, Extract Surrounding Method, Extract Variable, Form Template Method, Hide Delegate, Hide Method, Inline Class, Inline Method, Inline Module, Inline Temp, Introduce Assertion, Introduce Class Annotation, Introduce Expression Builder, Introduce Foreign Method, Introduce Gateway, Introduce Local Extension, Introduce Named Parameter, Introduce Null Object, Introduce Parameter Object, Isolate Dynamic Receptor, Lazily Initialized Attribute, Move Eval from Runtime to Parse Time, Move Field, Move Method, Parameterize Method, Preserve Whole Object, Pull Up Constructor Body, Pull Up Field, Pull Up Method, Push Down Field, Push Down Method, Recompose Conditional, Remove Assignments to Parameters, Remove Control Flag, Remove Middle Man, Remove Named Parameter, Remove Parameter, Remove Setting Method, Remove Unused Default Parameter, Rename Method, Replace Abstract Superclass with Module, Replace Array with Object, Replace Conditional with Polymorphism, Replace Constructor with Factory Method, Replace Data Value with Object, Replace Delegation With Hierarchy, Replace Delegation with Inheritance, Replace Dynamic Receptor with Dynamic Method Definition, Replace Error Code with Exception, Replace Exception with Test, Replace Hash with Object, Replace Inheritance with Delegation, Replace Loop with Collection Closure Method, Replace Magic Number with Symbolic Constant, Replace Method with Method Object, Replace Nested Conditional with Guard Clauses, Replace Parameter with Explicit Methods, Replace Parameter with Method, Replace Record with Data Class, Replace Subclass with Fields, Replace Temp with Chain, Replace Temp with

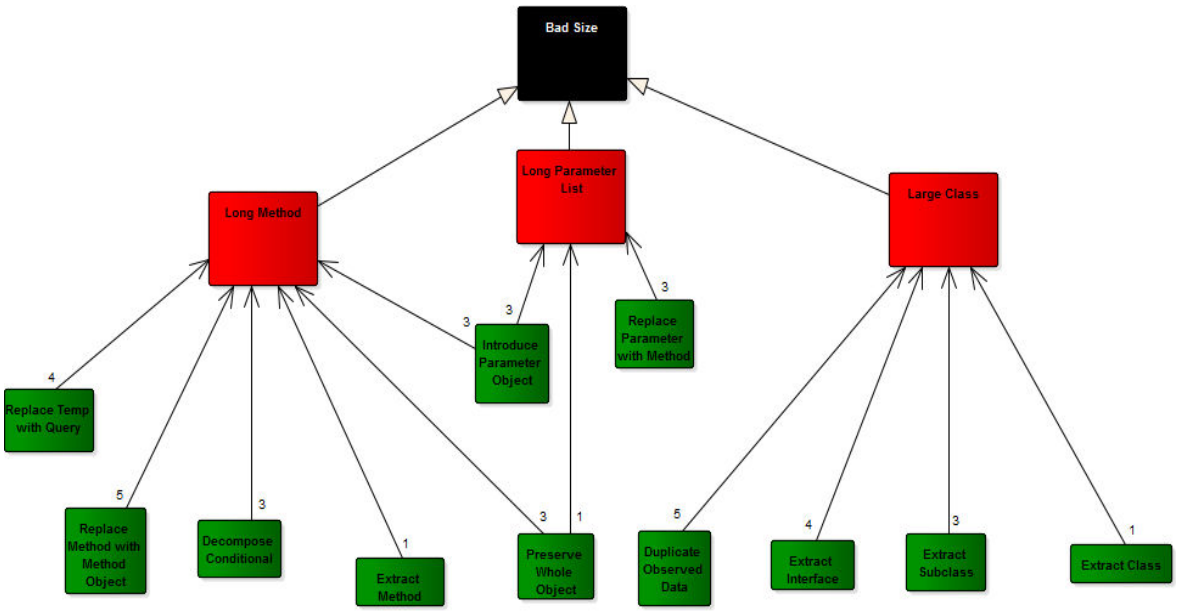
Query, Replace Type Code with Class, Replace Type Code with Module Extension, Replace Type Code With Polymorphism, Replace Type Code with State/Strategy, Replace Type Code with Subclasses, Self Encapsulate Field, Separate Query from Modifier, Split Temporary Variable, Substitute Algorithm

Konkrétne bolo zadefinovaných 91 refaktorovacích operácií, no len niekoľko z nich priamo súvisí s riešením týchto pachov v kóde. Každému z pachov kódu prislúcha niekoľko refaktorovacích operácií, ako možné riešenie. Výber konkrétneho riešenia závisí vo väčšine prípadov od aktuálnej situácie, no častokrát sú jednotlivé riešenia preferovanejšie, ako iné. Na nasledujúcich diagramoch sú ku každému pachu uvedené refaktorovacie operácie a stupeň ich použitia, pričom *1 predstavuje často používané riešenie* a *5 minimálne používané riešenie.*

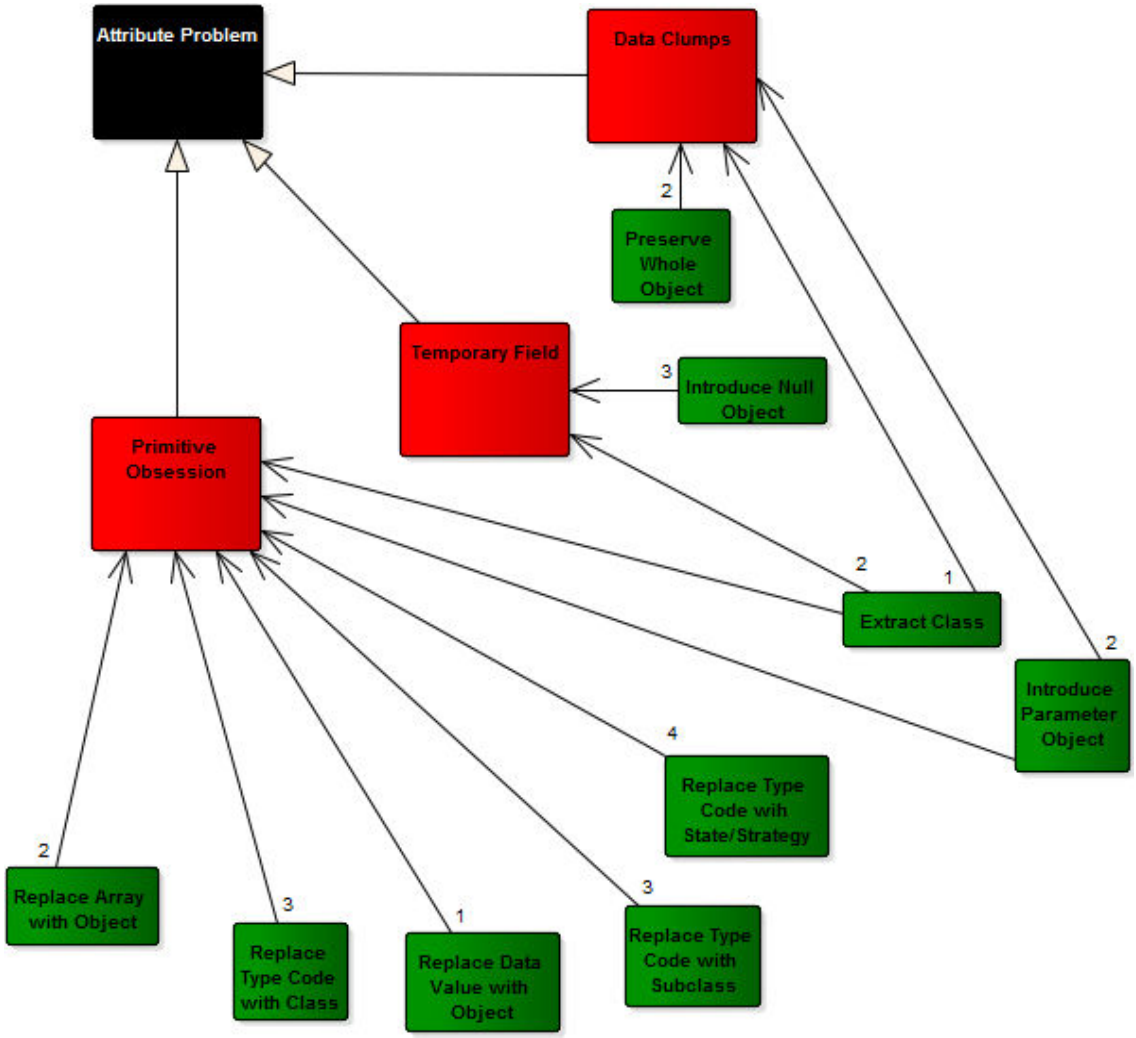


Obr. 22 Refaktorovacie operácie pachu kódu *„Incomplete Library Class“*

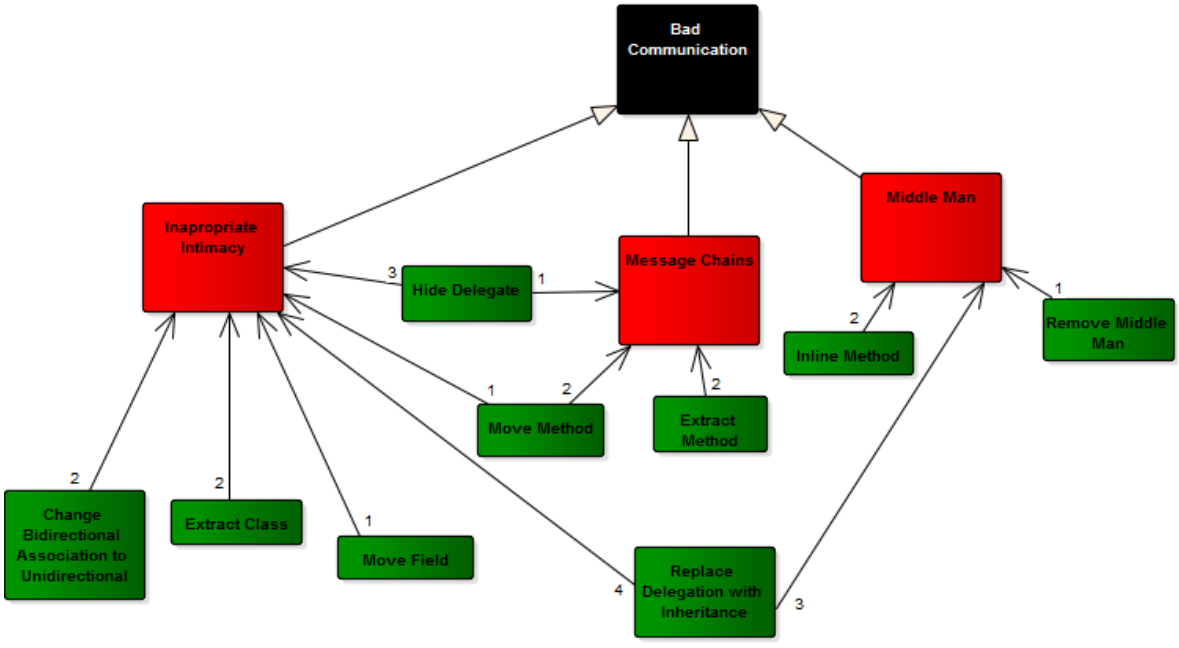
Na *Obr. 22, 23, 24, 25, 26, 27, 28 a 29* je uvedených niekoľko diagramov, ktoré je v podstate možné chápať ako jeden diagram, ktorý je však rozdelený kvôli prehľadnosti do viacerých častí.



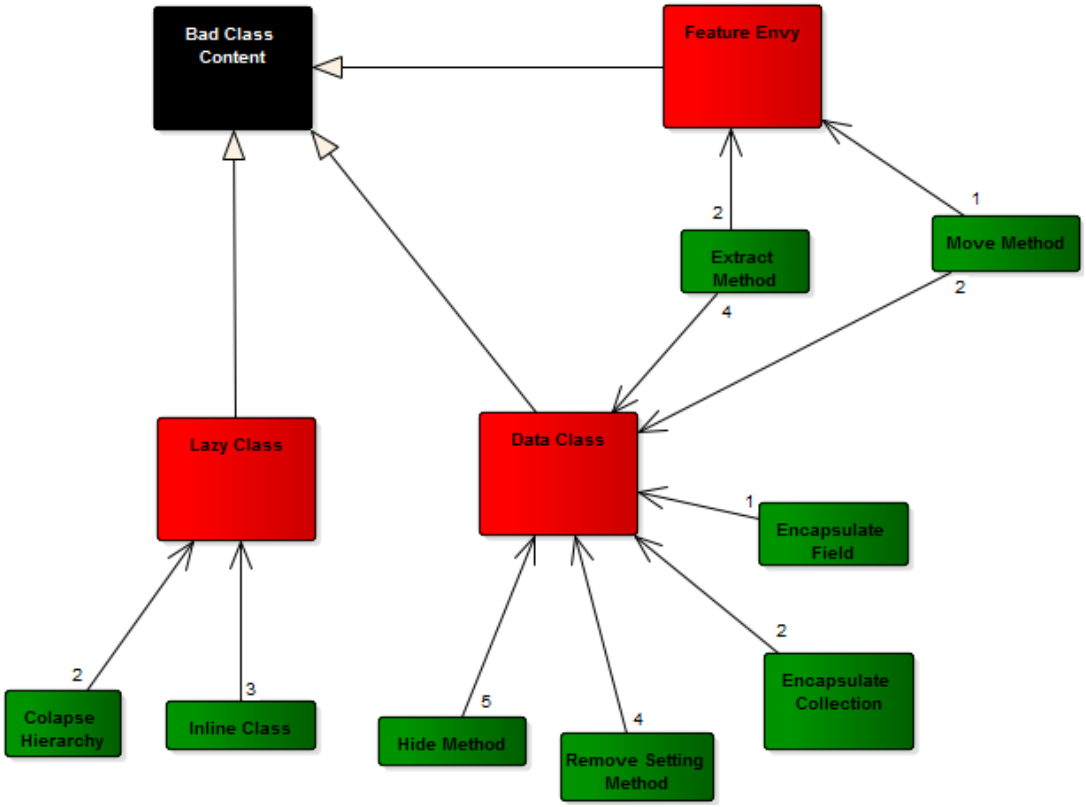
Obr. 23 Refaktorovacie operácie pachov triedy *„Bad Size“*



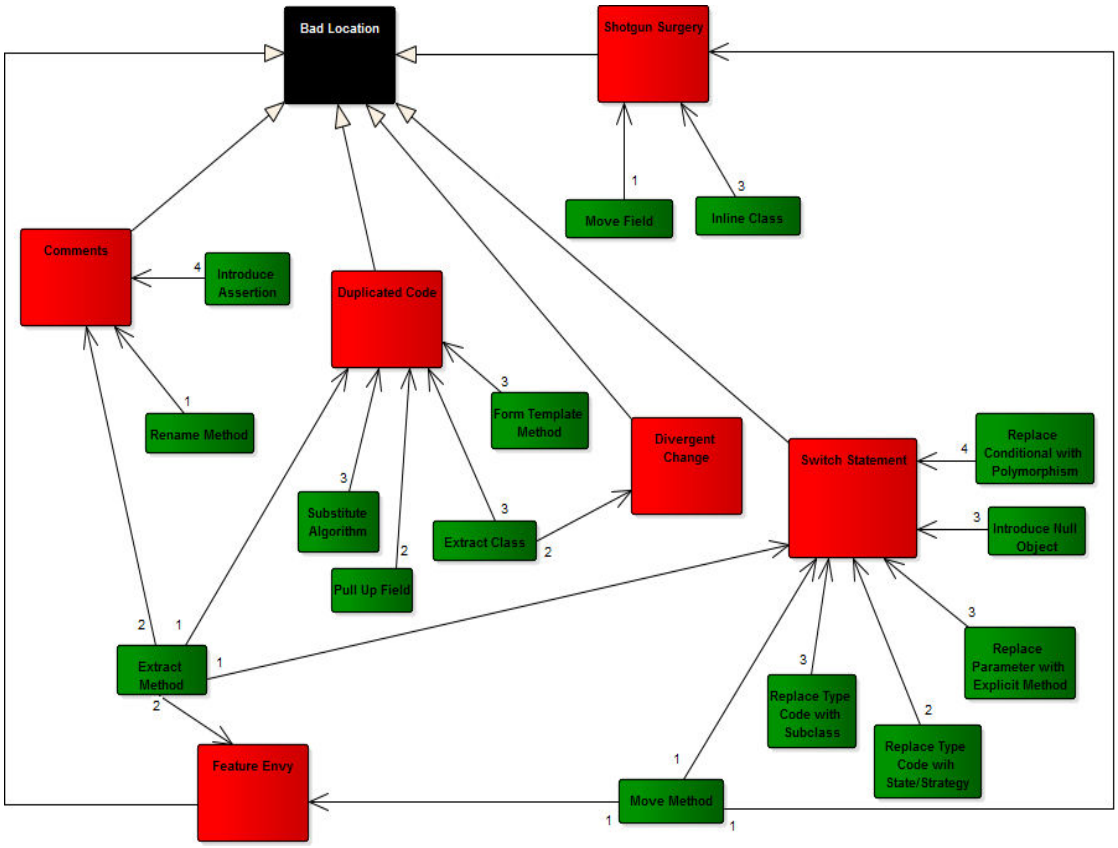
Obr. 24 Refaktorovacie operácie triedy *„Attribute Problem“*



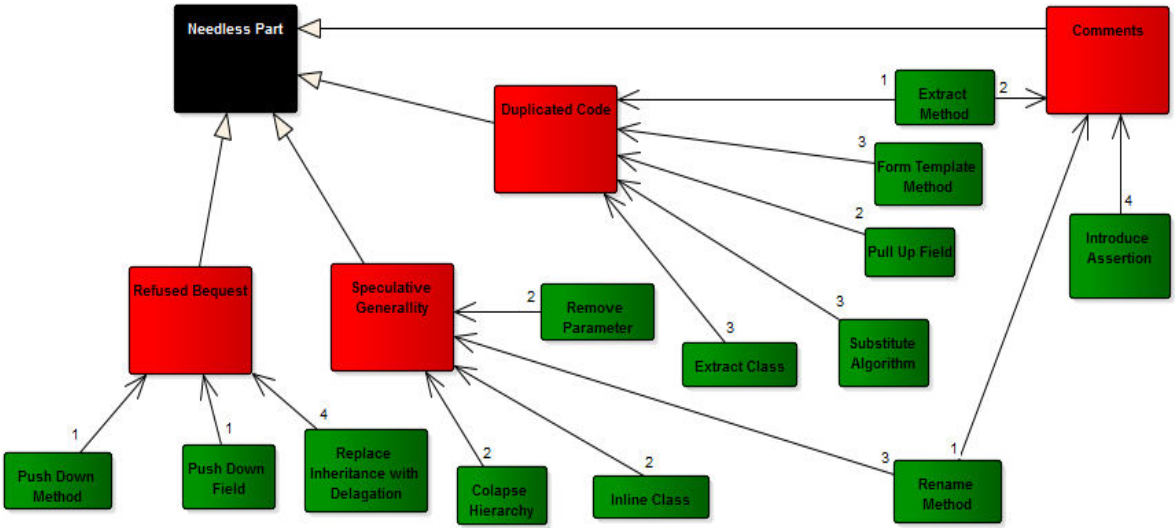
Obr. 25 Refaktorovacie operácie triedy *„Bad Communication“*



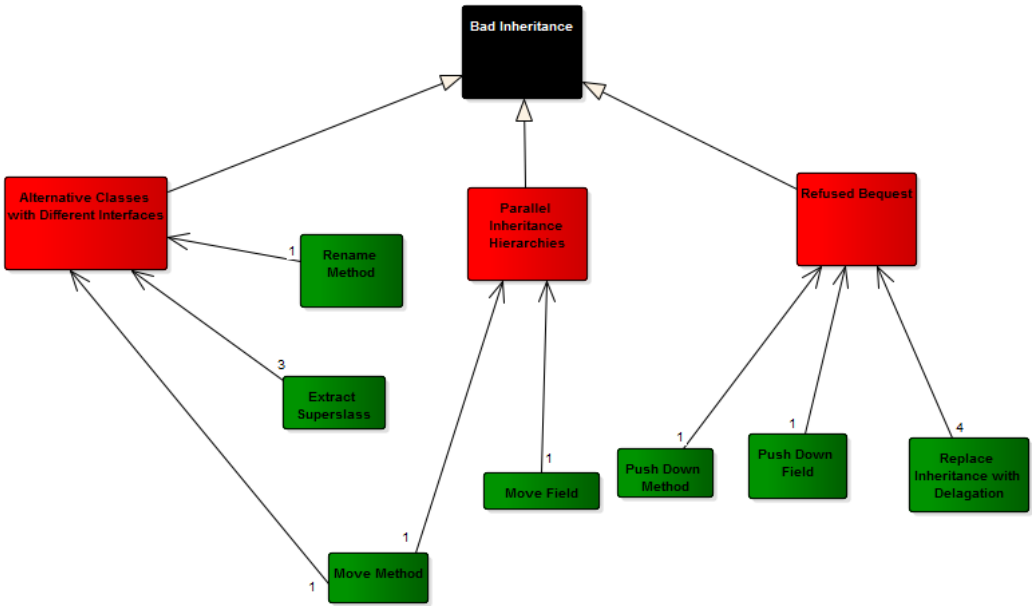
Obr. 26 Refaktorovacie operácie triedy *„Bad Class Content“*



Obr. 27 Refaktorovacie operácie triedy *„Bad Location“*



Obr. 28 Refaktorovacie operácie triedy *„Needless Part“*



Obr. 29 Refaktorovacie operácie triedy „Bad Inheritance“

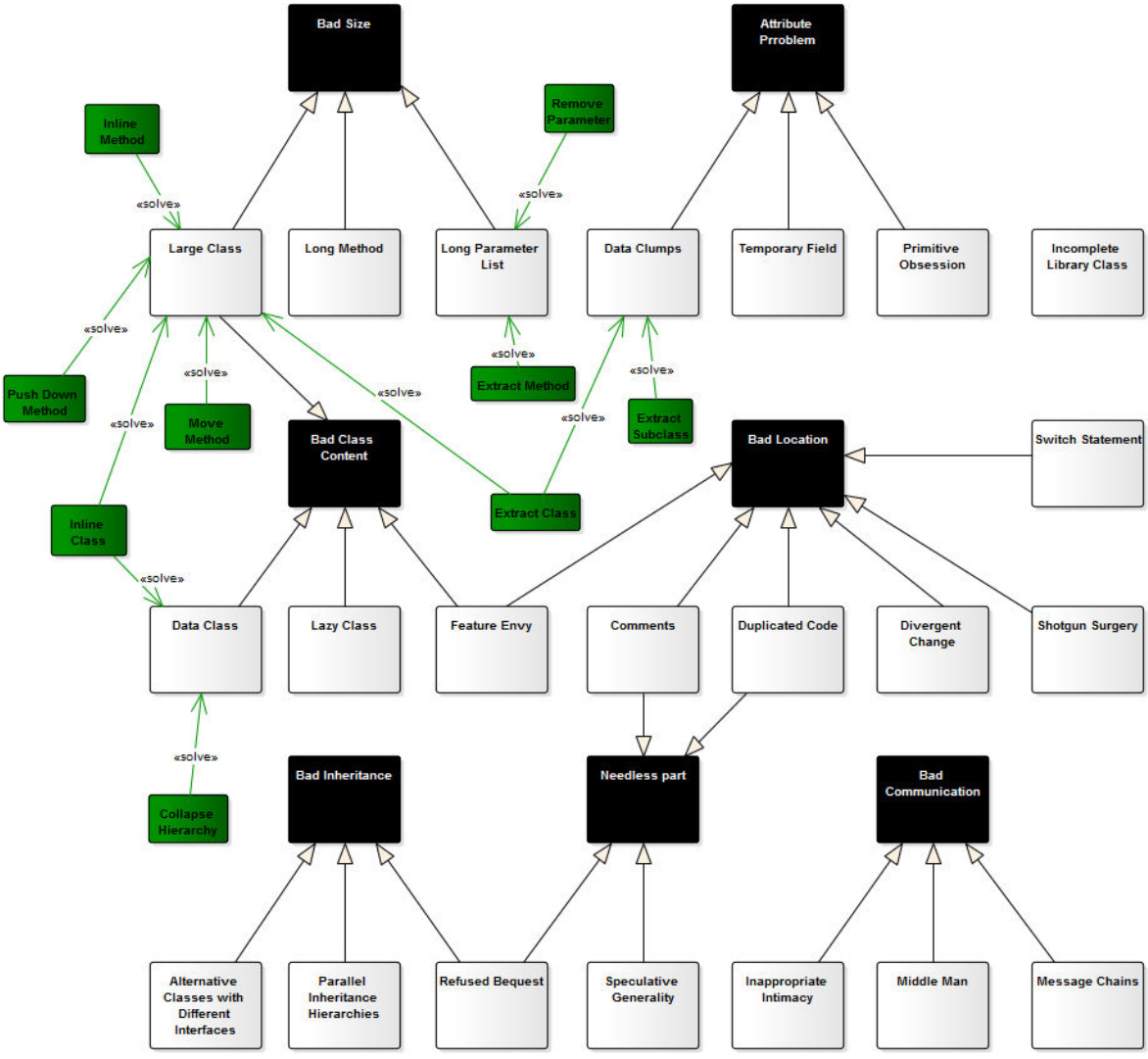
Pozn.: *Ohodnotenie vzťahov medzi pachom a refaktorovacou operáciou vychádza primárne z analýzy textov Martina Fowlera* [1] *a zároveň z odhadu autora práce.*

Definovanie vzťahov refaktorovacích operácií a pachov kódu je vstupným bodom pre prechod od závislostí medzi pachmi kódu k závislostiam medzi pachom, jeho refaktorovaním a vedľajším pozitívnym, alebo negatívnym účinkom. Zahrnutie informácie o frekvencii používania danej refaktorovacej operácie napomáha pri chápaní vzťahov, čo vedie k lepším výsledkom pri definovaní pravidiel pre expertný systém. Nasledujúca podkapitola opätovne rozoberá pozitívne a negatívne závislosti, no na hrany pridáva refaktorovacie operácie, ktoré dané závislosti zapríčiňujú.

5.5. Dopady refaktorovacích operácií na zdrojový kód

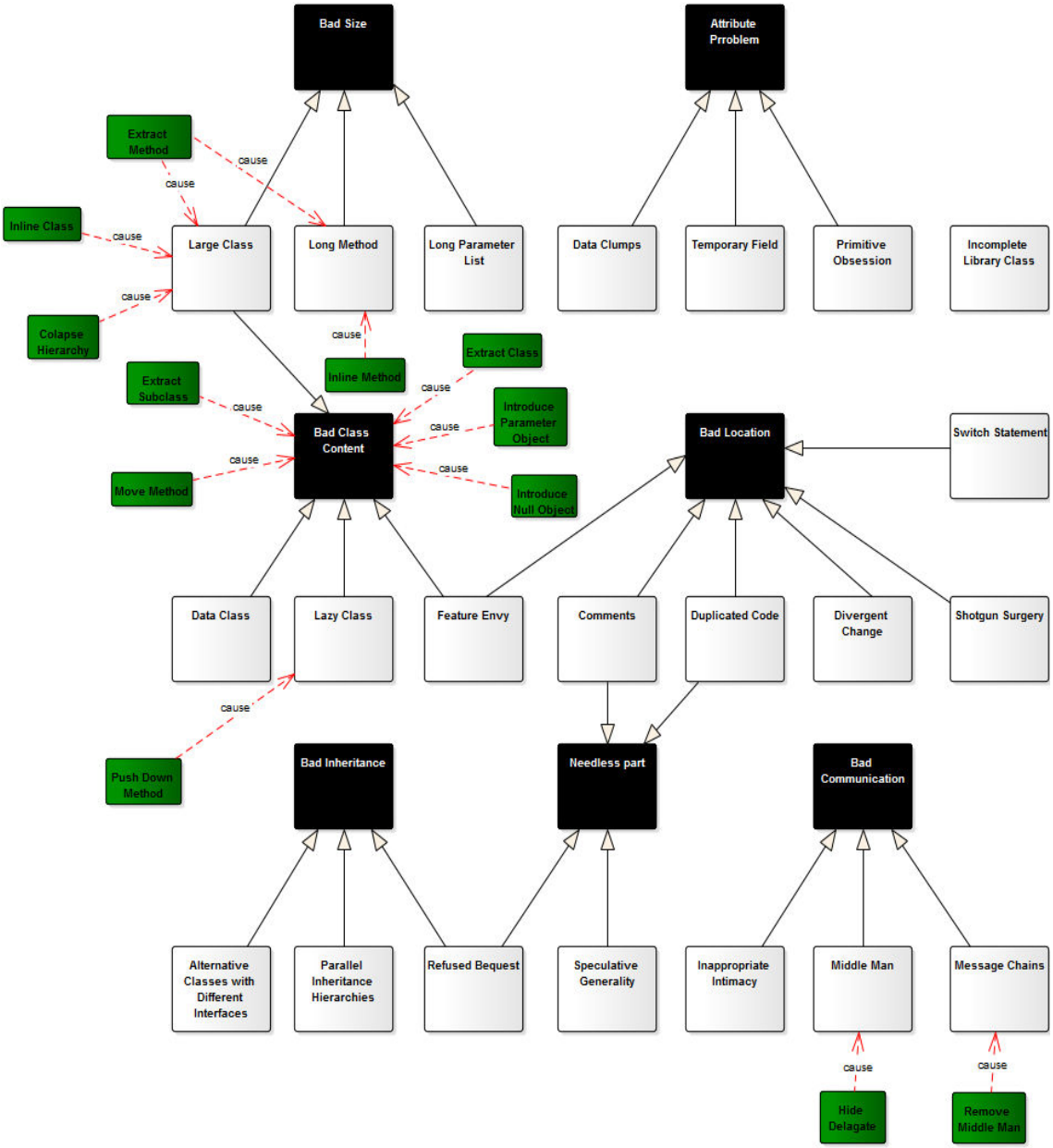
V tejto kapitole sú zachytené možné vedľajšie účinky aplikácie (pozitívne a negatívne) refaktorovacích operácií na zdrojový kód. Závislosti vychádzajú z predchádzajúcich návrhov – diagramov závislostí pachov a príslušných refaktorovacích metód pre pachy.

Pozitívne závislosti pachov uvedené na *Obr. 20* možno zobraziť s využitím refaktorovacích operácii oveľa prehľadnejšie. Diagram, uvedený na *Obr. 30* zobrazuje refaktorovacie operácie, ktoré môžu v určitých situáciách pozitívne vplývať aj na iné pachy na danom mieste. Ako pozitívna závislosť pachu a refaktorovacej operácie možno taktiež brať príslušnosť refaktorovacích operácii k pachom, čo je rozoberané v predchádzajúcej kapitole. Tento diagram však znázorňuje iba vedľajšie pozitívne efekty aplikácie refaktorovacej operácie, nie primárne, cielené efekty.



Obr. 30 Pozitívne závislosti refaktorovacích operácii na vedľajšie pachy kód

Na obrázku nižšie možno vidieť znázornenie možného negatívneho vplyvu refaktorovacích operácií na vznik pachov v kóde. *Obr. 31* predstavuje podrobnejší pohľad, prostredníctvom refaktorovacích operácií na závislosti uvedené na *Obr. 21*.



Obr. 31 Negatívne závislosti refaktorovacích operácií a pachov kódu

Ako možno vidieť na uvedených diagramoch, tak napriek pomerne náročným závislostiam medzi samotnými pachmi kódu, je možné tieto závislosti zovšeobecniť a preniesť na niekoľko zodpovedných refaktorovacích operácií. Na základe predchádzajúcich kapitol, tak ľahko možno určiť situácie, kedy daná závislosť vzniká a ich pravdepodobnosti. Práve zohľadňovanie týchto závislostí a ich uvažovanie v expertnom systéme je predmetom ďalšieho smerovania tejto práce.

6. Optimalizácia procesu refaktorovania

Definované závislosti pachov kódu slúžia ako vhodný základ pre optimalizáciu procesu automatizovaného refaktorovania. Na základe definovaných vzťahov sa možno rozhodovať ako efektívne aplikovať refaktorovacie operácie tak, aby bol dosiahnutý optimálny výsledok v podľa požiadavky. (Vychádza sa z jednoduchej závislosti pachov, nie zo závislosti pachov a refaktorovacích operácií)

Pri refaktorovaní možno klásť za prioritu viacero cieľov. Pre ohodnotenie splnenia cieľov možno definovať rôzne hodnotiace funkcie – fitness funkcie. Uvažovať možno napríklad o funkciách, ktoré vyhodnocujú aktuálny stav tak, že vyhodnocujú:

- *Aktuálny počet pachov v kóde* – najjednoduchšia funkcia, ktorá hodnotí iba aktuálny stav kódu a počet pachov v ňom. Situácia s troma pachmi je teda lepšia ako situácia so štyrmi pachmi, bez ohľadu na ďalšie okolnosti.
- *Aktuálna váha pachov v kóde* – pri takejto funkcii je nutné prioritizovať pachy kódu. Každý pach môže mať vlastnú váhu, respektíve prioritu, no možno uvažovať aj o prioritizácií na základe tried pachov. V takomto prípade môže byť situácia s 10 jednoduchšími pachmi v kóde vyhodnotená ako lepšia, ako situácia s dvoma zložitými pachmi v kóde.
- *Aktuálny počet použitých refaktorovacích operácií* – takáto funkcia sa uvažuje taktiež o výhodnosti aplikácií refaktorovacích operácií. Pri veľkom množstve refaktorovacích operácií kód výrazne degraduje a preto je neraz výhodnejšie použiť menší počet refaktorovacích operácií a zároveň akceptovať určité pachy v kóde, ako použiť veľké množstvo refaktorovacích operácií, pre malé zlepšenie počtu pachov v kóde.
- *Aktuálne váha použitých refaktorovacích operácií* – podobne, ako pri predchádzajúcej funkcii sa uvažuje počet refaktorovacích operácií, ktorý znižuje výhodnosť daného stavu. Na rozdiel od predchádzajúcej funkcie však nemajú refaktorovacie operácie rovnakú váhu, ale každá operácia znižuje výhodnosť daného stavu podľa pridelenej hodnoty. Možno tak uvažovať, že refaktorovacia operácia, ktorá manipuluje s umiestnením kódu je viac nevýhodná ako operácia, ktorá iba mení názov premennej.

- *Rôzne kombinácie* – okrem vyššie definovaných funkcií možno uvažovať aj také, ktoré berú do úvahy viaceré vyššie spomínané aspekty. Pri funkciách, ktoré uvažujú počet, respektíve váhy refaktorovacích operácií je nutné automaticky uvažovať aj dosiahnutý stav pachov v kóde. Vzniká tak možnosť definovať ďalšie štyri zložitejšie fitness funkcie, ktoré ohodnocujú aktuálny stav a to:
    - *Počet pachov v kóde a počet použitých refaktorovacích operácií*
    - *Váha pachov v kóde a počet použitých refaktorovacích operácií*
    - *Počet pachov v kóde a váha použitých refaktorovacích operácií*
    - *Váha pachov v kóde a váha použitých refaktorovacích operácií*

    Práve posledný definovaný typ fitness funkcie predstavuje najdokonalejší a zároveň najzložitejší prístup pre implementáciu funkcie. Práve pomocou nej však pravdepodobne možno dosiahnuť najlepšie výsledky pri automatizovanom refaktorovaní.
- *Ďalšie* – teoreticky je možné uvažovať aj iné fitness funkcie, napríklad komplikovanosť príslušných opráv, dostupnosť spôsobu opravy pachov a podobne.

Popísané fitness funkcie vyhodnocujú aktuálny stav. Pre nájdenie optimálneho stavu je preto potrebné prehľadávať stavový priestor, v ktorom uzol predstavujú aktuálne pachy v kóde. Prechod do iného stavu je vyvolaný aplikovaním refaktorovania niektorého z týchto pachov. Nový stav je následne ohodnotený fitness funkciou.

Samotné prehľadávanie stavového priestoru predstavuje pomerne zložitý problém. Možné prechody medzi stavmi vychádzajú z grafu závislostí medzi pachmi. Následne je možné na základe týchto grafov prehľadávať stavový priestor pomocou dátovej štruktúry – stromu. Nasledujúca podkapitola podrobnejšie popisuje prístup k hľadaniu postupnosti refaktorovacích operácií pre optimalizáciu automatizovaného refaktorovania.

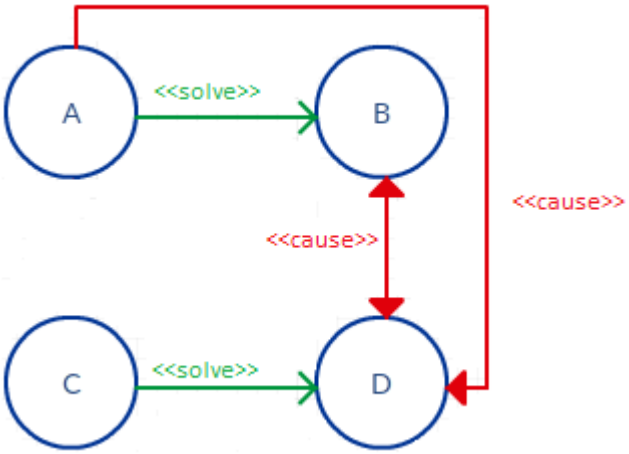
6.1. Hľadanie optimálneho refaktorovacieho postupu

Pri hľadaní refaktorovacieho postupu je potrebné vychádzať z definovaných pozitívnych a negatívnych vzťahov pachov kódu. Vyššie uvedené vzťahy zohľadňujú iba 22 pachov kódu, no v reálnom nástroji by bolo potrebné uvažovať aj rôzne anti-vzory, čím vznikajú podstatne zložitejšie vzťahy.

Pomocou takto definovaných vzťahov možno vytvoriť spoločný statický graf pozitívnych a negatívnych závislostí pachov kódu, respektíve anti-vzorov.

Pozn.: *Pre udržiavanie a dopytovanie v takto zložitom grafe je výhodné použiť grafové nástroje, ako napríklad grafová databáza.*

Pre ilustrovanie prístupu je na *Obr. 32* uvedený zjednodušený graf závislostí, predpokladajúci existenciu iba štyroch pachov kódu.



Obr. 32 Zjednodušený graf pozitívnych a negatívnych závislostí

Pomocou grafu uvedenom na *Obr. 32* je následne potrebné vyhľadať takú postupnosť odstraňovania pachov pomocou refaktorovania, ktorá vedie do stavu s optimálnym ohodnotením pomocou fitness funkcie. Existuje mnoho prístupov, o ktorých možno uvažovať pri hľadaní optimálnej cesty, napríklad:

- *Grafové algoritmy*
  - *Prioritizované prehľadávanie*
  - *Prehľadávanie do šírky*
  - *Prehľadávanie do hĺbky*
- *Grafové databázy*
- *Fuzzy systémy*

- *Lineárne programovanie*
- *Multi-agentové systémy*
    - *Mravčí algoritmus*
    - *Včelí algoritmus*

Pravdepodobne nie všetky z týchto prístupov sú na daný problém skutočne použiteľné. Určenie najvhodnejšieho prístupu k hľadaniu optimálneho postupu aplikácie refaktorovania ponúka vhodný priestor pre ďalšiu prácu v tejto oblasti.

V podkapitolách 6.1.1 a 6.1.2 sú čiastočne rozpracované dva prístupy - pomocou jednoduchého *prioritizovaného prehľadávania* a návrh využitia *včelieho algoritmu*, pri aplikácii na daný problém.

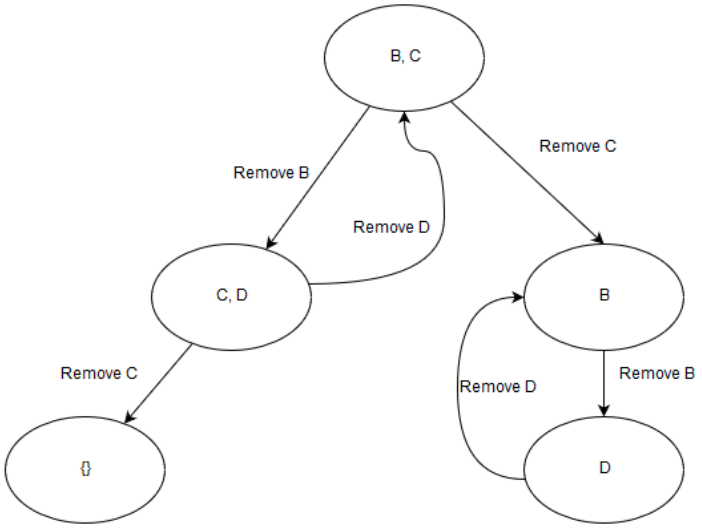
6.1.1. Prioritizované prehľadávanie

Prioritizované prehľadávanie (*best-first seach*) je jednoduché a na daný problém vhodne aplikovateľné hľadanie. Takéto hľadanie berie priamo do úvahy hodnotu fitness funkcie nasledujúceho stavu, pričom sa vyberá ten najlepší.

Oproti jednoduchším prehľadávaniam – do šírky a hĺbky má mnoho výhod. Pri použití fitness funkcií, ktoré neuvažujú počet, respektíve váhu aplikovaných operácií refaktorovania, hľadanie postupne prehľadáva aktuálne najvýhodnejšie stavy. Pri uvažovaní počtu, respektíve váhy použitých refaktorovacích operácií zachováva výhody prehľadávania do šírky, ktoré nachádza optimálny stav skôr, ako prehľadávanie do hĺbky.

Nižšie je uvedený príklad hľadania optimálneho stavu pomocou prioritizovaného hľadania. V príklade je uvažovaná fitness funkcia, ktorá uvažuje iba minimalizáciu počtu pachov v kóde, nie minimalizáciu refaktorovacích operácií. Hľadanie vychádza z grafu závislostí, uvedenom na *Obr. 32*.

Príklad uvažuje dva začiatočné pachy v kóde a to pach B a pach C. Pri tomto zjednodušenom príklade je možné určiť celý stavový priestor. Stavový priestor je zobrazený na *Obr. 33* uvedenom nižšie. Ako možno vidieť, strom znázorňujúci stavový priestor obsahuje cykly, ktoré je nutné pri prehľadávaní vhodne ošetriť.

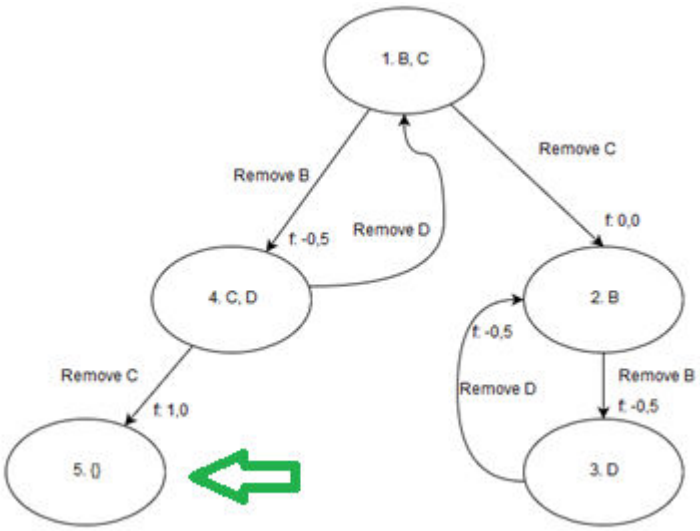


Obr. 33 Možné stavy pachov v kóde

Prioritizované hľadanie môže prebiehať nasledovne: refaktorovanie pri ktorom vzniká pach znižuje fitness funkciu stavu a 0,5 a refaktorovanie pri ktorom sa odstráni ďalší pach v kóde zvýši fitness funkciu o 1,0.

0. Začiatočné pachy: {B, C}
1. Remove C (f: 0,0) => {B}
2. Remove B (f: -0,5) => {D} => cyklus
3. Remove B (f: -0,5) => {C, D}
4. Remove C (f: 1,0) => { }

Na obrázku nižšie je uvedená príslušná vizualizácia prehľadávania.

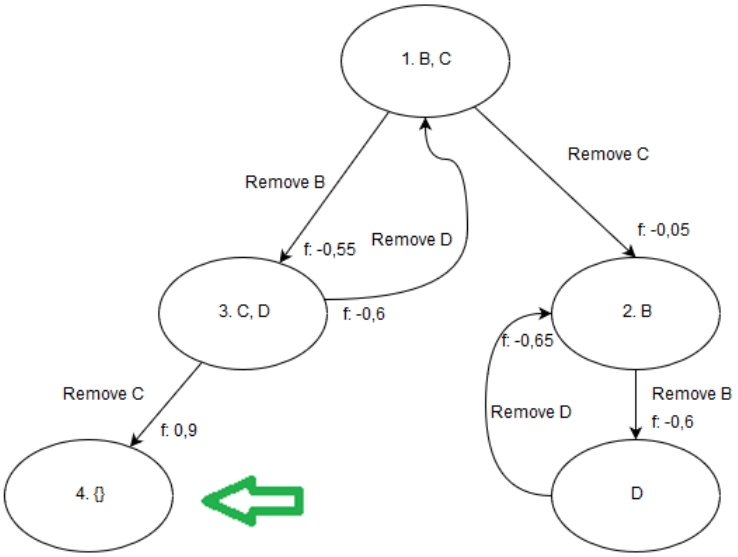


Obr. 34 Vizualizácia príkladu prehľadávania

V tomto príklade musel byť napriek prioritizácií prehľadaný celý stavový priestor. Možno však vidieť, že pri niektorých krokoch nie je voľba jednoznačná. V prípade uvažovania fitness funkcie, ktorá berie do úvahy taktiež počet refaktorovacích operácií sa situácia zlepší, ako možno vidieť v nasledovnom príklade.

Prioritizované hľadanie bude prebiehať nasledovne, pričom refaktorovanie pri ktorom vzniká pach znižuje fitness funkciu stavu a 0,5 a refaktorovanie pri ktorom sa odstráni ďalší pach v kóde zvýši fitness funkciu o 1,0. Každý krok taktiež znižuje hodnotu fitness funkcie o 0,05.

0. Začiatočné pachy: {B, C}
1. Remove C (f: -0,05) => {B}
2. Remove B (f: -0,55) => {C, D}
3. Remove C (f: 0,9) => {}



Obr. 35 Vizualizácia príkladu uvažujúceho počet refaktorovacích operácií

Ako možno vidieť na uvedených príkladoch, spôsob vyhľadávania optimálneho stavu závisí vo veľkej miere od fitness funkcie. Fitness funkciu možno postaviť veľmi jednoducho, napríklad tak, že je priamo závislá od počtu pachov v kóde, ale aj komplikovanejším spôsobom, ktorý počet pachov hodnotí na základe zmeny pachov v kóde, ako je uvedené v prvom príklade.

Ohodnotenie jedného kroku môže stanoviť rôzne, od čoho sa odvíja aj to, aký počet refaktorovacích operácií je prípustný pre opravu jedného pachu v kóde. Pokiaľ teda bude krok ohodnotený na 0.5 znamená to, že pach je prípustné opraviť maximálne 2 operáciami.

6.1.2. Včelí algoritmus

Multi-agentové systémy možno s výhodou využiť pri hľadaní vhodnej refaktorovacej cesty. Pre optimalizáciu a paralelné prehľadávanie možno uvažovať viacero biologicky inšpirovaných algoritmov. Typickým príkladom je včelí algoritmus.

Pod pojmom včelí algoritmus sa vo všeobecnosti rozumie viacero rôznych algoritmov inšpirovaných správaním včiel. Pre stanovený problém je vhodné uvažovať vyhľadávací algoritmus vychádzajúci zo správania včiel pri hľadaní potravy.

Včely pri hľadaní potravy prehľadávajú prostredie, pričom zisťujú kvalitu konkrétnych zdrojov potravy. Po návrate do úľa jednotlivé včely komunikujú zistenia ďalším včelám, ktoré sa následne v určitom počte, závisiacom od kvality zdroja pridajú k hľadaniu.

Dané správanie možno pravdepodobne aplikovať pri hľadaní optimálneho refaktorovacieho postupu. Na začiatku by tak bolo pridelených niekoľko málo agentov pre prehľadávanie stavového priestoru. Keďže ide o multi-agentový systém, jednotliví agenti by pracovali nezávisle a paralelne. Každý agent po ohodnotení svojej cesty následne komunikuje svoje zistenia ostatným agentom, ktorí sa nachádzajú v nečinnom stave. Na základe fitness cesty, ktorý daný agent prehľadával sa následne pridajú ďalší agenti, ktorí umožnia intenzívnejšie prehľadávanie danej cesty. Po prehľadaní konkrétnej vetvy by sa agent následne vrátil do stavu nečinnosti, pričom by mohol byť aktivovaný pomocou správy od iného agenta.

Takýto algoritmus pravdepodobne možno prispôsobiť pre hľadanie optimálnej refaktorovacej cesty. Napriek tomu, že prístup definovaný v kapitole 6.1.1 sa javí ako pomerne výhodný a jednoduchý, je nutné si uvedomiť, že pokiaľ neexistuje stav, ktorý možno jednoznačne určiť ako finálny bez ďalších informácií, je vždy nutné prehľadať celý stavový priestor, čo môže pri grafe závislostí s obrovským počtom vzťahov predstavovať pomerne vážny výkonnostný problém. Takýto multi-agentový systém je však možné použiť pre oba prípady – keď je potrebné prehľadať celý stavový priestor, aj keď existuje jednoznačne identifikovaný koncový stav, ktorý sa dá predpokladať, že bude na základe správania včiel nájdený pomerne rýchlo.

Výskumu možností včelieho, prípadne iného muleti-agentového algoritmu na vyhľadávanie dobrého postupu refaktorovania je potrebné venovať ďalší výskumný priestor.

7. Implementácia automatizovaného refaktorovacieho systému

Táto kapitola popisuje základné implementačné záležitosti automatizovaného refaktorovacieho nástroja.

Nástroj bol implementovaný v programovacom jazyku Java verzie 1.8. Pre pohodlnú prácu s programom je poskytnuté jednoduché webové rozhranie umožňujúce spravovať jednotlivé vyhľadávacie a opravovacie skripty a príslušné pravidlá. Jednotlivé skripty sú ukladané do klasickej relačnej databázy PostgreSQL vo verzii 9.5 tak, aby bolo možné rýchlo a ľahko získavať potrebné vyhľadávacie a opravovacie skripty. Na manažment projektu v XML podobe bola použitá XML dokumentová databáza BaseX vo verzii 8.4.4.

Systém funguje nad projektami spravovanými pomocou Git repozitárov. Pre refaktorovanie zdrojového kódu je teda potrebné uviesť príslušný git repozitár a prístupové údaje k nemu. Pri práci s projektom sa pôvodný zdrojový kód neovplyvní, aby v prípade zmien s ktorými používateľ nesúhlasí bolo možné ľahko odstrániť vykonané zmeny. Systém tak vytvára 2 dodatočné „branch“, ktorých názov je možné taktiež udať. Ide teda o „branch“ v ktorom sú vyznačené nájdené pachy v kóde a „branch“, v ktorom sa nachádzajú vykonané opravy.

V nasledujúcich podkapitolách sú bližšie popísané jednotlivé implementačné záležitosti navrhovaných konceptov.

7.1. Webové rozhranie

Aplikácia poskytuje jednoduché webové rozhranie. Toto rozhranie komunikuje s webovým serverom, na ktorý bol použitý web server *Apache Tomcat* vo verzii 8.0.33. Aplikácia vystavuje jednotlivé služby, ako *RestFull* webové služby. Webové rozhranie následne volá tieto služby, čím je zabezpečené vykonávanie aplikačnej logiky programu.

Na obrázku nižšie možno vidieť príklad vystavenej webovej služby programu. Konkrétne ide REST webovú službu typu PUT, ktorá zodpovedá za pridanie nového vyhľadávacieho skriptu.

```
@PUT
@Path("/search/")
@Consumes(MediaType.APPLICATION_JSON)
public void addSearchScript(String input) {
    resourceCommand.addSearchScript(input);
}
```

Obr. 36 Príklad vystavenej webovej služby

Podobným spôsobom, ako možno vidieť na obrázku sú vystavené všetky služby, ktoré využíva HTML webové rozhranie. Konkrétne sú to služby pre zobrazenie skriptov, zobrazenie pravidiel, aktualizáciu skriptov, aktualizáciu pravidiel, pridanie skriptov a vykonanie samotného refaktorovania nad udaným projektom.

7.2. SrcML – konverzia zdrojového kódu

Konverzia zdrojového kódu do XML reprezentácie a opačne je realizovaná pomocou nástroja SrcML. Tento nástroj predstavuje externý modul, ktorý je volaný pomocou systémového volania. Pre správny chod programu je tak potrebné mať daný modul nainštalovaný na stroji, kde je spustený webový server. Nástroj je voľne dostupný na vedecké účely, pričom posledná verzia 0.9.5 vyšla v máji 2015.

Na obrázku nižšie je uvedená metóda, ktorá umožňuje pomocou externého nástroja SrcML vykonať konverziu zdrojového kódu do XML reprezentácie a naspäť.

```
public void executeSrcML(String input, String output) {
    try {
        ProcessBuilder builder = new ProcessBuilder();
        builder.command("C:\\Program Files (x86)\\srcML\\bin\\srcML.exe", input, "-o", output);
        builder.redirectErrorStream(true);
        Process process = builder.start();
        process.waitFor();
        logger.info("File " + input + " was converted into " + output);
    } catch (IOException | InterruptedException e) {
        logger.error(e.getLocalizedMessage(), e);
    }
}
```

Obr. 37 Volanie nástroja SrcML z programovacieho jazyka Java

V uvedenej metóde možno vidieť volanie externého programu, ktorý sa nachádza na uvedenej štandardnej ceste. Vstup metódy predstavuje cesta k vstupnému súboru a požadovanému výstupnému súboru. Po volaní programu tak vznikajú príslušné XML, respektíve Java súbory podľa toho, či bola vykonaná konverzia zdrojového kódu do XML reprezentácie alebo spätná konverzia XML reprezentácie do zdrojového kódu.

7.3. Integrácia BaseX databázy do refaktorovacieho nástroja

Nástroj BaseX bol využitý ako základ celého refaktorovacieho nástroja. Keďže refaktorovací nástroj bol implementovaný primárne v programovacou jazyku Java, na pripojenie k XML databáze bolo využité štandardizované Java rozhranie pre napojenie na XML databázy XQJ API. Na *Obr. 38* je zobrazené vytvorenie pripojenia k databáze pomocou XQJ API.

```
public void createDatabase(String database) throws XQException {
    if(connection != null && !connection.isClosed()) {
        connection.close();
    }

    XQDataSource source = new BaseXXQDataSource();
    source.setProperty("serverName", "localhost");
    source.setProperty("port", "1984");

    connection = (XQConnection2) source.getConnection(name, password);
    connection.createExpression().executeCommand("CREATE DB " + database);
    connection.close();

    source = new BaseXXQDataSource();
    source.setProperty("serverName", "localhost");
    source.setProperty("port", "1984");
    source.setProperty("databaseName", database);
    connection = (XQConnection2) source.getConnection(name, password);
    getExpression().executeCommand("SET EXPORTER indent=no");
    getExpression().executeCommand("SET STRIPNS ON");
    getExpression().executeCommand("SET CHOP OFF");
}
```

Obr. 38 Pripojenie k BaseX databáze

Pred samotným pripojením sa uzatvára prípadné existujúce pripojenie. Následne sa vytvára pripojenie bez napojenia na konkrétnu databázu, pričom pre každého používateľa sa vytvára unikátna databáza. Toto pripojenie sa následne uzatvára a vytvára sa nové pripojenie už na novo vytvorenú konkrétnu databázu. Nad databázou sa hneď po pripojení vykonajú určité nastavenia tak, aby vyhovovala potrebám refaktorovacieho systému. Ide o nastavania, ktoré umožňujú zachovávať pôvodne biele znaky v dokumente a zabraňujú pridávaniu nových formátovacích bielych znakov pre XML dokument tak, aby formátovanie zdrojového kódu bolo možné zachovať.

Základ samotného navrhnutého procesu tvorí import a export dokumentov z databázy. Na obrázku nižšie sú uvedené metódy pre import dokumentov a export dokumentov z databázy.

```
public void insertDocument(FileInputStream file, String fileName) throws IOException, XQException {
    XQItem item = connection.createItemFromDocument(file, null, null);
    connection.insertItem(fileName, item, null);
}

public void exportDatabase(String path) throws XQException {
    expression.executeCommand("EXPORT " + path);
}
```

Obr. 39 Import a Export XML dokumentov z databázy pomocou XQJ API

Podobne, ako pri klasických SQL databázach, aj BaseX vyžaduje pred samotným pripojením mať spustený databázový server. Pred pripojením z jazyka Java musí byť spustený server pomocou príkazu BaseXServer. Následne je možné vykonávať nad databázou XQuery a XQuery Update skripty. Jeden zo spôsobov vykonania skriptu nad databázou z jazyka Java je zobrazený na *Obr. 40* uvedenom nižšie.

```
public void applyXQuery(String content) throws XQException {
    expression.executeQuery(content);
}
```

Obr. 40 Vykonanie XQuery skriptu nad BaseX databázou

Na *Obr. 40* je zobrazené vykonanie XQuery, alebo XQuery Update príkazov vo forme skriptu, ktorý je do metódy poslaný v textovej podobe pomocou premennej *content*. Následne je vytvorený XQuery výraz pomocou XQJ API a ten je vykonaný nad aktívnym pripojením k databáze.

Pre pokročilejšiu prácu s XQuery skriptami z jazyka Java je nutné uvažovať možnosť viazania externých premenných na volaný XQuery skript. V takomto prípade je potrebné v konkrétnom XQuery skripte zadefinovať premennú ako „*extern*“, čo znamená, že na ňu bude naviazaná hodnota pri vykonaní daného skriptu. Príklad takéhoto zadefinovania premennej môže byť nasledovný:

declare variable $tag external := “LPL1“

V takomto prípade je však nutné upraviť aj metódu zodpovednú za volanie XQuery skriptov. Metóda uvedená na obrázku nižšie umožňuje zavolať XQuery skript tak, že na základe udaného mena premennej je možné naviazať jednu externú hodnotu premennej na príslušný skript.

```
public void applyXQuery(String content, String variableName, String value) throws XQException {
    expression.bindString(new QName(variableName), value, null);
    expression.executeQuery(content);
}
```

Obr. 41 Viazanie premennej na skript pri vykonaní XQuery skriptu

Na *Obr. 41* možno vidieť naviazanie príslušnej hodnoty na konkrétnu premennú v skripte pomocou volania metódy XQJ API *bindString*. Metóda očakáva objekt typu *QName*, ktorý je taktiež súčasťou XQJ API a je vytvorený konštruktorom pomocou príslušného textového názvu premennej. Podobne ako na *Obr. 40,* je samotné vykonanie XQuery skriptu volané pomocou metódy *executeQuery* nad objektom s názvom *expression*, ktorý je typu *XQExpression*. Zabezpečenie vytvorenia príslušného *expression* objektu je uvedené v metóde na obrázku nižšie.

```
public XQExpression getExpression() throws XQException {
    if(expression == null || expression.isClosed()) {
        expression = connection.createExpression();
    }

    return expression;
}
```

Obr. 42 Vytvorenie Objektu zodpovedného za vykonávanie XQuery funkcionality nad XML databázou

7.4. Skripty v jazyku XQuery

Výsledkom doterajšej práce je niekoľko vyhľadávacích a opravovacích skriptov, ktoré obsahujú okrem samotného vyhľadania, respektíve opravovania, taktiež istú nutnú réžiu potrebné pre fungovanie automatizovaného refaktorovania. Opravovacie skripty okrem potrebnej logiky, ktorá bola rozoberaná v kapitole 4.3 neobsahujú dodatočné implementačné záležitosti, no dopyty pomocou jazyka XPath, ako boli uvedené v kapitole 4.2, je nutné obaliť do XQuery skriptov, ktoré zabezpečia aj dodatočné zaznamenávanie potrebných informácií a podobne. Nižšie sú podrobnejšie rozobraté niektoré skripty vychádzajúce z uvedených XPath dopytov.

Empty Catch Clausule

Tento jednoduchý anti-vzor predstavuje stav, kedy je zachytená výnimka úplne ignorovaná. Znamená to teda, že catch blok neobsahuje žiadny zdrojový kód.

Problém v kóde je možné jednoducho vyhľadať tak, že sa skontroluje každý uzol, ktorý predstavuje catch blok, či obsahuje nejaké ďalšie, vnútorné uzly. *Obr. 43* predstavuje skript na vyhľadanie a označenie takejto situácie.

```
declare variable $resultFile external;

let $code := "ECC"
for $node at $position in //catch[count(block/child::node()) <= 1]
return (
replace node $node with
  element {xs:QName(concat($code, $position))}
  {
    $node
  },
insert node
  <comment type="line">//Refactor - Empty Catch Clausule</comment>
  before $node,
file:append($resultFile, concat("NAME: ", $code, $position, "&#10;"))
)
```

Obr. 43 *Empty Catch Clausule* XQuery vyhľadávanie

Ako možno vidieť, na začiatku je uvedená premenná, na ktorú sa pri vykonaní viaže externá hodnota uvádzajúca cestu k súboru, do ktorého sú zapisované informácie o nájdených problémoch v zdrojovom kóde. Následne je definovaná premenná *code*, ktorá určuje všeobecný identifikátor daného pachu, respektíve anti-vzoru. Pre každý nájdený pach daného typu sa zvyšuje hodnota počítadla, čím sa vytvárajú unikátne kódy. Informácie o nájdených pachoch je následne zaznamenaná do udaného súboru, na základe ktorého sa vytvárajú informácie pre Jess expertný systém. V tomto bode je možné zaznamenať aj prípadný kontext problému, napríklad pozíciu alebo dĺžku, čo v tomto prípade nie je potrebné.

Catch and rethrow

Anti-vzor, kedy programátor vyhadzuje rovnakú výnimku, ako práve zachytil. V catch bloku sa teda nachádza vyhodenie zachytenej výnimky.

```
declare variable $resultFile external;

let $code := "CR"
for $node at $position in //try[catch/parameter_list/parameter/decl/name/text()=
catch/block//throw/expr/name/text()]
let $size := count($node/catch/block/child::node())
return (
replace node $node with
  element {xs:QName(concat($code, $position))}
  {
    $node
  },
insert node
    <comment type="line">//REFACTOR - Catch and Rethrow</comment>
    before $node,
file:append($resultFile, concat("NAME: ", $code, $position, "&#10;")),
file:append($resultFile, concat("SIZE: ", $size - 2, "&#10;"))
)
```

Obr. 44 *Catch and Rethrow* XQuery vyhľadávanie

Nájdenie spočíva v skontrolovaní, či meno vyhodenej výnimky je rovné menu zachytenej výnimky. Toto zabezpečuje vnorený XPath výraz navrhnutý v kapitole 4.2, ktorá sa udáva do centrálneho for výrazu.

V tomto XQuery dopyte je situácia podobná, ako v predchádzajúcom prípade. Okrem aspektov uvádzaných pri predchádzajúcom skripte je dôležité všimnúť si pridanie komentárovej značky identifikujúcej príslušný pach. Táto značka následne slúži pri vizualizácií nájdených pachov. Okrem identifikátora pachu možno vidieť aj záznam veľkosti *catch* bloku, čo slúži ako podklad pre rozhodovanie expertného systému.

Long parameter list

Ide o jeden z typických pachov definovaných Martinom Fowlerom. Prejavuje sa tak, že metóda má príliš mnoho parametrov. Takémuto zoznamu je vo väčšine prípadov možné sa vyhnúť, napríklad pomocou vytvorenia objektu, ktorý by takéto parametre reprezentoval.

Pach kódu je možné odhaliť kontrolou počtu parametrov jednotlivých metód. Vyhľadanie a označenie je zobrazené na *Obr. 45* uvedenom nižšie.

```
import module namespace functx = "http://www.functx.com";
declare variable $resultFile external;

let $code := "LPL"
for $node at $position in //function[parameter_list[count(parameter) > 5]]
return (
replace node $node with
  element {xs:QName(concat($code, $position))}
  {
    $node
  },
insert node
  <comment type = "line">//REFACTOR - Long Parameter List</comment>
  before $node,
  file:append($resultFile, concat("NAME: ", $code, $position, "&#10;")),
  file:append($resultFile, concat("PATH: ", functx:path-to-node($node), "&#10;"))
)
```

Obr. 45 *Long Parameter List* XQuery vyhľadávanie

Tento XQuery skript okrem vlastností prezentovaných na predchádzajúcich príkladoch taktiež zaznamenáva hierarchiu uzla, pomocou funkcie *path-to-node*. Toto predstavuje veľmi dôležitú informáciu, pri rozhodovaní o možných závislostiach pachov na danom mieste. Pre umožnenie vytvorenia daného záznamu je potrebné využiť externú knižnicu functx. Jej zadefinovanie je uvedené na prvom riadku daného skriptu.

7.5. Zapojenie expertného systému

Nástroj Jess sa okrem iného poskytuje aj ako jar knižnica, ktorú možno využiť pri implementácii aplikácií. Knižnice sú však obmedzené trial licenciou a je ich tak možné používať iba jeden mesiac. V tejto práci bola využitá trial licencia produktu, preto bol výskum expertného systému vykonávaný s obmedzeniami. Výsledný prototyp tak nie je možné používať na akékoľvek iné, ako vedecké účely. Prebieha však snaha o získanie plnej licencie určenej na vedecké účely.

Príslušné knižnice sa musia nachádzať jednak v *lib* adresári v priečinku, kde sa nachádzajú knižnice Javy a zároveň v *lib* adresári webového servera. Expertný systém číta pravidlá zo súboru s príponou *clp*. Pravidlá sú preto uvádzané a uchovávané v tomto súbore ukladanom v *config* adresári aplikácie, ku ktorému je následne možné pristúpiť z príslušného webového servera.

Expertný systém pracuje na základe *Rete* objektu, ktorý predstavuje jeden pravidlový stroj. Pomocou vytvárania viacerých inštancií *Rete* triedy je tak možné vytvárať viaceré pravidlové stroje. Vytvorenie nového *Rete* objektu je taktiež dôležité pri vykonávaní nového odvodzovania. Vykonanie odvodzovania požadovaných refaktorovacích metód je možné vidieť na obrázku nižšie.

```
public List<JessOutput> run(List<JessInput> searchResults) {
    List<JessOutput> refactoringRules = new ArrayList<>();

    for(JessInput input : searchResults) {
        try {
            rete = new Rete();
            reloadRules();
            rete.add(input);
            rete.run();
        } catch (JessException e) {
            e.printStackTrace();
        }

        Iterator<?> result = rete.getObjects(new Filter.ByClass(JessOutput.class));
        while(result.hasNext()) {
            refactoringRules.add((JessOutput) result.next());
        }
    }

    return refactoringRules;
```

Obr. 46 Vytvorenie nového Jess pravidlového stroja pomocou Java Jess API

Vstupom je zoznam identifikovaných faktov, pričom výstup je následne spracovaný zoznam požadovaných refaktorovacích metód.

8. Evaluácia riešenia a vyhodnotenie navrhovanej metódy

Táto kapitola popisuje v krátkosti spôsob vyhodnocovania navrhovaného prístupu k automatizovanému refaktorovaniu softvérových systémov. Na vyhodnocovanie prístupu bol využitý prototyp, ktorý bol implementovaný ako výstup tejto práce demonštrujúci navrhovaný prístup.

Nástroj bol testovaný na menších a stredne veľkých projektoch obsahujúcich určité problémy v kóde. Išlo buď o špeciálne zanesené problémy pre testovanie konkrétneho vyhľadávania a opravy, alebo všeobecné problémy bežne sa nachádzajúce v zdrojových kódoch.

Pri testovaní boli zistené nasledujúce nedostatky:

- *Pri vkladaní komentárovej značky môže vloženie pokaziť formátovanie kódu* – nedostatok vyplýva z problémov s formátovaním zdrojového kódu pri reprezentácií pomocou XML. Pri vkladaní nového uzla nie je možné zistiť správne odsadenie a podobne. Nedostatok je možné riešiť implementáciou vlastného prevádzača zdrojového kódu do XML reprezentácie tak, aby bolo možné pracovať aj s bielymi znakmi.
- *XML elementy zanesené pre identifikáciu pachu môžu zasiahnuť do navigáciu XPath dopytov* - pri vytváraní dopytov pomocou jazyka XPath je treba dôsledne uvažovať možnosť výskytu zanesenej značky identifikujúcej pach v kóde. Dopyty je potrebné vytvárať tak, aby neboli úplne závislé na štruktúre XML dokumentu ale dokázali pracovať aj s prípadnými dodatočnými značkami, ktoré je rovnako potrebné vkladať tak, aby ovplyvnili v čo najmenšej miere štruktúru dokumentu.
- *Vytváranie nových časti zdrojového kódu môže zaniesť objekty / metódy a podobne s rovnakým názvom* – pri refaktorovaní je problém automatizovane určiť vhodný názov pre novo zanášané elementy do zdrojového kódu. Aktuálne riešenie je také, že názov elementu je odvodený z existujúceho elementu, alebo je poskytnutý generický názov. Tento problém aktuálne nie je možné riešiť a je možné ho identifikovať aj pri manipulácií s kódom v niektorých vývojových prostrediach. Tejto téme by bolo potrebné venovať sa v samostatnej výskumnej práci.

- *Refaktorizácia výrazne poškodzuje formátovanie kódu* – pomocou aktuálne použitej metódy nie je možné vykonávať refaktorovanie tak, aby bolo zachované dobré formátovanie kódu pri novo vkladaných častiach. Riešením je ďalší, dlhodobý vývoj nástroja tak, aby umožňoval automaticky pracovať so zavedeným formátovaním kódu a vkladať nové elementy podľa zaužívanej formátovacej konvencie.

Program vykazuje veľmi dobré výsledky hlavne pri procese vyhľadávania pachov v kóde. Očakávaný výstup vyhľadávania bol dosiahnutý viac-menej v každom prípade, pokiaľ nenastáva druhý uvedený problém. Pri identifikácií takéhoto problému je potrebné upraviť vyhľadávacie dopyty.

Horšie výsledky sú dosahované pri procese refaktorovania. Samotné refaktorovanie prebieha bez problémov a podľa očakávaní, no problém predstavujú záležitosti vyžadujúce ďalší výskum a vývoj, ako napríklad zachovávanie formátovania a dodržovanie menného priestoru daného programu.

Systém nie je možné porovnať s existujúcimi systémami, keďže dostatočne dobrý, voľne dostupný nástroj, schopný okrem opravy aj vykonávať refaktorovanie nájdených problémov, v podstate neexistuje. Väčšina dostupných nástrojov, ako napríklad Sonar, alebo PMD dokáže dané problémy len identifikovať, nie však opravovať. Exituje niekoľko málo nástrojov, zameraných aj na opravu istej úzkej skupiny pachov, ako napríklad JDeodorant, no tento nástroj sa nezameriava na rovnakú skupinu pachov.

Oproti nástrojom Sonar, PMD, alebo iPlasma, ktoré oproti nástroju *Refactor* dokážu vyhľadávať podstatne väčšie množstvo pachov je výhodou nástroja to, že sa snaží tieto pachy aj opravovať. Nástroj je taktiež veľmi prispôsobiteľný pomocou doplnenia vyhľadávacích a opravovacích skriptov a možnosti riadiť spôsob refaktorovania na základe pravidiel s možnosťou uvažovania preferencii používateľa, či aktuálneho kontextu pachu v zdrojovom kóde.

Napriek nepopierateľnej kvalite dlhodobo vyvíjaných nástrojov, ako Sonar či PMD, s ktorými je nástroj ťažko porovnateľný aj v oblasti vyhľadávania pachov, keďže umožňuje vyhľadávať veľmi obmedzenú skupinu pachov, je potrebné povedať, že nástroj *Refactor* prináša zaujímavý koncept riadenia automatizovaného refaktorovania pomocou pravidiel, ktorý v uvedenej podobe nebol identifikovaný v žiadnom existujúcom nástroji.

9. Zhodnotenie a ďalšie smerovanie

Diplomová práca prezentovala základné možnosti automatizovania refaktorovania softvérových systémov pomocou jazyka XQuery. Ako súčasť práce boli zhodnotené viaceré prístupy k reprezentácii zdrojového kódu a možnosti refaktorovania nad jednotlivými prístupmi. Jadro tvorila analýza automatizovania refaktorovania pomocou jazyka XQuery, ktorý bol použitý na vyhľadávanie anti-vzorov a pachov kódu, ale aj samotné refaktorovanie.

Návrh samotného nástroja umožňujúceho automatizovanie refaktorovania tvorí druhú časť práce. V rámci návrhu boli identifikované možnosti a spôsoby použitia XML dokumentových databáz, ktoré tvoria vhodný zdroj údajov pre jazyk XQuery a XQuery Update, ktorý slúži ako dopytovací jazyk nad týmto druhom NoSQL databáz. Ako súčasť návrhu boli taktiež uvedené možnosti pre zakomponovanie expertného systému JESS, ktorý slúži pre rozhodovanie o efektívnom spôsobe refaktorovania a ponúka možnosť ovplyvňovať spôsob automatizovaného refaktorovania pre používateľa pomocou zmeny príslušných pravidiel.

Druhú dôležitú časť návrhu tvorila identifikácia pozitívnych a negatívnych vzťahov medzi pachmi kódu. Na základe tejto identifikácie boli navrhované prístupy k riešeniu optimalizácie refaktorovacieho procesu tak, aby bol dosiahnutý optimálny výsledok.

Práve táto oblasť ponúka priestor pre ďalšiu výskumnú prácu zameranú na výskum identifikácie vhodného refaktorovacieho postupu – pomocou grafových algoritmov, biologicky inšpirovaných algoritmov, či lineárneho programovania. V tejto oblasti existuje veľa priestoru pre ďalšiu výskumnú prácu, ktorej výsledkom môže byť veľmi vyspelý refaktorovací nástroj.

Ďalší výskum je nutné zamerať aj na riešenie technických problémov aktuálneho prototypu. Tými sú hlavne formátovanie zdrojového kódu pri vkladaní nových častí, čo pomocou jazyka XQuery nie je dostatočne možné, alebo aj identifikácia a dodržiavanie menného priestoru pri generovaní nových častí zdrojového kódu.

Ako možno vidieť, oblasť refaktorovania tvorí perspektívnu oblasť, kde je ešte mnoho priestoru na výskum. Metóda navrhovaná v tejto práci môže slúžiť ako podklad pre moderný refaktorovací nástroj umožňujúci automatizované refaktorovanie širokého počtu pachov v zdrojovom kóde. Taký nástroj však vyžaduje dlhodobý vývoj skúseným vývojovým tímom.

Bibliografia

[1] M. Fowler, K. Beck, J. Brant, W. Opdyke and D. Roberts, Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999.

[2] A. Koenig, "Patterns and Antipatterns," *Journal of Object-Oriented Programming,* vol. 8, pp. 46-48, 1995.

[3] L. Markovič, „Podpora refaktoringu pomocou transformácií medzi jazykom Java a XML," Slovenská technická univerzita v Bratislave, Bratislava, 2014.

[4] F. Simon, F. Steinbrückner a C. Lewerentz, „Metrics based refactoring," rev. *Fifth European Conference on Software Maintenance and Reengineering*, Lisbon, 2001.

[5] R. Mahouachi, M. Kessentini a M. Ó. Cinnédele, „Search-Based Refactoring Detection Using Software Metrics Variation," rev. *5th International Synoposium, SSBSE 2013*, St. Petersburg, 2013.

[6] T. Kuhn and O. Thomann, "Abstract Syntax Tree," 20 November 2006. [Online].

[7] P. Palát, "Automatizovaný vyhľadávač pachov a návrhových antivzorov," Fakulta Informatiky a Informačných Technológií STU, Bratislava, 2014.

[8] K. Kontogiannis and E. Mamas, "Towards Portable Source Code Representations Using XML," in *7th Working Conference on Reverse Engineering, WCRE 2000*, Brisbane, 2000.

[9] M. L. Collard, M. J. Decker a J. I. Maletic, „srcML: An Infrastructure for the Exploration, Analysis, and Manipulation of Source Code: A Tool Demonstration," rev. *20th IEEE International Conference on Software Maintenance (ICSM'2014)*, Eindhoven, 2013.

[10] J. Robie, D. Chamberlin a M. Dyck, „XQuery 3.0: An XML Query Language," W3C, 8 4 2014. [Online]. Available: http://www.w3.org/TR/xquery-30/.

[11] J. Robie, D. Chamberlin, M. Dyck, D. Florescu, J. Melton a J. Siméon, „XQuery Update Facility 1.0,“ W3C, 17 Marec 2011. [Online]. Available: http://www.w3.org/TR/xquery-update-10/.

[12] N. C. Mendonça, P. H. M. Maia, L. A. Fonseca a R. M. C. Andrade, „RefaX: A Refactoring Framework Based on XML,“ rev. *20th IEEE International Conference on Software Maintenance (ICSM'04)*, Chicago, 2004.

[13] G. J. Badros, "JavaML: A Markup Language for Java Source Code," in *9th International World Wide Web Conference, WWW 2000*, Amsterdam, 2000.

[14] F. A. Fontana, E. Mariani, A. Morniroli, R. Sorman a A. Tonello, „An experience report on using code smells detection tools,“ rev. *Fourth International Conference on Software Testing, Verification and Validation Workshops, ICST 2011*, Berlín, 2011.

[15] E. Murphy-Hill a A. P. Black, „An Interactive Ambient Visualization for Code Smells,“ rev. *5th international symposium on Software visualization, SOFTVIS 2010*, 2010.

[16] C. Marinescu, R. Marinescu, P. F. Mihancea, D. Ratiu a R. Wettel, „iPlasma: An Integrated Platform for Quality Assessment of Object-Oriented Design,“ rev. *21th IEEE International Conference on Software Maintenance and Evolution, ICSM 2005*, Budapest, 2005.

[17] L. Markovič, „Refactoring support using XSLT transformations,“ rev. *Informatics and Information Technologies Student Research Conference (IIT.SRC2014)*, Bratislava, 2014.

[18] E. Friedman-Hill, „Jess® The Rule Engine for the Java™ Platform,“ Sandia National Laboratories, 2008.

Príloha A – JSON reprezentácia kódu uvedeného na *Obr. 3*


```
{ "function" : { "block" : { "if" : { "condition" : { "expr" : { "name" : "a",
                      "text" : "== 0"
                    },
                  "text" : [ "(",
                      ")"
                    ]
                },
              "else" : { "block" : { "return" : { "expr" : { "call" :
{ "argument_list" : { "argument" : [ { "expr" : { "name" : "a" } },
                                          { "expr" : { "name" : [ "b",
                                                       "a"
                                                     ],
                                                   "text" : "%"
                                                 } }
                                        ],
                                      "text" : [ "(",
                                          ",",
                                          ")"
                                        ]
                                    },
                                  "name" : "nsd"
                                } },
                            "text" : [ "return",
                                ";"
                              ]
                          },
                        "text" : [ "{",
                            "}"
                          ]
                      },
                    "text" : "else"
                  },
                "text" : "if",
                "then" : [ { "return" : { "expr" : { "name" : "b" },
                             "text" : [ "return",
                                 ";"
                               ]
                           },
                         "text" : [ "{",
                             "}"
                           ]
                       } ]
            },
          "text" : "{"
        },
      "name" : "nsd",
      "parameter_list" : { "param" : [ { "decl" : { "name" : "a",
                     "type" : { "name" : "int" }
                   } },
               { "decl" : { "name" : "b",
                     "type" : { "name" : "int" }
                   } }
            ],
          "text" : [ "(",
              ",",
              ")"
            ]
        },
      "type" : { "name" : "int",
          "specifier" : "public"
        }
    } }
```


Obr. 47 JSON reprezentácia kódu uvedeného na *Obr. 3*

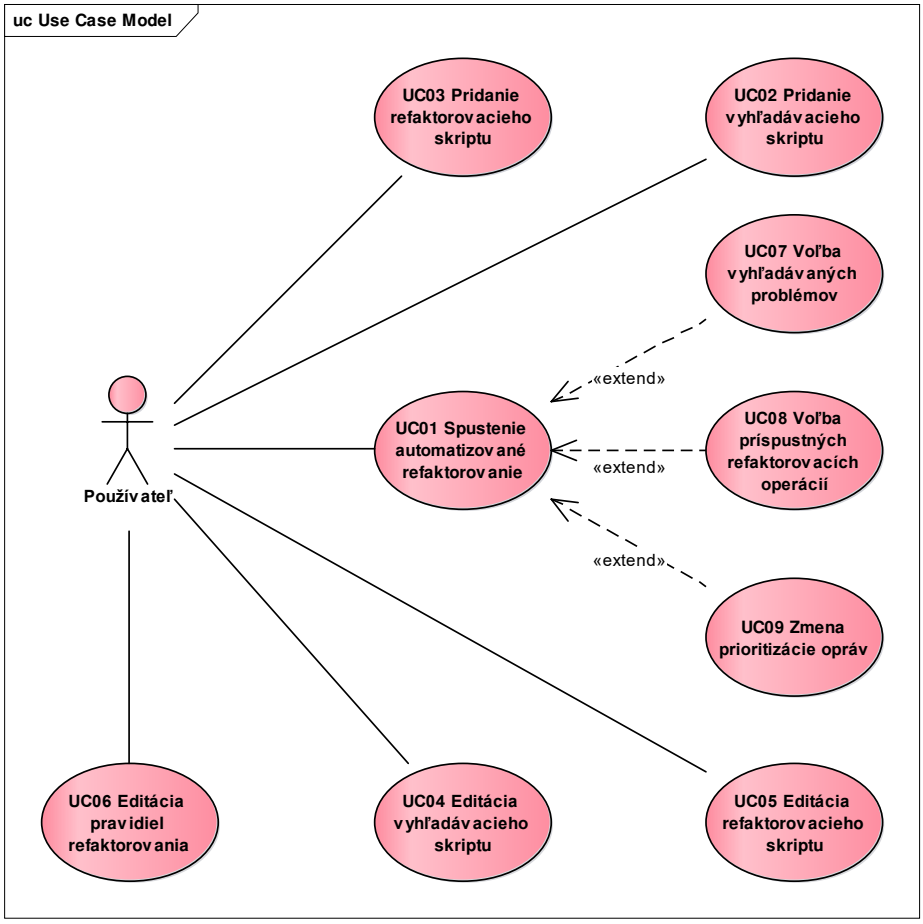
Príloha B – Technická dokumentácia

Táto príloha obsahuje stručnú technickú dokumentáciu. Dokumentácia obsahuje základné časti, ako zjednodušenú špecifikáciu systému, implementáciu, inštalačnú príručku a používateľskú príručku.

Špecifikácia systému

Systém *Refactor* slúži na vyhľadávanie a opravovanie anti-vzorov a pachov v kóde. Systém je postavený tak, aby bol plne prispôsobiteľný a umožňoval široké možnosti konfigurácie pre používateľa. Do konfigurácie sa zahŕňa možnosť pridávať, alebo upravovať dopyty pre vyhľadávanie problémov v zdrojovom kóde, refaktorovacie metódy zabezpečujúce samotné refaktorovanie a pravidlá, určujúce, akým spôsobom prebieha automatizovaná reakcia na nájdený problém.

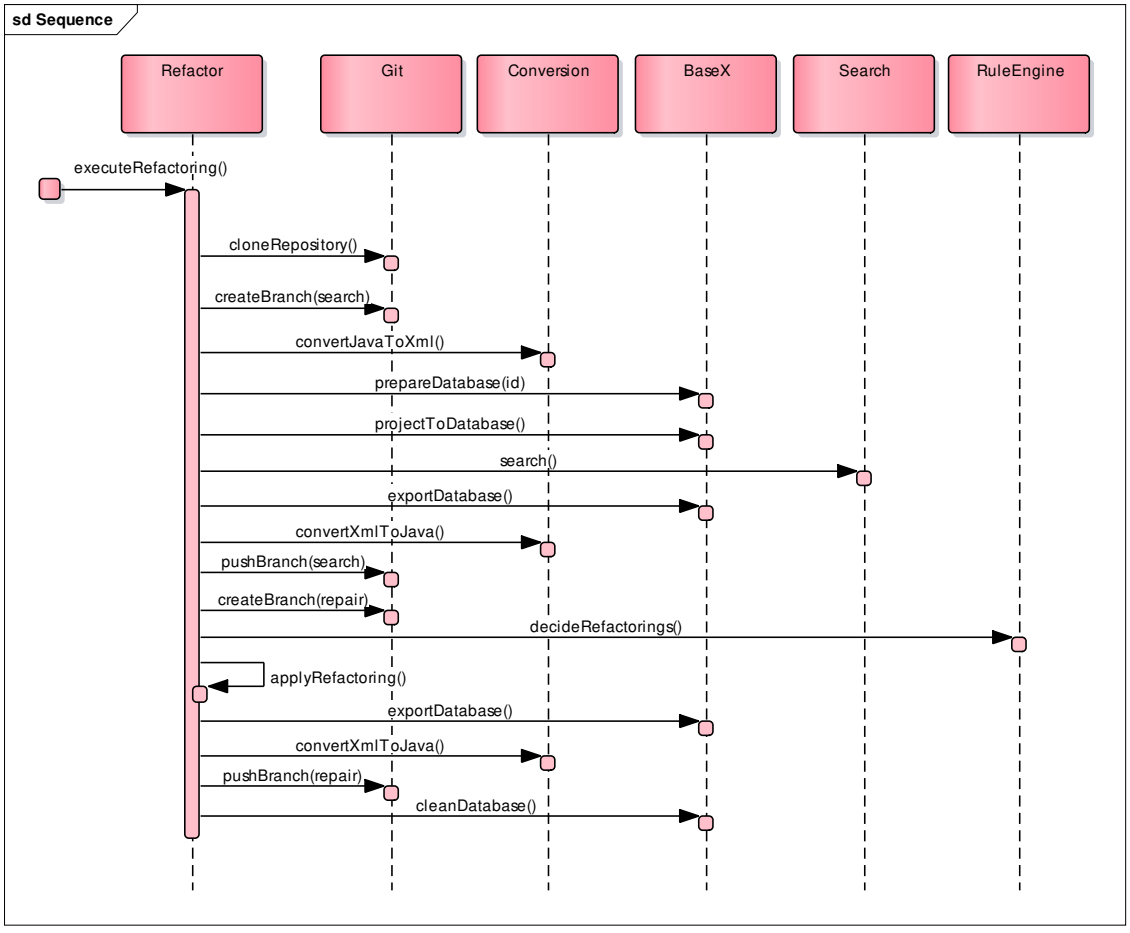
Na Obrázku nižšie sú uvedené prípady použitia systému *Refactor*.



Obr. 48 Prípady použitia systému

Prípady použitia sú navonok veľmi jednoduché a preto nebudú hlbšie popisované. Hlavný prípad použitia predstavuje *UC01 Spustenie automatizovaného refaktorovania*. Systém pracuje nad Git repozitárom, ktorý musí používateľ uviesť rovnako, ako prístupové údaje k nemu. Systém vytvorí na repozitári 2 nové branch, ktorých názov je taktiež nutné uviesť. V prvej branch sú následne vyznačené identifikované pachy a v druhej poskytnutý návrh refaktorovania daného systému. Refaktorovanie, rovnako ako vyhľadávanie je možné ovplyvniť pomocou výberu problémov pre vyhľadávanie a taktiež prípustných refaktorovacích metód. Ovplyvnenie prioritizácie predstavuje ďalšie informácie na základe ktorých možno riadiť proces refaktorovania pomocou pravidiel v prípade konfliktov.

Na obrázku uvedenom nižšie je zobrazený sekvenčný diagram činnosti refaktorovacieho nástroja. Diagram zobrazuje iba jadro priebehu refaktorovacieho procesu, nie celého systému.



Obr. 49 Sekvenčný diagram spracovania refaktorovacej požiadavky

Ako možno vidieť, systém je zložený z niekoľkých modulov, pričom proces refaktorovania riadi modul *Refactor*, ktorý postupne volá služby iných modulov. Ide o zjednodušený sekvenčný pohľad, kde sú zobrazené iba volania modulov, nie vnútorné operácie modulov.

Implementácia

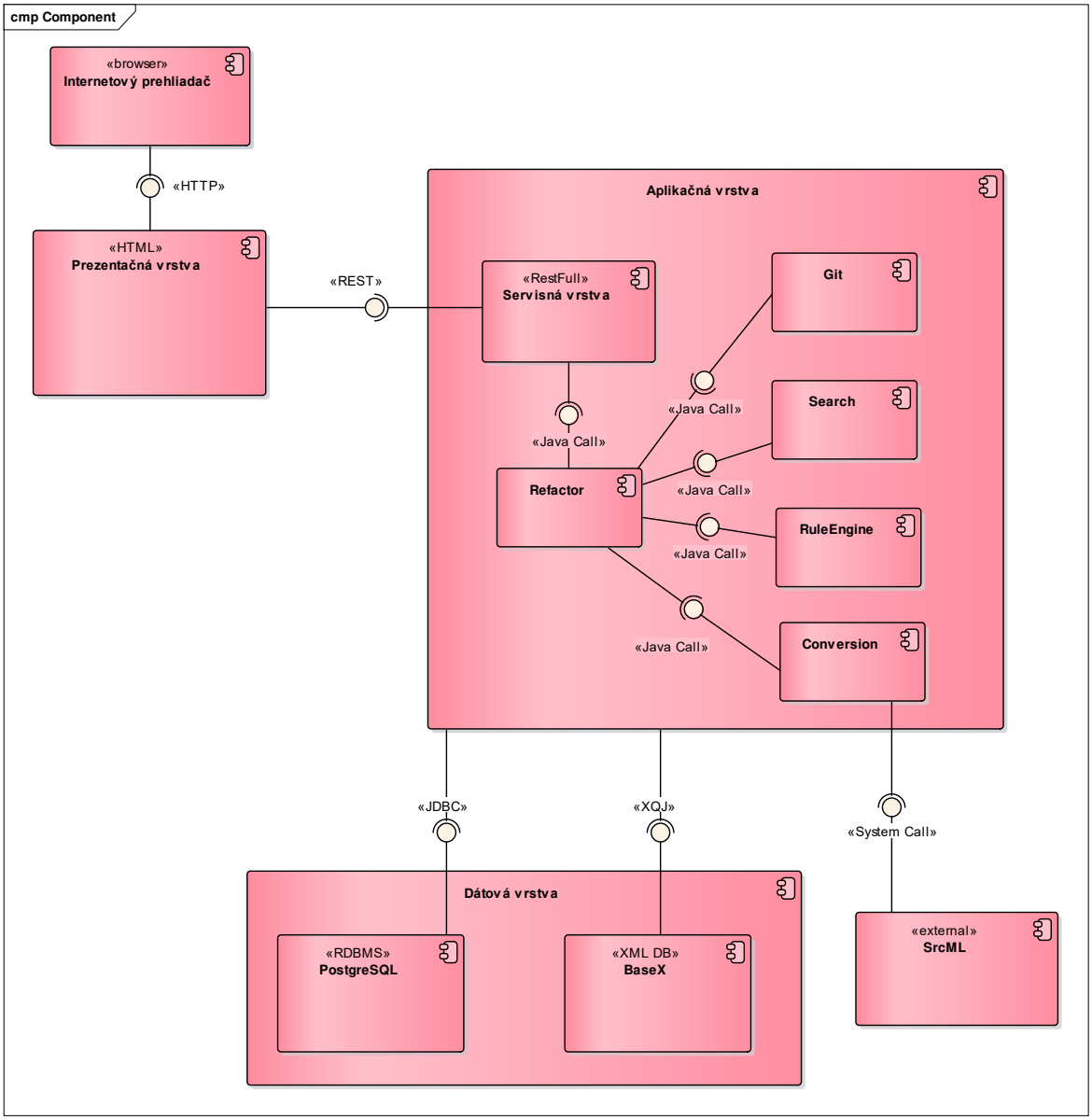
Táto časť technickej dokumentácie popisuje základné implementačné rozhodnutia, a poskytuje pohľad na architektúru systému implementačný pohľad prostredníctvom diagramu tried. Systém sa nachádza vo fáze prototypu, preto sa môžu vyskytovať chyby a iné technické problémy.

Prototyp systému bol implementovaný prostredníctvom viacerých technológií a externých modulov. Medzi najdôležitejšie použité technológie patria:

- Systém bol implementovaný v jazyku Java SE verzie 1.8.
- Ako databázový systém pre prácu s projektom v XML reprezentácii a ako procesor XPath, XQuery a XQuery Update bola použitá XML dokumentová databáza BaseX verzie 8.4.4
- Pre pripojenie k XML databáze bola využitá knižnica XQJ verzie 1.4.
- V programe je použitý expertný systém Jess vo verzii 7.1p2, ktorý je na akademické účely využitý prostredníctvom skúšobnej verzie a príslušné Java knižnice s rovnakého softvérového balíka.
- Projekt používa Git rozhranie pre prístup k projektovým repozitárom, na čo bola použitá knižnica JGit.
- Nástroj obsahuje externý modul SrcML vo verzii 0.9.5, pre konverziu zdrojového kódu do XML reprezentácie a naopak, použitý vo voľnej licencii pre akademické účely.
- Ako úložisko refaktorovacích metód a vyhľadávacích operácií sa využíva relačná databáza PostgreSQL vo verzii 9.5.
- Systém vystavuje webové služby pre komunikáciu s webovým rozhraním. Webové služby sú typu RestFull, pričom na implementáciu boli využité rôzne dodatočné knižnice pre prácu s rest webovými službami a formátom JSON.
- Systém poskytuje webové rozhranie, implementované pomocou jazyka HTML5 a CSS3.
- Pre dynamickú stránku webového rozhrania je využívaný jazyk JavaScript a framework jQuery.

Prototyp systému bol vytvorený vo vývojovom prostredí Eclipse, najnovšej verzie, ako Maven projekt, teda s využitím automatizovaného zostavovacieho nástroja Maven v najnovšej verzii. Počas implementácie bol využívaný webový server Apache Tomcat najnovšej verzie.

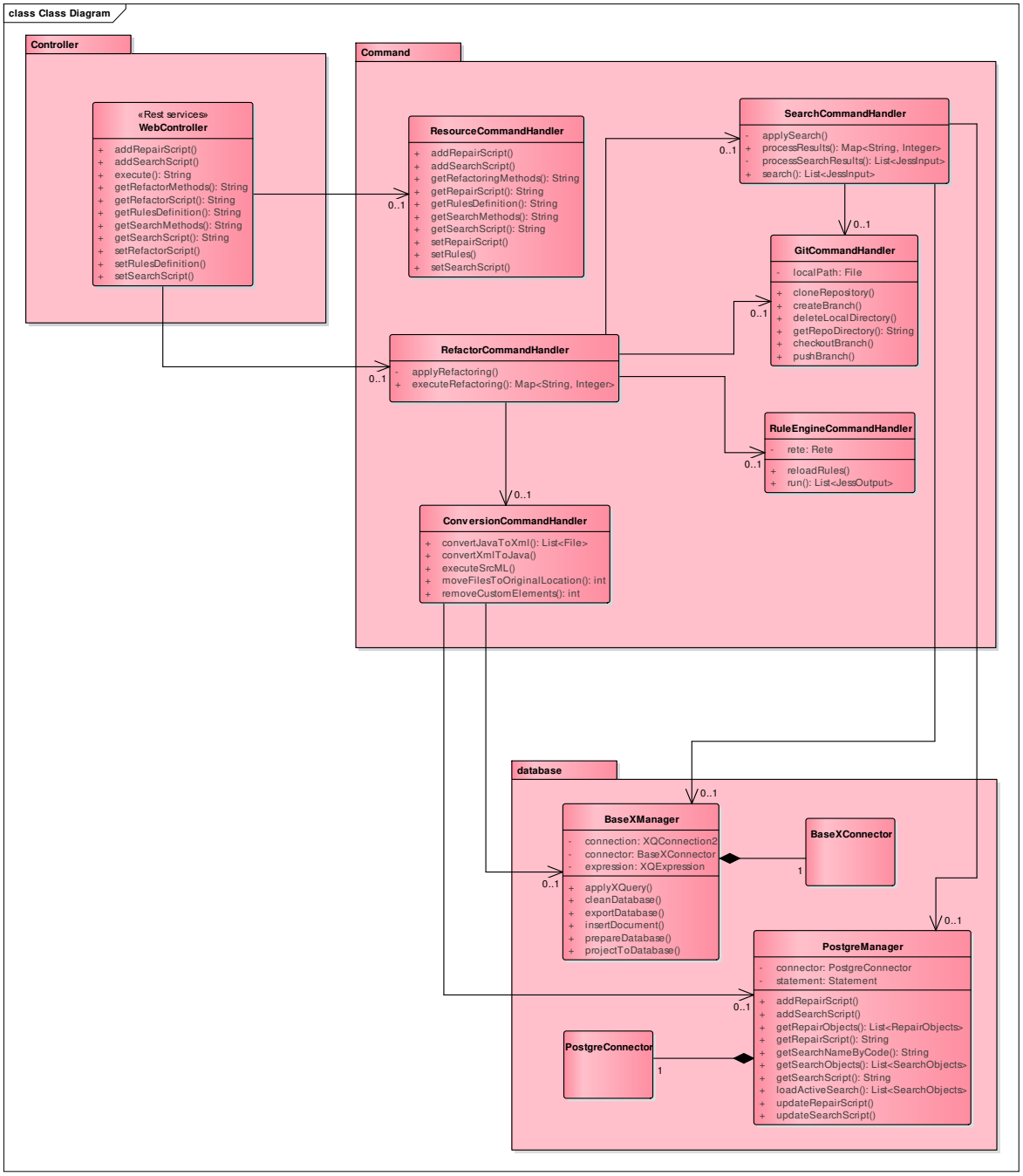
Na obrázku nižšie je uvedený funkcionálny model systému prostredníctvom diagramu komponentov znázorňujúci architektúru systému.



Obr. 50 Architektúra prototypu systému

Pre podrobnejší pohľad na architektúru aplikačnej vrstvy je možné pozrieť diagram komponentov uvedený pri návrhu systému v kapitole 4.

Systém do veľkej miery pozostáva z integrácie rôznych modulov slúžiacich na určitý účel. Veľká časť zdrojového kódu systému pozostáva z integrácie týchto modulov a ich manažmentu. Na obrázku nižšie je uvedený diagram tried systému. Z diagramu sú vynechané málo podstatné triedy a neobjektové časti systému – webové rozhranie, refaktorovacie skripty a podobne.



Obr. 51 Diagram tried systému *Refactor*

Diagram tried je uvedený v zjednodušenej podobe a zobrazuje iba základnú stavbu systému. V diagrame vzhľadom na rozmery taktiež nie sú uvedené doménové triedy, ako JessInput alebo JessOutput, ktoré sa používajú naprieč celým zdrojovým kódom na prenos údajov.

Inštalačná príručka

Napriek tomu, že systém obsahuje webové rozhranie, zatiaľ nie je možné pristupovať k systému prostredníctvom internetu. Pre spustenie programu na vlastnom stroji je nutné vykonať niekoľko krokov. Inštalačná príručka je platná pre systém Windows.

1. Inštalácia JDK – na systéme je potrebné mať nainštalovanú Javu o minimálnej verzii 1.8
2. Inštalácia BaseX – na stránke http://basex.org/products/download/all-downloads/ je potrebné stihnúť najnovšiu verziu BaseX databázy. Ideálne je stiahnúť inštalačný súbor vo forme *.exe súboru. Pre inštaláciu je potrebné mať v systéme nainštalovanú Javu.
3. Spustenie BaseX servera – po inštalácii BaseX databázy je potrebné spustiť BaseX server tak, aby bolo možné pripojiť sa k databáze. V príkazovom riadku je možné spustiť BaseX server pomocou príkazu BaseX.
4. Inštalácia PostgreSQL – do systému je potrebné nainštalovať databázu PostgreSQL a vykonať import databázy z príslušného dump súboru nachádzajúceho sa v adresári projektu.
5. Inštalácia SrcML – zo stránky http://www.srcml.org/downloads.html je potrebné stiahnuť inštalačný súbor pre program SrcML. Po inštalácií systému by sa mali automaticky vytvoriť systémové premenné tak, že program SrcML je možné zavolať ako príkaz z príkazového riadku.
6. Tomcat server – pre beh systému je potrebné nakonfigurovať Tomcat server (prípadne iný). Pridať a nakonfigurovať server je možné napríklad prostredníctvom vývojového prostredia Eclipse. Do konfiguračného súbory serveru web.xml je následne potrebné pridať nasledujúci XML fragment:

```
<filter>
        <filter-name>CorsFilter</filter-name>
        <filter-class>org.apache.catalina.filters.CorsFilter</filter-
class>
        <init-param>
         <param-name>cors.allowed.origins</param-name>
         <param-value>*</param-value>
           </init-param>
           <init-param>
               <param-name>cors.allowed.methods</param-name>
               <param-value>GET,POST,HEAD,OPTIONS,PUT,DELETE</param-value>
           </init-param>
```

```
</filter>
<filter-mapping>
  <filter-name>CorsFilter</filter-name>
  <url-pattern>/*</url-pattern>
      </filter-mapping>
```

Vďaka tomu bude umožnené správne volanie webových služieb.

7. Inštalácia Jess expertného systému – pre správny beh Jess systému je potrebné pridať príslušné knižnice do *lib* v inštalačnom priečinku prostredia Java a taktiež *lib* priečinku Tomcat serveru. Vzhľadom na to, že k dispozícii je iba trial verzia produktu, je potrebné knižnice stiahnuť s 30 dňovým obmedzením zo stránky http://www.jessrules.com/jess/download.shtml.
8. Následne je možné projekt nasadiť na server napríklad prostredníctvom prostredia Eclipse.
9. Rozhranie aplikácie je možné spustiť pomocou html súboru Refactor.html umiestneného v adresári Refactor-UI/Pages.

Spustenie projektu je pomerne komplikované a náročné. Spôsobené je to tým, že pre beh samotného programu sú vyžadované rôzne externé programy a moduly. Problémy spôsobuje hlavne nedostupnosť plnej verzie Jess expertného systému.

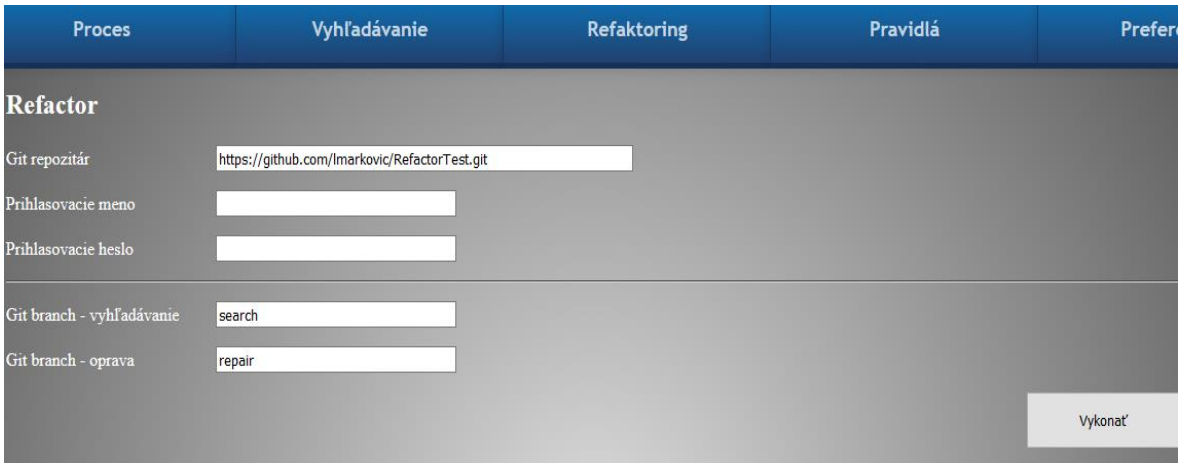
Používateľská príručka

Nástroj *Refactor* obsahuje veľmi jednoduché používateľské rozhranie, implementované ako webové. Pomocou nástroja je možné vykonávať manažment jednotlivých vyhľadávacích a opravovacích skriptov a príslušných pravidiel. Nastavenie a vykonanie procesu refaktorovania je hlavným účelom nástroja. Rozhranie nástroja je rozdelené na niekoľko častí:

1. Proces – hlavná obrazovka, na ktorej možno spúšťať vykonanie refaktorovania
2. Vyhľadávanie – manažment vyhľadávacích skriptov
3. Refaktoring – manažment refaktorovacích skriptov
4. Pravidlá – možnosť úpravy pravidiel expertného systému
5. Preferencie – úprava prioritizácie opráv
6. O programe – stručné informácie o pragrame

1. Proces

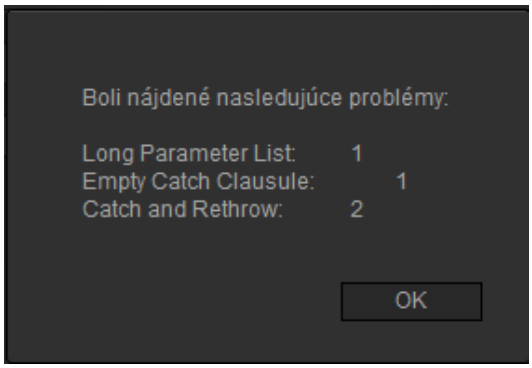
Na obrazovke proces je používateľovi umožnené nastaviť všetky potrebné informácie pre proces refaktorovania. Daná obrazovka je zachytená na obrázku nižšie.



Obr. 52 Hlavná obrazovka systému

- Do poľa *Git repozitár* používateľ zadá požadovaný projekt spravovaný pomocou systému Git
- Do polí *prihlasovacie meno* a *prihlasovacie heslo* je potrebné zadať prístupové údaje k príslušnému repozitáru
- Do polí *Git branch - vyhľadávanie* a *Git branch – oprava* je nutné zadať názvy branch, ktoré budú vytvorené pre identifikáciu pachov v kóde a návrh opráv daných pachov

Samotný proces refaktorovania je možné spustiť pomocou tlačidla vykonať. Po identifikovaní problémov v kóde sú výsledky zobrazené pomocou vyskakovacieho okna, ako je zobrazené nižšie.



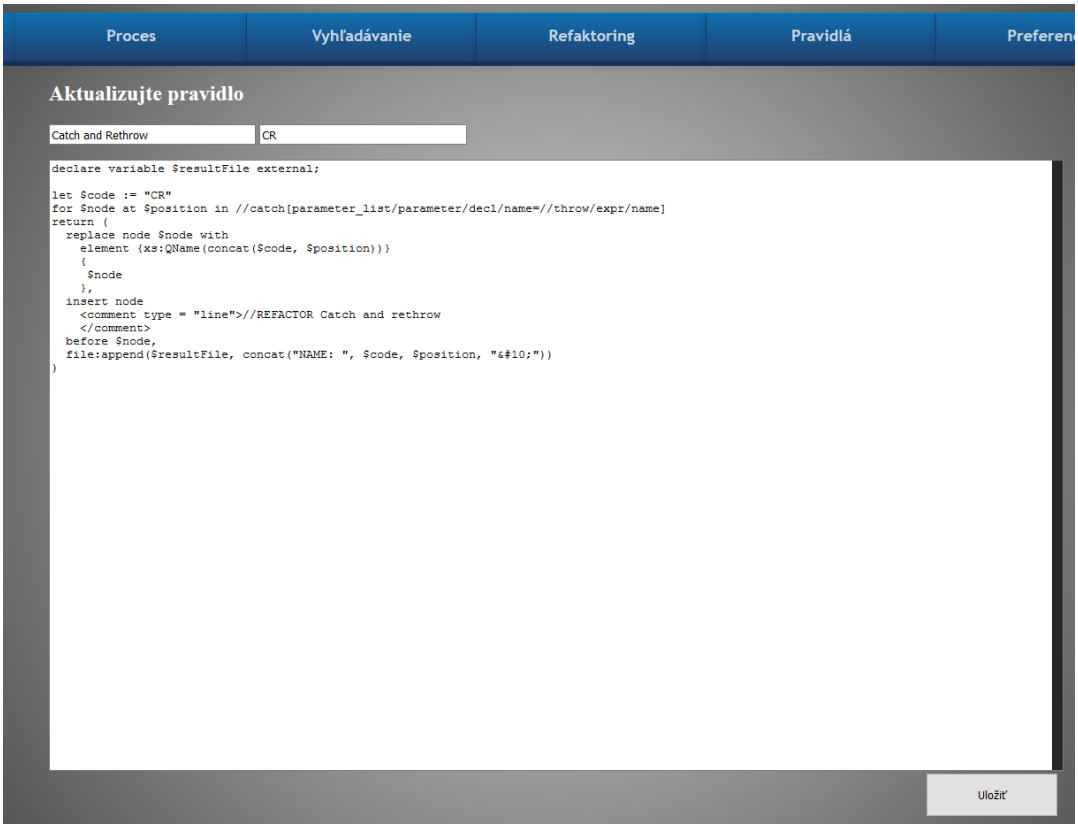
Obr. 53 Zobrazenie správy s výsledkom

2. Vyhľadávanie

Pomocou obrazovky vyhľadávanie je možné spravovať dostupné vyhľadávacie skripty. Príklad obrazovky je možné vidieť na obrázku nižšie.

Obr. 54 Zobrazenie dostupných možností vyhľadávania

Na obrazovke je uvedená tabuľka dostupných vyhľadávacích skriptov, pričom používateľ ich môže aktivovať alebo deaktivovať pre vyhľadávanie. Pri jednotlivých skriptoch je možné pomocou tlačidla *editovať* vyvolať editáciu príslušného skriptu, ako je možné vidieť na obrazovke nižšie.



Obr. 55 Editácia konkrétneho vyhľadávacieho pravidla

Pri editácii je možné meniť obsah skriptu a jeho názov, nie však jeho kód. Zmeny sú uložené pomocou tlačidla uložiť.

Tlačidlo pridať na obrazovke manažmentu vyhľadávania vyvolá podobné okno, ako pri editácií konkrétneho skriptu s tým rozdielom, že je nutné vyplniť aj kód pridávaného vyhľadávacieho pravidla. Pridávanie je následne ukončené pomocou tlačidla Uložiť.

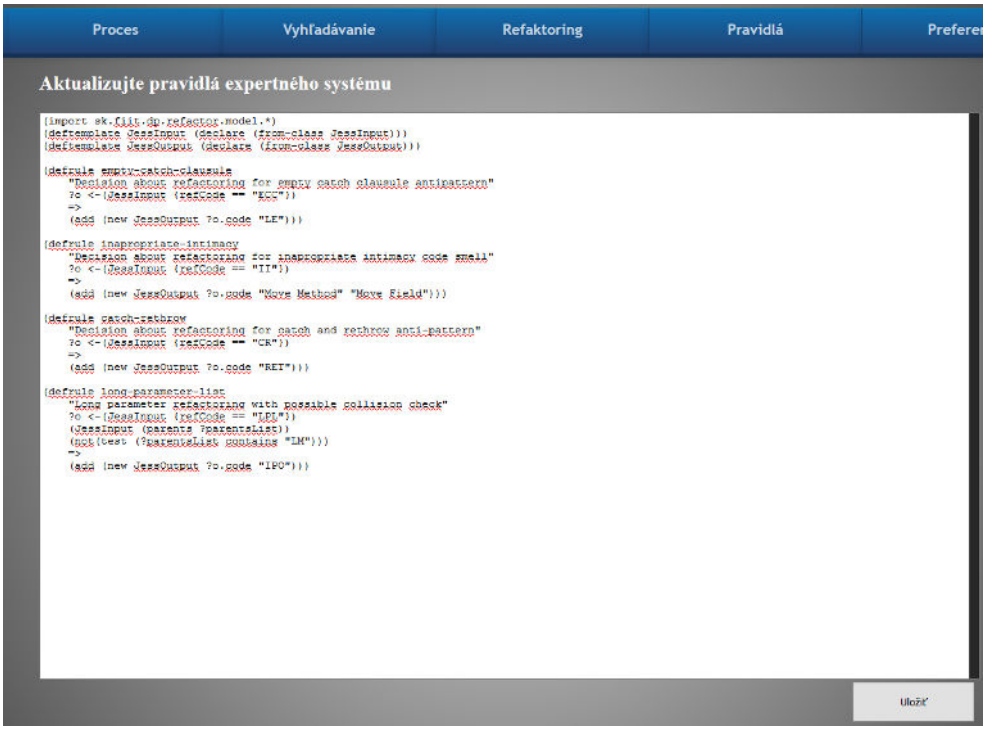
3. Refaktoring

Podobne ako obrazovka vyhľadávanie, slúži obrazovka refaktoring na manažment refaktorovacích skriptov. Dostupné skripty sú taktiež poskytnuté vo forme prehľadnej tabuľky, pričom je možné aktivovať alebo deaktivovať jednotlivé metódy. Na rozdiel od vyhľadávania sa neposkytujú spôsoby refaktorovania pre konkrétne pachy a anti-vzory, ale poskytujú sa všeobecné refaktorovacie metódy, ako možno vidieť na obrázku nižšie.

Obr. 56 Zobrazenie dostupných refaktorovacích metód

Editácia a pridávanie nových metód refaktorovania funguje rovnako, ako pri manažmente vyhľadávacích pravidiel.

4. Pravidlá

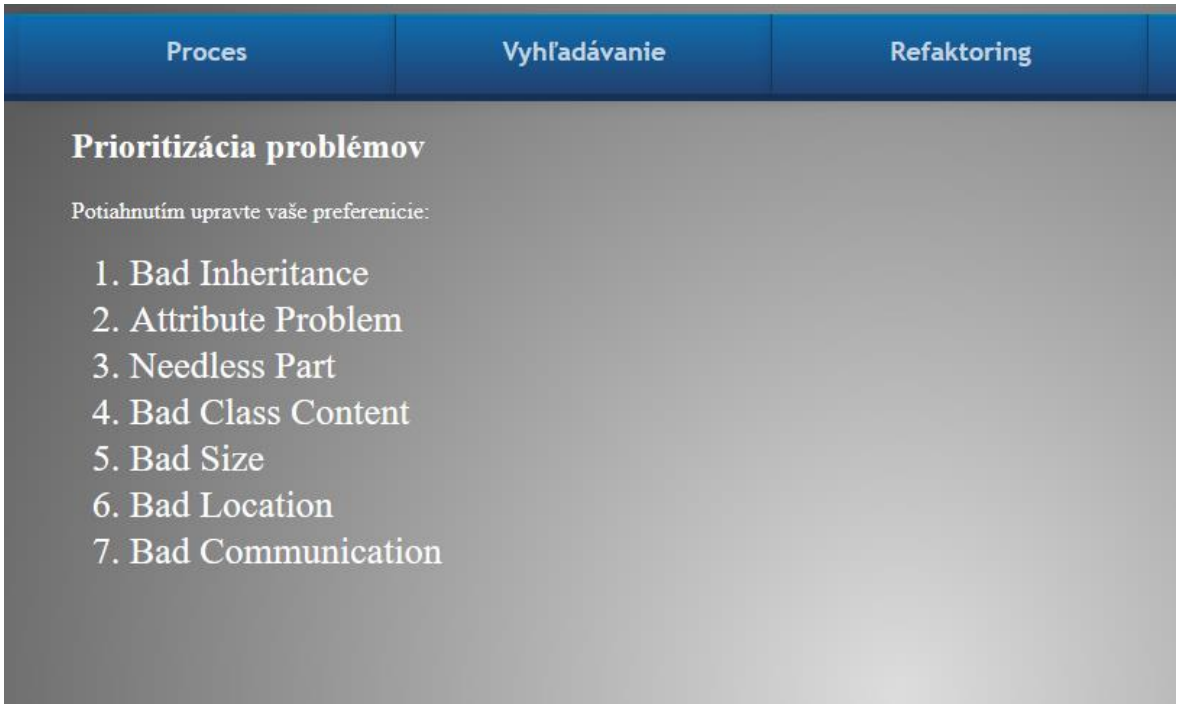


Obr. 57 Manažment pravidiel expertného systému

Pomocou systému je možné spravovať príslušné pravidlá expertného systému, ako možno vidieť na *Obr. 57*. Pravidlá sú zobrazované v jednoduchej textovej forme, pričom je umožnená ich editácia. Následné uloženie pravidiel sa vykonáva pomocou tlačidla uložiť.

5. Preferencie

Na obrazovke preferencie možno potiahnutím upraviť prioritizáciu opráv jednotlivých problémov. Daný zoznam sa poskytuje ako vstup expertného systému a je možné na základe neho vykonávať niektoré rozhodovania. Ukážku obrazovky možno vidieť nižšie



Obr. 58 Úprava prioritizácie refaktorovania

6. O programe

Obrazovka obsahuje iba stručné informácie o programe a jeho účeli. Na obrazovke sú taktiež poskytnuté informácie o autorovi programu.

Príloha C – Príspevok prijatý na konferenciu IIT.SRC 2016

Táto príloha obsahuje príspevok publikovaný na študentskej vedeckej konferencii IIT.SRC 2016, ktorá sa konala na Fakulte Informatiky a Informačných Technológií STU 28. apríla 2016. Príspevok v krátkosti prezentuje základné myšlienky tejto diplomovej práce. Príspevok bol uverejnený v príslušnej konferenčnej publikácií.

Towards Rule Based Refactoring

Lukáš MARKOVIČ*

Slovak University of Technology in Bratislava
Faculty of Informatics and Information Technologies
Ilkovičova 2, 842 16 Bratislava, Slovakia
lukass.markovic@gmail.com

Abstract. This paper presents the advanced approach to source code refactoring using XML technologies and rule based expert system. The article also describes particular refactoring problems such as dependencies between code smells, or order effectiveness of refactoring operations. Our attention is dedicated not only to well-known code smells, but also similar concept called anti-patterns. Some of these problems are used as example to explain possibilities of expert system with defined rules in refactoring process. Main part of the article describes a proposal of software system, able to perform automated refactoring using rule based decision making based on code smells dependencies analysis.

1 Introduction

Refactoring, first defined by Martin Fowler, is a process of changing a software system source code in a way, that it does not affect external behaviour, but yet improves its internal structure [2]. Besides refactoring definition, Fowler also describes the target of refactoring – code smells, as indications of a deeper problem in source code structure. In addition to code smells, there are also anti-patters, firstly mentioned by Andrew Koenig [3]. They are described as commonly occurring solutions that can cause problematic consequences [1]. There is only small difference between code smells and anti-pattern. These terms are therefore often being confused, or used without a difference. In fact, by definition, only code smell should be the real target of refactoring, because anti-patterns can also cause system problems. Removing an anti-pattern can therefore cause change in system behaviour, and that is not consistent with the refactoring definition. Despite of these terminology problems, refactoring process is nowadays used for removing both code smells and anti-patterns from the source code. Software system thus becomes easier to maintain, alternate, adapt and further develop.

However refactoring costs time, it is very necessary process during the life cycle of almost every software system. Therefore, there is a great effort in refactoring automatization. Also here in Faculty of Informatics and Information technologies was created a lot of research studies in field of refactoring automatization [7] [8] [9] [11]. This paper, which is related to previous research works proposing the rule-based refactoring [9] [11], is dedicated to the less known approach to refactoring automatization using XML technologies in combination with expert system based decision making, for obtaining the best possible result from automated refactoring process.

---

* Master degree study programme in field: Software Engineering
Supervisor: Dr. Ivan Polášek, Institute of Informatics and Software Engineering, Faculty of Informatics and Information Technologies STU in Bratislava

IIT.SRC 2016, Bratislava, April 28, 2016, pp. 1–6.

2 Automatized source code refactoring

Refactoring automatization is a very difficult process, that varies based on the selected technologies of representing the source code, searching for anti-patterns or code-smells and refactoring of these problems itself.

Code smell search and refactoring can be performed on many source code representations. Technique of code smell search and refactoring itself is subsequently unwound based on selected representation. Table 1 describes some of the possible source code representations and corresponding techniques of code smell search and refactoring.

Table 1. Possible representations and corresponding techniques of code smell search and refactoring

Source code representation	*Search technique*	*Refactoring technique*
plain text	metrics	text processing tools
AST	AST libraries, metrics	AST libraries
JSON	JavaScript, …	JavaScript, …
XML	XPath, XQuery	XSLT, XQuery, XQuery Update

As presented, automated refactoring can be performed on a clean code itself. Then, many well-known source code metrics can be used for code smell search [10], such as a *lines of code*. On the other hand, refactoring is very hard to perform on such a "*clean*" representation. One option is to use text processing tools, that are able to work with simple text, but this option is complicated and unnecessary.

Abstract syntax tree is probably the most widely used method for refactoring automatization. This method is based on a tree representation that removes all insignificant source code elements, for example parentheses or spaces. Many languages, or IDE's provides libraries, that are able to programmatically work with AST representation. Typical example is Eclipse AST library, that is used in many refactoring and source code manipulation tools under Eclipse [4].

Despite that most widely used technique for refactoring automatization is abstract syntax tree, in our work we focused on XML technologies. XML, similarly as JSON notation allows serialization of source code. Afterwards, searching and refactoring in such representation can be simply performed with technologies, that are able to manipulate with given representation [5].

3 XML based refactoring process

XML language, which is a widespread language for data serialization can be also used for a source code representation. There are several tools able to convert source code into XML representation and vice versa[1]. Many XML technologies can be then used for search of an anti-patterns and code smells [6]. There is also a lot of useful Technologies [6], such as XSLT[2] or XQuery[3], for refactoring itself.

One problem with XML representation is, that every source code file is represented as a single XML document. When refactoring is performed, there is often a needs to work with more than one file in time. For example, refactoring method "*push up method"* brings selected method into other class, that is often in its own file. This means, there is need to work with system source code as with single document, which removes the need to solve documents interconnections.

Solution to these problems are XML databases, that are kind of NoSQL documents databases. Project can be represented as a single collection of documents in this kind of databases. Each query

[1] http://www.srcml.org/
[2] https://www.w3.org/TR/xslt-30/
[3] https://www.w3.org/TR/xquery-3/

into project collection works with project de facto as one document, and this effectively solves presented problem.

Nowadays, there is only few actively supported native XML databases. These includes:

- *BaseX*
- *eXist*
- *MarkLogic*
- *Sedna*

Despite that XML databases are relative old, not a wide spread technology, they are still supported and they have actively developed an interface for many common languages, such as Java – XQJ API[4].

XML databases find only small usage nowadays, but are highly suitable for application in XML based refactoring process. They also usually offer processor of many XML technologies, such as XPath, XSLT or XQuery. XML Technologies can be used to search and mark a concrete problem and also to refactor problems in the source code, when using XML databases. XML based processes of anti-pattern or code smell search and refactoring will be described in the following subchapters.

3.1 Code smells search using XML technologies

As described in [7], XPath language is very suitable tool for XML based code smells search. Despite its simple, not hard to learn language, it is able to effectively locate and return desired nodes from XML documents. Using simple XPath expressions, it is possible to detect a lot of common, as well as little-known code smells. However, there are some reasons to select different XML associated language for code smell search.

XQuery, which used XPath for node locations, is a language for querying in XML documents. Native XML database systems are often using XQuery as main language to query into database. According to this and also needs to store information's about found problems, what XPath is not able to, the XQuery language is probably best technology for search and also refactor code smells, as described in subchapter 3.2.

3.2 Refactoring of code smells using XML technologies

Refactoring itself is the most difficult part of refactoring process. There are several possibilities to select a refactoring technology, for example XSLT [7]. Another options are XQuery and its extension XQuery Update. These are able of functional programming over XML documents, and are more sufficient in case of proposed tool.

Refactoring, as an operation of source code structure changing, requires modifying of XML documents in the database. Only XQuery itself is insufficient for this purpose, because it does not contain operations of modifying the document itself. This problem is solved by *XQuery Update* extension.

Example of XQuery script is provided in fragment code below. Very simple refactoring operation *Remove exception throw* is shown, in which *delete* operation is XQuery Update command.

```
declare variable $tag external := "CR1";

for $node in xquery:eval(fn:concat("//", $tag))
return
      delete node //throw
```

[4] http://xqj.net/

4 Rule based refactoring

Refactoring has many areas, which need to be considered during the process. One of them is the influencing of the code smells each other's.

Influencing means that by removing one code smell, other code smell can arise on a given place. Typical example is, that by removing *middle man, message chains* code smell often arises. Besides this, positive code smell influencing can also be taken into mind. This means, that by removing one code smell on given place, other code smell can be removed as well. This behaviour is very interesting, knowing that every refactoring affects the structure of the code. In general, the target of refactoring is to remove code smells in effective way. This means with as minimum as possible refactoring operations. Removing code smell, which removes also other code smell on given position is better, than firstly removing inner code smell and then removing outer code smell.

These two kind of influences between code smells are simple to take into consideration when refactoring is performed in a manual process. But during refactoring automatization, it is hard to consider these influences. One possibility is to introduce rule based expert system, able to decide about optimal or sub optimal refactoring sequence. An expert system, which contains knowledge of code smell influences, can be then included into automated refactoring system. An expert system can select refactoring operations and their order based on search analysis of source code. Nowadays, there are few tools able to create such an expert system. Jess[5] tool, which represents expert system programing language based on Java language is one possibility which was also used in proposed tool.

4.1 Dependencies between code smells

Identification of positive and negative influences between code smells is necessary in order to build such a rule based on refactoring system.

Identification of influences between each code smell and anti-patterns is very difficult, because more than one hundred source code problems are identified. In proposed tool we focused on identification of influences between the well-known 22 code smells defined by Martin Fowler. Despite of that, the number of problems is reduced into 22, there can still be identified a lot of relations between them. The reason for this is that each code smell can be removed by application not only one refactoring operation. Each operation can arise, or remove different type of code smell.

Due to these complicated relations, it is necessary to introduce the concept of code smell groups, which can be presented in form of super classes for concrete code smells. Consequently, it is possible to simplify relations between code smells using these subclasses. Identified group of code smells are presented in Table 2 below.

Table 2 Classification of code smells into groups

Group	*Code smells*
Bad Size	Large Class, Long Method, Long Parameter List
Bad Location	Feature Envy, Comments, Duplicated Code, Divergent Changes, Shotgun Surgery, Switch Statement
Bad Class Content	Data Class, Lazy Class, Feature Envy, Large Class
Bad Inheritance	Alternative Classes with Different Interfaces, Parallel Inheritance Hierarchies, Refused Bequest
Needless Part	Comments, Duplicated Code, Refused Bequest, Speculative Generality
Attribute Problem	Data Clumps, Temporary Field, Primitive Obsession
Bad Communication	Inappropriate Intimacy, Middle Man, Message Chains

[5] http://www.jessrules.com/jess/index.shtml

Given the complex character of some code smells, it is appropriate, to include them into more than one group. With such hierarchy of code smells, there is possibility to create complex views at both kinds of code smells relations. UML class diagram is a sufficient tool for creating such views, as can be seen on Figure 1, that presents positive dependencies between code smells.

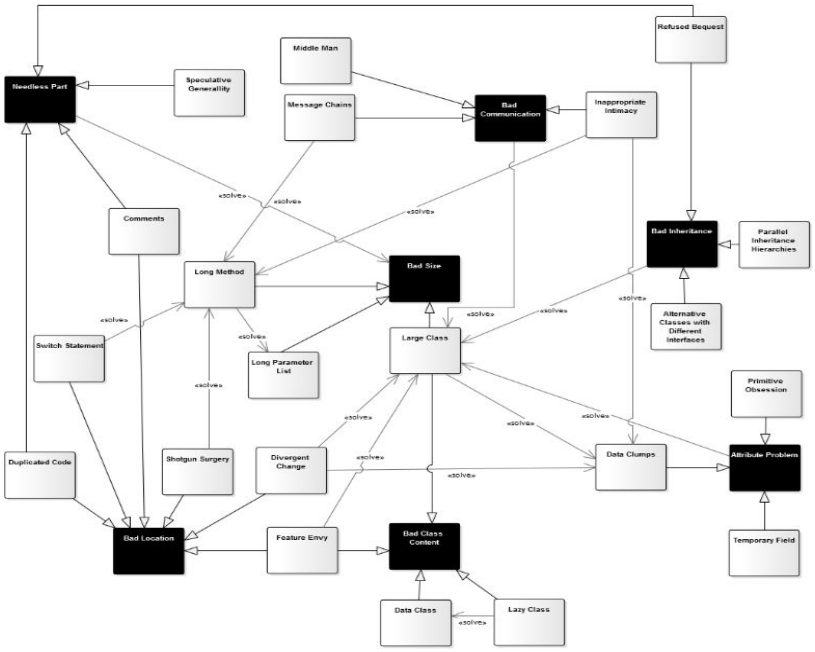


Figure 1 Positive dependencies between code smells

Solid arrow in combination with *<<solve>>* stereotype is used for positive dependencies, as can be seen in Figure 1. On the other hand, for the negative dependencies (refactoring can cause another smell) *<<cause>>* stereotype and dashed arrow can be used and similar diagram can be created.

4.2 Design of refactoring rules

It is possible to straightforward design refactoring rules based on previous analysis of code smells dependencies. Based on given rules, rule engine can decide about selected refactoring operations.

Such refactoring rule can be very simple, in form of *if – then* rule, as presented below.

```
(defrule empty-catch-clause
      "Empty catch clause refactoring"
      ?o <-(JessInput {refCode == "ECC"})
      =>(add (new JessOutput ?o.code "LE")))
```

However, it can also take into consideration other conditions, for example the size of concrete problem, or, as presented, dependency to other found problem on given place. On the code fragment below, there is a simple Jess rule, which takes such dependency between code smells into account.

```
(defrule long-parameter-list
      "Long parameter refactoring with possible collision check"
      ?o <-(JessInput {refCode == "LPL"})
      (JessInput (parents ?parentsList))
      (not(test (?parentsList contains "LM")))
      => (add (new JessOutput ?o.code "IPO")))
```

Based on the second presented rule, there is only selected refactoring operation *Introduce parameter object* for refactoring of code smell *Long parameter list*, only if code smell *Long method* was not found in place. Otherwise, *Long parameter list* refactoring will not be executed, before *Long method* refactoring. On the other hand, based on first presented rule, *Log Exception* refactoring method is selected unconditionally as solution for every *Empty Catch Clause* problem.

5 Conclusions

This paper presented new, advanced approach to refactoring. Rule based refactoring, in combination with XML technologies, offers great potential into the future. XML database, that has strong processing power is a very important part of XML based refactoring process. In future work, there is a need to examine dependencies between all known code smells more precisely. Refactoring rules and strong refactoring tool can be created based on such analysis.

Acknowledgement: This contribution is the partial result of the project Research of methods for acquisition, analysis and personalized conveying of information and knowledge, ITMS 26240220039, co-funded by the ERDF.

References

[1] Brown, H.B., Malveau, R.C., McCormick, H.W, Mowbray T.J.: *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis.* John Wiley & Sons, Inc., (1998).

[2] Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, (1999).

[3] Koenig, A.: Patterns and Antipatterns. In: *Journal of Object-Oriented Programming*, (1995), vol. 8, pp. 46-48.

[4] Kuhn, T., Thomann, O.: *Abstract Syntax Tree*. [Online; accessed November 20, 2006]. Available at: http://www.eclipse.org/articles/Article-JavaCodeManipulation_AST

[5] Mamas, E., Kontogiannis, K.: Towards Portable Source Code Representations Using XML. In: *WCRE '00 Proceedings of the Seventh Working Conference on Reverse Engineering*, IEEE Computer Society, Washington, (2000), pp. 172–182.

[6] Markovič. L.: *Refactoring support using transformations between Java a XML*. Bachelor's thesis, Slovak University of Technology in Bratislava, (2014).

[7] Markovič. L.: Refactoring Support Using XSLT Transformations. In.: *IIT.SRC 2014 Proceedings in Informatics and Information Technologies Student Research Conference*, Slovak University of Technology in Bratislava, (2014), pp. 457-462.

[8] Pipík, R., Polášek, I.: Semi-automatic refactoring to aspect-oriented platform. In: *CINTI 2013: proceedings of the 14th IEEE International Symposium on Computational Intelligence and Informatics*, Budapest, (2013), pp. 141-145.

[9] Polášek, I., Snopko, S. Kapustík, I.: Automatic identification of the anti-patterns using the rule-based approach. In: *SISY 2012: IEEE 10th Jubilee International Symposium on Intelligent Systems and Informatics*, Subotica (2012), pp. 283-286.

[10] Simon, F., Steinbrückner, F., Lewerentz, C.: Metrics based refactoring. In: *CSMR '01 Proceedings of the Fifth European Conference on Software Maintenance and Reengineering*, IEEE Computer Society, Lisbon, (2001), pp. 30–38.

[11] Štolc, M., Polášek, I.: A Visual based Framework for the Model Refactoring Techniques. In: *SAMI 2010, 8th IEEE International Symposium on Applied Machine Intelligence and Informatics*, Herľany, (2010), pp. 77-82.