Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológií

FIIT-5220-64365

Bc. Martin Konôpka

SOFTVÉROVÉ METRIKY VYCHÁDZAJÚCE Z AKTIVÍT PROGRAMÁTORA A KONTEXTU VÝVOJA SOFTVÉRU

Diplomová práca

Študijný program:	Softvérové inžinierstvo
Študijný odbor:	9.2.5. Softvérové inžinierstvo
Miesto vypracovania:	Ústav informatiky a softvérového inžinierstva, FIIT STU Bratislava
Vedúca práce:	prof. Ing. Mária Bieliková, PhD.

máj 2014

S T U
F I I T
SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
Fakulta informatiky a informačných technológií

Zadanie diplomovej práce

Meno študenta: **Bc. Martin Konôpka**

Študijný program: Softvérové inžinierstvo

Študijný odbor: Softvérové inžinierstvo

Názov práce: **Softvérové metriky vychádzajúce z aktivít programátora a kontextu vývoja softvéru**

Samostatnou výskumnou a vývojovou činnosťou v rámci predmetov Diplomový projekt I, II, III vypracujte diplomovú prácu na tému, vyjadrenú vyššie uvedeným názvom tak, aby ste dosiahli tieto ciele:

Všeobecný cieľ:

Vypracovaním diplomovej práce preukážte, ako ste si osvojili metódy a postupy riešenia relatívne rozsiahlych projektov, schopnosť samostatne a tvorivo riešiť zložité úlohy aj výskumného charakteru v súlade so súčasnými metódami a postupmi študovaného odboru využívanými v príslušnej oblasti a schopnosť samostatne, tvorivo a kriticky pristupovať k analýze možných riešení a k tvorbe modelov.

Špecifický cieľ:

Vytvorte riešenie, zodpovedúce návrhu textu zadania, ktorý je prílohou tohto zadania. Návrh bližšie opisuje tému vyjadrenú názvom. Tento opis je záväzný, má však rámcový charakter, aby vznikol dostatočný priestor pre Vašu tvorivosť.

Riaďte sa pokynmi Vášho vedúceho.

Pokiaľ v priebehu riešenia, opierajúc sa o hlbšie poznanie súčasného stavu v príslušnej oblasti alebo o priebežné výsledky Vášho riešenia alebo o iné závažné skutočnosti, dospejete spoločne s Vaším vedúcim k presvedčeniu, že niečo v texte zadania a/alebo v názve by sa malo zmeniť, navrhnite zmenu. Zmena je spravidla možná len pri dosiahnutí kontrolného bodu.

Miesto vypracovania: Ústav informatiky a softvérového inžinierstva FIIT STU v Bratislave

Vedúci práce: **prof. Ing. Mária Bieliková, PhD.**

Termíny odovzdania:

podľa harmonogramu štúdia platného pre semester, v ktorom máte príslušný predmet (Diplomový projekt I, II, III) absolvovať podľa Vášho študijného plánu

Predmety odovzdania:

V každom predmete dokument podľa pokynov na www.fiit.stuba.sk v časti: home > Informácie o > štúdiu > organizácia štúdia > diplomový projekt

V Bratislave dňa 18. 2. 2013

prof. Ing. Pavol Návrat, PhD.
riaditeľ Ústavu informatiky a softvérového inžinierstva

STU FIIT
SLOVENSKÁ TECHNICKÁ
UNIVERZITA V BRATISLAVE
FAKULTA INFORMATIKY
A INFORMAČNÝCH TECHNOLÓGIÍ

Návrh zadania diplomovej práce

Finálna verzia do diplomovej práce [1]

Študent:

Meno, priezvisko, tituly: Martin Konôpka, Bc.
Študijný program: Softvérové inžinierstvo
Kontakt: martkono@gmail.com

Výskumník:

Meno, priezvisko, tituly: Mária Bieliková, prof. Ing. PhD.

Projekt:

Názov:	Softvérové metriky vychádzajúce z aktivít programátora a kontextu vývoja softvéru
Názov v angličtine:	Software Metrics Based on Developer's Activity and Context of Software Development
Miesto vypracovania:	Ústav informatiky a softvérového inžinierstva, FIIT STU, Bratislava
Oblasť problematiky:	Metriky softvéru, modelovanie kontextu a používateľa, analýza dát

Text návrhu zadania[2]

Hodnotenie softvéru, resp. softvérových súčiastok, je dôležité ako z pohľadu manažmentu softvérového projektu, tak aj z pohľadu celkového vývoja projektu a jeho výsledku. Hodnotenie zdrojového kódu programátormi je dôležité z hľadiska vhodných (očakávaných) vlastností vytváraného softvéru. Programátori vo vývojom prostredí vykonávajú rôzne aktivity, ktoré môžu, ale aj nemusia byť zlúčiteľné s cieľom splnenia úlohy. Efektívnosť programátorov a aj výsledok, ktorý vytvárajú ovplyvňuje aj kontext, v ktorom softvér vzniká. Vlastnosti zdrojového kódu vieme určiť aj na základe aktivity viacerých programátorov, či množstva zmien v zdrojovom kóde.

Analyzujte existujúce metódy vyhodnocovania výstupov softvérového projektu, sústreďte sa na softvérové metriky vyhodnocujúce zdrojový kód. Analyzujte možnosti sledovania aktivít programátora, jeho kontextu a prostredia, v ktorom sa nachádza. Nájdite možnosti prepojenia aktivít a kontextu programátora s vlastnosťami zdrojového kódu. Navrhnite metódu pre vyhodnotenie a označkovanie zdrojového kódu na základe aktivít a kontextu programátora. Navrhnutú metódu vyhodnoťte. Diskutujte porovnanie s existujúcimi spôsobmi vyhodnocovania softvéru na základe zdrojového kódu.

---

[1] Vytlačiť obojstranne na jeden list papiera

[2] 150-200 slov (1200-1700 znakov), ktoré opisujú výskumný problém v kontexte súčasného stavu vrátane motivácie a smerov riešenia

Anotácia

Slovenská technická univerzita v Bratislave
FAKULTA INFORMATIKY A INFORMAČNÝCH TECHNOLÓGIÍ
Študijný program: Softvérové inžinierstvo

Autor: Bc. Martin Konôpka
Diplomová práca: Softvérové metriky vychádzajúce z aktivít programátora a kontextu vývoja softvéru
Vedúca práce: prof. Ing. Mária Bieliková, PhD.

máj 2014

Vývoj softvéru je komplexná činnosť, ktorú je náročné sledovať a riadiť. Nedostatočné sledovanie projektu nepriaznivo vplýva na produkovanie kvalitného softvéru, ako jedného z hlavných cieľov softvérového inžinierstva. Výskum v softvérovom inžinierstve sa preto zameriava aj na spôsoby merania softvéru a vyhodnocovania jeho vlastností. Existujúce metriky zdrojového kódu sa úspešne uplatnili aj napriek nevýhodám ich interpretácie a výpočtovej náročnosti pre vyhodnotenie. Pomocou metrík zdrojového kódu meriame výsledok práce programátorov, nezohľadňujeme však príčiny vzniku identifikovaných nedostatkov v zdrojovom kóde.

V tejto práci uvádzame metódu identifikácie skrytých závislostí v zdrojovom kóde ako metriky vychádzajúcej z aktivít programátora a kontextu vývoja softvéru. Predpokladáme, že udalosti počas vývoja softvéru a jeho údržby vplývajú na závislosti v zdrojovom kóde a jeho výsledné vlastnosti. Identifikovaním implicitných závislostí rozširujeme priestor závislostí medzi súčiastkami, čím prispievame k udržovateľnosti softvéru. Implicitnými závislosťami dokážeme nahradiť a doplniť explicitné závislosti vďaka nezávislosti ich identifikácie od syntaktickej analýzy zdrojového kódu.

Východiskom našej práce je projekt PerConIK (Personalized Conveying of Information and Knowledge) zameraný na vytváranie ľahkej sémantiky nad informačným priestorom vývoja softvérových projektov, kde vidíme uplatnenie našej práce.

Annotation

Slovak University of Technology in Bratislava
FACULTY OF INFORMATICS AND INFORMATION TECHNOLOGIES
Degree Course: Software Engineering

Author: Bc. Martin Konôpka
Master's Thesis: Software Metrics Based on Developer's Activity and Context of Software Development
Supervisor: prof. Ing. Mária Bieliková, PhD.

2014, May

Software development is an extensive process difficult to monitor and control. Insufficient monitoring of software project negatively affects production of quality software products, which is one of the main goals of software engineering. Therefore, research in software engineering aims its effort to monitoring and evaluation of software and its quality attributes. We are aware of several successful source code metrics, although with disadvantages of possible misinterpretation and high computational complexity. These metrics evaluate results of developer's work. However, they do not take into consideration the causes of the identified problems in source code.

In this thesis we propose method for identification of hidden source code dependencies as a metric based on developer's activity and context of software development. We assume that events occurred during the development process affect source code dependencies and attributes of the resulting software. By identifying implicit source code dependencies we broaden the space of known dependencies in source code and so we contribute to the software maintainability. Implicit dependencies can replace or supplement explicit dependencies because of their independence of syntactic analysis of source code.

We base our work on the PerConIK (Personalized Conveying of Information and Knowledge) project which focuses on creation of lightweight semantics over the information space of software projects development, where we see possible application of our work.

Obsah

Prílohy

Príloha A Technická dokumentácia
Príloha B Opis dát softvérových projektov pre vyhodnotenie práce
Príloha C Koncept analýzy priestoru informačných značiek pre prehliadku zdrojového kódu
Príloha D Prehľad metód dolovania v dátach
Príloha E Prehľad vlastností a objektovo-orientovaných metrík softvéru
Príloha F Príspevok na konferenciu IIT.SRC 2014
Príloha G Príspevok na konferenciu ESEM 2014
Príloha H Obsah elektronického média

Kapitola 1

Úvod

Vývoj softvéru je komplexná činnosť, ktorú je náročné sledovať a riadiť. Produkovanie kvalitného softvéru zaraďujeme medzi hlavné ciele softvérového inžinierstva (Bieliková, 2000; Sommerville, 2010). Nedostatočné sledovanie projektu vplýva na neplnenie termínov, plytvanie zdrojmi a celkový úspech projektu. Pre manažment softvérového projektu je dôležité vedieť dostatočne hodnotiť projekt pre zabezpečenie plnenia plánov, vytvárania odhadov blízkych realite, vznik kvalitnejších produktov a čo najvyššej produktivity tímu vývojárov.

Riadenie softvérového projektu závisí od možností ho sledovať, čo vyjadruje aj známy citát z prostredia softvérového inžinierstva:

„Nedokážete riadiť to, čo nedokážete merať.“ (DeMarco, 1982)

Meraním projektu získame znalosť o jeho stave, vlastnostiach a schopnosti plniť stanovené požiadavky. Neschopnosť merať projekt vedie k neefektívnosti jeho manažmentu, preto je dôležité projekt vyhodnocovať ako počas doby jeho vykonávania, tak aj spätnou analýzou po dokončení.

Výskum v oblasti softvérového inžinierstva priniesol množstvo metrík pre vyhodnotenie softvérového projektu, ktoré umožňujú jednoduchšie identifikovať problémy a merať proces vývoja (Fenton & Pfleeger, 1998; Bieliková, 2000; Sommerville, 2010). Najčastejším prístupom sú metriky obsahu zdrojového kódu, ktorých nevýhodou je ich výpočtová náročnosť a problém interpretácie ich definícií. Odporúčané hranice výsledkov sa líšia pre programovacie paradigmy, či dokonca aj jazyky. Aj napriek týmto problémom dokážeme merať obsah zdrojového kódu pre vyhodnotenie vlastností softvéru, akými je jeho kvalita, udržovateľnosť, znovupoužiteľnosť, efektívnosť alebo prenositeľnosť.

Vychádzajúc z podrobnej analýzy súčasného stavu v dobre rozpracovanej oblasti metrík softvérového projektu založených na obsahu zdrojového kódu, ktorých história siaha do 70. rokov minulého storočia, sa v našej práci zameriavame na prínos metriky založenej na analýze aktivít programátora a jeho kontextu. Metriky vychádzajúce z analýzy aktivít programátora vznikajú až v poslednej dobe, ako z dôvodu zvyšovania potreby presnejšieho vyhodnocovania softvérového projektu, tak aj z dôvodu netriviálnosti úlohy sledovania programátora. Metriky obsahu zdrojového kódu vyhodnocujú výsledky práce programátorov a nezohľadňujú príčiny vzniku chýb, pachov v kóde či porušenia konzistencie a konvencií – čím sú práve aktivity programátora a kontext vývoja softvéru. Vedecké práce v tejto oblasti sledovali vplyv aktivít na udržovateľnosť softvéru, vznik chýb na základe času práce alebo aj znalosť programátora (Gousios, et al., 2008; Eyolfson, et al., 2011; Kuric & Bieliková, 2013; Zeleník, 2013).

K udržovateľnosti softvéru prispieva znalosť o závislostiach jednotlivých softvérových súčiastok zdrojového kódu v procesoch jeho vývoja a údržby. Identifikácia závislostí medzi softvérovými súčiastkami je metrikou určujúcou orientované prepojenie medzi uvažovanými súčiastkami, t.j. vlastnosťou softvérových súčiastok s číselným ohodnotením vyjadrujúcim váhu závislosti. Súčasná identifikácia závislostí syntaktickou analýzou zdrojového kódu umožňuje identifikovať explicitne určené závislosti programátorom počas vývoja, zároveň je však obmedzená možnosťami syntaktickej analýzy konkrétneho programovacieho jazyka.

Motiváciou našej práce je možnosť využitia aktivít programátora a jeho kontextu ako ďalší zdroj pre identifikovanie závislostí v zdrojovom kóde, ktoré v tomto prípade označujeme ako *implicitné*. Metódou identifikácie implicitných závislostí zväčšujeme priestor všetkých identifikovaných závislostí medzi softvérovými súčiastkami, čím prispievame k jeho udržovateľnosti. Implicitné závislosti odrážajú prepojenia v zdrojovom kóde počas jeho vývoja, údržby, alebo aj počas jeho vykonávania. Výhodou našej metódy je jej nezávislosť od syntaktickej analýzy zdrojového kódu, čo nám umožňuje identifikovať aj závislosti v tom zdrojovom kóde, ktorý nedokážeme syntakticky analyzovať, napr. pri dynamicky-typovaných programovacích jazykoch. Prínos implicitných závislostí k aktuálnemu stavu poznania v meraní a vyhodnocovaní softvéru identifikujeme ako:

- identifikovanie vzťahov medzi súčiastkami zdrojového kódu vo forme implicitných závislostí, t.j. takých, ktoré súčasné metódy neidentifikujú;
- definovanie scenárov pre využitie implicitných závislostí v procese vývoja a údržby softvérového produktu, najmä pri jeho testovaní;
- možnosť vyhodnotiť závislosti súčiastok aj v prípade nedostupnosti syntaktickej analýzy zdrojového kódu.

V práci vychádzame a ďalej staviame na infraštruktúre výskumného projektu PerConIK[1], v ktorom autori uplatňujú tzv. „webifikáciu vývoja softvérových projektov“ (Bieliková, et al., 2013), t.j. aplikovanie metód modelovania, vyhľadávania a odporúčania z domény Webu na dáta domény softvérovej spoločnosti. Projekt predstavuje vytváranie ľahkej sémantiky nad informačným priestorom vývoja softvérových projektov, kde vidíme uplatnenie našej metódy merania softvéru. Pomocou implicitných závislostí dopĺňame zdroje použité v existujúcich metódach vyhľadávania, navigácie programátora alebo vyhodnocovania softvéru.

V kapitole 2 analyzujeme existujúce prístupy vyhodnocovania softvérových projektov a merania pomocou metrík obsahu zdrojového kódu. Kapitolu 3 sme venovali analýze vplyvu aktivít programátora a kontextu na vlastnosti softvérového projektu. V kapitole 4 opisujeme projekt PerConIK, v ktorom vidíme uplatnenie našej práce, a diskutujeme o možnostiach príspevku k údržbe softvéru. V kapitole 5 predstavujeme pôvodnú metódu identifikácie implicitných závislostí v zdrojovom kóde a jej experimentálne overenie opisujeme v kapitole 6. Kapitola 7 je venovaná zhodnoteniu tejto práce a identifikovaniu jej prípadného pokračovania.

Súčasťou práce sú aj prílohy obsahujúce vybrané časti technickej dokumentácie riešenia navrhnutej metódy. Na priloženom elektronickom médiu sa nachádza kompletná technická dokumentácia vygenerovaná zo zdrojového kódu riešenia. V prílohách ďalej uvádzame opis použitých dát softvérových projektov pre overenie metódy prostredníctvom infraštruktúry projektu PerConIK (dokumentácia rozhraní služieb je uvedená na elektronickom médiu). Počas riešenia práce sme okrem identifikácie implicitných závislostí navrhli aj metódu analýzy priestoru

[1] PerConIK – Personalized Conveying of Information and Knowledge. `http://perconik.fiit.stuba.sk/`

informačných značiek. Metódu sme nerealizovali, jej koncept však uvádzame v prílohe C. Prílohy ďalej dopĺňajú prácu o prehľady metód dolovania v dátach a vyhodnocovania vlastností softvéru.

Výsledky práce sme úspešne prezentovali na študentskej vedeckej konferencii IIT.SRC 2014, kde náš príspevok s názvom *Identifying Hidden Source Code Dependencies* bol ocenený cenou dekana. V prílohách uvádzame ocenený príspevok, a aj návrh príspevku na konferenciu ESEM 2014.

Kapitola 2

Vyhodnocovanie softvéru

Softvérový projekt sledujeme a vyhodnocujeme už počas jeho vykonávania aby sme zabezpečili dosiahnutie jeho cieľov v požadovanej kvalite. Pod softvérovým projektom rozumieme dočasné úsilie vykonávané s cieľom priniesť unikátny produkt, službu alebo iný výsledok prác (Project Management Institute, 2004). Na softvérový projekt sa môžeme pozerať z viacerých pohľadov a podľa nich aj rozlišovať aké vlastnosti a akými metrikami ich dokážeme vyhodnocovať.

2.1. Pohľady na softvérový projekt

Softvérový projekt spravidla zahŕňa rozsiahlu činnosť rozdeliteľnú na viaceré časti s cieľom ich ľahšej analýzy a sledovania. Na softvérový projekt sa môžeme pozerať z rôznych uhlov pohľadu.

2.1.1. Zainteresované strany

Na softvérovom projekte sa zúčastňuje rozličné množstvo účastníkov závisiac od veľkosti projektu, typu projektu a jeho cieľov. Účastníkov projektu rozdeľujeme do troch strán, ktoré sú do projektu zainteresované, no navzájom sa odlišujú podľa svojich úloh a očakávaní:

- *manažér* – sleduje a riadi projekt i jeho smerovanie,
- *vývojár, programátor* – aktívne pracuje na projekte,
- *používateľ* – zákazník alebo aj zadávateľ, ktorý má určité očakávania od produktu.

Napriek rozdielom v prínose jednotlivých účastníkov strán do projektu, všetky tri strany potrebujú mať možnosť sledovať vývoj projektu a kontrolovať, či projekt spĺňa ich očakávania. Ako príklad môžeme uviesť situáciu, keď zákazník, ktorý zadal projekt môže mať vedomosti o doméne projektu, ktorý požaduje. Na druhej strane ale nemá skúsenosti s vývojom softvéru, s programovacími jazykmi, či s hardvérom, aké má vývojár. Aj napriek tomuto rozdielu potrebujú všetky strany sledovať vývoj projektu, či spĺňa zadané požiadavky počas vývoja, či sa vyvíja správnym smerom.

Každá zo zainteresovaných strán sleduje projekt z iného pohľadu. Zákazník sleduje výstupy projektu či spĺňajú jeho požiadavky, a často aj využitie zdrojov potrebné pre vývoj projektu. Vývojár sleduje softvér z vnútorného pohľadu, skúma vlastnosti zdrojového kódu, akou je napr. jeho zložitosť, alebo aj úlohy, ktoré má splniť. Manažér sleduje projekt z vonkajšieho hľadiska, kontroluje plnenie úloh vývojárov, sleduje smerovanie projektu a využitie zdrojov.

2.1.2. Úrovne projektu

Softvérový projekt zahŕňa komplexnú činnosť, ktorú môžeme rozumieť na viacerých úrovniach. Podľa (Fenton & Pfleeger, 1998) na softvérovom projekte vieme identifikovať tri úzko prepojené úrovne záujmu, ktoré potrebujeme merať a sledovať:

- *procesy* – činnosti spojené s prácou na softvérovom projekte,
- *produkty* – výsledky vykonaných činností,
- *zdroje* – prostriedky potrebné pre vykonanie činností.

Medzi procesy zaraďujeme už činnosti zo základného rozdelenia etáp vývoja softvérového projektu, t.j. zber požiadaviek od zákazníka, analýza, návrh, implementácia, testovanie a údržba (Sommerville, 2010). Všeobecnejšie však môžeme procesy charakterizovať ako vykonávanie činností so stanoveným začiatkom, koncom, určenými prostriedkami (zodpovedajú zdrojom projektu) a očakávaným výstupom, t.j. produkt (Project Management Institute, 2004). Priebežné alebo konečné produkty sú výsledkom činností a plnenie činností spotrebúva poskytnuté zdroje, napr. ľudské alebo finančné zdroje.

Najčastejšie sa ako produkt softvérového projektu (softvérový produkt) uvažuje zdrojový kód alebo výsledný softvér (program). Zdrojový kód je hlavným výstupom programátorov a softvér je výsledkom samotného softvérového projektu. Medzi produkty patria však aj ostatné artefakty jednotlivých etáp projektu (najmä analýza a návrh), dočasné a priebežné výstupy, ktoré vznikajú počas procesov, napr. dokumenty, prototypy riešenia i inštalačné balíky a testy.

2.1.3. Klasifikácia zdrojového kódu softvérového produktu

Produkt softvérového projektu pozostáva zo zdrojového kódu, ktorého časti môžeme rozlišovať rôznymi pojmami. Všeobecným pojmom je *softvérová súčiastka,* niekedy označovaná aj všeobecnejšie ako *entita*:

> *Softvérová súčiastka je ucelená časť zdrojového kódu softvérového projektu na zvolenej úrovni granularity. Softvérová súčiastka môže byť deliteľná na ďalšie súčiastky nižšej úrovne granularity, kedy ju považujeme za zloženú súčiastku, vytvárajúc hierarchiu softvérových súčiastok.*

Konkretizácia softvérovej súčiastky závisí od zvoleného programovacieho jazyka. Ako príklad môžeme uviesť konkretizáciu pre objektovo-orientované programovacie jazyky *C#* a *Java*. Uvažujme hierarchiu od najvyššej úrovne granularity po najnižšiu, potom jednotlivé softvérové súčiastky označujeme takto:

- zložené softvérové súčiastky:
  - komponent – zoskupenie balíkov, knižnica, aplikácia,
  - balík, menný priestor – zoskupenie elementárnych softvérových súčiastok;
- elementárna softvérová súčiastka:
  - jednotka zdrojového kódu – trieda, rozhranie, číselník,
  - prvok jednotky zdrojového kódu – metóda, atribút, premenná.

2.2. Vlastnosti softvérového projektu

Pre každú z úrovní softvérového projektu môžeme sledovať a vyhodnocovať jej vlastnosti:

- *vonkajšie vlastnosti* – správanie v okolitom prostredí,
- *vnútorné vlastnosti* – správanie v rámci úrovne, priamo neovplyvnené prostredím.

Vnútorné vlastnosti sa jednoduchšie vyhodnocujú len v rámci sledovanej úrovne. Výsledky meraní vnútorných vlastností nezávisia od vplyvov okolitého prostredia, no vo výsledku vplývajú na vonkajšie vlastnosti, ktoré je častokrát možné vyhodnotiť až spätne po dokončení projektu.

Na procesnej úrovni softvérového projektu vyhodnocujeme jeho jednotlivé činnosti, najčastejšie procesy vývoja a údržby produktov (Sommerville, 2010). Spomedzi vonkajších vlastností sledujeme napr. kvalitu, cenu, kontrolovateľnosť či stabilitu projektu, a z vnútorných vlastností napr. cenu, čas, potrebné úsilie na vykonanie projektu, či aj množstvo zmien a chýb, ktoré sa vyskytli počas jeho vykonávania. Skvalitnením procesov ovplyvníme kvalitu výsledných produktov alebo kvalitnejšie využitie zdrojov. To podnietilo vznik štandardizovaných modelov pre kvalitu procesov, napr. ISO 9001[2] a CMMI[3].

Vyhodnotenie úspešnosti softvérového projektu z veľkej miery ovplyvňuje kvalita výsledného produktu a splnenie požiadaviek zákazníka. Softvérový produkt je výsledkom práce programátorov a jeho hlavnou súčasťou je zdrojový kód produktu. Preto kvalita produktu z dlhodobého pohľadu zákazníka závisí vo veľkej miere od jeho udržovateľnosti, t.j. úsilia potrebného pre:

- doplnenie novej funkcionality,
- upravenie existujúcej funkcionality,
- opravenie chýb v produkte.

Udržovateľnosť softvérového produktu vyplýva najmä z organizácie zdrojového kódu a jeho architektúry, čo môžeme vyhodnotiť meraním jeho vnútorných vlastností ako modulárnosť, rozšíriteľnosť alebo chybovosť. Na udržovateľnosť však vplývajú aj prostriedky a nástroje používané programátormi v procese údržby, ktoré umožňujú alebo zjednodušujú údržbu.

Prehľad vonkajších a vnútorných vlastností uvádzame v prílohe E.1, pretože ovplyvňujú vlastnosti zvyšných úrovní, a zároveň sú aj vstupom pre predpoveď vlastností zvyšných úrovní softvérového projektu, napr. predpoveď potrebného úsilia na základe odhadu veľkosti produktu.

2.3. Meranie softvérového produktu

Vlastnosti softvérového produktu vyhodnocujeme pomocou softvérových metrík, najčastejšie nad obsahom zdrojového kódu, sú definované ako (Fenton & Pfleeger, 1998; Lincke, et al., 2008):

> *Priame alebo nepriame meranie vlastnosti entity softvérového projektu. Výsledkom metriky je číselné vyjadrenie sledovanej vlastnosti.*

Priame meranie, na rozdiel od nepriameho, vyhodnocuje vlastnosť entity bez zahrnutia inej jej vlastnosti alebo ďalšej entity (Fenton & Pfleeger, 1998). Výsledky mnohých metrík, aj napriek snahe o jednoznačné definície, však závisia od ich implementácie (Lincke, et al., 2008).

Metriky softvérového produktu môžeme rozdeliť podľa pohľadov vnímania softvéru a jeho zdrojového kódu (Fenton & Pfleeger, 1998; Chidamber & Kemerer, 1994):

[2] ISO 9001:2008 Quality management systems – Requirements.

[3] Capability Maturity Model Integration, CMMI. `http://cmmiinstitute.com/`

- *pohľad na softvér ako textový obsah* – informácie v obsahu zdrojového kódu:
  - dĺžka zdrojového kódu a index udržovateľnosti
- *pohľad na softvér ako graf* – štruktúra zdrojového kódu:
  - cyklomatická zložitosť, závislosti súčiastok zdrojového kódu
- *pohľad na softvér ako ontológie* – vzťahy medzi súčiastkami zdrojového kódu, objektami:
  - objektovo-orientované metriky

2.3.1. Dĺžka zdrojového kódu

Základnou metrikou softvérového produktu je určenie jeho veľkosti, najčastejšie počtom riadkov zdrojového kódu. Výsledkom metriky je jednoznačné číslo, ktoré je však nutné špecifikovať, pretože môžeme uvažovať aj komentované riadky, ignorovať prázdne riadky, nemerať generované riadky alebo merať iba vykonateľné riadky. To podnietilo vznik viacerých odnoží tejto metriky, a tak jej výsledok závisí od použitého nástroja (Lincke, et al., 2008).

Výsledok merania veľkosti zdrojového kódu môžeme použiť pri určovaní jeho udržovateľnosti a zrozumiteľnosti. Ako príklad môžeme uviesť odporúčania použitia metriky, kedy metóda s viac ako 20 riadkami je ťažká na udržovanie (Fenton & Pfleeger, 1998). Riešením nepriaznivých výsledkov je zmena štruktúry kódu či odčlenenie funkcionality.

Iný prístup pre určenie dĺžky zdrojového kódu predstavila Maurice Halstead (Halstead, 1999), ktorej metrika umožňuje aj jej predpoveď. Metrika nepriamo určuje veľkosť na základe priamych meraní počtu operátorov a operandov vyskytujúcich sa v zdrojovom kóde.

2.3.2. Cyklomatické číslo

Vyhodnotenie zložitosti softvéru navrhol McCabe (McCabe, 1976) ako výpočet cyklomatického čísla toku riadenia programu. Tok riadenia vyjadríme grafom, ktorého vrcholy sú jednotlivé riadiace elementy programu (priradenie, vetvenie, cyklus) a jeho prepojenia vyjadrujú možnosť presunúť riadenie medzi týmito elementami (Fenton & Pfleeger, 1998; McCabe, 1976). Cyklomatickú zložitosť toku riadenia v grafe zdrojového kódu F s n vrcholmi a e hranami potom vyjadríme vzťahom:

$$v(F) = e - n + 2$$

Cyklomatické číslo vyjadruje počet nezávislých ciest v zdrojovom kóde, čo vieme okrem odhadu zložitosti využiť aj pri určovaní potrebných testov pre testovanie kódu. Výsledok však nehovorí o celkovej zložitosti programu, preto McCabe navrhol, že vyhodnotený modul s hodnotou cyklomatického čísla väčšou ako 10 má byť označený ako problematický.

Cyklomatické číslo vyhodnocuje zložitosť organizácie zdrojového kódu, ktorá vplýva napr. aj na udržovateľnosť, nie však algoritmickú zložitosť riešenia zadaného problému, ktorú vyhodnocujeme napr. meraním časovej alebo pamäťovej náročnosti výpočtu riešenia.

2.3.3. Index udržovateľnosti

Udržovateľnosť zdrojového kódu môžeme relatívne vyjadriť kombináciou Halsteadovej metriky, cyklomatickej zložitosti a počtu riadkov ako index udržovateľnosti (Oman & Hagemesiter, 1994):

$$MI = 171 - 5.2\ ln(aveV) - 0.23\ aveG - 16.2\ ln(aveLOC) + 50\ sin\left(\sqrt{2.4perCM}\right)$$

Vo vzťahu $aveV$ označuje priemernú veľkosť softvérovej súčiastky podľa Halsteadovej metriky, $aveG$ priemernú cyklomatickú zložitosť súčiastky, $aveLOC$ priemerný počet riadkov zdrojového kódu a $perCM$ priemerný počet komentovaných riadkov v súčiastke. Interpretácie výsledkov metriky rozdeľujú jej hodnoty na úrovne udržovateľnosti: nízka, stredná a vysoká. Tabuľka 2-1 uvádza odporúčané intervaly indexu podľa výskumu spoločností Microsoft a Hewlett-Packard (Microsoft, 2012; Coleman, et al., 1995).

Tabuľka 2-1 Odporúčané intervaly metriky index udržovateľnosti *MI* podľa výskumu spoločností Microsoft a Hewlett-Packard.

Miera udržovateľnosť	**Intervaly indexu udržovateľnosti**	
	Microsoft	**Hewlett-Packard**
Vysoká	$20 \leq MI \leq 100$	$85 < \mathrm{MI} \leq 100$
Stredná	$10 \leq \mathrm{MI} \leq 19$	$65 \leq \mathrm{MI} \leq 85$
Nízka	$0 \leq \mathrm{MI} \leq 9$	$0 \leq \mathrm{MI} < 65$

2.3.4. Objektovo-orientované metriky

Príchod objektovo-orientovanej paradigmy programovania podnietil vznik špecifických metrík, keďže vyjadrenie zložitosti na základe počtu riadkov alebo riadiaceho toku prestalo byť dostačujúce. Autori v (Chidamber & Kemerer, 1994) argumentujú, že existujúce metriky nebrali do úvahy pojmy objektovo-orientovaného prístupu ako triedy, dedenie, zapuzdrenie alebo posielanie správ. Objektovo-orientovaná paradigma podľa (Chidamber & Kemerer, 1994) predstavuje:

- *objekty* – indivíduá sveta a ich vlastnosti,
- *triedy* – množiny objektov majúce spoločné vlastnosti,
- *metódy tried* – operácie nad objektmi triedy.

To podnietilo vznik viacerých metrík (Morris, 1989; Lieberherr, et al., 1988), podobne aj týchto šesť od Chidambera a Kemerera, ktorým detailnejšiemu opisu je venovaná príloha E.2. (Chidamber & Kemerer, 1994):

- váhované metódy triedy,
- hĺbka stromu dedenia,
- počet potomkov triedy,
- zviazanosť tried objektov,
- odpoveď pre triedu,
- súdržnosť metód.

2.3.5. Graf závislostí

Zdrojový kód softvérového produktu pozostáva z jednotlivých súčiastok, ktoré sú medzi sebou prepojené, t.j. závislé na sebe, pre zabezpečenie požadovanej funkcionality. Pomocou syntaktickej analýzy obsahu zdrojového kódu dokážeme identifikovať nasledujúce typy prepojení medzi súčiastkami:

- volanie – napr. volanie metódy objektu,
- referencia – napr. uchovanie referencie na objekt,
- dedenie alebo implementácia – napr. dedenie triedy inou triedou.

Jednotlivé prepojenia súčiastok môžeme agregovať do spoločných závislostí medzi súčiastkami zvolenej úrovne granularity (napr. závislosti metód, tried, balíkov). Potom uvažovaním súčiastok zdrojového kódu ako vrcholov a závislostí medzi nimi ako hrán zostrojujeme graf závislostí. Hrany v grafe závislostí sú orientované podľa smeru agregovaných prepojení v zdrojovom kóde a ohodnotené počtom týchto prepojení. Identifikovanie závislostí v zdrojovom kóde je tak metrikou určujúcou číselné vyjadrenie váhy závislosti medzi dvomi zvolenými softvérovými súčiastkami.

Graf závislostí nachádza uplatnenie pri navigácii a študovaní zdrojového kódu programátorom, keďže prehľadne zobrazuje prepojenia v zdrojovom kóde. Identifikované závislosti môžeme aj algoritmicky vyhodnocovať, napr. pre hľadanie pachov a chýb v návrhu (Counsell, et al., 2006; Zimmermann & Nagappan, 2008).

Keďže graf závislostí zachytáva prepojenie tried, balíkov i komponentov, môžeme hľadať jeho podobnosť s obdobnými UML diagramami štruktúry. Ich podobnou vlastnosťou je pochopiteľne znázornenie závislostí na danej úrovni (dedenie tried, volanie a odkazovanie). Na rozdiel od príslušných UML diagramov však graf závislostí vizualizuje aj mieru prepojenia (silu závislostí) a kombinuje úrovne granularity (až po najelementárnejšie jednotky súčiastok).

Nedostatkom identifikácie závislosti v zdrojovom kóde je však jej závislosť na syntaktickej analýze zdrojového kódu, ktorou nedokážeme identifikovať:

- závislosti počas vykonávania programu – napr. vyplývajúce z tranzitívností závislostí,
- závislosti súčiastok počas ich vývoja a údržby,
- závislosti v dynamicky-typovaných programovacích jazykoch,
- závislosti súčiastok s konfiguračnými súbormi,
- závislosti súčiastok s neanalyzovanými súbormi zdrojového kódu – napr. dáta, definície používateľských rozhraní, webové stránky, atď.

Znalosť o závislostiach v zdrojovom kóde vysoko prispieva k jeho udržovateľnosti, no na základe týchto nedostatkov je graf závislostí iba čiastočnou pomôckou pri vývoji a údržbe.

2.3.6. Vyhodnotenie vlastností softvéru pomocou metrík

Uvedené softvérové metriky sú aplikovateľné na rôznej úrovni granularity softvérových súčiastok i ich podmnožine z celého zdrojového kódu softvéru, čím môžeme odvodiť vlastnosti softvérového produktu ako celku, alebo aj jeho zvolených častí.

Autori objektovo-orientovaných metrík (Chidamber & Kemerer, 1994) odporúčajú priebežné sledovanie ich hodnôt počas vývoja softvérového produktu. Metriky váhované metódy triedy, počet potomkov triedy a hĺbka stromu dedenia potvrdia dobrý návrh architektúry, pokiaľ sa ich hodnoty nemenia, t.j. hierarchia tried je stabilná. Počas vývoja sa však môže tvorením nových tried meniť prepojenie a komunikácia existujúcich tried, čo sa odzrkadlí na hodnotách zviazanosti tried objektov a odpovede pre triedu.

Objektovo-orientované metriky možno použiť pre predpoveď udržovateľnosti softvérového produktu (Dagpinar & Jahnke, 2003; Li & Henry, 1993; Dubey & Rana, 2011; Riaz, et al., 2009; Sjøberg, et al., 2012). V (Dagpinar & Jahnke, 2003) autori identifikovali, že na udržovateľnosť produktu najviac vplýva jeho veľkosť a zviazanosť tried. V (Dubey & Rana, 2011) autori odvádzajú vzťah pre udržovateľnosť M_t ako súčin prevrátených hodnôt metrík v súčine s konštantou K, ktorú je nutné určiť pre sledovaný projekt (autori neuvádzajú návrhy na jej zvolenie):

$$M_t = K \frac{1}{LCOM\ CBO\ WMC\ DIT\ NOC\ RFC}$$

Metriky Chidambera a Kemerera nie sú použiteľné len pri meraní zdrojového kódu, ale aj pri fáze návrhu. Autori v (Marinescu, 2001) hľadali chyby na úrovni návrhu softvéru, konkrétne dvoch pachov v kóde (Fowler, 1999) – údajová trieda (angl. data-class) a božská trieda (angl. god-class). Pre hľadanie každej z tried použili kombináciu troch metrík Chidambera a Kemerera a navrhli spôsob ako ich odstrániť. Keďže prístup používa údaje návrhu (a nie priamo zdrojový kód), tak je podľa autorov nezávislý na implementačnom jazyku.

Objektovo-orientované metriky vychádzajú z viacerých existujúcich metrík, medzi ktorými sú aj závislosti v zdrojovom kóde. Vplyv organizácie zdrojového kódu a správnej dekompozície zdrojového kódu na vlastnosti softvérového produktu môžeme vyhodnotiť aj pomocou grafu závislostí (Microsoft, 2013). Uplatnenie grafu závislostí pre vyhodnotenie vonkajších vlastností môžeme uviesť na nasledujúcich príkladoch:

- *testovateľnosť* – možnosť zameniť závislé softvérové súčiastky za ich falošné implementácie pre otestovanie správnosti sledovanej súčiastky,
- *prenositeľnosť* – odčlenenie platformovo-špecifickej funkcionality do samostatných súčiastok, ktoré je možné vymieňať podľa prostredia,
- *udržovateľnosť* – množstvo potrebných zmien v zdrojovom kóde po úprave súčiastky,
- *znovupoužiteľnosť* – minimalizovanie závislostí vybranej súčiastky a jej odčlenenie pre jej znovupoužitie na inom mieste.

Pomocou grafu závislostí dokážeme vyhodnotiť vnútorné vlastnosti softvérového produktu, napr. modulárnosť, zviazanosť či ich súdržnosť (Dietrich, et al., 2012), a tak vyhodnotiť uvedené vonkajšie vlastnosti.

2.4. Existujúce nástroje pre softvérové metriky

Manažéri i vývojári softvérového projektu používajú rozličné nástroje pre analýzu a monitorovanie jeho zdrojového kódu. Buď ide o samostatné softvérové riešenia alebo rozšírenia vývojových prostredí. Nástroje sa odlišujú aj množstvom podporovaných metrík a programovacích jazykov.

Základným prístupom je počítanie riadkov súborov zdrojového kódu, čo môžeme vykonať aj jednoduchým skriptovacím súborom. Hromadné vyhodnotenie metrík poskytujú aj nástroje ako napr. *Microsoft Line of Code Counter*[4] podporujúci aj pripojenie na systém správy zdrojových súborov, alebo *SLOCCount*[5] podporujúci 27 programovacích jazykov. Vyhodnotenie ďalších metrík zdrojového kódu poskytuje nástroj *Software Metrics* od spoločnosti *Semantics Designs*[6]. Medzi komplexné samostatné nástroje zaraďujeme aj nástroj *PMD*[7] identifikujúci pachy v zdrojovom kóde v jazyku Java.

V nasledujúcich častiach bližšie analyzujeme vyhodnotenie metrík zdrojového kódu v samotnom vývojovom prostredí *Microsoft Visual Studio* a ďalšie použitím jeho rozšírenia *NDepend* z dôvodu zamerania našej práce na tieto technológie.

[4] Microsoft Line of Code Counter. `http://archive.msdn.microsoft.com/LOCCounter`

[5] SLOCCount. `http://www.dwheeler.com/sloccount/`

[6] Semantics Designs: Software Metrics Tools. `http://www.semdesigns.com/Products/Metrics/index.html`

[7] PMD. `http://pmd.sourceforge.net/`

2.4.1. Microsoft Visual Studio

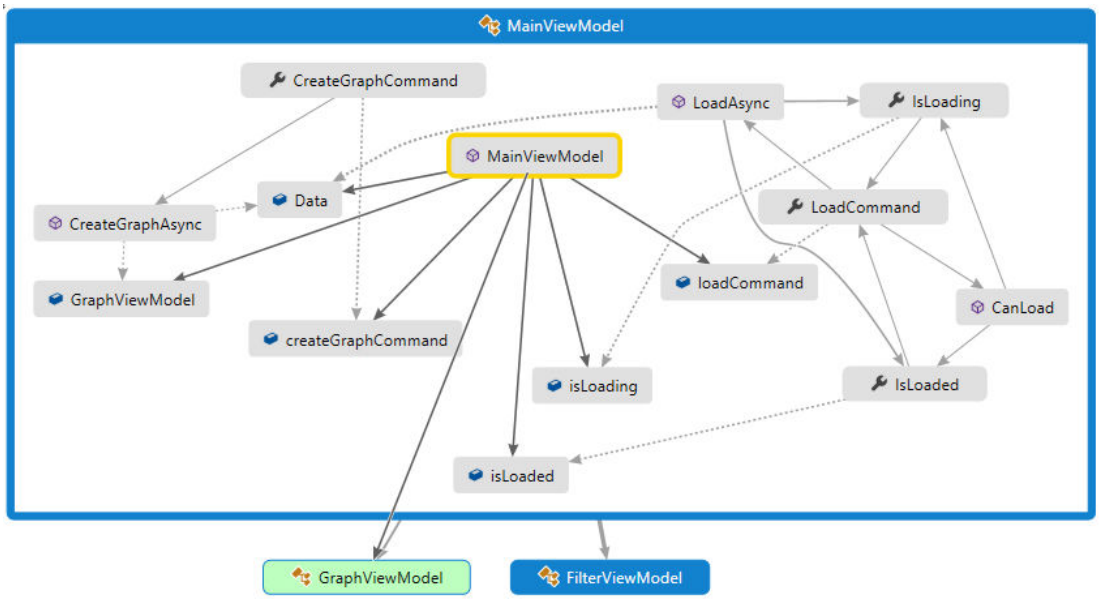
Vývojové prostredie od spoločnosti Microsoft umožňuje vývoj softvéru vo viacerých programovacích jazykoch. Pre projekty vyvíjané v manažovaných programovacích jazykoch (napr. *C#*, *Visual Basic*) umožňuje vyhodnotiť päť vstavaných metrík na používateľom zvolené projekty (Microsoft, 2012). Výsledky metrík je možné prezerať ako na úrovni celých projektov, tak aj jednotlivých tried a ich metód. Vyhodnocované metriky:

- index udržovateľnosti,
- cyklomatická zložitosť,
- hĺbka dedenia,
- zviazanosť tried,
- počet riadkov zdrojového kódu.

Udržovateľnosť projektu (triedy alebo metódy) vyhodnotí na základe odporúčaných hodnôt metriky index udržovateľnosti (Tabuľka 2-1, s. 9). Počet riadkov zdrojového kódu je vyjadrený počtom *IL* inštrukcií[8] vykonaných zdrojovým kódom a nie počtom riadkov, ktoré programátor sám vytvoril. Vďaka tomu je možné vyhodnotiť koľko inštrukcií vykonáva metóda.

Výsledky metrík je možné v nástroji uložiť do hárku tabuľkového procesora Excel balíka Microsoft Office, alebo vytvoriť novú úlohu na projekte, ak je pripojený na Microsoft Team Foundation Server. Nevýhodou tohto riešenia je, že výsledky meraní sa nearchivujú a nástroj neponúka možnosť ich vizualizácie, vytvárania prehľadov, ani určovania zmien od predchádzajúcich meraní.

Microsoft Visual Studio vo verzii Ultimate ponúka aj vytvorenie grafu závislostí na zvolené súčiastky zdrojového kódu (Obrázok 2-1). Výsledný graf je reprezentovaný súborom vo formáte DGML, je možné s ním interagovať a navigovať sa s ním priamo v zdrojovom kóde. Výhodou práce s grafom závislostí priamo vo vývojovom prostredí je možnosť pohľadu na závislosti súčiastok zvolenej úrovne granularity, napr. knižnice, triedy, metódy.



Obrázok 2-1 Graf závislostí v nástroji Microsoft Visual Studio.

[8] Microsoft Intermediate Language, MSIL

2.4.2. NDepend

Riešenie NDepend[9] pozostáva z rozšírenia pre vývojové prostredie Microsoft Visual Studio a samostatných aplikácií. NDepend prevyšuje možnosti merania projektu samotného vývojového prostredia z viacerých hľadísk. Rozšírenie dopĺňa prostredie o možnosť definovania vlastných metrík dopytovaním sa nad zdrojovým kódom pomocou jazyka *CQLINQ*, vychádzajúceho z jazyka *LINQ* pre aplikačný rámec *Microsoft .NET*. NDepend obsahuje 82 preddefinovaných metrík, ktoré vyhodnocujú organizáciu kódu, jeho kvalitu, štruktúru a pokrytie[10]. Autori rozdeľujú tieto metriky do šiestich skupín podľa ich úrovne zamerania:

- metriky aplikácie,
- metriky komponentov,
- metriky menných priestorov,
- metriky typov,
- metriky metód,
- metriky polí.

Príklad zapísania metriky v jazyku *CQLINQ* uvádza Ukážka 2-1. Výsledky metrík sú porovnávané s odporúčanými hodnotami a rozšírenie upozorňuje pri prekročení povolených hraníc. Výhodou riešenia je možnosť archivovať výsledky metrík, porovnávať výsledky s predchádzajúcimi analýzami a aj možnosť vyhodnocovať metriky v rámci kontinuálnej integrácie projektu.

Ukážka 2-1 Porovnanie pôvodnej a novej cyklomatickej zložitosti upravených metód pomocou CQLINQ.

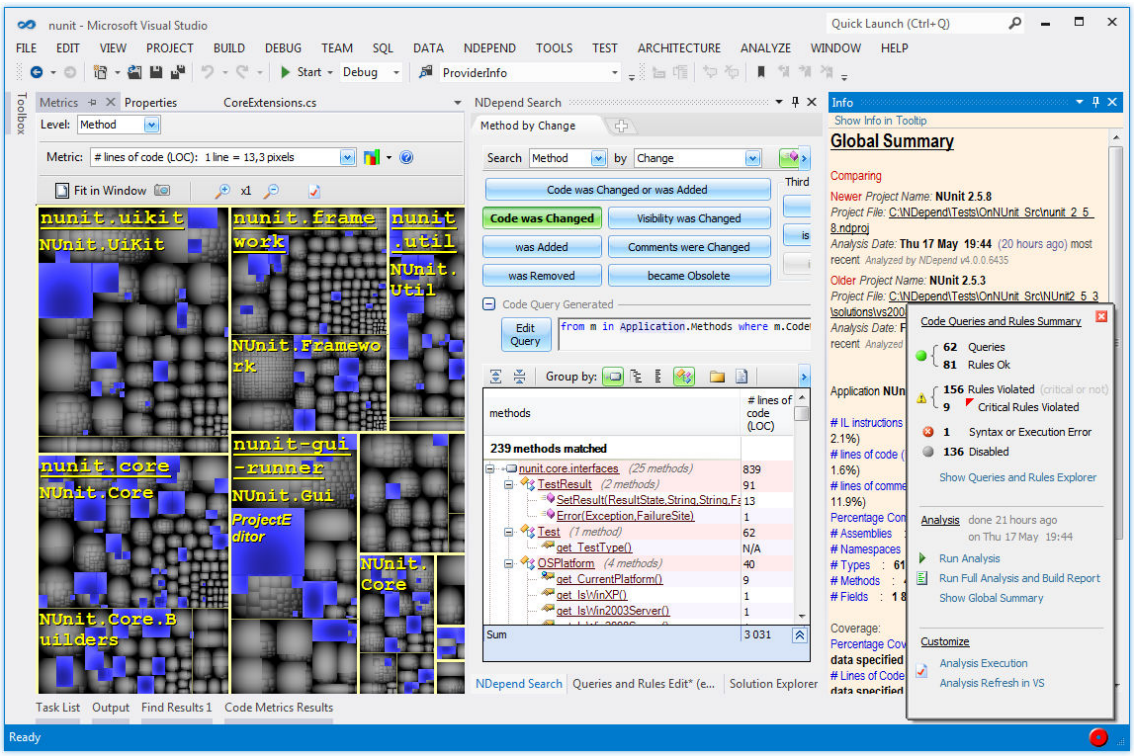
```
from m in JustMyCode.Methods.Where(m1 => !m1.WasAdded())
let oldComplexity = m.OlderVersion().CyclomaticComplexity
let newComplexity = m.CyclomaticComplexity
where oldComplexity > 8 && oldComplexity < newComplexity
select new { m, oldComplexity, newComplexity }
```

Rozšírenie NDepend pridáva do prostredia Visual Studio viacero grafických výstupov výsledkov zvolených metrík. Pre prehľadný prieskum výsledkov poskytuje mapu (Obrázok 2-2). Identifikovanie cyklov závislostí modulov, tried či aj metód triedy je možné pomocou grafu závislostí alebo matice (Obrázok 2-3).

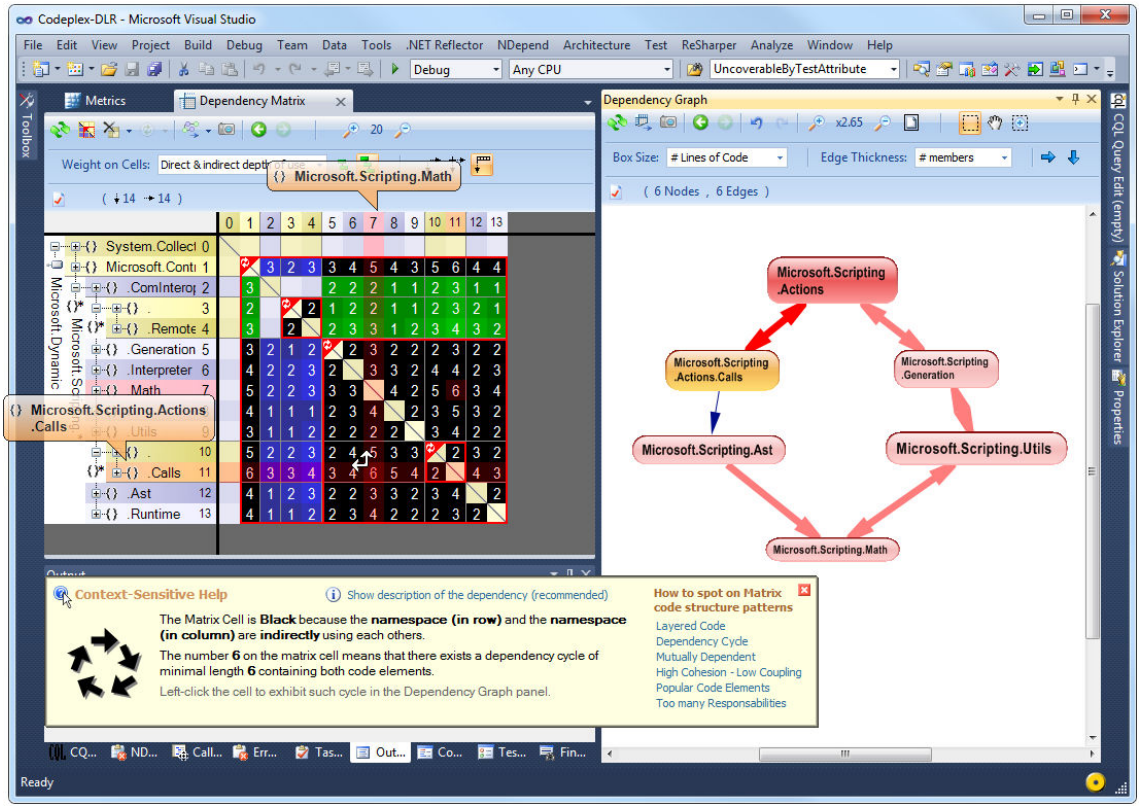
NDepend je komplexné rozšírenie do vývojového prostredia pre vyhodnocovanie softvérových metrík. Navyše poskytuje aplikačné rozhranie pre vytvorenie ďalších nástrojov vhodných napr. na vyhľadávanie v kóde alebo prepojenie NDepend s inými nástrojmi pre monitorovanie softvérového projektu.

[9] NDepend, URL: http://www.ndepend.com/

[10] Metriky v NDepend. URL: http://www.ndepend.com/Metrics.aspx



Obrázok 2-2 Mapa výsledkov vyhodnocovanej metriky v NDepend[11].



Obrázok 2-3 Matica a graf závislostí modulov softvéru v NDepend[11].

[11] Obrázok prevzatý z webstránky nástroja NDepend: `http://www.ndepend.com/`

2.5. Diskusia

Cestou k zabezpečeniu kvality softvérového projektu, k splneniu požiadaviek a využitia dostupných zdrojov je jeho dôsledné sledovanie a vyhodnocovanie. Vonkajšie a vnútorné vlastnosti úrovní softvérového projektu napovedajú o ich správaní a štruktúre. V tejto kapitole sme bližšie analyzovali metriky pre vyhodnotenie vlastností softvérového produktu. Vyhodnotením vnútorných vlastností a definovaním ich vzťahu k vonkajším vlastnostiam môžeme sledovať aj vysokoúrovňové vlastnosti projektu, napr. jeho použiteľnosť, udržovateľnosť alebo kvalitu.

Pri vyhodnocovaní softvéru sa však stretávame s problémom interpretácie definícií a výsledkov metrík, ich implementácie a definície závislostí vnútorných a vonkajších vlastností. Ako sme uviedli, existuje viacero modelov kvality softvéru, viacero možností ako vyhodnocovať rovnakú vlastnosť, ale aj viacero implementácií metrík. Pri meraní zdrojového kódu závisia výsledky metrík aj od použitého programovacieho jazyka. Výsledné vyhodnotenie vlastností softvéru tak závisí od konkrétneho projektu a kladených požiadavkách. Príkladom je už jednoduché určenie veľkosti súčiastok počtom riadkov zdrojového kódu. V praxi sa stretávame s odporúčaniami na maximálne 20 až 30 riadkov metódy triedy. V odôvodnených prípadoch môžeme toto odporúčanie samozrejme porušiť.

Medzi jednoznačne definovateľné metriky môžeme zaradiť identifikáciu závislostí zdrojového kódu, tá však naráža na obmedzenia syntaktickej analýzy, na ktorej je závislá. Napriek rozsiahlosti výskumu v oblasti metrík zdrojového kódu, nedokážeme uvedenými metrikami vyhodnotiť príčiny vzniku problémov, ktoré dokážeme identifikovať. Zdrojový kód je výsledkom práce programátorov, uvedené metriky vyhodnocujú práve len tento výsledok.

Odvodenie vlastností softvéru pomocou metrík umožňuje interpretovať jeho inak ťažko-opísateľný vývoj, čo prispieva k jeho manažmentu. Príkladom je sledovanie požadovaných vlastností produktu a vykonávanie rozhodnutí počas vývoja pre zlepšenie ich hodnôt. Aj z tohto dôvodu si získali popularitu nástroje podpory vyhodnocovania projektu. Tieto nástroje sú použiteľné ako pre vyhodnotenie, tak aj počas vývoja viacerými zúčastnenými stranami na softvérovom projekte. Identifikované závislosti v zdrojovom kóde nepoužívame len na vyhodnotenie zviazanosti softvérových súčiastok, ale aj pre navigáciu a analýzu zdrojového kódu samotnými programátormi počas vývoja a údržby, napr. pre predídenie negatívneho vplyvu úpravy súčiastky na iné závislé súčiastky. Z programátorského prostredia je známy výrok o vytvorení ďalších 2 nových chýb po opravení chyby v zdrojovom kóde. Práve preto vidíme vysoký význam metriky identifikácie závislostí, aby sme zamedzili negatívnym dopadom upravovania existujúceho kódu, zároveň je však potrebné ich identifikovať aj iným spôsobom než len syntaktickou analýzou.

Vyhodnocovanie softvéru na základe obsahu zdrojového kódu môžeme zhrnúť ako často nejednoznačné a náročné, pričom zanedbáva okolnosti jeho vzniku. To otvára priestor pre vznik softvérových metrík vychádzajúcich z aktivít programátora a kontextu vývoja softvéru. Zaujímavým prístupom je napr. sledovanie pozitívneho alebo negatívneho vplyvu prostredia na výsledky metrík zdrojového kódu.

.

Kapitola 3

Vplyvy na vlastnosti softvéru

Existujúce prístupy merania softvérových produktov založené na analýze zdrojového kódu nezohľadňujú príčiny, ktoré vplývali na jeho výsledné vlastnosti. Počas vykonávania softvérového projektu je práca programátora ovplyvňovaná rôznymi vplyvmi, deje sa vždy v určitom kontexte. Tieto vplyvy môžeme využiť pri analyzovaní vlastností vytváraných softvérových produktov, a tak usudzovať nad nimi nielen na základe ich obsahu, ale aj na základe toho za akých okolností boli vytvárané alebo modifikované.

Vplyvy na vlastnosti softvéru môžeme rozdeliť medzi aktivity a kontext programátora. Aktivity sa týkajú činností programátora a kontext stavu jeho znalostí, okolia a ostatných činností, ktoré programátor nevykonáva, ale majú vplyv na neho alebo vyvíjaný softvér.

3.1. Aktivity programátora

Činnosti, ktoré programátor vykonáva počas práce na projekte môžeme rozdeliť do troch skupín:

- *priamo spojené s vývojom* – napr. programovanie, modelovanie komponentov,
- *nepriamo spojené s vývojom* – napr. hľadanie riešení, študovanie dokumentácie,
- *nespojené s vývojom* – napr. prehliadanie Internetu zo zábavy.

Sledovaním skupín aktivít a ich vplyvu na výsledok práce programátora sa venovalo viacero vedeckých prác. Aktivity spojené s vývojom priamo zahŕňajú prácu vo vývojovom prostredí, kde programátori vytvárajú zdrojový kód, revidujú ho, či testujú. Zvyšné skupiny aktivít sa vykonávajú aj mimo vývojového prostredia, v iných aplikáciách a operačnom systéme.

3.1.1. Chyby v zdrojovom kóde

Autori v (Purushuthaman & Perry, 2005) sledovali vplyv periodicity zmien a ich veľkosti na vznik chýb v zdrojovom kóde. Motivácia ich výskumu vychádza z vyvrátenia presvedčenia, že malá zmena (napr. jeden znak, reťazec alebo riadok) nemôže mať negatívny vplyv. Svoj experiment vykonali na projekte o veľkosti 200 miliónov riadkov zdrojového kódu a zmeny v kóde triedili do skupín podľa ich dôvodu vykonania (oprava, vylepšenie, prispôsobenie kódu a zmena po prehliadke) a typu zmeny (pridanie, úprava, zmazanie). Výsledkom experimentu je odhalenie, že 4% jednoriadkových zmien vnesú do zdrojového kódu chybu. Taktiež odhalili, že 40% zmien pre opravu chýb vytvorí novú chybu. Sledovaním vykonávania zmien v zdrojovom kóde môžeme hľadať ich prepojenie na chybovosť kódu.

Vznikom chýb v zdrojovom kóde sa zaoberajú aj autori v (Abreu & Premraj, 2009) sledovaním aký vplyv má frekvencia komunikácie pracovníkov na vznik chýb. Ich predpoklad je, že kód, ktorý je vytvorený v čase častej komunikácie vývojárov bude kvalitnejší než kód napísaný v čase, kedy vývojári komunikujú sporadicky. Autori sledovali komunikáciu vývojárov a nahlasovania chýb v kóde na projekte Eclipse[12]. Medzi výkyvmi v komunikácii a vznikom chýb našli koreláciu, a tak tvrdia, že frekvencia komunikácie môže slúžiť ako indikátor výberu súborov na testovanie voči chybám. Zároveň odhalili, že vývojári komunikujú medzi sebou viac v čase, keď vzniká viac nových chýb. Autori predpokladali, že tieto výkyvy budú ovplyvnené aj termínmi odovzdávania verzií projektu, ale tento vzťah neidentifikovali.

3.1.2. Znalosť programátora

Programátor počas vývoja zasahuje do rôznych častí kódu, alebo aj preberá prácu od iných programátorov. Zisťovaním znalostí programátora o konkrétnom kóde sa zaoberali autori v (Fritz, et al., 2007). Frekvencia práce s kódom vplýva na programátorovu znalosť. Kvalitatívnym vyhodnocovaním s programátormi autori zistili, že model znalostí na základe aktivít vie pomôcť programátorom pri návrate ku kódu, alebo aj pri odporúčaní komu priradiť úlohu opravy chyby. V práci (Fritz, et al., 2010) autori vytvárajú model programátora, v ktorom rozlišujú medzi jeho záujmami (pohyb v zdrojových kódoch) a aktivitami. Model programátora ovplyvňujú aj zmeny vykonané inými programátormi v kóde, ktorý on vytvoril.

Odporúčacím systémom pre vývoj softvéru sa zaoberali autori v (Christidis, et al., 2012). Systém je zameraný na odporúčanie programátora, ktorému má byť priradená nahlásená chyba. V práci navrhujú model programátora s jeho znalosťami zo sémantickej analýzy zdrojového kódu a s kvantitatívnym meraním jeho aktivity. Znalosť programátora závisí od:

- zdrojového kódu, ktorý vytvoril,
- časov odovzdania vykonaných zmien,
- histórie komunikácie.

Modelovaním programátora sa venujú aj autori v práci (Kuric & Bieliková, 2013), zameriavajú sa však na vyhodnotenie miery znalostí technológií, ktoré programátor používa. Iným zameraním autorov je vyhodnotenie „karmy“ programátora na základe kódu, ktorý vytvoril, pretože ten môže byť používaný, vyhľadávaný a hodnotený inými programátormi.

3.1.3. Prínos programátora do projektu

Meranie prínosu programátorov do projektu sa najčastejšie vykonáva meraním počtu vytvorených riadkov. Ako sme uviedli, vytváranie zdrojového kódu však nie je jediná činnosť programátora. Autori v (Gousios, et al., 2008) navrhujú pre meranie prínosu sledovať aj komunikáciu v tíme, príspevky do dokumentácie, oznamovanie a uzatváranie chýb, alebo aj rozlišovať medzi typmi odovzdaných súborov. Sledované aktivity rozdeľujú do piatich skupín:

- práca so zdrojovým kódom v repozitári projektu,
- komunikácia prostredníctvom e-mailov a diskusných fór,
- správa chýb v projekte,
- príspevky do dokumentácie,
- komunikácia prostredníctvom konferenčného systému.

[12] Eclipse Project. `http://www.eclipse.org/`

Pre každú aktivitu je určené, či má pozitívny alebo negatívny prínos do projektu. Prínos do projektu CF (angl. contribution factor) programátora d vyjadrujú následne vzťahom, v ktorom sčítajú všetky typy aktivít s nastavenými váhami α až ε, kde $A_i^a(d)$ je programátorova aktivita jednej z uvedených skupín a (Gousios, et al., 2008):

$$CF(d) = \alpha \sum_{i=0}^{n} w_i^{rep} A_i^{rep}(d) + \cdots + \varepsilon \sum_{i=0}^{n} w_i^{irc} A_i^{irc}(d)$$

Hodnota w_i^a vyjadruje pozitívny alebo negatívny prínos aktivity.

Autori overovali metódu na viacerých projektoch s otvoreným zdrojovým kódom (Kalliamvakou, et al., 2009), ktorých výsledkom bolo potvrdenie predpokladu, že príspevok do projektu nesúvisí len so zmenami v riadkoch zdrojového kódu. Autorom sa zároveň podarilo potvrdiť platnosť Paretovho pravidla, kedy 70% prínosu do projektu prinieslo 30% vývojárov.

3.2. Kontext vývoja softvéru

Na prácu programátora a jeho interakciu so zdrojovým kódom vplýva aj kontext, ktorý môžeme považovať ako akúkoľvek informáciu o situácii programátora, v ktorej sa nachádza. Kontext môžeme všeobecne opísať štyrmi kategóriami – miesto, čas, okolie, identita (Dey & Abowd, 2000). Pri vývoji softvéru ich zhrnieme ako:

- *interné vplyvy* – znalosť programátora, jeho nálada, fyzický stav, ale aj termíny odovzdania práce,
- *externé vplyvy* – prostredie, v ktorom sa programátor nachádza, napr. čas, počasie, miesto vykonávania práce,
- *kontext úlohy, ktorú programátor vykonáva* – zadanie, dokumentácia projektu, študovaný a upravovaný zdrojový kód.

Medzi interné vplyvy zaraďujeme aj znalosti programátora, pretože tie vplývajú na kvalitu jeho výsledkov. Fyzický stav programátora vplýva na jeho pozornosť počas práce a únavu, čo môže mať za následok zvýšenie pravdepodobnosti vzniku chýb. Medzi interné vplyvy zaraďujeme aj emocionálny stav programátora, keďže jeho nálada môže ovplyvňovať jeho sústredenosť, vznik chýb a kvalitu zdrojového kódu, ktorý vytvorí (Fejes, 2013). Prirodzene na vznik chýb vplýva aj stres programátora, blížiace sa termíny odovzdania a nestíhanie dokončenia prác.

Externé vplyvy dopĺňajú interné vplyvy o okolie, v ktorom sa programátor nachádza, najmä prostredie, čas, počasie i jeho sociálne väzby s ľuďmi v okolí.

Prostredie

Vplyv chaotického prostredia vývoja projektu na výsledný produkt sledovali autori v (Hassan & Holt, 2003). Ich predpokladom bolo tvrdenie, že chaotický (neriadený) proces vývoja negatívne vplýva na výsledný produkt a jeho zdrojový kód. Pod chaotickým vývojom rozumieme zmenu v zdrojových kódoch na rôznych miestach a v rôznom čase, ale aj zmenu cieľov projektu. Programátor sa nachádza v prostredí požiadaviek a zadaní, ktoré sa neustále menia, a preto sa musí neustále adaptovať na zmeny. Pre vyhodnotenie chaosu vo vývoji použili autori v práci Shannonov model neistoty (angl. Shannon Entropy) nad súbormi zdrojového kódu a informáciami o jeho vývoji. Vyhodnotením experimentu identifikovali súbory, ktoré môžu byť náchylnejšie na chyby.

Čas

Čas zmien v kóde, ich vplyv na kvalitu kódu a prínos nových chýb vyhodnocovali autori v (Eyolfson, et al., 2011; Zeleník, 2013). Ich cieľ bol odhaliť, či vznik kódu s chybami závisí od skúseností jeho autora a od času, kedy bol vytvorený. Výsledkom je, že zmeny vykonané neskoro večer obsahujú viac chýb, kým zmeny vykonané pred obedom najmenej. Autori odhalili aj nepriaznivý vplyv termínov odovzdávania projektu na vznik chýb. Je pravdepodobné, že rozdelenie práce do pracovných dní týždňa by mohlo mať vplyv na prácu programátorov (Bryson & Forth, 2007), autori to však jednoznačne neidentifikovali.

Počasie

Programátor pracuje na vývoji softvéru v miestnosti, a tak nie je priamo ovplyvnený počasím. Počasie však vplýva na jeho sústredenosť a produktivitu. Vplyv počasia na počet odovzdaných príspevkov na projektoch skúmal autor v (Davis, 2013). Vo svojom výskume odhalil, že na odovzdávanie príspevkov vplýva počasie tak, že počet odovzdaní vzrástol o 10,1% počas daždivých dní oproti slnečným. Natrénovaním štatistického modelu odhalil kopírovanie priebehu počasia počtom odovzdaní zdrojového kódu.

Kontext úlohy

Pri práci na softvérovom projekte dostane programátor úlohu vyššej úrovne než činnosti, ktoré nakoniec vykonáva, napr. opravenie chyby a úprava konkrétneho zdrojového kódu. Splnenie zadania si vyžaduje analýzu možných riešení, existujúcich častí systému, nájdenie riešenia a jeho samotné aplikovanie. Cieľom práce v (Coman, 2007) je návrh odhadovania úloh vyššej úrovne od činností sledovaných na nízkej úrovni (práca vo vývojovom prostredí, otváranie súborov), a tak identifikovania kontextu úlohy, t.j. činností súvisiacich s úlohou.

Sledované aktivity programátora môžeme podľa (Coman & Sillitti, 2008) rozdeľovať do sedení na základe histórie interakcií vo vývojovom prostredí, a tak identifikovať miesta v kóde, ktoré sa zdajú byť kľúčové pre riešenie úlohy. Kľúčové miesta (metódy, triedy, balíky) sú pre splnenie úlohy častejšie navštevované v rámci jej riešenia než inokedy, a tiež v približne rovnakom čase. Autori v (Antunes, et al., 2012) identifikovali, že kontext úlohy môžu využiť iní programátori pri študovaní implementácie riešenia inej úlohy, alebo pri odporúčaní výsledkov hľadania v zdrojovom kóde.

Softvérové súčiastky, s ktorými programátor pracuje pre splnenie úlohy vyplývajú z jeho záujmu. Autori v (Kersten & Murphy, 2006) využívajú model stupňa záujmu (angl. degree of interest) pre reprezentovanie činností nad súbormi, ktoré majú význam pre plnenie vykonávanej úlohy. Ich práca vychádza z projektu *Mylyn*[13], ktorý integruje úlohy programátora na projekte do vývojového prostredia *Eclipse*. Rozšírenie prostredia zaznamenáva programátorovu činnosť a dáva ju do kontextu s úlohou, na ktorej programátor pracuje. Ak sa programátor vracia k práci, alebo jeho prácu preberá iný programátor, má tak možnosť efektívnejšieho zorientovania sa v zdrojovom kóde s prehľadom aktivít, ktoré pri plnení práce vykonal. Výhodou riešenia je priame naviazanie činností na plnené úlohy, takže návrat k práci pri oprave chyby v súčiastke zdrojového kódu je efektívnejší (Kersten & Murphy, 2006). Rekonštrukcia kontextu úlohy programátora pozostáva z ponúknutia súborov zdrojového kódu, na ktorých v rámci úlohy pracoval.

[13] Eclipse Mylyn. `http://www.eclipse.org/mylyn/`

3.3. Diskusia

Vyhodnocovať softvérový projekt môžeme nie len na základe obsahu zdrojového kódu, ale aj na základe zaznamenaných aktivít programátora a kontextu. Činnosť programátora, jeho znalosti a stav prostredia majú vplyv na výsledky jeho práce. Ak pri vyhodnocovaní softvérového projektu zanedbáme príčiny, ktoré ovplyvnili výsledok práce programátora, tak sa oberáme o množstvo informácií a sémantiky v procesoch softvérového projektu.

Aktivity programátora sme rozdelili do skupín podľa nástrojov, ktoré programátor používa. Rozlišujeme medzi priamou (vo vývojovom prostredí) a nepriamou prácou na projekte (hľadanie riešení, komunikácia). Taktiež si uvedomujeme, že na vlastnosti softvéru vplývajú aj programátorove aktivity, ktoré nie sú zlučiteľné s prácou na projekte. Ako príklad môžeme uviesť časté prerušovanie práce prehliadaním Internetu, odbiehaním od počítača alebo telefonovaním s rodinnými príslušníkmi. Mnohé vedecké práce sa zaoberajú vplyvom aktivít na vlastnosti softvéru, hlavne jeho chybovosť, napr. odovzdávanie výsledkov prác, alebo aj práca v neskorých hodinách.

Kontext vývoja softvéru charakterizujeme ako informácie o situácii programátora, jeho znalostí a okolia. Pritom ale neuvažujeme len jeho základné rozdelenie (miesto, čas, okolie, identita), ale ho špecifikujeme pre doménu vývoja softvéru. Konkrétne nás zaujíma kontext úlohy, v ktorom vidíme dôležitý zdroj pre podporu udržovateľnosti softvéru. Znalosť o súčiastkach, ktoré súviseli s vykonanou prácou programátora je veľmi prínosná pri návrate k práci, opravovaní chyby alebo preberaní úlohy po inom programátorovi. Kontext úlohy tak predstavuje prepojenie súčiastok na základe aktivít programátora, podobne ako identifikujeme závislosti ich syntaktickou analýzou.

Sledovaním aktivít programátora a kontextu môžeme hľadať ich vzťahy s výslednými vlastnosťami softvérového produktu, čo za bežných okolností vykonávame iba na základe obsahu zdrojového kódu. Nastáva otázka, či dokážeme vyhodnotiť vlastnosti s podobnou úspešnosťou, prípadne či je takéto vyhodnocovanie výpočtovo efektívnejšie ako periodická a hĺbková analýza zdrojového kódu.

Zaznamenávanie aktivít programátora a jeho kontextu je náročné z dôvodu presného zachytenia skutočnej činnosti programátora, a zároveň aby nebolo invazívne a nevyrušovalo programátora pri tvorivej práci. V poslednej dobe sa uplatňuje zaznamenávanie činnosti na pozadí v rámci operačného systému a vývojových nástrojov, ktoré podporujú túto možnosť prostredníctvom vlastných rozšírení. Zaznamenávanie činností počas práce môže u programátorov vzbudzovať narušanie ich súkromia, preto majú tendenciu sledovanie obmedzovať, prípadne zabúdať ho aktivovať, čo spôsobuje stratu informácií. Keďže ide o formu implicitnej spätnej väzby, ďalším úskalím je interpretácia zaznamenaných informácií a ich spätné vyhodnotenie, najmä za účelom identifikácie dôvodov rozhodnutí. Intenzívne zaznamenávanie aktivít programátora a kontextu vývoja softvéru sprevádza veľké množstvo dát, častokrát reprezentovaných prúdmi záznamov, čo vyžaduje špecifické spôsoby ich ukladania a spracovania. Z týchto dôvodov je zaznamenávanie a vyhodnocovanie aktivít programátora a kontextu pre vyhodnotenie softvéru aktuálnym smerom výskumu v doméne softvérového inžinierstva (Aalst, et al., 2012; Bieliková, et al., 2014; Kersten & Murphy, 2006).

Kapitola 4

Východiská a ciele práce

V softvérových projektoch identifikujeme množstvo artefaktov, ktoré sú medzi sebou prepojené, vytvárané a používané rôznymi účastníkmi projektu. Artefakty softvérového projektu vznikajú počas všetkých jeho etáp, najmä zdrojový kód, dokumentácia alebo zadania práce. Aktivity programátora a kontext vývoja softvéru sú ďalším zdrojom informácií o týchto artefaktoch v podobe implicitnej spätnej väzby, ktorú je náročné interpretovať. Preto sa na posúdenie využitia dát aktivít programátora a kontextu vývoja softvéru pre jeho vyhodnotenie pozeráme z týchto pohľadov (Aalst, et al., 2012):

- spôsoby získavania dát a ich vplyv na zaznamenávaný objekt,
- možnosti ich vyhodnotenia – manuálne, automatické, poloautomatické,
- možnosti využitia zaznamenaných dát – záverečné vyhodnotenie alebo doplnenie procesu,
- cieľoví používatelia získaných informácií – programátori, testeri, manažéri.

Explicitné informácie o vývoji softvéru bežne vyhodnocujeme:

- manuálne – prehliadky zdrojového kódu, testovanie výsledkov,
- automaticky – vyhodnotenie metrík zdrojového kódu, vyhodnotenie jednotkových testov,
- poloautomaticky – manuálne spracovanie výsledkov automatického vyhodnotenia.

Počas riešenia našej práce sme vytvorili návrh poloautomatickej identifikácie zaujímavých miest v zdrojovom kóde na prehliadku využitím implicitne aj explicitne získaných. Tento návrh uvádzame v prílohe C. V rámci riešenia našej práce sme sa však detailne zamerali na automatické vyhodnotenie aktivít programátora. Našu prácu, a aj uvedený návrh, staviame na základoch projektu PerConIK, do ktorého bol autor tejto práce aktívne zapojený.

4.1. Priestor softvérových artefaktov a informačný priestor Webu

Priestory artefaktov softvérového projektu a ich prepojení môžeme prirovnať k Webu, ktorý môžeme taktiež opísať ako priestor webových stránok (artefaktov), ich prepojení a používateľov. Vzťahy medzi artefaktami v jednom aj druhom priestore sú zaujímavé, keďže nájdu uplatnenie pri podpore riešenia rôznych úloh:

- prepojenie zadaní úloh k súčiastkam zdrojového kódu riešenia,
- analýza prepojení súčiastok zdrojového kódu pre potreby ich úpravy alebo doplnenia,
- dosiahnutie požadovanej funkcionality prepojením súčiastok zdrojového kódu,
- navigácia medzi webovými stránkami pomocou odkazov,
- odvodzovanie znalostí o entitách opísaných prepojenými webovými stránkami,
- odporúčanie nových informácií podobnosťou prepojených webových artefaktov.

Výskum v doméne Webu sa venuje zaznamenávaniu a vyhodnocovaniu vzťahov medzi webovými artefaktmi (Knutov, et al., 2009). Nájdenie paralely medzi priestormi softvérových projektov a Webu nás motivuje aplikovať podobné metódy aj v tejto doméne (Bieliková, et al., 2013).

Medzi uvažované metódy patrí vyhľadávanie a odporúčanie informácií, ale aj modelovanie používateľa či domény (Knutov, et al., 2009):

- *Vyhľadávanie* – zjednodušenie a zrýchlenie prístupu k informáciám vychádzajúce zo vzťahov medzi nimi a používateľmi. Vyhľadávanie ponúkne používateľom relevantnejšie výsledky využitím získanej sémantiky z prepojených webových artefaktov.
- *Odporúčanie* – ponúknutie informácií používateľom, ktoré by ich zaujímali bez ich vlastnej aktivity vyhľadania. Jednoduchým príkladom je odporučenie filmu používateľovi, ktorý by si mohol chcieť pozrieť na základe jeho vlastných preferencií.

V prípade softvérového projektu môžeme uplatniť tieto metódy napr. keď programátor hľadá konkrétnu súčiastku, na ktorej v minulosti pracoval (Antunes, et al., 2012), alebo je mu odporučený dokument opisujúci upravenú súčiastku. Prostredie, s ktorým používateľ pracuje je v oboch prípadoch čiastočne podobné. Vývojové prostredia softvéru zobrazujú súbory zdrojového kódu v podobe kariet, podobne ako prehliadač webových stránok. Používateľ sa môže prepínať medzi kartami a navigovať sa medzi súbormi tak, ako medzi webovými stránkami.

Výsledky personalizovaného vyhľadávania či odporúčania závisia od použitého modelu pre ich vyhodnotenie. Zbierať a uchovávať informácie môžeme v modeli ako o používateľovi, tak aj o doméne, v ktorej používateľa sledujeme, alebo aj o iných hľadiskách nášho záujmu. Kvalita modelu ovplyvňuje kvalitu výsledkov metód, preto je dôležité použiť model správne opisujúci používateľove záujmy, znalosti a ciele, čo je netriviálna úloha.

Informácie o webových artefaktoch môžeme získavať automaticky zaznamenávaním aktivity používateľa a kontextu, alebo aj priamo od samotného používateľa, ak je motivovaný opisovať používané artefakty. Jednou z motivácií pre získanie explicitnej spätnej väzby používateľa k artefaktom je jeho možnosť rýchlejšie sa k nim vrátiť, keď má pri nich zaznamenané vlastné vedomosti a v budúcnosti ich môže použiť bez zbytočného dohľadania.

4.2. Projekt PerConIK

Projekt PerConIK[14] (Výskum metód získavania, analýzy a personalizovaného poskytovania informácií a znalostí) sa zaoberá sledovaním vývoja softvérových projektov a používaním metód typických pre doménu Webu, tzv. „webifikáciou vývoja softvérových projektov“ (Bieliková, et al., 2013). Cieľom softvérového inžinierstva je vytváranie kvalitného softvéru (Project Management Institute, 2004) a podobne ako pri Webe, požadujeme zlepšenie možností vyhľadávania v artefaktoch projektu, nie len na základe ich obsahu. Jednotliví programátori pracujú na podobných funkcionalitách a odporúčanie vhodného riešenia im vie pomôcť pri práci. Znalosti ostatných

[14] PerConIK – Personalized Conveying of Information and Knowledge.
http://perconik.fiit.stuba.sk/

programátorov uchované pri ich výsledku vedia zamedziť vzniku zlých riešení, a taktiež pomôcť novým programátorom sa rýchlejšie zorientovať v existujúcich artefaktoch. Tak môžu jednoduchšie nájsť dobré programátorské postupy. Zamedzíme tým vnášaniu zlých praktík do kódu a umožníme samovzdelávanie. V rámci projektu PerConIK sa pracuje s dátami domény softvérovej spoločnosti ako vstupmi pre metódy modelovania, vyhľadávania a odporúčania (Bieliková, et al., 2014):

- zdrojové kódy prepojené na programátorov, projekty, použité technológie,
- komunikácia programátorov,
- dokumenty prislúchajúce k vývoju a zdroje pôvodných riešení,
- informačné značky vytvorené k artefaktom manuálne alebo automaticky.

Okrem získavania informácií o artefaktoch vývoja softvéru sa projekt zameriava aj na získavanie informácií o aktivitách a kontexte programátora:

- Práca v operačnom systéme:
  - prepínanie okien aplikácií,
  - využitie systémových prostriedkov.
- Práca vo vývojovom prostredí:
  - manipulácia so zdrojovým kódom – napr. kopírovanie, vyhľadávanie,
  - operácie so súbormi zdrojového kódu – napr. pridanie, mazanie, presúvanie,
  - odovzdanie súborov do projektového úložiska,
  - zmena stavu prostredia – napr. kompilácia, krokovanie, upravovanie
  - operácie nad projektami.
- Prehliadanie Webu, ktorý je možným zdrojom informácií pre programátora:
  - navštevovanie stránok,
  - kopírovanie obsahu stránok do zdrojového kódu.
- Komunikačný kanál:
  - zmena stavu prihlásenia.

Uvedené aktivity sú zaznamenávané pomocou rozšírení do vývojových prostredí Microsoft Visual Studio a Eclipse, a aj webového prehliadača Mozilla Firefox. Rozšírenia komunikujú s aplikáciou pre odosielanie zaznamenaných udalostí do centrálneho úložiska. Sledovanie udalostí však nie je presné z dôvodu obmedzení aplikačných rozhraní rozširovaných prostredí, napr. nemožnosť určiť pôvodný súbor, z ktorého sa programátor prepol na nasledujúci.

Autor tejto práce sa podieľal na vývoji modulu pre automatické generovanie informačných značiek. Projekt PerConIK je nasadený v školskom aj firemnom prostredí. Množstvo a kvalita nazbieraných dát o aktivitách a kontexte programátorov tak závisí aj od typov projektov, pretože študenti nepracujú v rovnakom režime ako profesionálni vývojári, majú tendenciu sledovanie vypnúť alebo ho vôbec nepoužívať.

4.3. Informačné značky

V rámci projektu PerConIK boli zavedené informačné značky (Bieliková & Rástočný, 2012), ktoré umožňujú uchovávať dodatočné informácie k zdrojovému kódu a tiež kontextu, v ktorom vznikol, resp. bol modifikovaný, a tak vytvárať ľahkú sémantiku nad informačným priestorom softvérových projektov (Rástočný & Bieliková, 2013). Informačné značky, ako metadáta označeného obsah, sú definované ako trojice:

- *typ* – definuje typ a význam informačnej značky,
- *ukotvenie* – identifikácia označeného informačného artefaktu,
- *telo* – reprezentuje štruktúrovanú informáciu (podľa typu informačnej značky).

Informačné značky vznikajú *manuálne*, t.j. zadané používateľmi počas práce so zdrojovým kódom, alebo *automaticky*, vyhodnotené a odvodené zo zaznamenaných aktivít a kontextu programátorov. Samotná informačná značka je v zdrojovom kóde reprezentovaná štruktúrovaným komentárom, tým pádom ich je možné synchronizovať podobne ako samotný zdrojový kód.

4.3.1. Manuálne zadávané informačné značky

Použitím rozšírení vývojových nástrojov Microsoft Visual Studio a Eclipse môžu programátori vytvárať nasledujúce informačné značky k zdrojovému kódu:

- *TODO* – vyjadrenie potreby dokončenia úlohy,
- *Bug* – označuje identifikovanú chybu v zdrojovom kóde,
- *Ranking* – hodnotenie označeného zdrojového kódu,
- *Smell* – výskyt pachu v zdrojovom kóde,
- *CodeReview* – identifikovanie porušenia konvencie v kóde pri jeho prehliadke,
- *Recommendation* – odporúčanie ako zlepšiť označený zdrojový kód.

Manuálne informačné značky sú vhodné na hodnotenie zdrojového kódu po iných programátoroch, hlavne vo fáze jeho prehliadky. Aj napriek možnostiam automatickej identifikácie pachov ich môže programátor sám označiť značkou *Smell*, a tým podnietiť jeho autora k opraveniu.

4.3.2. Generované informačné značky

Automaticky generované značky vznikajú vyhodnotením činnosti programátora vo vývojovom prostredí i mimo neho, ktorú zaznamenáva aplikácia „logovača“ vykonávaná v pozadí a odosiela do centrálneho úložiska. Spomedzi automaticky generovaných značiek môžeme spomenúť:

- *Zložitosť zdrojového kódu* – zložitosť vypočítaná po odovzdaní zdrojového kódu,
- *Pôvod zdrojového kódu* – adresa webovej stránky alebo názov súboru, z ktorého bol kód skopírovaný,
- *Záujem o metódu* – miera hľadania zdrojového kódu,
- *Intenzita zmien* – množstvo zmien vo fragmente zdrojového kódu za jednotku času,
- *Autor zdrojového kódu,*
- a ďalšie.

Aktivity programátora, zmeny kontextu a zmeny v zdrojovom kóde vznikajú postupne v čase, tým pádom tvoria prúd udalostí. Zaznamenané udalosti sú však nízkoúrovňovou reprezentáciou. Pre obohatenie sledovaného projektu je vhodné udalosti spracovať, vyhodnotiť a reprezentovať obsiahnutú sémantiku na vyššej úrovni (Aalst, et al., 2012), napr. práve ako informačné značky.

Zaznamenané udalosti môžeme okrem uchovania v relačnej databáze reprezentovať ako grafy RDF trojíc[15], ktorými vyjadríme vzťahy v udalostiach. RDF trojica pozostáva zo subjektu, predikátu a objektu, pričom všetky tri zložky majú priradený jednoznačný identifikátor. Pomocou predikátov reprezentujeme vzťahy prvkov (subjekt alebo objekt). Príklad spracovania udalosti pomocou RDF

[15] Resource Description Framework (RDF). `http://www.w3.org/RDF/`

trojíc môžeme ukázať na udalosti prepnutia medzi dvomi súbormi vo vývojovom prostredí. O zaznamenaných udalostiach uchovávame:

- autor udalosti – programátor,
- projekt, v ktorom programátor vykonal aktivitu,
- názov cieľového súboru, do ktorého sa prepol,
- čas, kedy programátor prepnutie vykonal.

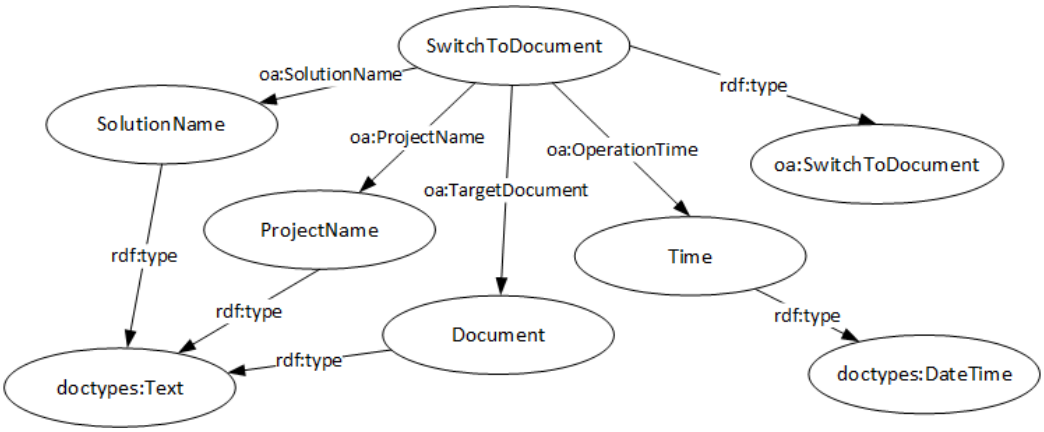
Každá udalosť má určený čas výskytu, preto pri RDF reprezentácii musíme rozlišovať medzi jednotlivými inštanciami typov udalostí (dve udalosti prepnutia medzi súbormi nemajú rovnaký čas). Celú udalosť prepnutia medzi súbormi rozdeľujeme na viacero trojíc (Obrázok 4-1), v reprezentácii používame stereotypy „doctypes“ pre dátové typy a „oa“ pre vlastné typy.

Takto reprezentované udalosti v podobe prúdu vyhodnocujeme priebežne v čase, podobne ako by sme ich uchovávali v statickom repozitári (Le-Phuoc, et al., 2012). Vyhodnocovaniu prúdov prepojených údajov (angl. Linked Stream Data) sa venovali viaceré práce (Le-Phuoc, et al., 2012), ktoré spájajú znalosti zo systémov prúdového spracovania údajov a spracovania prepojených údajov (Aalst, et al., 2012). Vyhodnocovanie umožňujú zápisom dopytov v jazykoch podobných jazyku SPARQL, napr. C-SPARQL (Continuous SPARQL) (Barbieri, et al., 2010), s rozdielom, že sa vyhodnocujú priebežne nad prichádzajúcimi RDF trojicami. Ukážka 4-1 uvádza príklad dopytu v jazyku C-SPARQL. Z výsledkov dopytov následne generujeme informačné značky, ktoré sú ukotvené do zdrojového kódu (Rástočný & Bieliková, 2013).

V projekte PerConIK sa používa generátor informačných značiek pre konverziu zaznamenaných udalostí na prúd RDF trojíc a ich vyhodnocovanie C-SPARQL dopytmi (Bieliková, et al., 2014). Generátor umožňuje definovať nové značky za behu, určovať dopyty a akcie, ktoré sa majú vykonať nad výsledkami dopytov v podobe zdrojového kódu alebo šablón informačných značiek. Takto je možné definovať existujúce i nové značky, ktoré sa budú kotviť do zdrojového kódu a rozširovať sémantiku vývoja softvéru.

Ukážka 4-1 Príklad dopytu nad prúdom RDF trojíc v jazyku C-SPARQL.

```
SELECT ?s (AVG(?o) AS ?avg)
  FROM STREAM <http://perconik.fiit.stuba.sk/itgstream> [RANGE TRIPLES 10]
  WHERE { ?s ?p ?o }
  GROUP BY ?s
HAVING (?avg > 10);
```



Obrázok 4-1 Graf RDF trojíc udalosti prepnutia medzi súbormi vo vývojovom prostredí.

4.4. Dáta sledovaných projektov

Pre potreby našej práce použijeme dáta o viacerých softvérových projektoch vyvíjaných študentmi, poskytnuté prostredníctvom projektu PerConIK. V čase riešenia práce sme mali dostupné dáta aj z firemného prostredia, ale bolo by náročné diskutovať naše výsledky so sledovanými programátormi. Študentské projekty nám naopak umožnili lepšie vyhodnotiť výsledky, hoci na menšej vzorke dát, pretože diskusia so študentmi bola pre nás prístupnejšia. Príloha B obsahuje opis použitých projektov, ich veľkosť a rozsah v podobe času a množstva zaznamenaných dát. Uvedené projekty sú rôznej veľkosti – jednotlivci až tímy o 7 študentoch, pričom z nich sme vyhodnocovali dáta len jedného z členov. Aktivita programátorov bola na projektoch sledovaná v období jedného roka štúdia, kedy sa podarilo zaznamenať 152606 záznamov aktivít programátorov nad súbormi zdrojového kódu.

K dátam týchto projektov v podobe záznamov činností programátorov a zdrojových kódov pristupujeme prostredníctvom webových služieb infraštruktúry projektu PerConIK, alebo priamym dopytovaním jeho databáz. Použitú dokumentáciu k rozhraniam služieb a databázy uvádzame na priloženom elektronickom médiu (príloha H).

4.5. Diskusia k udržovateľnosti softvéru

Udržovateľnosť softvéru považujeme za jednu z najdôležitejších z pohľadu vyhodnotenia kvality softvéru. Udržovateľnosť vyhodnocuje úsilie potrebné pre opravu, úpravu alebo doplnenie funkcionality. Na udržovateľnosť softvéru má vplyv proces vývoja softvéru, ako pristupujeme k jeho samotnej údržbe a aké máme pri nej možnosti.

4.5.1. Štýl programovania a údržba softvéru

Prístup programátora k riešeniu problémov v implementácii vplýva na jeho budúcu údržbu. Napriek existencie konvencií pre organizáciu zdrojového kódu a rôznych odporúčaní, prístup programátora vždy závisí od jeho samotného a od situácie, v akej sa nachádza. Sledovaním postupu prác programátora pri plnení úlohy si môžeme všimnúť, že existuje rozdiel medzi programovaním novej funkcionality alebo udržiavania existujúcej implementácie. Ak programátor začína s implementáciou novej funkcionality, má väčšiu voľnosť práce a môže sa sebarealizovať pri riešení. V prípade, že programátor udržiava už existujúcu implementáciu (napr. opravuje chybu, alebo dopĺňa funkcionalitu), je zviazaný existujúcimi štruktúrami a prepojeniami v zdrojovom kóde. Ak potrebuje zmeniť časť zdrojového kódu, musí si byť vedomý prípadných dopadov jeho zmien. Jednoducho sa môže stať, že opravou chyby vytvorí dve ďalšie, pretože poruší inú funkcionalitu závisiacu od upravenej časti. Pri vytváraní novej funkcionality, alebo na začiatku vývoja nového projektu, toto riziko nie je vysoké alebo neexistuje, pretože štruktúry v zdrojovom kóde sa len vytvárajú.

Vznik závislostí v zdrojovom kóde samozrejme závisí aj od prístupu k jeho vytváraniu. Vývoj softvéru môže byť riadený, vopred analyzovaný a navrhnutý. V tom prípade má programátor k dispozícii napr. návrh architektúry a riadi sa ním pri implementácii. Podobne môže fungovať aj samotný skúsený programátor, ktorý pred implementovaním funkcionality vopred premyslí úlohu a hľadá použitie návrhových vzorov v jeho riešení. Opačný prístup je chaotickejšieho rázu, kedy programátor začne implementovať riešenie bez vážnejšieho premyslenia, zdrojový kód neorganizuje a snaží sa len docieliť výsledok v čo najkratšom čase. Po dosiahnutí výsledku môže pristúpiť k reorganizácii vytvoreného kódu do správnych štruktúr, odčlení funkcionalitu,

refaktoruje svoj zdrojový kód. Pokiaľ sa projekt vyvíja rýchlo, nie sú dostatočné zdroje, alebo je cieľom vytvoriť iba prototyp, k tejto refaktorizácii ani nemusí prísť.

Efektivita programátora, či už ide o implementáciu novej funkcionality alebo údržbu, závisí od skúseností a poznania existujúceho zdrojového kódu, ktorého sa jeho úloha týka. Programátor sa musí zorientovať v zdrojovom kóde, ktorý predtým nepoznal alebo aj zabudol. Ak nemá prístup k osobnej konzultácii a vysvetleniu riešenia od autora pôvodného zdrojového kódu, musí venovať čas jeho štúdiu a sledovať ako funguje, alebo študovať dokumentáciu. Štúdiom zdrojového kódu môže identifikovať miesta, ktoré riešia podobnú úlohu, inšpirovať sa pre svoje riešenie alebo nájsť odporúčaný spôsob v danom projekte. Príkladom môže byť registrovanie novej služby, ktorú aplikácia poskytuje na svojom rozhraní. Všetky služby sa registrujú rovnakým spôsobom a ich rozhranie je implementované podobne pre všetky služby.

Rýchlosť programátora pri plnení úlohy a kvalita jeho výsledkov samozrejme závisia aj od jeho skúseností s programovaním a od schopností myslieť analyticky. Pochopiteľne nemôžeme považovať profesionálneho programátora za rovnocenného so študentom informatiky, ktorý sa prvýkrát stretáva s prácou na tímových projektoch počas svojho štúdia. Štýly programovania môžeme zhrnúť vyjadrením aspektov, ktoré pri tom vplývajú na programátora:

- *Pôvod funkcionality, s ktorou pracuje:*
  - vytvára novú funkcionalitu, alebo
  - udržuje existujúcu implementáciu.
- *Znalosť zdrojového kódu:*
  - pozná zdrojový kód, alebo
  - je neznalý, potrebuje sa zorientovať.
- *Prístup programátora:*
  - organizovaný a premyslený prístup,
  - chaotický vývoj s následným refaktoringom, alebo
  - vývoj riadený testami.
- *Skúsenosť programátora:*
  - vo všeobecnosti s návrhom a vývojom softvéru, alebo
  - s danou technológiou.

Výsledok práce programátora závisí od kombinácie týchto aspektov a situácie (kontext), v ktorej sa nachádza (stres, časová tieseň). Vlastnosti vytvoreného zdrojového kódu nesúvisia len s ním samotným, ale majú pôvod v programátorovi, ktorý zdrojový kód vytvoril.

4.5.2. Závislosti súčiastok zdrojového kódu

Počas údržby softvéru sa programátori vracajú k práci. Pri úprave existujúceho kódu, ktorí vytvorili dávno, alebo je výsledkom iných programátorov, potrebujú študovať ako kód funguje, čo ovplyvňuje a aký ma dopad zavádzaná zmena. Závislosti jednotlivých súčiastok sú vhodným zdrojom pre identifikovanie týchto miest v zdrojovom kóde. Syntaktickou analýzou zdrojového kódu získavame explicitne vyjadrené závislosti. Tie však nereflektujú logické prepojenia tried alebo ich prepojenia počas vykonávania softvéru. Situáciu zhoršuje dekomponovanie súčiastok a programovanie ich aplikačných rozhraní. V súčasnej dobe nedokážeme identifikovať závislosti pre každý programovací jazyk, týka sa to najmä dynamicky-typovaných jazykov a používania heuristík pre odhad odkazov na súčiastky.

Identifikovanie závislostí je dôležité aj medzi konfiguračnými súbormi projektu alebo medzi webovými stránkami a zdrojovým kódom, ktorý upravuje ich obsah, čo v dnešnej dobe nedokážeme jednoznačne vykonať syntaktickou analýzou.

Riešením uvedených problémov je zaznamenanie aktivity programátora so súčiastkami zdrojového kódu, napr. otváranie a prepínanie sa medzi nimi pri ich vývoji alebo študovaní. Aktivita programátora môže odzrkadliť závislosti súčiastok, ktoré navyše nemusia byť medzi sebou prepojené na úrovni kódu.

4.6. Ciele práce

Na základe prístupu k zaznamenaným aktivitám programátorov infraštruktúrou vytvorenou v rámci projektu PerConIK sa nám otvára priestor pre návrh vlastných metód vyhodnocujúcich softvér, t.j. metrík softvérového produktu vychádzajúcich z aktivít programátora a kontextu vývoja softvéru. Jednou z možností je použitie metód dolovania v dátach, napr. klasifikácia, zhlukovanie alebo aj hľadanie asociačných pravidiel. Prehľad metód dolovania v dátach uvádzame v prílohe D. V rámci projektu PerConIK môžeme aplikovanie nových softvérových metrík rozumieť ako zdroj pre vznik informačných značiek vyššej úrovne, vyjadrujúcich vlastnosti zdrojového kódu. Keďže sú značky viditeľné pre používateľa (programátor, manažér), tí ich môžu spätne revidovať a vyhodnocovať.

Cieľom našej práce je prepojenie zaznamenaných aktivít programátora a kontextu vývoja softvéru s jeho vlastnosťami, s cieľom odhalenia aj nových informácií o zdrojovom kóde, neidentifikovateľných len jeho syntaktickou analýzou. Konkrétne dopĺňame existujúcu metódu identifikácie závislostí v zdrojovom kóde, ako príspevok k podpore udržovateľnosti softvéru. Pre hlbší prínos našej práce do projektu PerConIK môžu byť identifikované závislosti medzi softvérovými súčiastkami reprezentované aj informačnou značkou.

Kapitola 5

Metóda identifikácie skrytých závislostí v zdrojovom kóde

Sledovanie aktivít a kontextu programátorov nám umožňuje hľadať ich súvis s priestorom softvérových súčiastok, ich vzťahy a zaujímavé miesta, podobne ako metrikami obsahu zdrojového kódu. Identifikácia závislostí je metrikou vyhodnocujúcou vzťah dvojice softvérových súčiastok. Tradične identifikujeme závislosti syntaktickou analýzou, no v zdrojovom kóde sa nachádzajú mnohé ďalšie závislosti, ktoré nie sú jednoducho odvoditeľné.

Z aktivity programátora a kontextu vývoja softvéru implicitne vyplývajú závislosti v zdrojovom kóde, ktoré sú inak skryté medzi explicitne vyjadrenými závislosťami. Vychádzame pritom z predpokladu spoločného pre identifikáciu kontextu úlohy (Kersten & Murphy, 2006), t.j., že softvérové súčiastky, s ktorými programátor pracuje v blízkom čase navzájom súvisia s riešením jeho úlohy. V našej práci sme navrhli metódu pre rozšírenie priestoru identifikovaných závislostí v zdrojovom kóde o implicitné závislosti, ktoré vyplývajú zo zaznamenaných aktivít programátora.

Pre vyjadrenie našej metódy opíšeme softvérový projekt na úrovni zdrojového kódu a aktivít programátorov. Vychádzame pritom zo zavedenej terminológie pre hierarchiu softvérového produktu (kapitola 2.1.3). Nech U je množina programátorov softvérového projektu, S je množina softvérových súčiastok projektu zvolenej úrovne, O určuje množinu možných operácií nad súčiastkami a T vyjadruje množinu časov, potom:

- množinou C vyjadríme priestor verzií súčiastok projektu v čase:

$$C = S \times T$$

- množinou A vyjadríme priestor zaznamenaných aktivít programátorov projektu:

$$A = U \times S \times O \times T$$

- aktivitu programátora definujeme ako funkciu $activity$, t.j. činnosť programátora v čase vyjadrená operáciou nad konkrétnou softvérovou súčiastkou:

$$activity\colon U \times T \rightarrow S \times O$$

Z týchto označení budeme vychádzať v opise našej metódy.

5.1. Závislosti softvérových súčiastok

Informácie o závislostiach softvérových súčiastok používame v procesoch ich vývoja a údržby. Závislosti odrážajú prepojenia, ktoré existujú v zdrojovom kóde. Tie môžeme rozlíšiť:

- *syntaktické* – prepojenia softvérových súčiastok v zdrojovom kóde, ponúkajú programátorom možnosť analyzovať fungovanie zdrojového kódu jeho prehliadaním (Holmes & Notkin, 2011),
- *sémantické* – vyjadrujú logické prepojenia súčiastok pre splnenie zadanej úlohy.

Tým pádom môžu medzi súčiastkami existovať vzťahy aj vtedy, keď nie sú na úrovni kódu prepojené. Sú to skryté závislosti, ktoré vznikajú počas vykonávania softvéru, alebo opisujú význam práce programátora so súčiastkami v rovnakom čase, či počas plnenia podobnej úlohy. Ako príklad môžeme uviesť tieto situácie:

- Programátor číta zdrojový kód súčiastky od iného programátora, v ktorej sa vykonáva operácia použiteľná v jeho kóde. Programátor inšpirovaný študovaným kódom následne implementuje vlastný kód. Obe implementácie na seba v kóde neodkazujú, ale sú významovo previazané. Ak by sa zmenil pôvodný kód, môže byť vhodné aktualizovať aj programátorov kód.
- Programátor skopíruje časť kódu z inej súčiastky do vlastnej. Obe súčiastky sa podobajú vo fragmentoch zdrojového kódu, ale priamo na seba neodkazujú.
- Programátor implementuje súčiastku vychádzajúcu zo schémy pre serializáciu objektov do súboru, napr. XML. Výsledný kód závisí od použitej schémy, ale nie je s ňou previazaný.
- Programátor študuje súčiastky súvisiace s tou, v ktorej chce opraviť chybu. Programátor sa rozhodne ako najvhodnejšie upraviť uvažovanú súčiastku a následne, či si táto úprava vyžiada zmenu aj vo zvyšných súčiastkach.

Skryté závislosti softvérových súčiastok implicitne vyplývajú z činnosti programátora, preto rozlišujeme:

- *explicitné závislosti* – prepojenia súčiastok na úrovni zdrojového kódu, sú získateľné analýzou ich obsahu a organizácie (tradičný graf závislostí).
- *implicitné závislosti* – prepojenia súčiastok vychádzajúce z činnosti programátora počas práce so zdrojovým kódom.

Závislosť d (podľa angl. dependency) dvoch súčiastok s_1 a s_2 definujeme ako štvoricu $d = (s_1, s_2, w, t)$ s váhou w a časovou pečiatkou výskytu t. Potom D vyjadruje priestor všetkých závislostí d medzi súčiastkami zdrojového kódu:

$$D = S \times S \times T \times \mathbb{R}$$

$$D = \{ d \mid \forall s_1 \in S, \exists s_2 \in S, t \in T, w \in \mathbb{R}: d = (s_1, s_2, w, t) \}$$

Priestor závislostí rozlišujeme na priestory explicitných a implicitných závislostí (D_{exp} a D_{imp}, resp. d_{exp} a d_{imp}). Pomocou funkcií $depend_{exp}$ a $depend_{imp}$ definujeme produkciu týchto závislostí z množín vybraných verzií súčiastok alebo aktivít pre vybranú súčiastku v čase:

$$depend_{exp}: S \times C^n \times T \rightarrow D_{exp}$$

$$depend_{imp}: S \times A^n \times T \rightarrow D_{imp}$$

Explicitné a implicitné závislosti vznikajú z rôznych zdrojov údajov a spoločne vyjadrujú prepojenia medzi súčiastkami. Z dôvodu rozdielneho pôvodu týchto typov závislostí ich tak nemôžeme zamieňať. Zároveň si však uvedomujeme, že tieto dva typy sa navzájom nevylučujú.

5.1.1. Váhovanie implicitných závislostí

Výsledné váhy závislostí sú určené podľa implementácie príslušnej funkcie $depend$. V prípade tradičných explicitných závislostí môžeme priradiť každému z identifikovaných výrazov v zdrojovom kóde súčiastky váhu *1* (referencia, volanie, dedenie). Implicitné závislosti môžeme identifikovať z rôznych aktivít programátora a podľa ich typu im priradiť váhu, napr.:

- prepnutie medzi súčiastkami vo vývojovom prostredí – váha určujúca význam prepnutia,
- kopírovanie zdrojového kódu medzi súčiastkami – váha množstva skopírovaného kódu.

Konkrétny príklad určenia váh implicitných závislostí uvádzame v kapitole 6 pri realizácii metódy.

5.1.2. Časová platnosť implicitných závislostí

Podobne ako sa môžu úpravami kódu v čase meniť explicitné závislosti súčiastok, tak aj implicitné závislosti postupom času nadobúdajú alebo strácajú na význame (Coman & Sillitti, 2008; Fritz, et al., 2007). Implicitné závislosti vyjadrujú záujem programátora, jeho potrebu iných súčiastok pri plnení úlohy, alebo aj odrážať kontext práce programátora so závislými súčiastkami. Preto definujeme funkciu platnosti závislostí $validity$ zohľadňujúcu čas kedy boli súčiastky prepojené voči zvolenému času:

$$validity_{exp}: D_{exp} \times T \rightarrow \{0,1\}$$

$$validity_{imp}: D_{imp} \times T \rightarrow \langle 0,1 \rangle$$

Explicitné závislosti vyplývajú z prepojení v zdrojovom kóde, t.j. platia alebo neplatia v danom čase. Obor hodnôt funkcie platnosti implicitných závislostí sme ale stanovili ako interval, pretože ich význam vyjadrujeme starnutím v čase (Ebbinghaus, 1885/1913). Funkciu starnutia môžeme zamieňať podľa potreby použitia aj podľa vyhodnocovaných implicitných závislostí. Ako základnú funkciu sme zvolili nasledujúcu podľa (White, 2001):

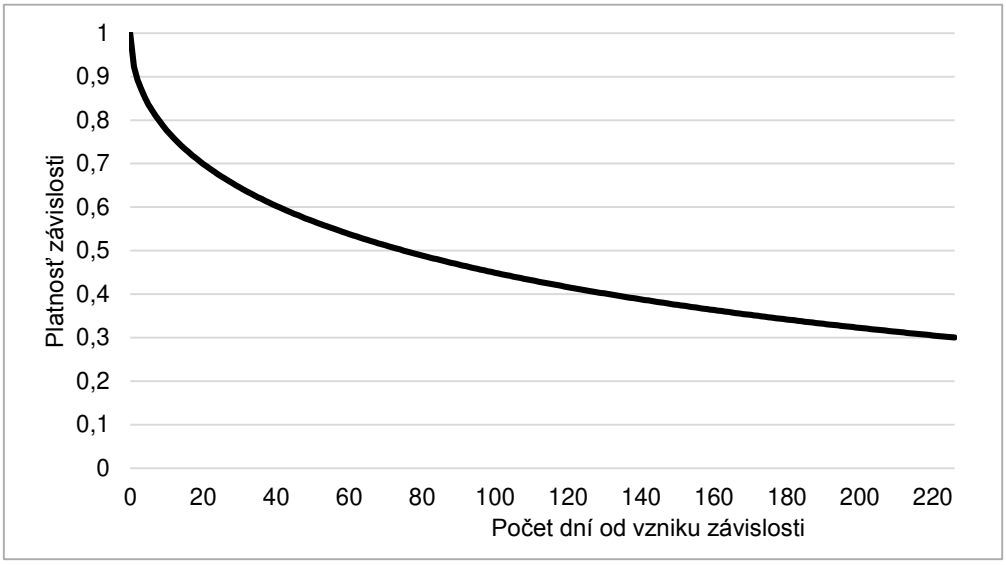
$$y = ae^{(-b\sqrt{t})}$$

Aby sme vyjadrili pomalé zabúdanie na intervale $\langle 0,1 \rangle$, parametrom sme nastavili hodnoty $a = 1$ a $b = 0.08$. Obrázok 5-1 znázorňuje priebeh funkcie, definičným oborom je počet uplynulých aktívnych dní od vzniku závislosti, t.j. dní, v rámci ktorých existujú závislosti (vynechávame prázdne dni bez aktivity).

5.2. Kontexty softvérových súčiastok

V súvislosti so softvérovým projektom používame pojem *kontext* podľa toho, čo sledujeme:

- kontext programátora – sledujeme programátora, jeho stav a situáciu v okolitom prostredí,
- kontext úlohy – sledujeme riešenie úlohy programátorom, najmä použité softvérové súčiastky.



Obrázok 5-1 Priebeh funkcie zabúdania pre určenie platnosti implicitných závislostí.

Pomocou identifikovaných závislostí dokážeme pre zvolenú súčiastku vybrať okolité súčiastky v zdrojovom kóde, ktoré s ňou implicitne alebo explicitne súvisia. Množinu okolitých súčiastok potom rozumieme ako kontext sledovanej súčiastky, rozlišujeme:

- *explicitný kontext* – súčiastky prepojené so sledovanou súčiastkou v zdrojovom kóde,
- *implicitný kontext* – súčiastky, s ktorými programátor pracoval pri sledovanej súčiastke.

Kontext sledovanej súčiastky s je podmnožinou všetkých súčiastok S uvažovaného softvérového projektu, označujeme ho DC_s (podľa angl. dependency context):

$$DC_s \subseteq S \setminus \{s\}$$

Keďže rozlišujeme medzi implicitným a explicitným kontextom softvérovej súčiastky, ich produkciu pre zvolený čas t vyjadrujeme zvlášť funkciami $context_{exp}$ a $context_{imp}$:

$$context_{exp}: D_{exp}^{n} \times S \times T \rightarrow DC_{s,exp}^{m}, n \geq m$$

$$context_{exp}(s,t) = \{x \mid \forall x \in S, \exists u \in T, \exists d \in D_{exp}: (d = (s,x,w,u) \vee d = (x,s,w,u)) \wedge w > 0 \wedge validity_{exp}(d,t) = 1\}$$

$$context_{imp}: D_{imp}^{n} \times S \times T \rightarrow DC_{s,imp}^{m}, n \geq m$$

$$context_{imp}(s,t) = \{x \mid \forall x \in S, \exists u \in T, \exists d \in D_{imp}: (d = (s,x,w,u) \vee d = (x,s,w,u)) \wedge w > 0 \wedge validity_{imp}(d,t) > 0\}$$

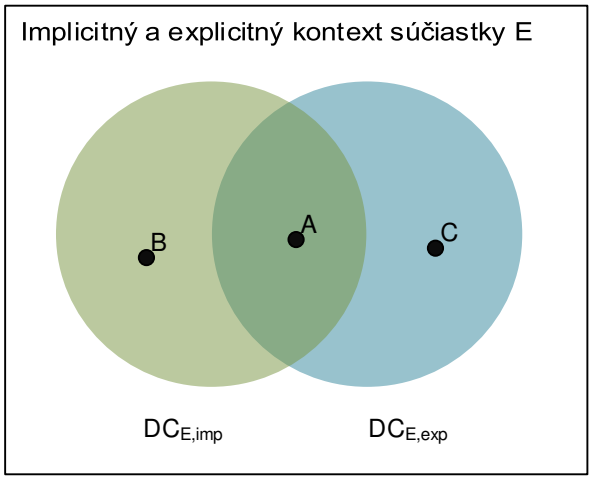
5.2.1. Ohraničenie kontextov softvérových súčiastok

Kontext softvérovej súčiastky určujeme zo závislostí, v ktorých vystupuje, preto ho rozumieme ako priamy kontext. Ďalej však môžeme uvažovať aj súčiastky patriace do priamych kontextov súčiastok, ktoré sú v priamom kontexte pôvodne sledovanej súčiastky, t.j. sú vzdialené od pôvodnej súčiastky na 2 alebo viac krokov. Tieto súčiastky už označujeme ako nepriamy kontext sledovanej súčiastky, sú to nepriame a tranzitívne prepojenia medzi súčiastkami. Rozlišujeme kontexty:

- *priamy kontext* – priamo zviazané súčiastky s danou súčiastkou (krok dĺžky 1),
- *nepriamy kontext* – tvorený súčiastkami, ktoré sú zviazané s danou súčiastkou cez ďalšie súčiastky (krok väčší ako 1).

5.2.2. Vzťahy typov závislostí a kontextov softvérových súčiastok

V definícií závislostí i kontextov rozlišujeme medzi implicitným a explicitným typom, ktoré nemôžeme zamieňať. V skutočnosti však môžu byť dve súčiastky závislé obidvoma typmi vzťahov v danom čase, a tak tieto množiny nie sú disjunktné. Obrázok 5-2 uvádza príklad výskytu súčiastky A v implicitnom aj explicitnom kontexte sledovanej súčiastky E. Súčiastky B a C však patria len do jedného z kontextov.



Obrázok 5-2 Implicitný a explicitný kontext súčiastky E môže mať nenulový prienik.

5.2.3. Hierarchický kontext softvérovej súčiastky

Implicitné a explicitné kontexty však nie sú jediné, ktoré môžeme pre softvérovú súčiastku určiť. Pre úplnosť uvádzame aj hierarchický kontext, aj keď ho môžeme považovať za špeciálny prípad explicitného kontextu. Zdrojový kód je organizovaný do hierarchie, čím sú explicitne vyjadrené hierarchické vzťahy súčiastok. Hierarchický kontext môžeme rozlišovať ako:

- *vertikálny* – súčiastky, ktoré sú nadradené a podradené danej súčiastke,
- *horizontálny* – súčiastky na rovnakej úrovni v zdrojovom kóde spadajúce pod spoločnú nadradenú súčiastku.

Vychádzajúc z klasifikácie softvérových súčiastok (kapitola 2.1.3) môžeme uviesť tento príklad:

- vertikálny hierarchický kontext pre triedu objektov v zdrojovom kóde je jej zaradenie do balíka a jej prvky, t.j. metódy a atribúty.
- horizontálny hierarchický kontext je zaradenie tried do spoločného balíku alebo aj súboru.

5.3. Graf implicitných a explicitných závislostí

Identifikovaním závislostí medzi súčiastkami sa pozeráme na zdrojový kód ako na graf $G(V, E)$, ktorého vrcholmi V sú softvérové súčiastky a hrany E reprezentujú identifikované závislosti medzi týmito súčiastkami. V našom prípade rozširujeme tradičný graf o implicitné závislosti, preto rozlišujeme množinu hrán na implicitné a explicitné hrany:

$$E = E_{exp} \cup E_{imp}$$

Množiny hrán E_{exp} a E_{imp} definujeme zobrazením množín závislostí medzi súčiastkami D_{exp} a D_{imp}, kedy agregujeme jednotlivé závislosti funkciou $edge$ do spoločných hrán:

$$E_{exp} = S \times S \times \mathbb{R}\,, \quad E_{imp} = S \times S \times \mathbb{R}$$

$$edge : (S \times S \times T \times \mathbb{R})^n \rightarrow S \times S \times \mathbb{R}$$

$$E_{exp} = \{e_{exp} \mid \forall s_1, s_2 \in S, w \in \mathbb{R}, p > 0 : e_{exp} = edge(D_{s_1,s_2,exp}) = (s_1, s_2, p)\}$$

$$E_{imp} = \{e_{imp} \mid \forall s_1, s_2 \in S, w \in \mathbb{R}, p > 0 : e_{imp} = edge(D_{s_1,s_2,imp}) = (s_1, s_2, p)\}$$

Pomocou $D_{s_1,s_2,exp}$ a $D_{s_1,s_2,imp}$ označujeme tie podmnožiny množín závislostí D_{exp} a D_{imp}, ktorých prvkami sú len závislosti súčiastky s_1 na s_2. Funkcia $edge$ agreguje všetky závislosti medzi dvomi súčiastkami do spoločnej hrany s váhou p. Váhu hrany určí funkcia na základe dôležitosti závislostí, prípadne aj ich časových platností (použitie funkcie $validity$).

Pomocou množín hrán definujeme množinu vrcholov grafu V ako výber tých softvérových súčiastok, medzi ktorými existuje hrana závislosti:

$$V \subseteq S\,, \qquad V = \{v \mid \forall v \in S,\ \exists x \in S,\ \exists e \in E,\ p > 0 : e = (v, x, p) \vee e = (x, v, p)\}$$

Výsledný graf softvérových súčiastok obsahuje implicitné aj explicitné závislosti. V prípade potreby môžeme graf ohraničiť podľa rôznych kritérií, čo sa odzrkadlí na ohraničení tvorby závislosti a hrán, napr. závislosti podľa projektov, programátorov, súčiastok, alebo aj času.

5.4. Použitie implicitných závislostí

Pomocou implicitných závislostí rozširujeme množinu súčiastok, ktoré môžu súvisieť so sledovanou súčiastkou. To môže výrazne podporiť procesy vývoja a údržby softvéru, napr. pri dopĺňaní alebo zmene funkcionality, oprave chýb, alebo vrátení sa k práci. Implicitné závislosti sú použiteľné aj počas manažovania projektu a jeho vyhodnocovania. Preto ich realizáciu navrhujeme uplatniť pri:

- rozšírení vizualizácie grafu závislostí o implicitné závislosti – ako nástroj pre prehliadanie a analýzu priestoru súvisiacich softvérových súčiastok,
- ponúknutí zoznamu súvisiacich súčiastok z kontextu zvolenej súčiastky.

Vychádzajúc z prelínania implicitných a explicitných kontextov súčiastok môžeme implicitné závislosti použiť aj ako náhradu explicitných závislostí. Tie v dnešnej dobe nedokážeme vždy jednoznačne identifikovať, hlavne pri dynamicky-typovaných programovacích jazykoch (*JavaScript*, *PHP*, *Ruby*, atď.), alebo aj pri ich kombinácii so staticky-typovanými jazykmi či značkovacími jazykmi. Identifikovanie explicitných závislostí v týchto situáciách je založené na rôznych heuristikách a môže byť výpočtovo náročné. Implicitné závislosti identifikujeme zo zaznamenaných aktivít programátora a ich vyhodnotenie je tak jednoduchšie.

Z možností použitia implicitných závislostí pri vývoji a údržbe softvéru odvádzame tieto hypotézy:

> *H1: Implicitné závislosti odrážajú explicitné závislosti súčiastok, s ktorými programátor pracoval počas plnenia úlohy.*

Programátor pri vývoji alebo údržbe softvéru postupne pracuje so súčiastkami prepojenými na úrovni zdrojového kódu, napr. pri navigácii, ale aj pri postupnom vývoji súčiastok závislých na sebe. Tým pádom programátorova aktivita v zdrojovom kóde odráža explicitné závislosti súčiastok.

> *H2: Implicitné závislosti obohacujú graf explicitných závislostí o nové prepojenia, ktoré sú použiteľné pri vývoji a údržbe zdrojového kódu.*

V mnohých prípadoch bývajú súčiastky na úrovni kódu veľmi slabo prepojené, alebo napr. závisia od konfiguračných súborov. To identifikujeme z aktivít programátora nezávisle od obsahu zdrojového kódu, a tak graf závislostí obohacujeme aj o voľné závislosti súčiastok a ostatných zdrojových súborov.

Vizualizácia grafu závislostí

Vizualizácia a možnosť interakcie s grafom identifikovaných závislostí je pre programátora vhodný spôsob pre navigáciu medzi súčiastkami porozumenie ich prepojení. Pri veľkom počte vrcholov a hrán však môže byť graf značne neprehľadný. Pre odstránenie tohto nedostatku navrhujeme umožniť používateľovi obmedziť zobrazenie grafu pomocou:

- časového okna pre tvorbu závislostí,
- výberu typu zobrazovaných hrán – implicitné alebo explicitné,
- nastavenia prahu váh zobrazenia hrán,
- výberu programátorov, ktorí sú uvažovaní pri identifikácii závislostí.

Zároveň sme navrhli rozšíriť graf o ďalšie údaje, napr. počet elementárnych súčiastok a ich zložitosť, a tak grafické zobrazenie prvkov grafu zodpovedá nasledujúcim bodom:

- Vrcholy grafu predstavujú softvérové súčiastky zvolenej úrovne granularity:
  - obsah vrcholu: názov súčiastky a počet elementárnych súčiastok v nej obsiahnutých,
  - farba vrcholu: vybraná metrika súvisiaca s kvalitou súčiastky (napr. zložitosť),
  - veľkosť vrcholu: vybraná metrika súvisiaca s rozsahom súčiastky (napr. počet riadkov).
- Hrany grafu predstavujú závislosti medzi súčiastkami:
  - rozlíšenie implicitných a explicitných hrán,
  - ohodnotenie a hrúbka hrany: počet a agregovaná váha závislostí medzi súčiastkami,
  - orientácia hrany: smer závislosti medzi súčiastkami.

Vizualizovaním grafu oboch typov závislostí umožníme programátorovi navigovať sa v zdrojovom kóde nie len využitím syntaktických informácií o zobrazovaných súčiastkach. Zahrnutím implicitných závislostí totiž zväčšujeme priestor pre identifikáciu sémantických vzťahov medzi

súčiastkami. Príkladom môže byť situácia, kedy by programátor identifikoval problém v zdrojovom kóde súčiastky. Keďže graf závislostí vychádza z aktivít programátora, jeho použitím môže identifikovať aj ostatné problematické miesta od pôvodného programátora.

Sémantické vzťahy nevyplývajú jednoznačne z implicitných či explicitných závislostí. Namiesto náročného vyhodnocovania vzťahov všetkých súčiastok naprieč celým zdrojovým kódom, aspoň rozširujeme inak ohraničený priestor explicitnými závislosťami o ďalšie závislosti medzi súčiastkami, ktoré môžu byť použité pre získanie sémantických vzťahov.

Ponúknutie implicitného kontextu zvolenej softvérovej súčiastky

Okrem vizualizácie grafu môže v mnohých prípadoch programátorovi postačovať zobrazenie kontextu zvolenej súčiastky v podobe usporiadaného zoznamu. Implicitné závislosti môžu pri údržbe súčiastky programátorovi odhaliť miesta, v ktorých by mal skontrolovať vplyv jeho úprav, napr.:

- Programátor, ktorý sa vracia k práci na súčiastke po dlhšej prestávke má záujem o ďalšie súčiastky, ktoré s ňou súvisia. Ide o súčiastky, s ktorými predtým pracoval.
- Programátor preberajúci úlohu alebo opravujúci súčiastku, ktorej nebol pôvodným autorom, potrebuje získať prehľad o jej vývoji.

Získanie zoznamu implicitne závislých súčiastok je vhodné realizovať ako rozšírenie do vývojového prostredia, čím umožníme rýchly a jednoduchý prístup o jeho dopytovanie a okamžité použitie.

5.5. Diskusia

V našej práci sa zameriavame na prínos softvérovej metriky vychádzajúcej z aktivít programátora a kontextu vývoja softvéru ako ďalšieho zdroja informácií o zdrojovom kóde, než len jeho obsah. Počas vývoja a údržby softvéru programátori používajú závislosti v zdrojovom kóde, ktorých identifikácia je metrikou vyhodnotenia váh orientovaných prepojení dvojíc softvérových súčiastok.

V predstavenej metóde identifikovania závislostí v zdrojovom kóde na základe aktivity programátora sme definovali a rozšírili existujúci graf o typ implicitných závislostí. Programátor sa počas práce na úlohe naviguje v priestore zdrojového kódu a zasahuje do neho, a tak implicitne vyjadruje prepojenia súčiastok. Tradičné explicitné závislosti vyhodnocujeme syntaktickou analýzou zdrojového kódu. Tá však nie je vždy realizovateľná alebo úplná, napr. ako sme uviedli pri dynamicky-typovaných jazykoch. Tým, že implicitné závislosti vychádzajú z aktivít programátora, máme možnosť v grafe závislostí zachytiť aj tieto logické prepojenia súčiastok:

- vplyv upravovanej súčiastky na iné súčiastky, ktoré nie sú explicitne závislé,
- prevzatie riešenia alebo jeho inšpirácia z inej súčiastky, alebo aj
- závislosti konfiguračných súborov a iných artefaktov softvérového projektu.

Podobne ako obmedzenia explicitných závislostí vyplývajú z procesu ich identifikácie, identifikovanie implicitných závislostí závisí od dostupných záznamov o aktivite programátorov vo vývojovom prostredí. Implicitné závislosti dokážeme identifikovať len z aktivít medzi tými súčiastkami, medzi ktorými sme zaznamenali aktivitu programátorov, a zároveň tieto aktivity boli časté a signifikantné, t.j. budú mať dostatočnú váhu pri identifikácii. Ďalšou nevýhodou implicitných závislostí je ich nepresnosť podľa obsahu zaznamenaných aktivít a úrovne súčiastok, na ktorých boli aktivity zaznamenané, napr.:

- pokiaľ sa programátor často pohybuje medzi súčiastkami, ktoré spolu nesúvisia,
- chyby v záznamoch aktivít, ktoré programátor vykonal, alebo aj nevykonal,
- aktivity sú zaznamenané na úrovni súborov zdrojového kódu.

V prípade použitia aktivít na úrovni súborov zdrojového kódu neidentifikujeme závislosti medzi súčiastkami nižšej úrovne (triedy a ich metódy), ako je tomu pri explicitných závislostiach.

Použitie našej metódy sme identifikovali v procesoch vývoja a údržby softvéru, napr. realizovaním ako:

- rozšírenia vizualizácie grafu závislostí,
- ponúknutie zoznamu implicitne závislých súčiastok pre zvolenú súčiastku, alebo
- ich využitie ako náhrady explicitných závislostí.

Definícia grafu implicitných závislostí a ich identifikácie nie je závislá od použitia s jedným konkrétnym zdrojom dát, je adaptovateľná podľa potreby jej aplikácie. Ohraničenia metódy však vyplývajú z povahy použitých dát. Implicitnú spätnú väzbu je náročné interpretovať, keďže pracujeme len s dátami činnosti programátorov a nepoznáme ich vlastné dôvody, ktoré ich k aktivite viedli. Ďalším rizikom úspešnosti našej metódy je množstvo dát, ktoré použijeme pre identifikáciu závislostí. Príkladom je aj sledovanie študentských projektov, ktoré sa dotklo našej práce. Počet tímov a študentov, ktorí boli ochotní sa sledovať záviselo od ich motivácie a podpory pedagogickými vedúcimi. Bolo náročné prelomiť ich vlastné zábrany a motivovať ich k pomoci výskumu, v konečnom dôsledku pomoci iným študentom a aj im samým.

Kapitola 6

Overenie metódy

Identifikáciu skrytých závislostí v zdrojovom kóde sme experimentálne overovali s predpokladom doplnenia alebo nahradenia explicitných závislostí implicitnými. Navrhnutú metódu sme implementovali v podobe knižníc a prototypov potrebných pre vykonanie našich dvoch experimentov. Dôležitou časťou overenia je použitie infraštruktúry projektu PerConIK ako platformy pre prístup k aktivitám programátorov a softvérovým projektom, na ktorých pracovali. Zamerali sme sa na využitie dát zo študentských projektov, ktorých opis je uvedený v prílohe B. V čase riešenia práce sme nemali dostatočne voľný prístup k dátam z firemného prostredia a možnosť diskusie s ich programátormi. Príloha A obsahuje technickú dokumentáciu riešenia.

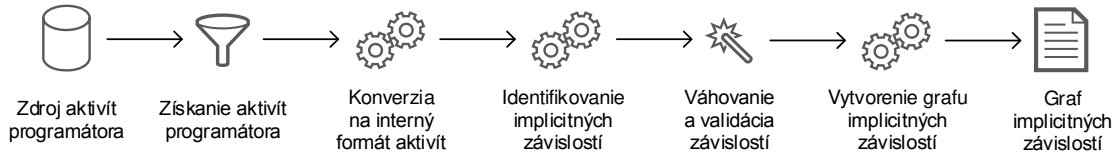
6.1. Realizácia metódy identifikácie implicitných závislostí

Implicitné závislosti v zdrojovom kóde identifikujeme z aktivít programátora, ktoré sú na nízkej úrovni.
Navrhnutý graf závislostí sme adaptovali na dáta zaznamenanej činnosti programátorov vo vývojovom prostredí na úrovni súborov zdrojového kódu poskytnuté infraštruktúrou projektu PerConIK. Neodhalíme tak závislosti na úrovni metód, ale častou konvenciou v programovacích jazykoch *C#* a *Java* je však umiestňovanie jednotiek zdrojového kódu do vlastných súborov, napr. trieda, rozhranie.

Identifikáciu implicitných závislostí a vyhodnotenie výsledného grafu závislostí zo záznamov aktivít na nízkej úrovni realizujeme ako viackrokový proces (Obrázok 6-1):

1. Výber a spracovanie dát so záznamami aktivít programátora.
2. Identifikácia implicitných závislostí rekonštrukciou aktivít vo vývojovom prostredí.
3. Agregácia závislostí do hrán grafu závislostí – vytvorenie grafu závislostí.

Technickú dokumentáciu realizácie uvádzame v prílohe A.



Obrázok 6-1 Proces identifikácie implicitných závislostí a vytvorenie grafu závislostí z aktivít programátora.

6.1.1. Záznamy aktivity programátora

Z dostupných typov aktivít programátora vo vývojovom prostredí a mimo neho sme pre realizáciu našej metódy identifikovali použiť nasledujúce operácie nad súbormi:

- otvorenie nového súboru,
- zatvorenie súboru,
- prepnutie medzi súbormi,
- skopírovanie kódu medzi súbormi.

Vymenovaním týchto typov operácií sme určili podmnožinu O' všetkých zaznamenávaných operácií nad súbormi $O_{PerConIK}$ infraštruktúrou projektu PerConIK:

$$O' \subseteq O_{PerConIK}$$

$$O' = \{open, close, switchTo, copyPaste\}$$

Ďalej uvažujeme ako množinu operácií pre graf závislostí práve O'. Dôvodom jej zmenšenia je význam pre určenie implicitných závislostí. Napr. aktivity označovania zdrojového kódu, vyhľadávanie výrazu v súbore, alebo aj práca v iných aplikáciách pre nás nemajú uplatnenie z pohľadu identifikovania implicitných závislostí.

Okrem práce so súbormi nás zaujímajú aj tieto aktivity vo vývojovom prostredí:

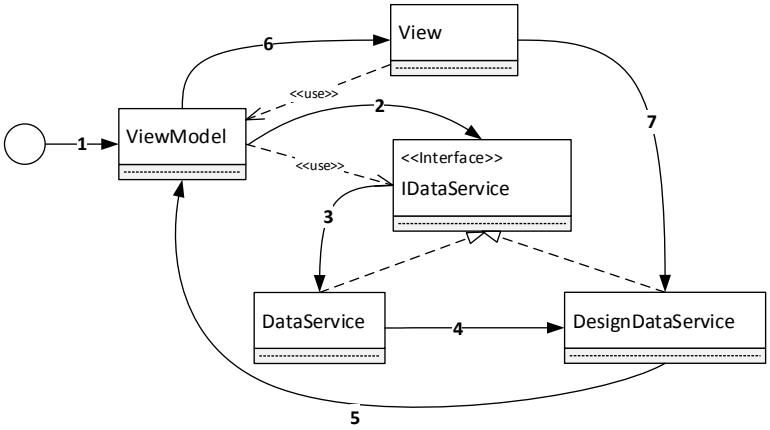
- prepínanie stavov vývojového prostredia *I* – vývoj, kompilácia a krokovanie vykonávania,
- odovzdávanie výsledkov práce – zoznam odovzdaných súborov.

$$I = \{design, build, debug\}$$

Príklad situácie práce programátora a vzniku sledovaných operácií so súbormi ukážeme na jeho práci doplnenia funkcionality do používateľského rozhrania aplikácie. Obrázok 6-2 znázorňuje situáciu očíslovanými prepojeniami v diagrame tried:

1. Programátor upravuje triedu prepájajúcu vrstvu používateľského rozhrania a aplikačnú vrstvu aplikácie (trieda *ViewModel*).
2. Programátor otvorí kód rozhrania dátovej služby (rozhranie *IDataService*), ktoré používa *ViewModel*. Programátor upraví rozhranie a následne musí upraviť aj jeho implementácie. Rozhranie je implementované dvomi triedami – skutočná implementácia a falošná implementácia, ktorá sa používa pri vývoji používateľského rozhrania.
3. Programátor upraví skutočnú implementáciu dátovej služby (trieda *DataService*).
4. Programátor sa vráti k rozhraniu a upraví falošnú implementáciu *DesignDataService*.
5. Programátor sa vráti k triede *ViewModel* a potom upraví triedu *View*.
6. Pri úprave triedy *View* programátor zistí, že falošnú implementáciu dátovej časti potrebuje ešte upraviť, a preto sa k nej vráti.
7. Programátor upraví falošnú implementáciu (trieda *DesignDataService*) a vráti sa k rozhraniu (trieda *View*).

V zdrojovom kóde neexistuje medzi triedami *View* a *DesignDataService* priama explicitná závislosť. Z tohto príkladu však vidíme, že programátor implicitne vyjadril ich závislosť prechodmi medzi nimi.



Obrázok 6-2 Ilustrácia sekvencie aktivít programátora na úprave používateľského rozhrania aplikácie.

6.1.2. Rekonštrukcia aktivity programátora vo vývojovom prostredí

Vstupom našej metódy sú aktivity programátora, ktoré boli zaznamenané vo vývojovom prostredí. Z dôvodu nízkej úrovne záznamov, kedy pre operácie poznáme iba cieľový súbor, nie zdrojový, a zároveň nás zaujíma aj stav vývojového prostredia počas aktivity, modelujeme vývojové prostredie a rekonštruujeme v ňom programátorove aktivity. Vývojové prostredie každého programátora, ktorého aktivity spracúvame, opisujeme pomocou:

- otvorený softvérový projekt,
- zásobník otvorených súborov zdrojového kódu (zoradený podľa poradia ich otvárania),
- aktuálne otvorený súbor,
- zásobník obsahov schránky kopírovania (zoradený podľa času),
- stav vývojového prostredia.

Postupným spracovaním záznamov aktivít vytvárame inštancie týchto dvoch špecializácií implicitných závislostí medzi súbormi d_{imp}, ktoré nastali za stavu prostredia ($ide \in I$):

- *časové závislosti* $d_{imp,time}$ – orientované prepnutie medzi dvojicou súborov s dĺžkou stráveného času v cieľovom súbore (časové okno $window$),
- *obsahové závislosti* $d_{imp,content}$ – orientované kopírovanie obsahu ($content$) medzi súbormi.

$$d_{imp,time} = (s_1, s_2, w, t, window, ide)$$

$$d_{imp,content} = (s_1, s_2, w, t, content, ide)$$

Zo záznamov odovzdávania výsledkov práce ďalej získame implicitné závislosti $d_{imp,commit}$ medzi každou dvojicou súborov v množine odovzdaných súborov, kde $count$ je počet všetkých odovzdaných súborov:

$$d_{imp,commit} = (s_1, s_2, w, t, count)$$

Použitím dostupných dát z projektu PerConIK sme takto identifikovali tri špecializované typy implicitných závislostí, ktoré sú výstupom kroku rekonštrukcie aktivít programátora. Ukážka 6-1 uvádza algoritmus identifikácie časových implicitných závislostí.

Ukážka 6-1 Pseudokód identifikácie časových implicitných závislostí z množiny aktivít.

```
Identify-Time-Dependencies(Activities : A) : D_imp
1.  Dps ← ∅              // identifikované závislosti
2.  Files ← ∅            // zásobník otvorených súborov
3.  dp ← NIL             // posledná časová implicitná závislosť
4.  file ← NIL           // aktuálne otvorený súbor
5.  for each a ∈ Activities                                  // a = (target, operation, t, ide)
6.  do if (a.operation = open ∨ switchTo) ∧ file ≠ a.target   // operation ∈ O'
7.     then if dp ≠ NIL
8.          then dp.window ← a.t − dp.t      // určí časové okno poslednej závislosti
9.          if file ≠ NIL
10.         then dp ← (file, a.target, a.t, 0, a.ide)        // d_imp,time = (s1, s2, t, window, ide)
11.              Dps ← Dps ∪ dp
12.         end
13.         file ← a.target
14.         Files ← Push(Files, file)        // uloží súbor na vrch zásobníka
15.    else-if a.operation = close
16.    then Docs ← Docs \ {a.target}        // súbor bol zavretý
17.         if dp ≠ NIL ∧ dp.s1 = a.target
18.         then dp.window ← a.t − dp.t
19.              dp ← NIL
20.         end
21.         file ← Top(Docs)                 // vráti najvrchnejší súbor zásobníka
22.    end
23. end
24. return Dps
```

6.1.3. Váhovanie implicitných závislostí

Váhu špecializovaných implicitných závislostí určujeme zvlášť pre každý typ funkciami $weight$:

$$weight : D_{imp} \rightarrow \mathbb{R}$$

- $weight_{time}$ – váhu časovej implicitnej závislosti určujeme ako význam programátorovho prepnutia na základe času stráveného v cieľovom súbore (Obrázok 6-3 znázorňuje priebeh funkcie). Hodnota $window$ a parametre a, b a c musia byť v rovnakej časovej jednotke:

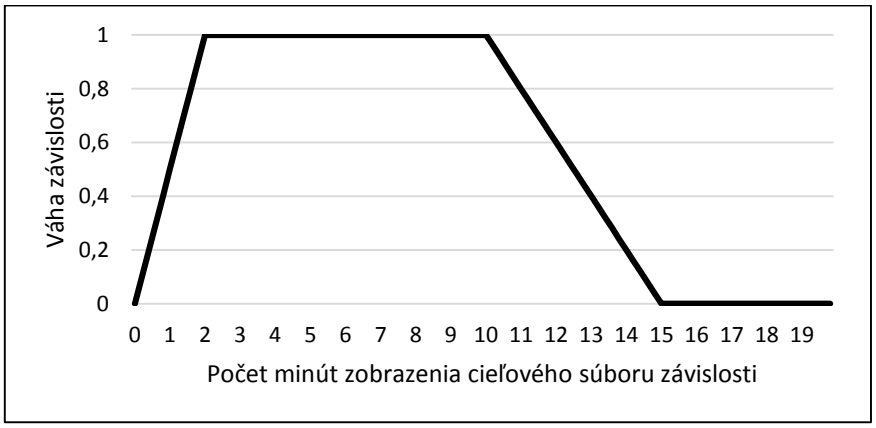
$$weight_{time}(d_{imp,time}) = \begin{cases} \dfrac{window}{a}, & window \leq a \\ 1, & window > a \wedge window \leq b \\ \dfrac{c - window}{c - b}, & window > b \wedge window \leq c \\ 0, & window > c \end{cases}$$

- $weight_{content}$ – váhu obsahovej implicitnej závislosti môžeme určiť podľa množstva kopírovaného kódu. Každú obsahovú závislosť sme však zvolili ohodnotiť váhou *1*, pretože bez analýzy kopírovaného obsahu nedokážeme určiť jeho význam.

$$weight_{content}(d_{imp,content}) = 1$$

- $weight_{commit}$ – váhu určujeme rovnomerne pre každú dvojicu súborov z odovzdania.

$$weight_{commit}(d_{imp,commit}) = \frac{1}{count}$$



Obrázok 6-3 Priebeh funkcie váhovania časových implicitných závislostí pri nastavení parametrov $a = 2$, $b = 10$ a $c = 15$ *minút*.

Pri voľbe priebehu váhovania časových implicitných závislostí sme vychádzali z vlastných skúseností pri programovaní. Krátke časové okno môže znamenať, že sa programátor pomýlil pri prepnutí. Naopak veľmi dlhé časové okno naznačuje, že pôvodné prepnutie do súboru nemalo veľký význam, pretože programátor stále pracuje s cieľovým súborom. Nízkou váhou pri dlhom časovom okne zároveň eliminujeme chybné situácie odvodené z aktivít programátora. Parametre funkcie a, b a c sme volili zvlášť podľa experimentov.

6.1.4. Vytvorenie grafu implicitných závislostí

Identifikované závislosti uchovávame aby sme sa vyhli ich opakovanému identifikovaniu. Potom podľa potrieb našich experimentov alebo používateľa dokážeme vyberať a filtrovať závislosti, napr. podľa programátorov, časového okna alebo požadovanej množiny súborov. Vybrané závislosti agregujeme do spoločných hrán e_{imp} medzi súbormi funkciou $edge$, a tak zostrojujeme graf implicitných závislostí. Váhu výslednej hrany závislosti p zo závislostí medzi súčiastkami s_1 a s_2 potom určujeme podľa potreby ako:

- sumu váh jednotlivých implicitných závislostí, t.j.:

$$p = \sum_{d \in D_{imp,s_1,s_2}} w$$

- sumu validovaných váh voči zvolenému časovému bodu t':

$$p = \sum_{d \in D_{imp,s_1,s_2}} validity(t, t')\, w$$

Výsledný graf implicitných závislostí s vrcholmi reprezentujúcimi súbory zdrojového kódu sme ďalej vizualizovali alebo vyhodnocovali podľa potrieb overenia.

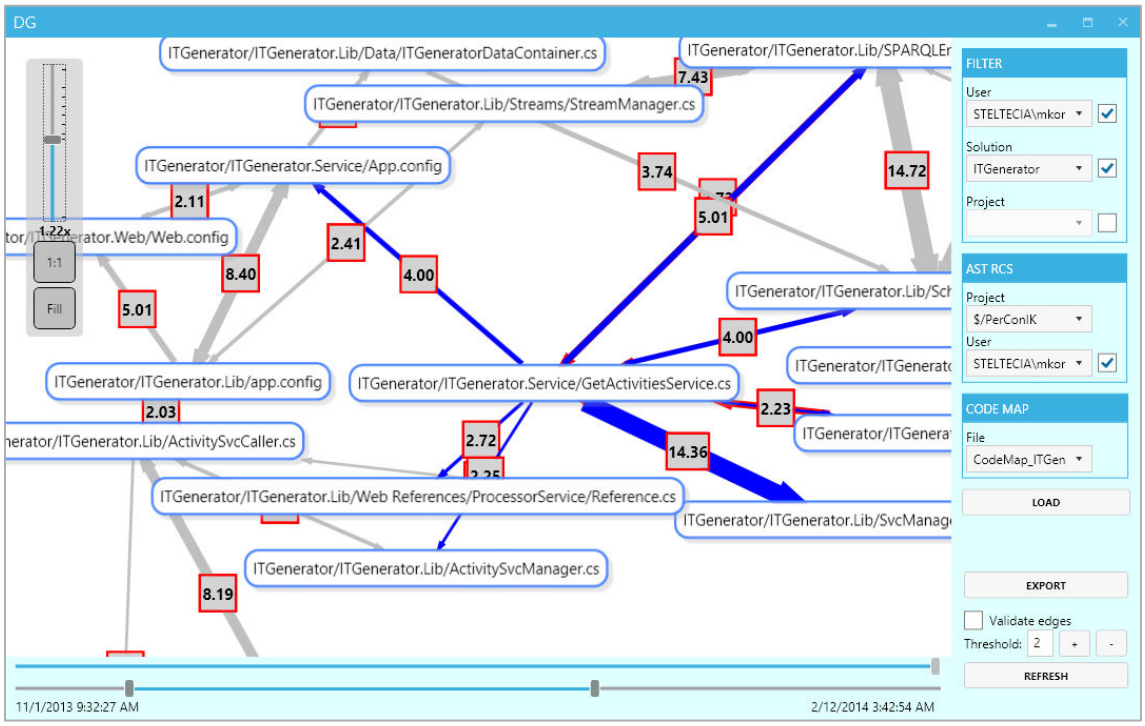
6.1.5. Vizualizácia grafu závislostí

V rámci overovania metódy sme používali dve riešenia vizualizácie grafu závislostí – vlastný prototyp a vizualizáciu grafu v prostredí Microsoft Visual Studio.

Vlastný prototyp sme implementovali v podobe aplikácie v prostredí *.NET* a jazyku *C#*, z dôvodov použitia týchto technológií aj v projekte PerConIK a praktických skúseností autora práce. Pre vizualizáciu grafu sme použili existujúci komponent *Graph#*[16], ktorý poskytuje algoritmy pre rozmiestnenie a možnosť úpravy vzhľadu jednotlivých prvkov grafu. Používateľ má v prototype možnosti:

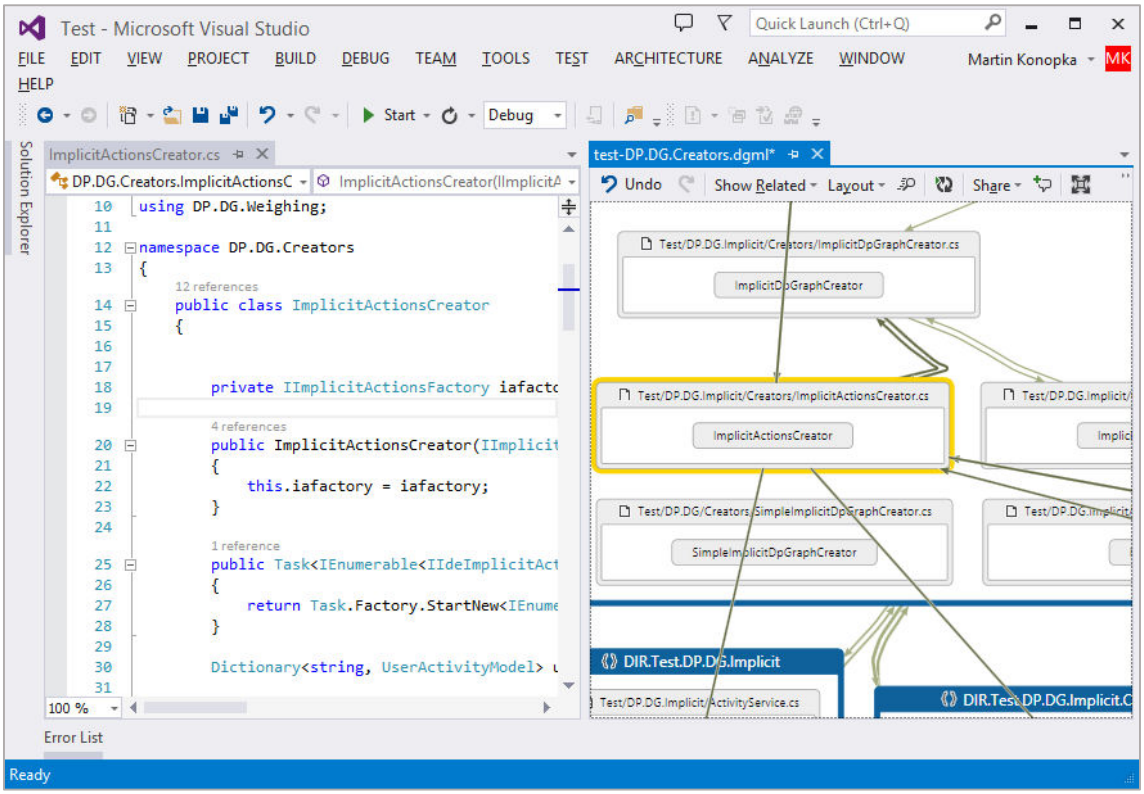
- zvoliť vstupné údaje pre vytvorenie grafu závislostí – projekt a programátor,
- nastaviť vytváranie hrán grafu závislostí pomocou časového okna,
- nastaviť prah váh hrán závislostí,
- zobraziť graf závislostí a interagovať s jeho hranami a vrcholmi,
- exportovať výsledný graf závislostí do súboru vo formáte DGML.

Obrázok 6-4 znázorňuje hlavnú obrazovku prototypu, na ktorej môže používateľ pracovať s grafom závislostí, no nemôže sa z neho dostať do zdrojového kódu. Preto sme umožnili export grafu do súboru DGML a jeho následné otvorenie v nástroji Microsoft Visual Studio, kde je možné priamo pristupovať k obsahom zdrojových súborov z grafu závislostí. Obrázok 6-5 znázorňuje nástroj s otvoreným grafom závislostí a obsahom jedného zo súborov zdrojového kódu.



Obrázok 6-4 Prototyp nástroja pre prácu s grafom implicitných závislostí.

[16] Graph#. http://graphsharp.codeplex.com/



Obrázok 6-5 Vizualizácia grafu implicitných závislostí v nástroji Microsoft Visual Studio.

6.2. Vyhodnotenie hypotéz

Overenie metódy identifikácie implicitných závislostí sme vykonali kvantitatívnym vyhodnotením hypotéz týmito experimentami s reálnymi dátami sledovaných študentských projektov (príloha B):

- automatické porovnanie grafov implicitných a explicitných závislostí,
- manuálne vyhodnotenie významu implicitných závislostí programátormi.

Od zúčastnených programátor sme zároveň získali pozitívnu spätnú väzbu na našu metódu.

6.2.1. Porovnanie grafov implicitných a explicitných závislostí

Z predpokladu nenulového prieniku implicitných a explicitných kontextov softvérových súčiastok, a aj príkladu práce programátora v tejto kapitole, sme skúmali podobnosť grafov implicitných a explicitných závislostí pre vyhodnotenie hypotézy *H1*:

> *Implicitné závislosti odrážajú explicitné závislosti súčiastok, s ktorými programátor pracoval počas plnenia úlohy.*

Grafy závislostí študentských projektov sme porovnali automaticky, študenti vyhodnocovaných projektov neboli do experimentu zapojení.

Dáta experimentu

Na experiment sme použili dáta všetkých dostupných študentských projektov:

- implicitné závislosti sme identifikovali z aktivít všetkých sledovaných programátorov,
- explicitné závislosti sme identifikovali z verzií súborov k času posledných aktivít.

Metodika experimentu

Všetky projekty použité pre tento experiment boli vytvorené v technológiách *C#* a *.NET*. Pre získanie explicitných závislostí sme použili funkcionalitu *Code Map* vo vývojovom prostredí Microsoft Visual Studio, aj keď jej výstup nebolo možné získavať automaticky.

Príprava grafov implicitných a explicitných závislostí medzi súbormi zdrojového kódu pozostávala z týchto krokov pre každý vyhodnocovaný projekt:

1. Identifikovanie explicitných závislostí v zdrojovom kóde:
   1.1. Vygenerovanie grafu explicitných závislostí funkcionalitou *Code Map*.
   1.2. Manuálne rozbalenie všetkých komponentov zdrojového kódu, aby výstupný DGML súbor s grafom obsahoval závislosti tried (závislosti sa lenivo pridávajú do vygenerovaného súboru podľa používateľovej práce s grafom).
   1.3. Exportovanie DGML súboru, ktorý obsahuje explicitné závislosti tried.
2. Identifikovanie implicitných závislostí:
   2.1. Získanie aktivít všetkých sledovaných programátorov na projektoch.
   2.2. Identifikácia a uloženie implicitných závislostí súborov zdrojového kódu.
3. Zjednotenie hierarchickej úrovne vrcholov v grafoch:
   3.1. Identifikovanie súborov, v ktorých sa nachádzajú triedy v grafe explicitných závislostí pomocou služby *AST-RCS* infraštruktúry projektu PerConIK.
   3.2. Nahradenie vrcholoch tried s vrcholmi súborov.
   3.3. Zoskupenie viacerých tried v rovnakých súboroch do spoločných vrcholov a agregovanie závislostí zoskupených tried.

Graf implicitných závislostí typicky obsahoval menej existujúcich súborov, pretože sme nemali k dispozícií záznamy o aktivite práce so všetkými súčiastkami (napr. neboli sledovaní všetci programátori, alebo prerušili sledovanie). Získané grafy implicitných a explicitných závislostí sme potom porovnávali podľa obsiahnutých hrán, ignorovali sme však ich orientáciu. Sledovali sme:

- pomer spoločných hrán voči hranám v grafe implicitných závislostí medzi existujúcimi súbormi, t.j. aká časť identifikovaných implicitných hrán bola explicitná:

$$\frac{\left|E_{exp} \cap E_{imp}\right|}{\left|E'_{imp}\right|}$$

- pomer spoločných hrán voči tým hranám v grafe explicitných závislostí, ktoré sú medzi súbormi zahrnutými v grafe implicitných závislostí:

$$\frac{\left|E_{exp} \cap E_{imp}\right|}{\left|E'_{exp}\right|}$$

Pri vyhodnocovaní sme sledovali aj vplyv nastavenia parametrov váhovania časových implicitných závislostí a obmedzenia prahu váh hrán postupne na *0* až *3*. Jednotlivé závislosti sme pri tomto experimente nevalidovali.

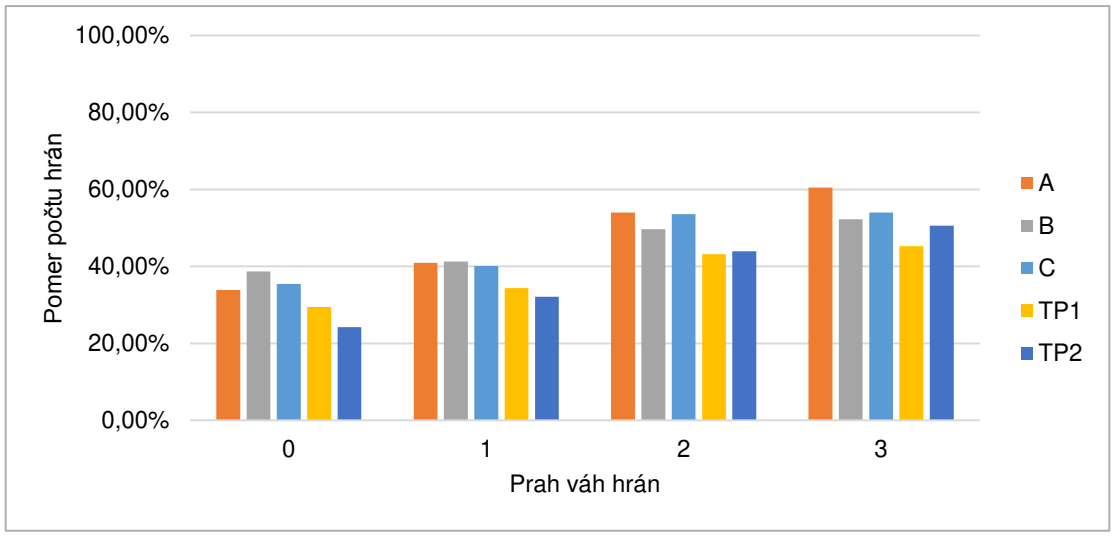
Tabuľka 6-1 Výsledky porovnania grafov implicitných a explicitných závislostí.

Projekt	Prah váh hrán	$\lvert E_{exp} \cap E_{imp} \rvert$	$\lvert E'_{imp} \rvert$	$\lvert E'_{exp} \rvert$	$\frac{\lvert E_{exp} \cap E_{imp} \rvert}{\lvert E'_{imp} \rvert}$	$\frac{\lvert E_{exp} \cap E_{imp} \rvert}{\lvert E'_{exp} \rvert}$
A	0	217	640	232	33,91%	93,53%
	1	173	423	232	40,90%	74,57%
	2	108	200	232	54,00%	46,55%
	3	75	124	232	60,48%	32,33%
B	0	196	507	274	38,66%	71,53%
	1	140	339	274	41,30%	51,09%
	2	70	141	274	49,65%	25,55%
	3	46	88	274	52,27%	16,79%
C	0	281	792	528	35,48%	53,22%
	1	210	524	528	40,08%	39,77%
	2	120	224	528	53,57%	22,73%
	3	67	124	528	54,03%	12,69%
TP1	0	235	797	270	29,49%	87,04%
	1	191	556	270	34,35%	70,74%
	2	123	285	270	43,16%	45,56%
	3	91	201	270	45,27%	33,70%
TP2	0	183	755	189	24,24%	96,83%
	1	149	464	189	32,11%	78,84%
	2	108	246	189	43,90%	57,14%
	3	83	164	189	50,61%	43,92%

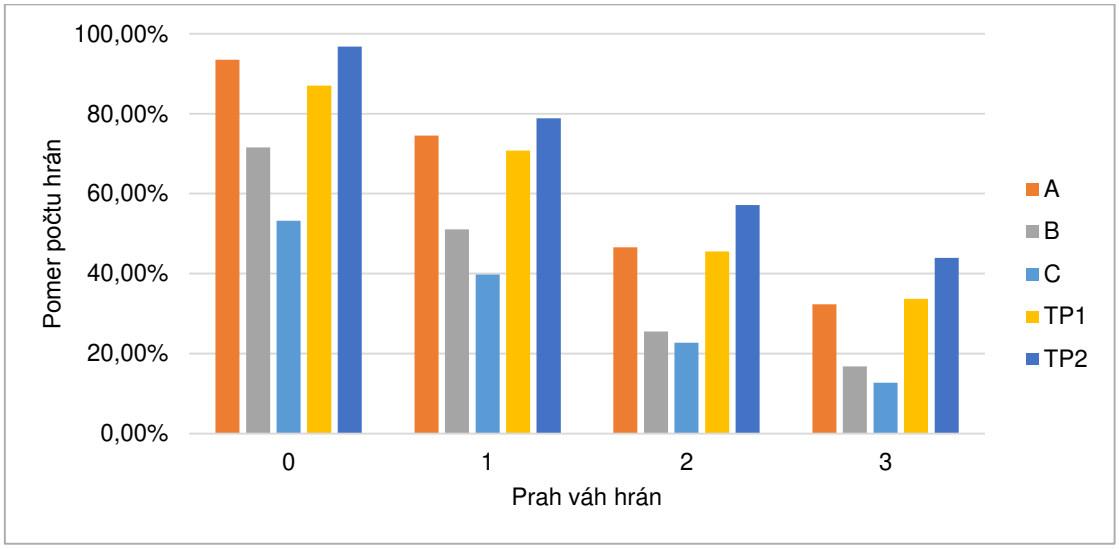
Výsledky experimentu

Vďaka automatickému vyhodnoteniu experimentu sme mohli sledovať vplyv nastavení funkcie váhovania závislostí. Vo výsledkoch experimentu sme však neodhalili signifikantný vplyv nastavenia prvého parametra a. Maximálna skúmaná hodnota parametra b bola 30 minút. V konečnom dôsledku bolo dôležité oddeliť málo významné závislosti minimálnou váhou *0*.

Tabuľka 6-1 uvádza výsledky vyhodnotenia podobnosti grafov závislostí pri uvažovaní implicitných hrán s váhou nad prahovou hodnotou *0* až *3*. Identifikované závislosti sme váhovali s parametrami *a = 10 sekúnd, b = 10 minút, c = 15 minút*. Obrázok 6-6 znázorňuje výsledky prvého porovnania grafov závislostí, môžeme si všimnúť relatívne zväčšovanie prieniku grafov znižovaním počtu závislostí. Obrázok 6-7 znázorňuje druhé porovnanie grafov a opačnú situáciu, pretože počet explicitných závislostí sa nemení a počet implicitných hrán zvyšovaním prahu klesá.



Obrázok 6-6 Výsledné pomery počtov spoločných hrán v grafoch implicitných a explicitných závislostí voči implicitným hranám, t.j. $|E_{exp} \cap E_{imp}|/|E'_{imp}|$.



Obrázok 6-7 Výsledné pomery počtov spoločných hrán v grafoch implicitných a explicitných závislostí voči explicitným hranám medzi súbormi zahrnutými v grafoch implicitných závislostí, t.j. $|E_{exp} \cap E_{imp}|/|E'_{exp}|$.

Diskusia k experimentu

Vyhodnotením prekrývania sa oboch typov závislostí sme potvrdili hypotézu, že z programátorovej aktivity môžeme odvodiť explicitné závislosti v zdrojovom kóde. Aj keď sme nedosiahli úplnú úspešnosť, výsledok experimentu hodnotíme pozitívne. Hypotézu sme vyhodnocovali s projektami v jazyku *C#*, ktorý je najmä staticky-typovaný. Použité dáta nám postačili na potvrdenie použiteľnosti implicitných závislostí v prípade dynamicky-typovaných programovacích jazykov, alebo v prípadoch, keď nemáme možnosť identifikovať explicitné závislosti v zdrojovom kóde. Monitorovanie programátorovej aktivity na úrovni práce so súbormi zdrojového kódu považujeme za jednoduchší proces než heuristicky odhadovať dynamické prepojenia v zdrojovom kóde.

6.2.2. Význam implicitných závislostí

Implicitné závislosti identifikujeme zo záznamov aktivity programátora, preto nás zaujímalo ich vyjadrenie na identifikované závislosti, či odrážajú prepojenia súborov podľa hypotézy *H2*:

> *Implicitné závislosti obohacujú graf explicitných závislostí o nové prepojenia, ktoré sú použiteľné pri vývoji a údržbe zdrojového kódu.*

Experimentu sa zúčastnili 2 programátori spolu s autorom tejto práce na vyhodnotení štyroch projektov.

Dáta experimentu

Na experiment sme použili dáta štyroch projektov: *A, B, C* a *TP1*. Graf implicitných závislostí sme vytvárali z aktivít len zúčastnených programátorov, aby vyhodnocovali závislosti tej časti zdrojového kódu, s ktorou skutočne pracovali. Z identifikovaných implicitných závislostí sme odstránili tie, ktoré sa prekrývali s explicitnými závislosťami, kedy sme využili aj výsledky z prvého experimentu.

Metodika experimentu

V rámci experimentu dostali zúčastnení programátori úlohu postupne prejsť hranami grafu implicitných závislostí a zvážiť ich význam. Ak hrana v grafe nepredstavovala súvis medzi prepojenými súbormi, programátori ju odstránili. Význam hrán sme stanovili ako:

> *Prepojené súbory medzi sebou súvisia v prípade, ak zmena v jednom zo súborov vyžiada zmenu alebo kontrolu druhého súboru, alebo ak vývoj súboru vychádzal z druhého súboru.*

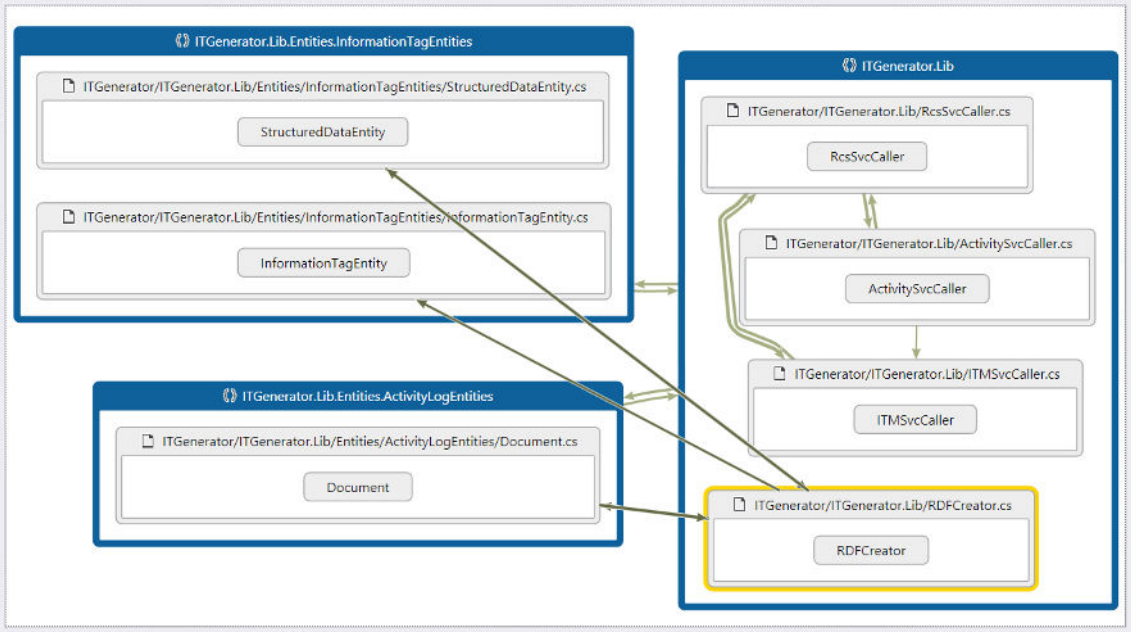
Vykonávaniu experimentu zúčastnenými programátormi predchádzalo pripravenie grafu implicitných závislostí, ktorý sme pre prehľadnosť rozdelili na menšie časti a odstránili z neho explicitné závislosti. Postupovali sme týmito krokmi:

1. Identifikovanie implicitných závislostí z aktivít zúčastnených programátorov.
2. Váhovanie časových implicitných závislostí s parametrami $a = 10$ sekúnd, $b = 10$ minút a $c = 30$ minút.
3. Odstránenie explicitných závislostí podľa postupu v predchádzajúcom experimente prekrývania závislostí a hrán s váhou nižšou ako 2 (pre projekt *TP1* bol prah váh zvolený na 3 kvôli veľkému počtu závislostí).
4. Získanie softvérových súčiastok obsiahnutých v súboroch zdrojového kódu (vrcholy grafu) pomocou služby *AST-RCS*, t.j. menné priestory, triedy, rozhrania, číselníky, atď.
5. Odhadnutie menných priestorov pre ostatné súbory podľa ich cesty, napr. pre webové stránky súbor *ITGenerator/ITGenerator.Web/Views/Schemas/Index.cshtml* je v mennom priestore *ITGenerator.Web.Views.Schemas*.
6. Rozdelenie grafu celého projektu na viacero podgrafov podľa menných priestorov. Každý podgraf obsahoval súbory zvoleného menného priestoru, ich závislosti navzájom a ich závislosti s ostatnými súbormi. Jednotlivé závislosti sa v podgrafoch neopakovali.
7. Uloženie podgrafov v DGML formáte.

Výsledné podgrafy bolo možné vďaka formátu DGML otvoriť v prostredí Microsoft Visual Studio pri zdrojovom kóde projektov, čo umožnilo zobraziť obsah súborov priamo z grafu. Obrázok 6-8 uvádza ukážku vyhodnocovaného grafu počas experimentu. Zúčastnení programátori postupovali počas experimentu týmito krokmi:

1. Otvorenie súboru s grafom závislostí.
2. Pre každú hranu v grafe:
   2.1. Zváženie významu závislosti. Pokiaľ závislosť nemala význam, hranu odstránili.
   2.2. Otvorenie skutočného súboru zdrojového kódu z grafu v prípade potreby.
3. Uloženie upraveného grafu závislostí.

Vyhodnotenie experimentu, t.j. úspešnosti identifikácie implicitných závislostí, sme vykonali porovnaním počtu hrán v upravených DGML súboroch grafov voči počtu hrán pôvodných súborov.



Obrázok 6-8 Vizualizácia grafu implicitných závislostí medzi súbormi zdrojového kódu pre experiment.

Výsledky experimentu

Počet vyhodnocovaných hrán v grafe závislostí závisel od veľkosti a povahy vyhodnocovaného projektu a množstva zaznamenanej aktivity zúčastneného programátora. Programátor projektu *TP1* vyhodnotil mnohonásobne viac hrán ako programátori projektov *A* a *B*. Hrany v grafe pre projekt A vyhodnocovali dvaja programátori zvlášť. Pri experimente sme nepoužili validáciu závislostí a z grafu sme odstránili už neexistujúce súbory s ich závislosťami. Tabuľka 6-2 uvádza výsledky vyhodnotenia hrán, dosiahli sme priemernú presnosť 82,55%. V tabuľke uvádzame aj pôvodný počet hrán a počet vrcholov v jednotlivých grafoch.

Tabuľka 6-2 Výsledky vyhodnotenia presnosti identifikácie implicitných závislostí 4 študentských projektov.

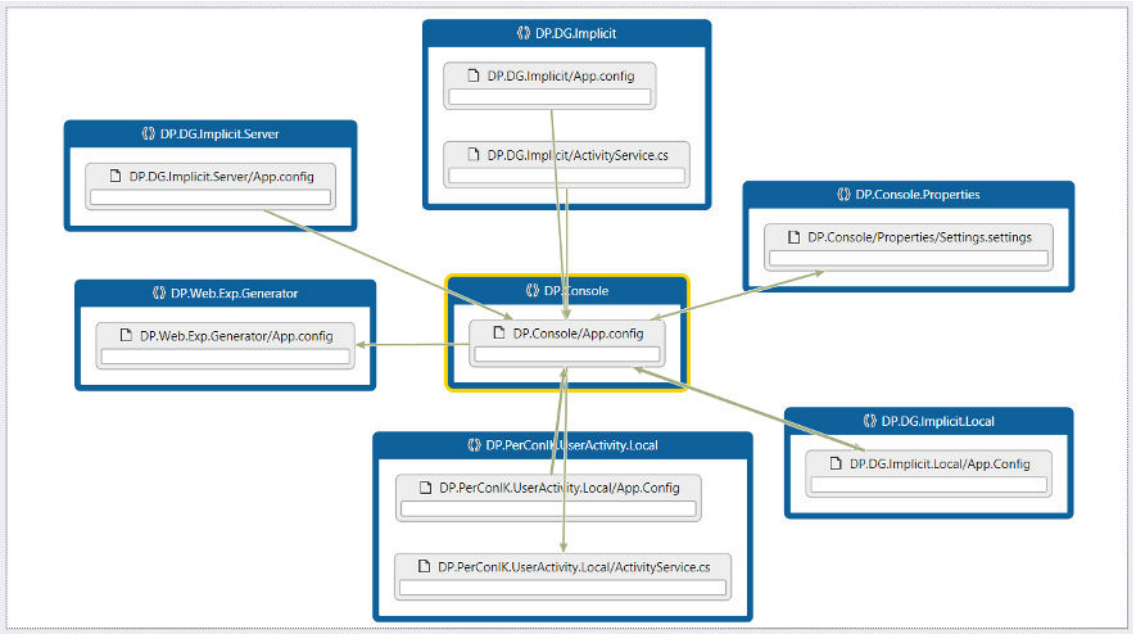
Projekt	Prah váh hrán	Počet vrcholov	Pôvodný počet hrán	Výsledný počet hrán	Presnosť
A-1	2	81	132	103	**78,03%**
A-2	2	47	48	35	**72,92%**
B	2	80	112	103	**91,96%**
C	2	146	257	203	**78,99%**
TP1	3	295	634	576	**90,85%**

Diskusia k experimentu

Manuálne vyhodnotenie závislostí sa počas vykonávania experimentu ukázalo ako časovo náročné. Programátori však aj napriek rozsiahlosti úlohy vedeli zareagovať na všetky závislosti, pričom možnosť otvorenia konkrétneho súboru zdrojového kódu a študovanie jeho obsahu vykonal každý programátor maximálne 3-krát. Programátori si počas experimentu vybavovali situácie, ktoré podľa nich podnietili vznik závislostí, napr. si pamätali na úlohy, ktoré plnili. V prípade, že závislosť uvedená v grafe už v skutočnosti nebola platná, pamätali si na obdobie, kedy závislosť platila a prečo už neplatí. Takto neplatné závislosti programátori odstraňovali z grafu pri vyhodnocovaní.

Programátori ocenili rozdelenie grafu na menšie časti a organizáciu súborov do menných priestorov spolu s ich obsiahnutými súčiastkami. Pri vyhodnocovaní závislostí webových projektov (*A* a *TP1*) programátori ocenili identifikované závislosti medzi webovými stránkami a súčiastkami, ktoré s nimi skutočne súviseli. Zároveň ocenili aj závislosti súborov, ktoré v zdrojovom kóde nie sú vôbec previazané. Obrázok 6-9 uvádza príklad správne identifikovaných závislostí medzi konfiguračnými súbormi a súbormi, ktoré ich používajú.

Jedným z ďalších pozitívnych výsledkov bolo, keď si programátor projektu *A* pri vykonávaní experimentu uvedomil, že si zabudol v zdrojovom kóde vytvoriť informačnú značku *TODO*. Uvedomil si to na základe identifikovanej závislosti a spomenutia si starej úlohy, ktorú riešil.



Obrázok 6-9 Identifikované závislosti konfiguračných súborov zdrojového kódu tejto práce.

6.3. Diskusia

Implicitné závislosti súčiastok zdrojového kódu rozširujú priestor známych prepojení v zdrojovom kóde. Metódu identifikácie implicitných závislosti sme overili pomocou dvoch kvantitatívnych experimentov a priameho zapojenia programátorov. Podarilo sa nám získať pozitívne výsledky pre potvrdenie stanovených hypotéz využitia implicitných závislostí pre podporu udržovateľnosti softvéru. Experimenty sme však vykonali na obmedzenej vzorke dát študentských projektov. To vyplýva aj zo všeobecných problémov získavania implicitnej spätnej väzby, napr. pocit zásahu do súkromia, vypínanie sledovania a zabudnutie ho opätovne zapnúť, ale aj z týchto dôvodov:

- zaznamenaná činnosť bola často len jedného člena tímu na projekte,
- projekty boli študentské, menšieho rozsahu,
- štruktúra zdrojového kódu projektov prechádzala častými zmenami,
- študenti nepracovali na projektoch pravidelne.

Ako vhodný spôsob overenia metódy sa ukázal byť experiment s priamym zapojením sledovaných programátorov, ktorý poznali zdrojový kód projektu. Preto sme identifikovali ako možné pokračovanie experimentu vytvorenie webovej aplikácie, v ktorej programátor postupne hodnotí ponúkané závislosti softvérových súčiastok. Rozdielom oproti vykonanému experimentu je ponúkanie dvojíc súčiastok namiesto zobrazenia celého grafu, a tak zinteraktívnenie úlohy. Navyše sa tohto experimentu môžu zúčastniť aj nesledovaní programátori projektov, ktorí so zdrojovým kódom taktiež pracovali.

Pre realizáciu navrhnutého použitia metódy identifikácie implicitných závislostí počas práce so zdrojovým kódom (kapitola 5.4, ponúknutie zoznamu závislých súčiastok), ale aj pre uvedený návrh experimentu, je vhodné graf implicitných závislostí realizovať ako službu. Tým sa vyhneme opakovanej identifikácii závislostí a zbytočnému dopytovaniu webových služieb infraštruktúry projektu PerConIK. To nám navyše zjednoduší aj integrovanie grafu implicitných závislostí ako rozšírení vývojových prostredí, najmä. Microsoft Visual Studio a Eclipse.

Kapitola 7

Zhodnotenie

Meranie a vyhodnocovanie softvérového projektu je dôležité pre zabezpečenie splnenia stanovených cieľov v požadovanej kvalite. Softvérový projekt môžeme sledovať z rôznych uhlov pohľadu a zainteresovaných strán. Výskum v oblasti merania softvéru je rozsiahle rozpracovaný a najčastejšie sa stretávame s vyhodnocovaním obsahu zdrojového kódu. Softvérový produkt je výsledkom činnosti softvérových inžinierov, preto nemôžeme zanedbať ich aktivity, ktoré počas práce vykonávajú, a ani kontext vplývajúci na vývoj. Vplyv aktivít programátorov a kontextu vývoja softvéru na jeho vlastnosti je motiváciou aktuálneho výskumu v oblasti softvérového inžinierstva, v mnohých prípadoch motivovaný výskumom v doméne Webu. Medzi priestormi softvérových artefaktov a informačných priestorov Webu môžeme nájsť paralelu zúčastnených strán, vzťahov medzi artefaktmi a ich použitia. Aj napriek pôvodne rozdielnym úlohám v oboch oblastiach môžeme uplatniť spoločné poznatky sledovania a modelovania používateľa, získavania spätnej väzby, alebo aj objavovania znalostí o artefaktoch pri vyhodnocovaní softvéru.

Riziká a obmedzenia získavania implicitnej spätnej väzby sa rovnako objavujú aj pri sledovaní programátorov. Zaznamenávanie činnosti prirodzene vzbudzuje vnútorné nepohodlie alebo pocit straty súkromia. To môže ovplyvniť prácu programátorov, prípadne oni sami prerušia sledovanie, čím stratíme cenné informácie. Sledovanie programátorov v prostredí firmy sa odlišuje od Webu jeho uzavretosťou. Je v záujme firiem ochrániť obchodné tajomstvo, preto sa pri výskume sledovania a vyhodnocovania údajov o vývoji softvéru môžeme stretnúť s problémami nepovolenia prístupu k údajom tretím stranám. V našom prípade sme mali možnosť pracovať s dátami softvérovej firmy, ale rozhodli sme sa overenie našej práce vykonať na študentských projektoch. Študenti sa od pracovníkov firmy odlišujú najmä nepravidelným prístupom k práci, ale aj väčších zábran sledovania, keďže vývoj softvéru nie je ich jedinou činnosťou na osobných počítačoch. Firmy v súčasnosti sledujú činnosť svojich pracovníkov z dôvodu kontroly, ale aj zlepšenia procesov. Sledovanie je však uzavreté, a aj pracovníci sú ochotní byť sledovaní v rámci firmy, keďže prispievajú k pracovným procesom. Študentské projekty pre overenie našej práce sme si vybrali z dôvodov jednoduchšieho prístupu k samotným študentom, možností častej diskusie a s príbuznosťou školského prostredia pre autora tejto práce.

Činnosť programátorov najčastejšie sledujeme počas vývoja a údržby softvéru, kedy nás zaujíma jeho udržovateľnosť. Produkovaný softvér pozostáva z prepojených súčiastok zdrojového kódu. Tieto prepojenia vznikajú úpravami v zdrojovom kóde, sú používané pri študovaní kódu, jeho opravovaní a rozširovaní o novú funkcionalitu. Tradične identifikujeme ich závislosti syntaktickou analýzou. Tým však neodhalíme všetky závislosti, ktoré medzi nimi existujú, alebo ich nedokážeme odhaliť vôbec pri dynamicky-typovaných programovacích jazykoch.

V našej práci sme predstavili metódu identifikácie skrytých závislostí súčiastok zdrojového kódu z aktivít programátora a kontextu vývoja softvéru. Z povahy vstupných dát ich označujeme ako *implicitné*. Identifikácia závislostí je softvérovou metrikou určujúcou vlastnosť miery prepojenia dvojice softvérových súčiastok. Implicitnými závislosťami rozširujeme existujúci graf závislostí o nové prepojenia.

Navrhnutú metódu sme experimentálne overili na dátach študentských projektov s cieľom potvrdenia hypotéz použiteľnosti implicitných závislostí pri vývoji a údržbe softvéru:

- použitie implicitných závislostí ako náhrady explicitných závislostí v prípade, keď ich nedokážeme identifikovať, napr. pri dynamicky-typovaných programovacích jazykoch,
- identifikovanie závislostí medzi súčiastkami, ktoré nie sú na úrovni zdrojového kódu prepojené, ale súvisia s plnením programátorovej úlohy,
- identifikovanie závislostí konfiguračných súborov so súčiastkami zdrojového kódu.

V našej práci sme vychádzali z výskumného projektu PerConIK, ktorý uplatňuje „webifikáciu vývoja softvérových projektov“. Infraštruktúru projektu sme použili pre overenie našej práce a pre prístup k dátam študentských projektov. Začlenenie našej práce do infraštruktúry projektu a jej ďalšie experimentálne overenie vidíme ako vhodnú možnosť budúcej práce, rovnako ako aj nasadenie realizovanej metódy a jej použitie programátormi pri reálnych úlohách. V diskusiách s programátormi sme získali pozitívnu spätnú väzbu na našu prácu, a preto vidíme potenciál v praktickej použiteľnosti implicitných závislostí.

Zároveň si uvedomujeme, že výsledky experimentov záviseli aj od spôsobu identifikovania a váhovania identifikovaných závislostí. To je ďalšou možnosťou ako v budúcnosti rozšíriť našu prácu, napr. metódami strojového učenia.

K aktuálnemu stavu poznania v oblasti vyhodnocovania softvéru prispievame výsledkami:

- identifikovanie vzťahov súčiastok zdrojového kódu:
  - takých, ktoré súčasné metódy neidentifikujú,
  - aj v prípade nedostupnosti syntaktickej analýzy zdrojového kódu,
  - aj s ostatnými artefaktmi softvérového projektu;
- definovanie scenárov použitia implicitných závislostí v procese vývoja a údržby softvérového produktu, najmä pri jeho testovaní;

Výsledky našej práce sme úspešne publikovali a prezentovali na konferencii IIT.SRC 2014 s príspevkom *Identifying Hidden Source Code Dependencies* (Konôpka, 2014). V overovaní práce a publikovaní výsledkov plánujeme pokračovať. V prílohách práce uvádzame aj návrh príspevku na konferenciu ESEM 2014.

Literatúra

1. AALST, W., et al.: Process Mining Manifesto. In: Business Process Management Workshops, Lecture Notes in Business Information Processing, vol. 99, Springer-Verlag, 2012, s. 169-194.
2. ABREU, R., PREMRAJ, R.: How Developer Communication Frequency Relates to Bug Introducing Changes. In: Proc. of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops (IWPSE-Evol '09), ACM, 2009, s. 153-158.
3. ANTUNES, B., CORDEIRO, J., GOMEZ, P.: An Approach to Context-based Recommendation in Software Development. In: Proc. of the 6th ACM conference on Recommender systems (RecSys '12), ACM, 2012, s. 171-178.
4. BAMIS, A., FANG, J., SAVVIDES, A.: A Method for Discovering Components of Human Rituals from Streams of Sensor Data. In: Proc. of the 19th ACM International Conference on Information and Knowledge Management (CIKM '10), ACM, 2010, s. 779-788.
5. BARBIERI, D.F., BRAGA, D., CERI, S., et al.: An Execution Environment for C-SPARQL Queries. In: Proc. of the 13th International Conference on Extending Database Technology (EDBT '10). ACM, 2010, s. 441-452.
6. BIELIKOVÁ, M.: Softérové inžinierstvo: Princípy a manažment. FEI STU Bratislava, Vydavateľstvo STU, 2000, 220 s.
7. BIELIKOVÁ, M., NÁVRAT, P., CHUDÁ, D., et al.: Webification of Software Development: General Outline and the Case of Enterprise Application Development. In: AWERProcedia Information Technology & Computer Science, vol. 3, 3rd World Conference on Information Technology (WCIT-2012), 2013, s. 1157-1162.
8. BIELIKOVÁ, M., RÁSTOČNÝ, K.: Lightweight Semantics over Web Information Systems Content Employing Knowledge Tags. In: ER Workshops 2012 (WISM 2012), Springer, 2012, s. 327-336.
9. BIELIKOVÁ, M., POLÁŠEK, I., BARLA, M., et al.: Platform Independent Software Development Monitoring: Design of an Architecture. In: Proc. of the 40th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM), Springer, 2014, s. 126-137.
10. BOEHM, B.W., BROWN, J.R., LIPOW, M.: Quantitative Evaluation of Software Quality. In: Proc. of the 2nd international conference on Software Engineering (ICSE '76), IEEE Computer Society Press, 1976, s. 592-605.
11. BRYSON, A., FORTH, J.: Are There Day of the Week Productivity Effects?. In: Manpower Human Resources Lab, discussion paper, Manpower Human Resource Lab, London, 2007.
12. CAVANO, J.P., MCCALL, J.A.: A Framework for the Measurement of Software Quality. In: ACM SIGSOFT Software Engineering Notes, vol. 3, no. 5. ACM, 1978, s. 133-139.

13. COLEMAN, D., LOWTHER, B., OMAN, P.: The Application of Software Maintainability Models in Industrial Software Systems. In: Journal of Systems and Software, vol. 29, no. 1, Elsevier Science Inc., 1995, s. 3-16.
14. COMAN, I.D.: An Analysis of Developers' Tasks Using Low-Level, Automatically Collected Data. In: Proc. of the 6th Joint Meeting on European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: companion papers (ESEC-FSE companion '07), ACM, 2007, s. 579-582.
15. COMAN, I.D., SILLITTI, A.: Automated Identification of Tasks in Development Sessions. In: Proc. of the 2008 The 16th IEEE International Conference on Program Comprehension (ICPC '08), IEEE Computer Society, 2008, s. 212-217.
16. COUNSELL, S., HASSOUN, Y., LOIZOU, G., et al.: Common Refactorings, A Dependency Graph and Some Code Smells: An Empirical Study of Java OSS. In: ACM/IEEE International Symposium on Empirical Software Engineering, ACM, 2006. s. 288-296.
17. DAGPINAR, M., JAHNKE, J.H.: Predicting Maintainability with Object-Oriented Metrics - An Empirical Comparison. In: Proc. of 10th Working Conference on Reverse Engineering (WCRE 2003), IEEE Computer Society, 2003, s. 155-164.
18. DAVIS, J.: Coding in the Rain, 2003, Jul., 8.
[online http://drjasondavis.com/2013/07/08/coding-in-the-rain/] [citované 14. máj 2014]
19. DEMARCO, T.: Controlling Software Projects: Management, Measurement, and Estimates. 1st. Prentice Hall/Yourdon Press, 1982.
20. DEMOVIČ, Ľ., KONÔPKA, M., LÁNI, M., et al.: Enhancing Web Surfing Experience in Conditions of Slow and Intemittent Internet Connection. In: Information Sciences and Technologies, Bulletin of the ACM Slovakia, vol. 4, no. 2, ACM, 2012, s. 25-29.
21. DEY, A.K., ABOWD, G.D.: Towards a Better Understanding of Context and Context-Awareness. In: CHI 2000 Workshop on The What, Who, Where, When, and How of Context-Awareness, The Hague, 2000, s. 12.
22. DIETRICH, J., MCCARTIN, C., TEMPERO, E., et al.: On the Existence of High-impact Refactoring Opportunities in Programs. In: Proc. of the 35th Australasian Computer Science Conference (ACSC '12), Australian Computer Society, 2012, s. 37-48.
23. DUBEY, S.K., RANA, A.: Assessment of Maintainability Metrics for Object-Oriented Software System. In: ACM SIGSOFT Software Engineering Notes, vol. 36, no. 5, ACM, 2011, s. 7.
24. EBBINGHAUS, H.: Memory: A Contribution to Experimental Psychology. Preklad: RUGER, H.A., BUSSENIUS, C.E., New York: Teachers College, 1885/1913.
25. EYOLFSON, J., TAN, L., LAM, P.: Do Time of Day and Developer Experience Affect Commit Bugginess?. In: Proc. of the 8th Working Conference on Mining Software Repositories (MSR '11), ACM, 2011, 153-162.
26. FEJES, M.: Facial Expression Recognition for Semantic User Modeling. In: Proc. of 9th Student Research Conference in Informatics and Information Technologies Bratislava (IIT.SRC 2013). Nakladatelstvo STU, 2013, s. 6.
27. FENTON, N.E., PFLEEGER, S.L: Software Metrics: A Rigorous and Practical Approach. 2nd. PWS Pub. Co., Boston, USA, 1998.
28. FOWLER, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, 1999.
29. FRITZ, T., MURPHY, G.C., HILL, E.: Does a Programmer's Activity Indicate Knowledge of Code?. In: Proc. of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC-FSE '07), ACM, 2007, s. 341-350.

30. FRITZ, T., JINGWEN, O., MURPHY, G.C., et al.: A Degree-of-Knowledge Model to Capture Source Code Familiarity. In: Proc. of the 32nd ACM/IEEE International Conference on Software Engineering, ACM, 2010, s. 385-394.
31. GOUSIOS, G., KALLIAMVAKOU, E., SPINELLIS, D.: Measuring Developer Contribution from Software Repository Data. In: Proc. of the 2008 International Working Conference on Mining Software Repositories (MSR '08), ACM, 2008, s. 129-132.
32. HALSTEAD, M.: Elements of Software Science (Operating and Programming Systems Series). Elsevier North-Holland, Inc., Amsterdam, 1999.
33. HAN, J., KAMBER, M., PEI, J.: Data Mining: Concepts and Techniques. 3rd. Morgan Kaufmann Publishers, 2001.
34. HASSAN, A.E., HOLT, R.C.: Studying the Chaos of Code Development. In: Proc. of the 10th Working Conference on Reverse Engineering (WCRE '03). IEEE Computer Society, 2003, s. 123-134.
35. HOLMES, R., NOTKIN, D.: Identifying Program, Test, and Environmental Changes That Affect Behaviour. In: Proc. of the 33rd International Conference on Software Engineering (ICSE '11). ACM, 2011, s. 371-380.
36. CHIDAMBER, S.R., KEMERER, C.F.: A Metrics Suite for Object Oriented Design. In: IEEE Transactions on Software Engineering, vol. 20, no. 6, IEEE Press, 1994, s. 476-493.
37. CHRISTIDIS, K., PARASKEVOPOULOS, F., PANAGIOTOU, D., et al.: Combining Activity Metrics and Contribution Topics for Software Recommendations. In: Proc. of 3rd International Workshop on Recommendation Systems for Software Engineering (RSSE), Switzerland, 2012, s. 43-46.
38. KALLIAMVAKOU, E., GOUSIOS, G., SPINELLIS, D., et al.: Measuring Developer Contribution from Software Repository Data. In: Proc. of the 4th Mediterranean Conference on Information Systems (MCIS 2009), Greece, 2009, s. 600-611.
39. KERSTEN, M., MURPHY, G.C.: Using Task Context to Improve Programmer Productivity. In: Proc. of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '06/FSE-14). ACM, 2006, s. 1-11.
40. KNUTOV, E., DE BRA, P., PECHENIZKIY, M.: AH 12 Years Later: a Comprehensive Survey of Adaptive Hypermedia Methods and Techniques. In: New Review of Hypermedia and Multimedia, vol. 15, no. 1, Taylor & Francis, UK, 2009, s. 5-38.
41. KONÔPKA, M.: Identifying Hidden Source Code Dependencies. In: Proc. of 10th Student Research Conference in Informatics and Information Technologies Bratislava (IIT.SRC 2014), Nakladateľstvo STU, 2014, s. 474-479.
42. KURIC, E., BIELIKOVÁ, M.: Search in Source Code Based on Identifying Popular Fragments. In: SOFSEM 2013: Theory and Practice of Computer Science, Lecture Notes in Computer Science, vol. 7741, Springer-Verlag, 2013, s. 408-419..
43. LE-PHUOC, D., PARREIRA, J.X., HAUSWIRTH, M.: Linked Stream Data Processing. In: Reasoning Web, Semantic Technologies for Advanced Query Answering, Lecture Notes in Computer Science, vol. 7487, Springer-Verlag, 2012, s. 245-289.
44. LI, W., HENRY, S.: Object-oriented Metrics That Predict Maintainability. In: Journal of Systems and Software, vol. 23, no. 2, Elsevier Science Inc, 1993, s. 111-122.
45. LIEBERHERR, K., HOLLAND, I., RIEL, A.: Object-oriented Programming: An Objective Sense of Style. In: Conference Proc. on Object-oriented Programming Systems, Languages and Applications (OOPSLA '88), ACM, 1988, s. 323-334.
46. LINCKE, R., LUNDBERG, J., LÖWE, W.: Comparing Software Metrics Tools. In: Proc. of the 2008 International Symposium on Software Testing and Analysis (ISSTA '08), ACM, 2008, s. 131-142.

47. MARINESCU, R.: Detecting Design Flaws via Metrics in Object-Oriented Systems. In: Proc. of the 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS '01), IEEE Computer Society, 2001, s. 173-182.
48. MCCABE, T.J.: A Complexity Measure. In: Proc. of the 2nd International Conference on Software Engineering (ICSE '76), IEEE Computer Society, 1976, s. 407-419.
49. Microsoft: Code Metrics Values. 2012.
[online http://msdn.microsoft.com/en-us/library/bb385914.aspx] [citované 14. máj 2014].
50. Microsoft: Find Potential Problems in Code on Dependency Graphs. 2013.
[online http://msdn.microsoft.com/en-us/library/vstudio/hh264270.aspx]
[citované 14. máj 2014].
51. MORRIS, K.L.: Metrics for Object Oriented Software Development, Master Thesis (M.S.). Massachusetts Institute of Technology, Cambridge, MA, 1989.
52. NAZERFARD, H., RASHIDI, P., COOK, D.J.: Using Association Rule Mining to Discover Temporal Relations of Daily Activities. In: Proc. of the 9th International Conference on Toward Useful Services for Elderly and People with Disabilities: Smart Homes and Health Telematics (ICOST'11), Springer-Verlag, 2011, s. 49-56.
53. OMAN, P.W., HAGEMEISTER, J.R.: Construction and Testing of Polynomials Predicting Software Maintainability. In: Journal of Systems and Software, vol. 24, no. 3, Elsevier Science Inc., 1994, s. 251-266.
54. Project Management Institute: A Guide to the Project Management Body of Knowledge. 2004.
55. PURUSHUTHAMAN, R., PERRY, E.T.: Toward Understanding the Rhetoric of Small Source Code Changes. In: IEEE Transactions on Software Engineering, vol. 31, no. 6, IEEE Computer Society, 2005, s. 511-526.
56. RÁSTOČNÝ, K., BIELIKOVÁ, M.: Human and Machine Metadata over the Web Content Maintenance. In: Proc. of ICWE 2012 Workshops, Doctoral Consortium, Springer-Verlag, 2013, s. 216-220.
57. RIAZ, M., MENDES, E., TEMPERO, E.: A Systematic Review of Software Maintainability Prediction and Metrics. In: Proc. of the 2009 3rd Intern. Symposium on Empirical Software Engineering and Measurement (ESEM '09), IEEE Computer Society, 2009, s. 367-377.
58. SJØBERG, D.I.K., ANDA, B., MOCKUS, A.: Questioning Software Maintenance Metrics: A Comparative Case Study. In: Proc. of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '12). ACM, 2012, s. 107-110.
59. SOMMERVILLE, I.: Software Engineering, 9th Edition. Addison-Wesley, 2010.
60. WHITE, K.G.: Forgetting Functions. In: Animal Learning & Behavior, 2001, s. 193-207.
61. WITTEN, I., FRANK, E., HALL, M.: Data Mining: Practical Machine Learning Tools and Techniques, 3rd Edition. Morgan Kaufmann Publishers, 2011.
62. ZELENÍK, D.: Beyond Code Review: Detecting Errors via Context of Code Creation. In: Proc. of 9th Student Research Conference in Informatics and Information Technologies Bratislava (IIT.SRC 2013). Nakladateľstvo STU, 2013, s. 8.
63. ZIMMERMANN, T., NAGAPPAN, N.: Predicting Defects Using Network Analysis on Dependency Graphs. In: Proc. of the 30th International Conference on Software Engineering, ACM, 2008, s. 531-540.

Príloha A

Technická dokumentácia

V rámci riešenia našej práce sme implementovali:

- knižnice reprezentácie grafu explicitných a implicitných závislostí, realizáciu metódy,
- knižnice pre získanie vstupných dát z infraštruktúry projektu PerConIK,
- prototyp vizualizácie grafu závislostí a aplikácie pre experimentálne overenie,
- systémovú službu periodickej identifikácie implicitných závislostí.

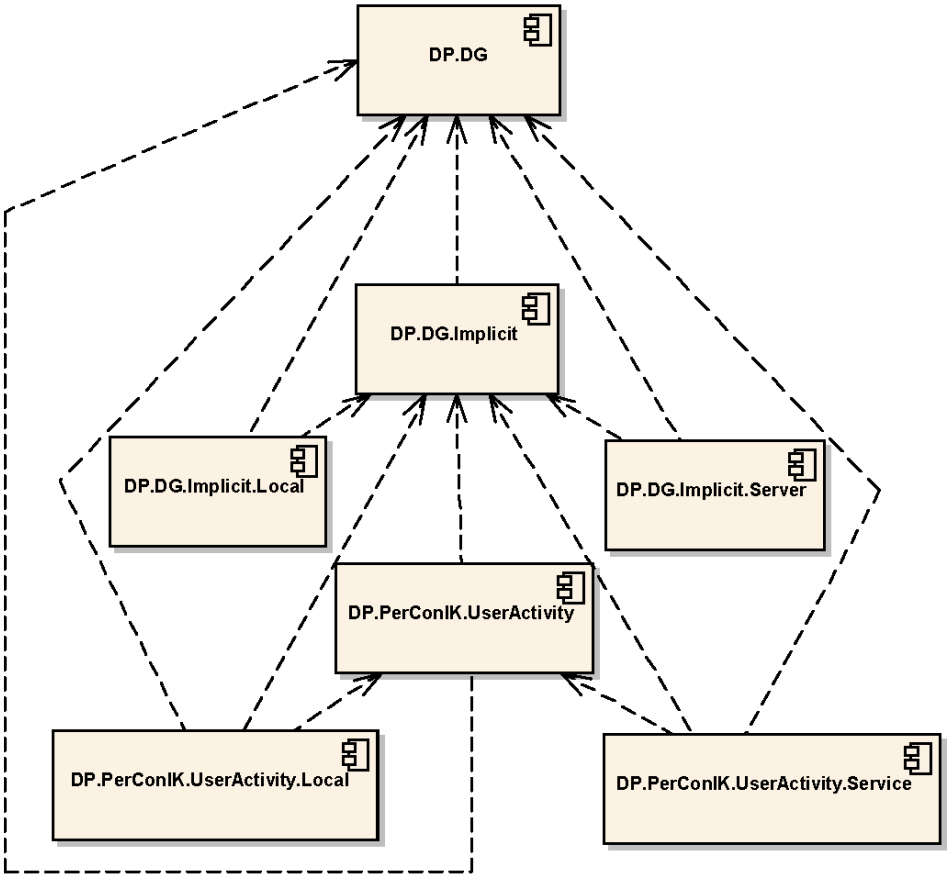
Riešenie je implementované v programovacom jazyku *C#* použitím aplikačného rámca *.NET*. V implementácii prototypu vizualizácie sme použili technológiu *WPF*.

A.1. Architektúra riešenia

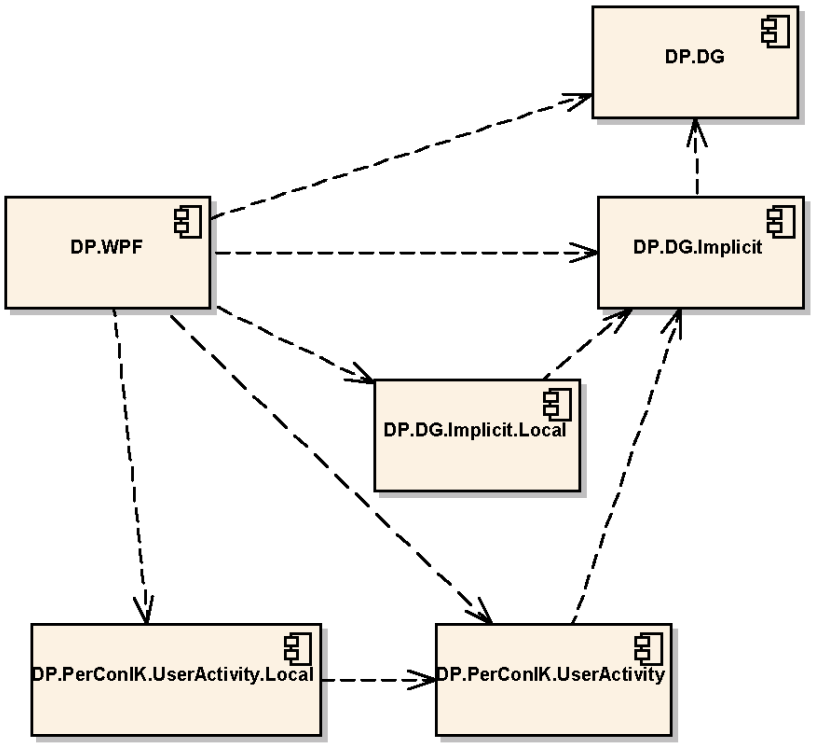
Pri návrhu architektúry riešenia sme sa rozhodli o vysokú mieru dekompozície. Zdrojový kód sme organizovali do viacerých komponentov, najmä knižnice s rozhraniami a všeobecnými realizáciami a knižnice so špecializovanými realizáciami. Dôvodom dekompozície je zjednodušenie vytvorenia koncovej aplikácie podľa potrieb, napr. získavať aktivity programátora môžeme z lokálnej databázy aktivít alebo webových služieb, či identifikované závislosti ukladať do databázy.

Obrázok A-1 uvádza diagram komponentov knižníc pre prácu s grafom:

- *DP.DG* – základná reprezentácia grafu závislostí, rozhrania entít, ich tovární a lokátorov, práca s grafom explicitných závislostí.
- *DP.DG.Implicit* – reprezentácia grafu implicitných závislostí, implementácia metódy identifikácie implicitných závislostí, váhovania a validovania závislostí.
- *DP.DG.Implicit.Local* – základná realizácia grafu implicitných závislostí, graf je uchovaný len v operačnej pamäti počas behu aplikácie.
- *DP.DG.Implicit.Server* – realizácia grafu implicitných závislostí s uchovaním závislostí a entít v lokálnej databáze.
- *DP.PerConIK.UserActivity* – všeobecné rozhranie pre prístup aktivitám programátora cez infraštruktúru projektu PerConIK.
- *DP.PerConIK.UserActivity.Local* – realizácia rozhrania pre prístup k aktivitám programátora z lokálnej databázy.
- *DP.PerConIK.UserActivity.Service* – realizácia rozhrania pre prístup k aktivitám programátora cez webové služby infraštruktúry projektu PerConIK.



Obrázok A-1 Diagram komponentov knižníc grafu závislostí.



Obrázok A-2 Diagram komponentov použitia knižníc grafu závislostí WPF aplikáciou vizualizácie grafu.

Prototyp vizualizáciu grafu *DP.WPF* používa lokálnu databázu aktivít a lokálne vytváranie grafu závislostí (Obrázok A-2). Opačná situácia je pri systémovej službe identifikácie závislostí, ktorá sa vykonáva na webovom serveri. Tá používa webové služby prístupu k aktivitám (*DP.PerConIK.UserActivity.Service*) a výsledky ukladá do databázy (*DP.DG.Implicit.Server*).

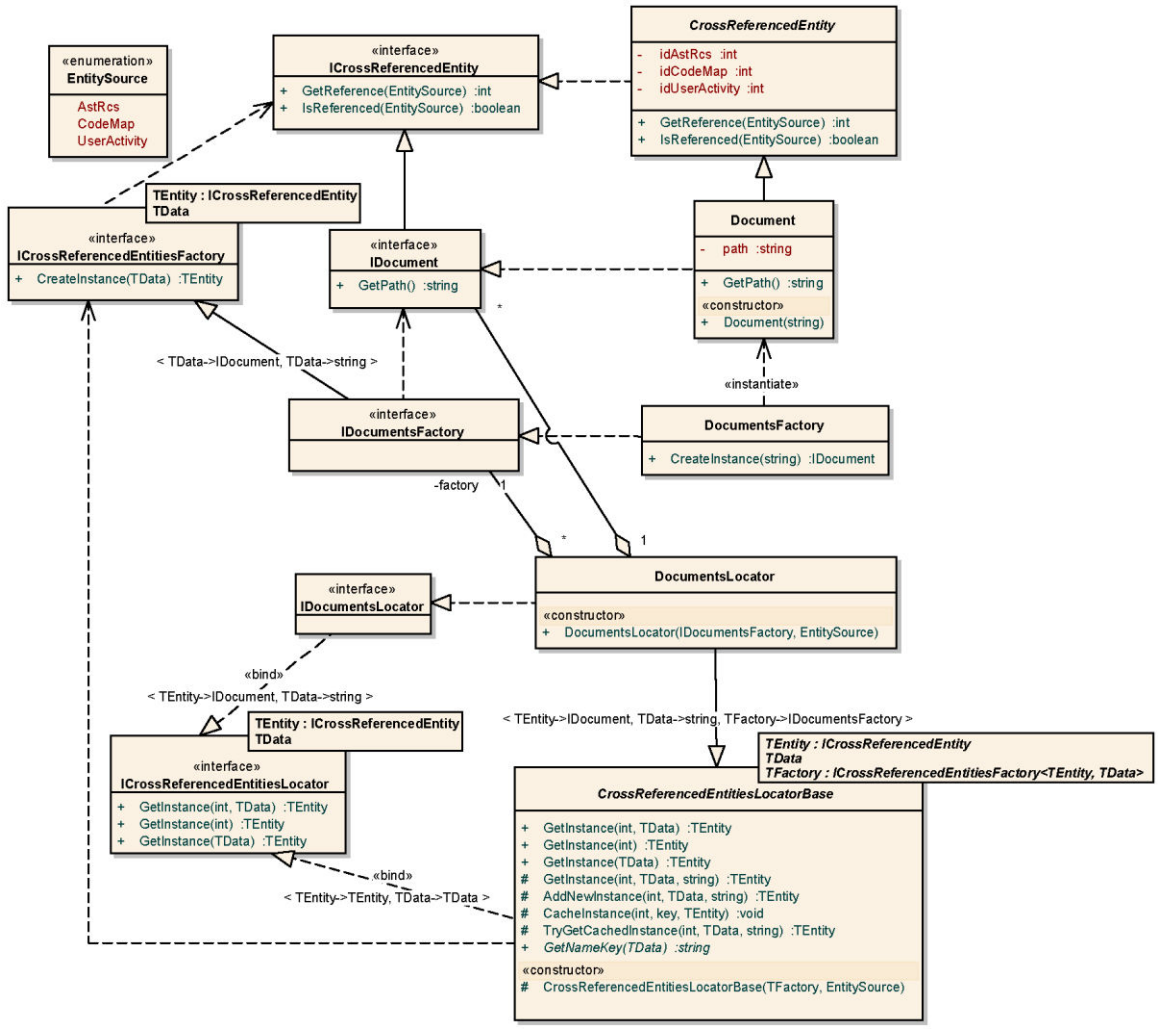
A.2. Reprezentácia grafov závislostí a ich entít

V implementácii riešenia reprezentujeme tieto tri grafy závislostí:

- graf explicitných závislostí na úrovní tried zdrojového kódu – import z DGML súboru,
- graf explicitných závislostí súborov zdrojového kódu – mapovanie tried na súbory,
- graf implicitných závislostí súborov zdrojového kódu.

Grafy závislostí vytvárame pomocou týchto troch rôznych zdrojov údajov:

- exportovaný DGML súbor z Microsoft Visual Studio,
- webová služba *AST-RCS* infraštruktúry projektu PerConIK,
- lokálna databáza alebo webová služba aktivít programátorov infraštruktúry PerConIK.



Obrázok A-3 Diagram tried reprezentácie súboru zdrojového kódu, jeho továreň a lokátor.

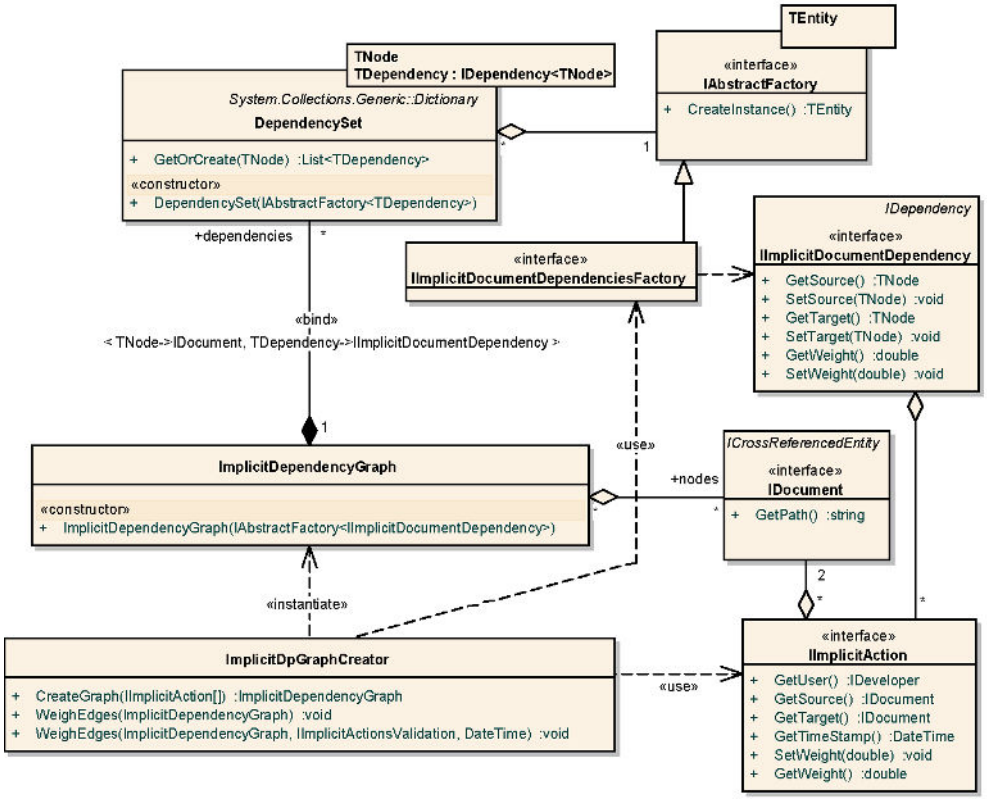
Viacnásobne identifikované entity

Výstupné dáta z uvedených zdrojov sme pre overenie našej metódy potrebovali rôzne kombinovať, a tak párovať rovnaké entity, ktoré v nich vystupovali, hoci boli rôzne identifikované. Entity, s ktorými pracujeme sú v implementácii reprezentované týmito rozhraniami:

- súbory zdrojového kódu – *IDocument*,
- softvérové projekty – *IIdeSolution* a *IIdeProject*,
- programátori – *IDeveloper*.

Entity sme zovšeobecnili do spoločného typu *ICrossReferencedEntity*, pomocou ktorého reprezentujeme jednu entitu (napr. jeden programátor) vždy jednou inštanciou. A to aj v prípade, ak sme ju spracovali z dvoch rôznych zdrojov. Inštancie týchto entít vytvárame pomocou tovární (*ICrossReferencedEntitiesFactory,* návrhový vzor *Abstraktná továreň*). Uchovávanie existujúcich inštancií a prevenciu vytvárania viacerých inštancií pre rovnakú entitu (napr. rovnaký zdrojový súbor alebo programátor) zabezpečuje *lokátor*. Ten pri požiadavke o entitu najprv overí, či už neexistuje a vráti existujúcu inštanciu, inak ju pomocou dodanej továrne vytvára. Obrázok A-3 uvádza diagram tried s entitou súboru zdrojového kódu (*IDocument*), jej realizáciou (*Document*), továrňou (*DocumentsFactory*) a lokátorom (*DocumentsLocator*). Zvyšné uvedené entity v implementácií sú reprezentované analogicky vlastnými rozhraniami, továrňami a lokátormi.

Knižnica *DP.DG.Implicit.Server* obsahuje vlastné implementácie realizácií rozhraní, tovární a lokátorov s použitím lokálnej databázy. Pred vytvorením novej inštancie najprv lokátor kontroluje vlastný zoznam inštancií, potom dopytuje lokálnu databázu, či entita skutočne neexistuje, a až potom vytvorí novú inštanciu a uloží ju do databázy.



Obrázok A-4 Diagram tried reprezentácie grafu implicitných závislostí.

Graf implicitných závislostí

Graf implicitných závislostí reprezentujeme triedou *ImplicitDependencyGraph* v knižnici *DP.DG.Implicit*, obsahuje množinu vrcholov a hrán medzi nimi. Každá hrana *IImplicitDocumentDependency* obsahuje agregované závislosti súborov *IImplicitAction* (v návrhu metódy označovaných ako d_{imp}). Vytvorenie grafu implicitných závislostí zabezpečuje trieda *ImplicitDpGraphCreator*. Obrázok A-4 uvádza diagram tried reprezentácie grafu implicitných závislostí.

A.3. Identifikácia implicitných závislostí z aktivít programátora

Implicitné závislosti reprezentujeme v zdrojovom kóde rozhraniami a ich realizáciami od *IImplicitAction* a jeho potomkov:

- *ITimedUserDocumentAction* – $d_{imp,time}$,
- *IContentUserDocumentAction* – $d_{imp,content}$,
- *ICommitAction* – $d_{imp,commit}$.

Identifikáciu závislostí vykonáva trieda *ImplicitActionsCreator* pomocou *UserActivityModel* a modelu vývojového prostredia *IdeModel*. Činnosť programátora (*IDeveloper*) je rozdelená do viacerých vývojových prostredí podľa projektov, na ktorých pracuje. Rozhranie *IIdeOperationEvent* je realizované záznamami aktivít, ktoré sú vstupom pre identifikáciu závislostí. Obrázok A-5 uvádza diagram týchto tried.

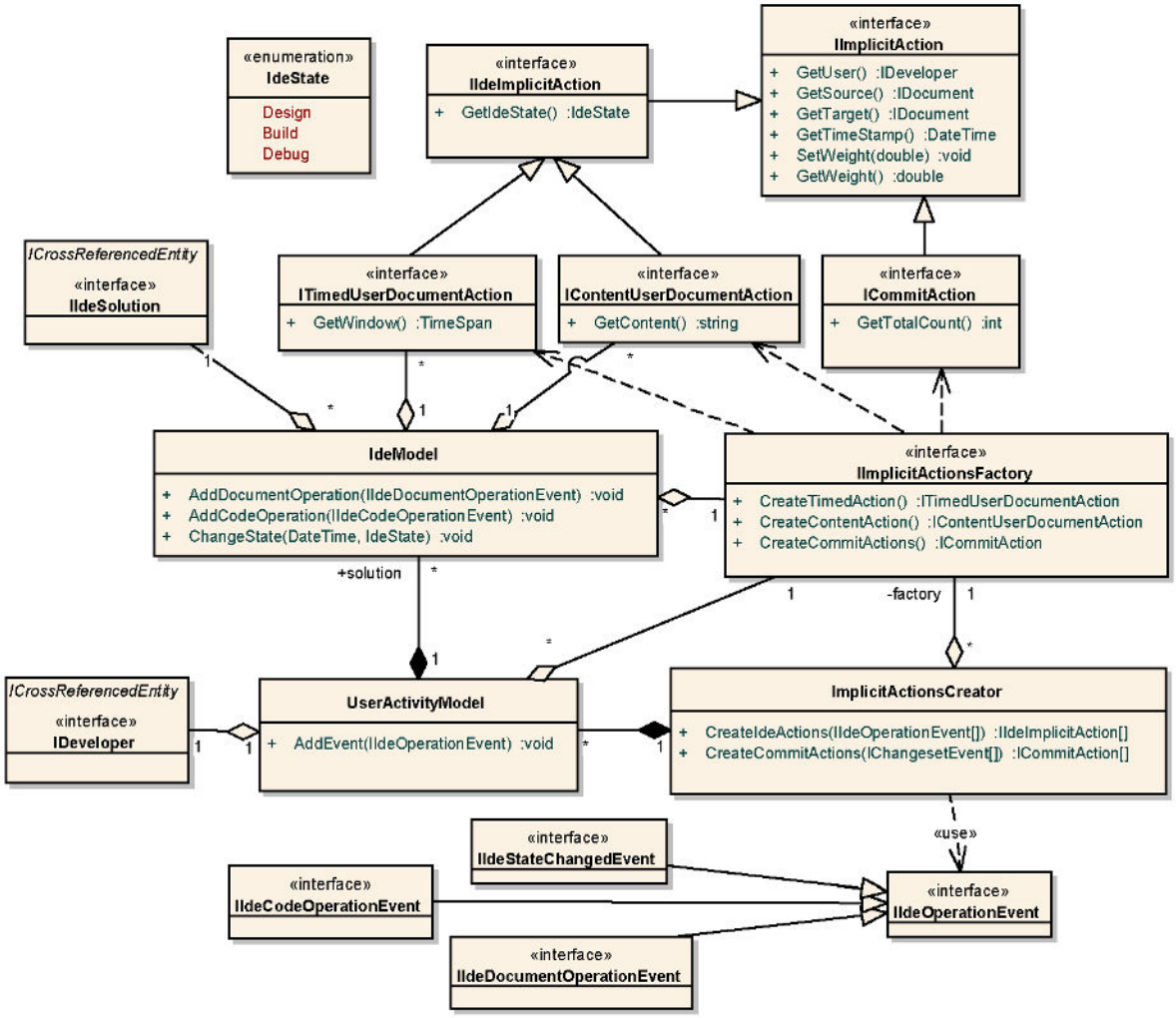
Proces identifikácie implicitných závislostí z kapitoly 6.1 uvádzame sekvenčnými diagramami na obrázkoch A-6 a A-7. Aplikácia používajúca knižnice s grafom implicitných závislostí najprv získa aktivity pomocou *IActivityService*. Z aktivít pomocou triedy *ImplicitActionsCreator* získa implicitné závislosti. *ImplicitActionsCreator* rekonštruuje používateľovu aktivitu s *UserActivityModel* a *IdeModel*. Po spracovaní všetkých aktivít aplikácia získa zoznam implicitných závislostí *IImplicitAction*.

Z identifikovaných závislosti vytvárame graf implicitných závislostí pomocou triedy *IImplicitDpGraphCreator*. Ten pre každú závislosť overí, či už obsahuje závislé súbory medzi vrcholmi. Ak nie, tak ich pridá, a zároveň pridá hranu medzi týmito súbormi do grafu (pokiaľ ešte neexistuje). Spracovanú závislosť pridá do zoznamu závislostí (metóda *AddAction*) hrany, keďže jedna hrana grafu agreguje viacero závislostí. Dodatočné váhovanie hrán prebieha agregovaním všetkých závislostí hrán, príp. aj ich validovaním pomocou *ForgettingValidation* (funkcia zabúdania, kapitola 5.1.2).

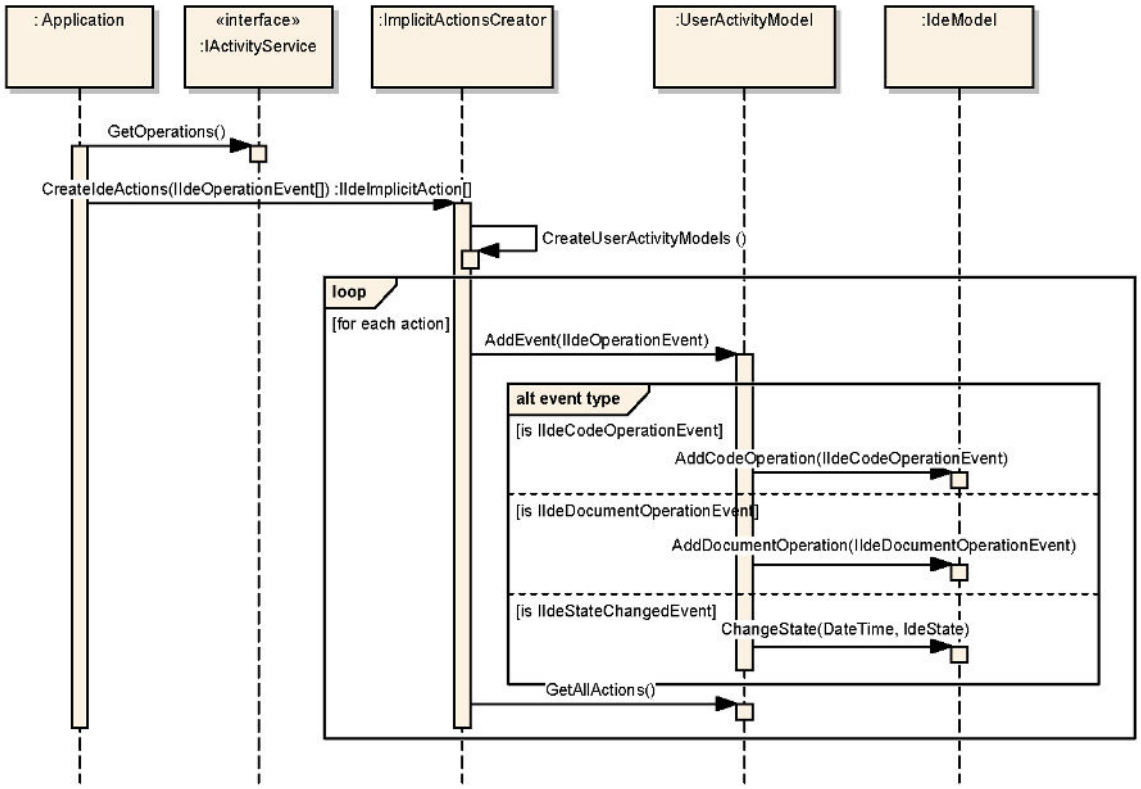
A.4. Ostatné časti zdrojového kódu

V zdrojovom kóde riešenia sa nachádzajú aj tieto projekty:

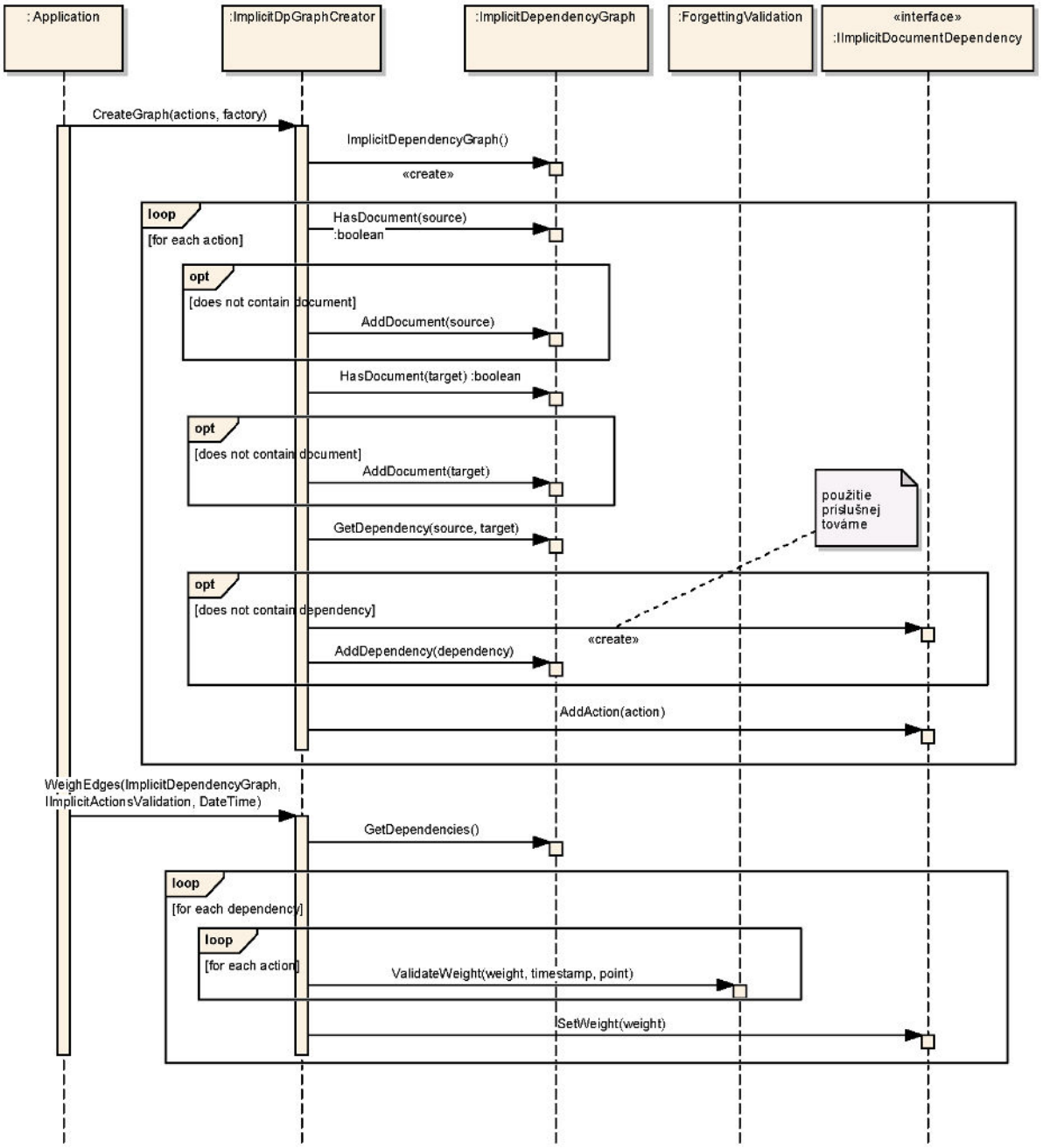
- *DP.Console.Exp1* – vyhodnotenie prvého experimentu,
- *DP.Console.ExpEval* – vyhodnotenie výsledkov druhého experimentu,
- *DP.ImplicitGraph.Service* – rozpracovaná systémová služba grafu implicitných závislostí,
- *DP.Web.Exp.** – rozpracovaná webová aplikácia pre tretí experiment overenia metódy,
- *DP.WPF* – prototyp vizualizácie grafu implicitných závislostí.



Obrázok A-5 Diagram tried implicitných závislostí a ich identifikácie modelovaním vývojového prostredia.



Obrázok A-6 Sekvenčný diagram získania aktivít programátorov zo služieb infraštruktúry projektu PerConIK a identifikácia implicitných závislostí modelom vývojového prostredia.



Obrázok A-7 Sekvenčný diagram vytvorenia grafu implicitných závislostí a validácie jeho hrán z identifikovaných implicitných závislostí.

Príloha B

Opis dát softvérových projektov pre vyhodnotenie práce

Pre vyhodnotenie metódy predstavenej v tejto práci sme použili zaznamenané aktivity programátorov viacerých študentských softvérových projektov prostredníctvom infraštruktúry projektu PerConIK. Projekty sa odlišujú vo veľkosti tímov a projektov, skúsenostiach ich členov, či aj v množstve zaznamenaných údajov. Pre každý projekt uvádzame krátky opis, charakteristiku a jeho trvanie. Tabuľka B-1 uvádza výsledky vyhodnotenia metrík zdrojového kódu pomocou nástroja Microsoft Visual Studio a celkový počet zaznamenaných aktivít práce so súbormi zdrojového kódu. Obrázok B-1 uvádza mesačné rozdelenie počtu zaznamenaných aktivít.

Projekt A

- Popis: Generátor informačných značiek pre projekt PerConIK.
- Typ projektu: webová aplikácia, systémová služba, knižnice
- Použité technológie: C#, .NET, ASP.NET MVC
- Veľkosť tímu: 3 programátori (2 sledovaní)
- Zaznamenaná aktivita: júl 2013 – máj 2014

Projekt B

- Popis: Knižnica pre údržbu informačných značiek pre projekt PerConIK.
- Typ projektu: knižnice
- Použité technológie: C#, .NET
- Veľkosť tímu: 1 programátor (sledovaný)
- Zaznamenaná aktivita: september 2013 – máj 2014

Projekt C

- Popis: Zdrojový kód implementovaný v rámci tejto práce.
- Typ projektu: knižnice, desktopová aplikácia, systémová služba
- Použité technológie: C#, .NET
- Veľkosť tímu: 1 programátor
- Zaznamenaná aktivita: september 2013 – máj 2014

Projekt TP1

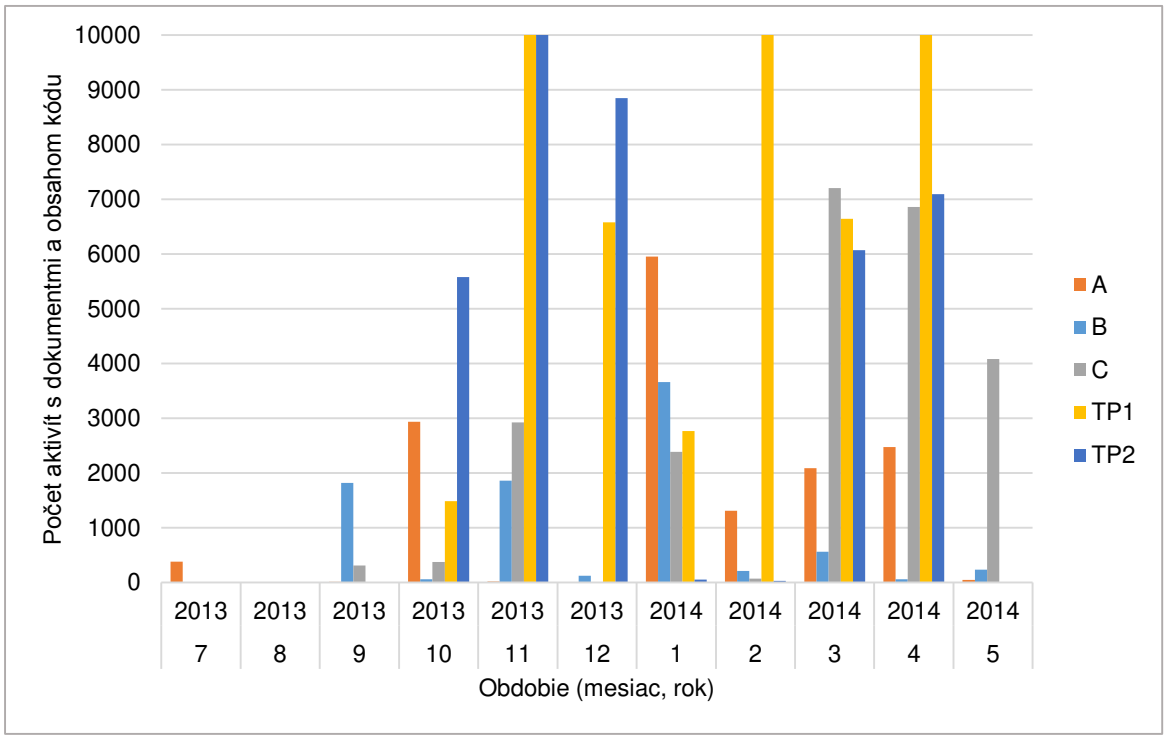
- Popis: Tímový projekt zaznamenávania implicitnej spätnej väzby používateľa.
- Typ projektu: knižnice, webová služba, webová aplikácia.
- Použité technológie: C#, .NET, ASP.NET MVC, WinForms
- Veľkosť tímu: 7 programátorov (2 sledovaní, 1 vyhodnocovaný)
- Zaznamenaná aktivita: október 2013 – máj 2014

Projekt TP2

- Popis: Tímový projekt vyhodnocovania zdrojového kódu a informačných značiek.
- Typ projektu: webová aplikácia.
- Použité technológie: C#, .NET, ASP.NET MVC
- Veľkosť tímu: 7 programátorov (4 sledovaní, 1 vyhodnocovaný)
- Zaznamenaná aktivita: október 2013 – máj 2014

Tabuľka B-1 Výsledky metrík zdrojového kódu študentských projektov a celkový počet zaznamenaných aktivít v období júl 2013 – máj 2014.

Projekt	Počet riadkov	Index udržovateľnosti	Cyklomatická zložitosť	Počet zaznamenaných aktivít
A	2402	82	1285	15215
B	4383	74	2164	8584
C	4993	81	3213	24213
TP1	5342	81	1839	55877
TP2	3721	81	1779	48717



Obrázok B-1 Mesačný počet zaznamenaných aktivít na jednotlivých študentských projektov.

Príloha C

Koncept analýzy priestoru informačných značiek pre prehliadku zdrojového kódu

Vo fáze návrhu metód vyhodnocovania softvéru z aktivít programátora a kontextu vývoja softvéru sme navrhli metódu poloautomatickej analýzy priestoru informačných značiek v projekte PerConIK. Cieľom metódy je identifikácia zaujímavých miest v zdrojovom kóde pre prehliadku. Tento návrh nakoniec nebol realizovaný.

Zdrojový kód softvérového projektu, okrem riadiacich prvkov a štruktúr, obsahuje aj iné informácie, ktoré programátori uviedli počas jeho tvorby. Jednoduchým príkladom je písanie komentárov, ktoré by mali vyjadrovať dôvody zvoleného riešenia problému alebo vyjasniť inému programátorovi problematiku. Informácie o zdrojovom kóde môžeme získavať automaticky syntaktickou analýzou zdrojového kódu, alebo vyhodnocovaním aktivity programátora. Získané informácie je však potrebné uchovať so zdrojovým kódom a náročné ich ďalej udržiavať.

Informačné značky v projekte PerConIK uchovávajú informácie viazané k softvérovým súčiastkam a môžu byť vytvorené buď manuálne programátorom, alebo automaticky vygenerované z jeho aktivít a zdrojových kódov sledovaného projektu. Informačné značky predstavujú metadáta o zdrojovom kóde, jeho obsahu, aktivite programátora a kontexte reprezentované štruktúrovaným komentárom. Tým pádom sú komplexným zdrojom informácií o softvérovom projekte, v ktorých môžeme taktiež hľadať prepojenia na vlastnosti jeho zdrojového kódu. To nás motivovalo navrhnúť metódu, ktorou dokážeme určiť problematické miesta v zdrojovom kóde využitím informácií poskytnutých informačnými značkami, vychádzajúc z nasledujúcej hypotézy:

> *Prehľadávaním priebehov výskytu informačných značiek a ich filtrovaním dokáže analytik identifikovať sekvencie aktivít programátora nad softvérovými súčiastkami, ktoré vedú k vzniku problematických miest v nich.*

Pre veľký rozmer priestoru informačných značiek a ich riedkosť ich nedokážeme vyhodnocovať automaticky a dedukovať vlastnosti zdrojového kódu. Preto sme navrhli vytvoriť metódu pre poloautomatickú prácu s informačnými značkami. Na základe hypotézy predpokladáme, že manažér (alebo analytik) dokáže sledovaním priebehov výskytu informačných značiek identifikovať v zdrojovom kóde miesta s podobnými vlastnosťami, alebo miesta, ktoré chce hlbšie analyzovať. Našu metódu môžeme rozdeliť na tieto časti:

- Zmenšenie priestoru informačných značiek pomocou filtrov.
- Vizualizácia pre navigáciu v priestore informačných značiek.
- Definovanie a aplikovanie pravidiel nad informačnými značkami.

C.1. Filtrovanie informačných značiek

Priestor informačných značiek je aj napriek ich vyššej úrovni abstrakcie nad záznamami aktivít programátorov stále veľký, viažu sa k súčiastkam zdrojového kódu a vznikajú v čase podľa programátorovej aktivity. Informačné značky môžeme rozlišovať podľa nasledujúcich kritérií:

- druh značky – manuálna alebo generovaná,
- čas vzniku,
- autor značky – programátor,
- softvérová súčiastka, ku ktorej značka prislúcha,
- konkrétny typ značky a jej obsah – napr. záujem o metódu, odporúčanie,
- stav značky – vznik, úprava, zánik.

Uvedené kritériá sme navrhli použiť pri filtrovaní značiek ako ich ohraničenie na:

- uvažované softvérové projekty,
- autori značky,
- časové okno posledných úprav značiek,
- súčiastky zdrojového kódu, ku ktorým sú značky ukotvené,
- typy značiek a ich obsah,
- spôsob vzniku.

Definovaním filtra ako kombinácie ohraničení dopytujeme priestor značiek a vyberáme iba značky spĺňajúce všetky ohraničenia filtra. Výsledkom je zmenšený priestor značiek, ktoré môžeme prehliadať, alebo ich aj ďalej filtrovať.

C.2. Vizualizácia priebehov výskytu informačných značiek

Informačné značky vznikajú a menia sa v čase, čo považujeme ako priebehy ich výskytu. Analytik dokáže pomocou filtrov vybrať iba relevantné značky podľa jeho záujmu. Vhodná vizualizácia potom umožní analytikovi získať prehľad nad výskytom značiek. Príkladom môže byť výskyt značiek určitého typu v určité dni, alebo tesne pred termínom dokončenia úlohy.

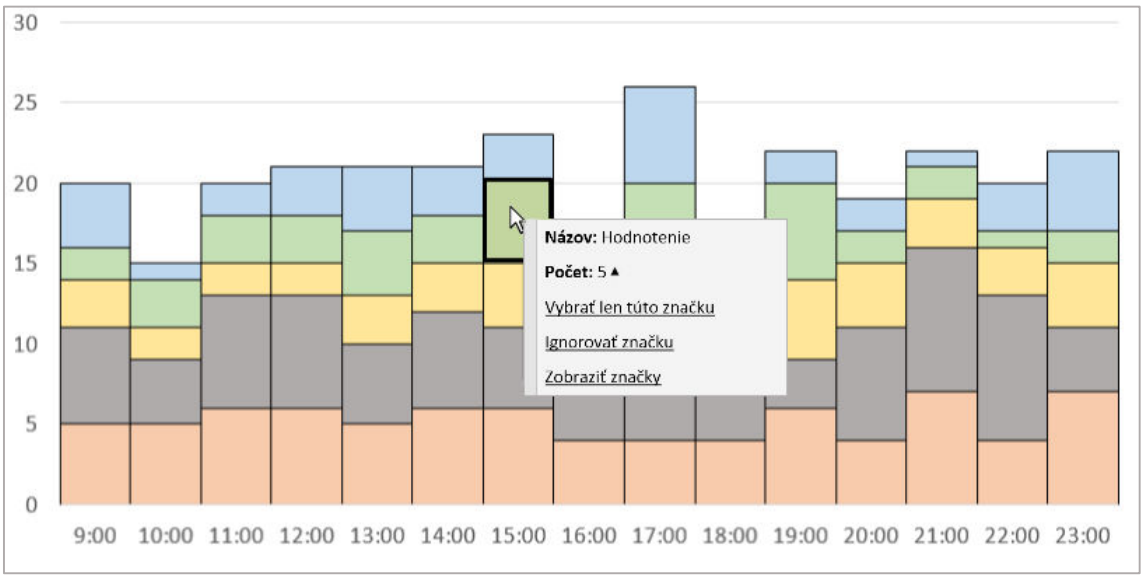
Pod výskytom značky považujeme jej existenciu v čase. Zjednotením značiek rovnakého typu do spoločných skupín potom môžeme vizualizovať zmeny v existencii značiek, t.j. ako priebehy ich výskytu. Vytvorením značky rastie veľkosť skupiny značiek rovnakého typu, odstránením značky klesá.

Grafické znázornenie priebehov výskytu značiek sme navrhli vykonať histogramom značiek v čase, ktoré spĺňajú ohraničenia filtra. Obrázok E-1 znázorňuje návrh grafického zobrazenia priebehov. Samotný stĺpec zodpovedá počtu existujúcich značiek v časovej jednotke. Stĺpec je farebne rozdelený na úseky, ktoré zodpovedajú typom značiek, z ktorých sa celý stĺpec skladá. Veľkosť farebných úsekov sa priebežne mení podľa počtu existujúcich značiek v zvolenom časovom úseku.

Histogram priebehov výskytu značiek doplníme o zoznam značiek, ktoré sú v ňom obsiahnuté. Zvolením úsekov v histograme analytik mení obsah zoznamu, v ktorom môže prehľadávať detaily značiek, a zároveň ich ignorovať. Ignorovanie značky vo vizualizácii prebieha v skutočnosti dodatočným vytvorením filtra, ktorý ďalej ohraničuje už vyfiltrované značky. Obrázok E-2 znázorňuje zoznam značiek zoskupený podľa tried, ku ktorým sú kotvené.

Zoznam značiek je možné ďalej zoskupiť podľa kritérií:

- autor značky,
- súčiastka, ku ktorej je značka kotvená,
- typ značky.



Obrázok E-1 Návrh grafického zobrazenia priebehov výskytu informačných značiek.

Zoskupiť podľa: Trieda > Vyberte...

Typ informačnej značky	Autor	Čas	Detaily
OrderWindow			
TODO	Juraj	12.1.2014 9:00	...
Úprava kódu	Ondrej	12.2.2014 9:20	...
CustomerEntity			
Úprava kódu	Martin	12.1.2014 14:20	...
Hodnotenie kódu	Peter	14.1.2014 12:00	...
Skopírovanie kódu	Martin	14.1.2014 13:42	...
Úprava kódu	Martin	14.1.2014 13:43	...

Obrázok E-2 Návrh zobrazenia vyfiltrovaných informačných značiek v zozname.

Vizualizáciu priebehov môžeme rozdeliť na 3 časti: manažovanie filtrov, histogram značiek a zoznam zvolených značiek. Pre každú identifikovanú značku a jednotku zdrojového kódu, ku ktorej je kotvená, má analytik možnosť prezerať detaily. Cieľom metódy je umožniť analytikovi identifikovať jednotky zdrojového kódu vhodné pre analýzu. Tie analyzuje a vyhodnocuje ich vlastnosti, ktoré opäť vyjadrí informačnou značkou.

C.3. Pravidlá priebehov výskytu informačných značiek

Pomocou postupného vytvárania filtrov dokáže analytik identifikovať súčiastky, ktoré potom bližšie analyzuje (napr. na úrovni zdrojového kódu) a priradí im ďalšiu informačnú značku. Vytvorenými filtrami identifikuje sekvencie značiek, ktoré mohli vplývať na vlastnosť súčiastky, ktorú v nej následne identifikoval. Ak by sa situácia v priebehoch výskytu značiek opakovala, analytik by hľadal rovnaké sekvencie. Preto mu umožníme vytvoriť pravidlá zo zadaných filtrov a výsledkov pre použitie neskôr.

Ak analytik identifikuje sekvencie značiek, ktoré vplývajú na vlastnosť skúmanej softvérovej súčiastky, potom v skutočnosti identifikoval softvérovú metriku vychádzajúcu z aktivít programátora.

Pri vytváraní pravidiel musíme vykonať nasledujúce úpravy vo filtroch:

- Zmeniť absolútne ohraničenia času &na relatívne, aby boli filtre platné aj v budúcnosti.
- Zovšeobecniť konkrétne hodnoty vo filtroch do premenných, napr. aby sa pravidlo mohlo týkať ktoréhokoľvek programátora.
- Pridať filtre pre všetky relevantné značky, ktoré analytik identifikoval.

Použitie pravidiel spočíva z výberu vytvorených pravidiel a zvolenia času, od ktorého sa majú uplatniť. Následne sa pravidlom prehľadá priestor priebehov výskytu informačných značiek. Ak sa v priestore identifikovali sekvencie spĺňajúce pravidlo, tak sa ponúknu analytikovi v podobe výsledkov, ktoré môže prezerať podobne ako pri pôvodnom filtrovaní informačných značiek zobrazením histogramu a zoznamu značiek s ich detailmi.

Príklady možných pravidiel

Naša metóda umožňuje analytikovi definovať pravidlá nad informačnými značkami, ktoré pozostávajú z filtrov. Uvedieme niekoľko príkladov pravidiel, ktoré vyjadrujú existenciu značiek postupne v čase:

- Nesprávne splnenie úlohy programátorom:
  1. Dokončenie metódy (značka *TODO*) pre programátora *A*.
  2. Programátor *A* vytvoril zdrojový kód o dĺžke 100 riadkov.
  3. Hodnotenie zdrojového kódu programátorom *B*.
  4. Programátor *B* pridá značku *TODO* na prerobenie kódu.
- Kontrola opravy chybného kód:
  1. Triedu v zdrojovom kóde upravil programátor a od 23:00 do 6:00 ráno.
  2. Programátor A skopíroval kód do tejto triedy.
  3. Programátor *B* identifikoval chybu v tejto triede.
  4. Programátor A upravil kód.
  5. Programátor A odstránil značku o pachu.

- Pravdepodobná chyba v prevzatom zdrojovom kóde:
  1. Programátor A skopíroval kód z triedy *X* do triedy *Y*.
  2. Programátor *B* označil v triede *X* chybu.
  3. Programátor *C* opravil chybu v triede *X*.
- Kód na revíziu:
  1. Programátor A vytvoril kód v noci.
  2. Kód, ktorý programátor vytvoril má viac ako 100 riadkov.
  3. Tento kód má vysokú zložitosť.
  4. Programátor *B* pridal tomuto kódu malé hodnotenie.

Použitím týchto pravidiel dokáže analytik identifikovať značky, ktoré spĺňajú kritériá, a tak nájsť softvérové súčiastky pre jeho analýzu.

Príloha D

Prehľad metód dolovania v dátach

V tejto prílohe uvádzame prehľad metód dolovania v dátach, ktorých uplatnenie nachádzame v generátore informačných značiek projektu PerConIK a analyzovali sme ich aj pre možné použitie v návrhu a riešení našej práce.

Rozšírenie informačných technológií a počítačov umožnilo získavanie a zaznamenávanie veľkého množstva dát. Dáta sú však často ukladané v podobe, ktorá nie je pre človeka jednoducho čitateľná a použiteľná pre získanie znalostí. Dolovaním v dátach máme možnosť získať použiteľné informácie a znalosti, ktoré nemusia byť vopred zrejmé v sledovanej doméne, no pomôžu nám zlepšiť procesy (napr. obchodné stratégie) alebo odhaliť skryté vzťahy. Ako príklad môžeme uviesť, kedy dolovaním v dátach histórie nákupov v obchodných reťazcoch bolo zistené, že zákazníci kupujúci detské plienky zakúpili vo svojom nákupe aj alkoholický nápoj. S takouto informáciou mohol obchodný reťazec lepšie adaptovať rozloženie produktov pre týchto zákazníkov. Iným príkladom je sledovanie závislosti vykonávania športovej aktivity od stavu počasia (Witten, et al., 2011) alebo poradie navštevovania webových stránok používateľom pre ich prednačítanie (Demovič, et al., 2012).

Podobne môžeme pracovať aj s dátami z oblasti vývoja softvéru, kedy máme k dispozícii dáta o zdrojovom kóde, ale taktiež aj činnosti programátorov a situácií, za ktorých programátori vytvorili výsledný zdrojový kód. To nás motivuje hľadať využitie metód dolovania v dátach aj na dáta zaznamenané počas vývoja softvéru. Odhalenie skrytých vzťahov záznamov o činnosti programátorov a ich výsledkov nám otvára priestor pre výskum nových prístupov merania softvéru.

D.1. Metódy dolovania v dátach

Metódy dolovania v dátach rozdeľujeme podľa ich úlohy na prediktívne alebo deskriptívne. Deskriptívne metódy sú určené na zatriedenie inštancií údajov, prediktívne na odvodzovanie ich vlastností. V nasledujúcich podkapitolách uvedieme vybrané metódy dolovania v dátach.

Klasifikácia

Klasifikácia patrí medzi metódu riešenia úlohy predikcie. Metóda zobrazuje vstupné dáta do vopred definovaných tried na základe klasifikátorom odvodených pravidiel. Metóda prebieha v dvoch krokoch:

1. Učenie klasifikátora charakteristík tried na základe zatriedených dát – trénovacia množina.
2. Použitie klasifikátora na určenie tried dát s neurčenými triedami – testovacia množina.

Inštancie dát množiny $D = \{ d_1, d_2, \ldots, d_n \}$ sú *n*-rozmerné vektory $d_i = (x_1, x_2, \ldots, x_n)$. Úlohou klasifikácie (Han, et al., 2001) je identifikovať zobrazenie f množiny D na množinu tried $C = \{ C_1, C_2, \ldots, C_n \}$:

$$f : D \rightarrow C$$

Trieda C_j obsahuje práve tie inštancie z D, ktoré sa funkciou f zobrazia na ňu:

$$C_j = \{ d_i \,|f(d_i) = C_j, \forall\, d_i \in D\}$$

Klasifikátor sa učí vplyvy jednotlivých atribútov vektoru inštancie na výslednú triedu z trénovacej množiny. Klasifikátor tak vytvorí klasifikačné pravidlá určujúce klasifikačnú funkciu f. Ako klasifikátor môžeme použiť napr. modely Naïve Bayes, rozhodovacie stromy alebo *k*-najbližších susedov (Witten, et al., 2011).

Ako príklad klasifikácie môžeme uviesť predpoveď výšky príjmu obyvateľov krajiny, ktorých opíšeme atribútmi ich veku, pohlavia, vzdelania alebo aj zamestnania[17]. Triedy obyvateľstva môžeme určiť podľa hranice výšky platu. Klasifikátor učíme na trénovacej množine so známymi priradeniami inštancií k triedam, následne overíme úspešnosť na testovacej množine klasifikovaním inštancií bez priradených tried.

Zhlukovanie

Zhlukovanie zatrieďuje inštancie dát do tried podobne ako klasifikovanie, ale s rozdielom, že ich triedy nie sú vopred známe. Zhlukovanie je preto deskriptívnou metódou dolovania dát. Objekty v identifikovaných zhlukoch sú si navzájom podobné, resp. odlišné ak patria do rôznych zhlukov. Rozlišujeme štyri druhy zhlukov:

- neprekrývajúce sa zhluky – objekt patrí práve do jedného zhluku,
- prekrývajúce sa zhluky – objekt patrí aspoň do jedného zhluku,
- pravdepodobnostné zhluky – určujeme pravdepodobnosti zaradenia objektov do zhlukov,
- hierarchické zhluky – zhluky je možné ďalej zhlukovať .

Príkladom použitia zhlukovania je zatriedenie platobných transakcií. Časté alebo drahé transakcie môžu byť znakom podvodov. Ich zatriedenie do zhlukov nám umožní získať lepší pohľad na ich distribúciu a odhalenie charakteristík zhlukov (Han, et al., 2001).

[17] UCI Machine Learning Repository: Adult Data Set.
`http://archive.ics.uci.edu/ml/datasets/Adult`

Asociačné pravidlá

Asociačné pravidlá sú, podobne ako klasifikačné pravidlá, vyhodnotené z testovacej množiny údajov. Od klasifikačných pravidiel sa však líšia v tom, že:

- môžu predpovedať aj ostatné atribúty inštancií, nie len triedy,
- môžu predpovedať viacero atribútov naraz,
- sú aplikovateľné jednotlivo, nie len ako množina pravidiel.

Dolovanie asociačných pravidiel môžeme definovať (Han, et al., 2001) ako hľadanie všetkých pravidiel $X \rightarrow Y$ pre množinu inštancií $D = \{ d_1, d_2, \ldots, d_n \}$ kde $d_i = \{ I_{i1}, I_{i2}, \ldots, I_{ik} \}$ je inštancia s atribútmi $I_{ij} \in I$, kde $I = \{ I_1, I_2, \ldots, I_m \}$. Dvojica atribút-hodnota $p = (I_{ij}, v)$ vyjadruje, že atribút I_{ij} inštancie d_i nadobúda hodnotu v, kde $v \in I$. Potom vzor $X = (p_1, p_2, \ldots, p_n)$ je konkjukciou dvojíc atribút-hodnota p_k, ktoré musí inštancia d_i spĺňať aby sme pre ňu mohli odvodiť vlastnosti Y, t.j. $X \rightarrow Y$.

Na základe tejto definície a neobmedzovania množín X a Y si môžeme všimnúť, že pravidiel môže vzniknúť veľké množstvo (v najhoršom prípade variácie všetkých atribútov objektov). Preto sa pre výber pravidiel používajú metriky:

- *podpora* (angl. support) – pomer inštancií spĺňajúcich pravidlo zo všetkých inštancií množiny D,
- *spoľahlivosť* (angl. confidence) – pomer inštancií, ktoré spĺňajú vzor X, a zároveň Y zo všetkých inštancií, na ktoré je pravidlo aplikovateľné, t.j. spĺňajúcich vzor X.

Z dôvodu náročnosti dolovania týchto pravidiel boli navrhnuté viaceré algoritmy, napr. Apriori alebo FP-Tree (Witten, et al., 2011). Asociačné pravidlá umožňujú odhaliť v dátach skryté vzťahy. Ako príklad môžeme uviesť už spomenutý jav nakupovania zákazníkov obchodného centra. Iným použitím je odhaľovanie sekvencií a závislostí v dátach. V prípade domény vývoja softvéru môžeme asociačné pravidlá použiť pre hľadanie vzorov v aktivite programátorov.

D.2. Dolovanie v prúdoch dát

Uvedené metódy dolovania v dátach sú určené pre statické množiny dát, napr. relačné databázy alebo dátové sklady. V mnohých prípadoch však potrebujeme analyzovať dáta, ktoré sa v čase často menia alebo sa postupne generujú, t.j. vytvárajú prúd dát. Príkladom môže byť vyhodnocovanie vlhkosti a osvetlenia v miestnosti pomocou senzorov. Ak potrebujeme sledovať v týchto dátach vzory, potrebujeme ich vyhodnocovať v reálnom čase.

Prúd dát môžeme charakterizovať ako dočasne usporiadaný, rýchlo sa meniaci, obsiahly a prípadne nekonečný zdroj dát (Han, et al., 2001). Uchovávať históriu všetkých dát z prúdu by bolo nevhodné kvôli ich rastúcemu počtu. To podnietilo vznik viacerých techník, napr. vzorkovanie údajov na časti, pohyblivé okno, histogramy a dopyty nad prúdom. Vyhodnocovaním prúdov RDF trojíc sa napr. venujú autori v (Le-Phuoc, et al., 2012).

Uplatnenie dolovania v prúdoch dát nájdeme pri predpovedaní vývoja burzy alebo predajov v obchodných reťazcoch na základe časových radov udalostí. Iným príkladom dolovania v prúdoch je vyhodnocovanie sekvencií udalostí, napr. zvyky používateľa (Bamis, et al., 2010), jeho pohyby medzi miestnosťami v budove (Nazerfard, 2011; Le-Phuoc, et al., 2012) alebo prehliadanie Webu (Demovič, et al., 2012).

D.3. Diskusia

Dolovanie v dátach nám umožňuje v nich hľadať skryté vzťahy a súvislosti. V tejto kapitole sme uviedli vybrané prediktívne a deskriptívne metódy dolovania dát. V našom prípade vidíme použitie metód klasifikácie a dolovania asociačných pravidiel nad prúdmi aktivít programátora a kontextu z projektu PerConIK. Klasifikáciu môžeme uplatniť pri určovaní vlastností zdrojového kódu na základe aktivít, ktoré sa na ňom vykonali. Na druhej strane, dolovanie asociačných pravidiel nám môže poodhaliť skryté vzťahy v aktivitách. Metódu zhlukovania však neignorujeme, vidíme jej pravdepodobné uplatnenie pri zhlukovaní aktivít podľa času, ako predspracovanie pre dolovanie asociačných pravidiel.

Príloha E

Prehľad vlastností a objektovo-orientovaných metrík softvéru

V rámci analýzy aktuálneho stavu merania a vyhodnocovania softvéru sme sa venovali vyhodnocovaniu vlastností softvéru. V tejto prílohe uvádzame prehľad vlastností, ktoré sledujeme pre softvérový produkt a prehľad objektovo-orientovaných metrík zdrojového kódu.

E.1. Vlastnosti softvérového produktu

Vonkajšie vlastnosti

Vonkajšie vlastnosti charakterizujú softvérový produkt bez potreby poznania ich vnútornej štruktúry. Do úvahy však musíme zohľadniť prostredie, v ktorom sa produkt používa alebo vyskytuje. Ako príklad môžeme uviesť vlastnosť efektívnosti produktu. Komplexné vývojové prostredie môže pre začínajúceho programátora vytvárať dojem zbytočnej zložitosti. Začínajúci programátor nie je v tomto prostredí efektívny pri svojej práci. Naopak skúsený programátor, požadujúci množstvo nástrojov a možností v prostredí, zároveň majúci znalosť ovládania tohto prostredia, je v ňom efektívny. Jednoduché vývojové prostredie by pre neho nebolo vhodné riešenie pre zefektívnenie práce.

Pri vyhodnocovaní vonkajších vlastností uvažujeme:

- zainteresovaná strana – používateľ a vývojár sledujú odlišné vlastnosti,
- spôsob a kontext používania produktu.

Medzi vonkajšie vlastnosti produktov môžeme zaradiť nasledujúce vlastnosti. Pri definíciách a príkladoch vychádzame z existujúcich prác (Boehm, et al., 1976; Cavano & McCall, 1978; Bieliková, 2000; Anon., 2001):

- *Spoľahlivosť* – miera produktu vykonať požadované funkcionality za určitých podmienok s očakávanou presnosťou výsledkov:
  - Zdrojový kód sa skompiluje, načíta, spustí a vráti presné výsledky.
  - Softvér nespôsobí pri výpadku ani fyzické, ani ekonomické škody.
  - Opotrebovanie alebo starnutie pri softvéri neexistuje, budúce nespĺňanie kritérií spoľahlivosti môže byť spôsobené len z dôvodu chýb v pôvodných požiadavkách, návrhu, implementácií alebo nesprávnym použitím produktu.

- *Testovateľnosť* – miera úsilia potrebného na testovanie produktu a overenie či spĺňa očakávané vlastnosti:
  - Úsilie potrebné na testovanie softvéru pre zaistenie vykonávania požadovaných funkcií.
  - Úsilie potrebné pre zistenie či softvér vykazuje požadované správanie vo zvolených podmienkach použitia.
- *Efektívnosť* – miera využitia produktu pre splnenie cieľov a využitia prostriedkov systému poskytnutých pre primeraný výkon produktu:
  - Množstvo a zložitosť zdrojového kódu a výpočtových prostriedkov potrebných aby program vykonal požadovanú funkcionalitu.
  - Čas potrebný na vykonanie funkcionality produktu.
- *Použiteľnosť* – miera úsilia potrebného na používanie výrobku v určených podmienkach:
  - Zdrojový kód je použiteľný do tej miery, akej je spoľahlivý, efektívny a zrozumiteľný.
  - Úsilie potrebné na porozumenie, naštudovanie, ovládanie a používanie programu používateľom (pre neho v atraktívnej podobe).
- *Správnosť* – miera spĺňania špecifikácie a cieľov produktu pri používaní používateľom.
- *Prenositeľnosť* – miera úsilia potrebného na prenesenie produktu medzi rôznymi prostrediami:
  - Softvér je možné použiť na rôznych platformách a zariadeniach, než len na tých, na ktorých sa práve vykonáva. Produkt nie je zviazaný na konkrétnu platformu.
  - Úsilie potrebné pre prenesenie softvéru z jednej hardvérovej konfigurácie a systémového prostredia na iné.
- *Udržovateľnosť* – miera úsilia potrebného na modifikovanie produktu z dôvodu opravy, vylepšenia alebo adaptácie na zmeny v prostredí, požiadavkách alebo špecifikácie funkcionality. Udržovateľnosť závisí od iných vlastností produktu:
  - Možnosť upraviť produkt pre splnenie nových požiadaviek alebo opravenie chýb je jednoduchšie ak je kód zrozumiteľný, testovateľný a modifikovateľný.
  - Úsilie potrebné pre lokalizovanie, porozumenie a opravenie chyby v zdrojovom kóde.
  - Úsilie potrebné na ďalší vývoj a údržbu výrobku podľa meniacich sa potrieb zákazníka, ale aj meniaceho sa okolia.
- *Bezpečnosť* – miera odolnosti produktu voči nepovolenému prístupu k údajom:
  - Kontrolovaný prístup k údajom softvéru nepovoleným osobám.
  - Odolnosť voči neoprávneným zásahom do systému.
- *Znovupoužiteľnosť* – miera použiteľnosti produktu v iných aplikáciách než bola jeho pôvodná, pri zachovaní požadovaných vlastností prostredia, v ktorom je používaný, alebo pri obmedzení poskytovaných možností produktu:
  - Ako náročné je použiť existujúci softvérový komponent iným systémom.
  - Ako sa obmedzí poskytovaná funkcionalita softvérového komponentu, keď sa použije v prostredí s inými vlastnosťami, než bol preň určený.
- *Interoperabilita* – miera úsilia potrebného pre prepojenie produktu s iným produktom:
  - Úsilie potrebné pre zabezpečenie spolupráce dvoch systémov.
- *Modifikovateľnosť* – miera úsilia potrebného pre vykonanie zmien produktu:
  - Ako náročné je v prípade potreby priniesť zmeny do kódu.
  - Úsilie potrebné pre modifikovanie systému v prevádzke.
- *Funkčnosť* – miera schopnosti produktu poskytnúť funkcie, ktoré spĺňajú požadované a očakávané potreby v čase jeho používania.

Vnútorné vlastnosti

Pohľad na produkty z vnútra je oproti sledovaniu ich vonkajších vlastností jednoduchší, pretože pracujeme priamo s ich obsahom a hodnotami, ktoré vznikli počas ich vytvárania a nezohľadňujeme použitie produktu po jeho dokončení. Ako príklad môžeme uviesť meranie dĺžky zdrojového kódu, vďaka ktorému vieme identifikovať, ktoré funkcionality sú implementované príliš dlhým zdrojovým kódom.

Napriek jednoduchosti merania vnútorných vlastností sa však stretávame s problémom ich interpretácie, určenia ich hraníc a celkovo ich použitím pri vývoji a údržbe softvéru. Ako príklad môžeme uviesť zložitosť zdrojového kódu, kde existuje viacero pohľadov, buď z hľadiska časovej i pamäťovej zložitosti, alebo aj na základe zložitosti toku riadenia.

Ako najčastejší produkt softvérového projektu uvažujeme zdrojový kód a softvérové súčiastky. Medzi ich vnútorné vlastnosti zaraďujeme najmä nasledujúce dve (Fenton & Pfleeger, 1998):

- *Veľkosť* – miera veľkosti produktu v nasledujúcich rozmeroch:
  - dĺžka – množstvo zdrojového kódu,
  - funkcionalita – možnosti ponúkané používateľovi,
  - algoritmická zložitosť algoritmu riešiaceho problém,
  - výnimočnosť – miera znovupoužitia v rámci produktu.
- *Štruktúrovanosť* – zložitosť štruktúry produktu:
  - zložitosť toku riadenia,
  - zložitosť toku údajov,
  - zložitosť štruktúry dát.

Veľkosť produktu sa ako jeho vnútorná vlastnosť často používa pre vyhodnotenie a predpovedanie úsilia, ceny alebo produktivity programátorov, keďže existuje väzba medzi týmito vlastnosťami. Pre exaktnejšie určenie veľkosti zohľadňujeme aj algoritmickú zložitosť implementovanej funkcionality. Veľkosť zdrojového kódu môže byť však umelo zvyšovaná, napr. redundantným kopírovaním existujúceho kódu namiesto jeho vhodnejšej organizácie, alebo aj jeho automatickým generovaním nástrojmi (napr. vytváranie používateľských rozhraní). Zaradením výnimočnosti kódu zabezpečíme relevantné vyhodnocovanie veľkosti produktu.

Zložitosť produktu môžeme rozdeliť na štyri typy, čím adresujeme problém interpretácie tejto vlastnosti:

- *Výpočtová zložitosť problému* – náročnosť riešenia problému.
- *Algoritmická zložitosť* – zložitosť algoritmu riešiaceho problém:
  - časová náročnosť – čas vykonávania algoritmu,
  - pamäťová náročnosť – pamäť spotrebovaná algoritmom.
- *Zložitosť štruktúry produktu* – vnútorná organizácia produktu,
- *Kognitívna zložitosť* – úsilie potrebné pre porozumenie produktu.

Veľkosť a štruktúrovanosť patria medzi základné vnútorné vlastnosti produktov. Priamym meraním produktov (najmä zdrojového kódu a softvérových súčiastok) však vieme určiť aj nasledujúce vlastnosti:

- *Modulárnosť* – miera organizácie produktu na zameniteľné časti.
- *Rozšíriteľnosť* – miera možnosti rozšírenia produktu o novú funkcionalitu.
- *Zviazanosť* – miera prepojenia softvérových súčiastok, ich väzieb medzi nimi.
- *Súdržnosť* – miera uzavretosti súčiastok, ich nezávislosti od ostatných.

- *Chybovosť* alebo *syntaktická správnosť* – miera množstva chýb v produkte, ktoré vieme identifikovať ešte pred jeho spustením, napr. syntaktické chyby v zdrojovom kóde.
- *Konzistentnosť* – miera jednotnosti produktu, ako v jeho zápise (notácia, terminológia), tak aj voči jeho požiadavkám.

Uviedli sme vnútorné vlastnosti produktu súvisiace najmä so zdrojovým kódom a softvérovými súčiastkami, na ktorých vyhodnotenie sa zameriavame v našej práci. Pre iné druhy produktov (dokumenty alebo testy softvéru) určujeme aj ďalšie vlastnosti, napr. konzistentnosť.

E.2. Objektovo-orientované metriky softvérového produktu

Príchod objektovo-orientovaných programovacích jazykov prinútil vznik špecifických metrík. Vyjadrenie zložitosti na základe počtu riadkov alebo riadiaceho toku prestalo byť dostačujúce. Autori v (Chidamber & Kemerer, 1994) argumentujú, že existujúce metriky nebrali do úvahy pojmy objektovo-orientovaného prístupu ako triedy, dedenie, zapuzdrenie alebo posielanie správ. Objektovo-orientovaná paradigma taktiež chápe (Chidamber & Kemerer, 1994):

- *objekty* – indivíduá sveta a ich vlastnosti,
- *triedy* – množiny objektov majúce spoločné vlastnosti,
- *metódy tried* – operácie nad objektmi triedy.

Preto bolo uvedených viacero metrík (Morris, 1989; Lieberherr, et al., 1988), podobne aj nasledujúcich šesť metrík od Chidambera a Kemerera (Chidamber & Kemerer, 1994).

Váhované metódy tried

Metrika (angl. Weighted Methods per Class, WMC) zastupuje vyhodnotenie zložitosti tried. Pre metódy $M_1, M_2, \dots, M_n$ triedy C, ktorých zložitosť je vyjadrená funkciou c vyhodnotíme váhovanie triedy pomocou vzťahu:

$$WMC = \sum_{i=1}^{n} c(M_i)$$

Návrh metriky neurčuje akú metriku zložitosti metód pre *c* je potrebné použiť (môže ňou byť aj Halsteadovej metrika alebo cyklomatické číslo (Marinescu, 2001)). Vyhodnotením zložitosti metód môžeme predpovedať koľko času a úsilia je potrebných pre udržovanie triedy. Počet metód ovplyvňuje aj potomkov vyhodnocovanej triedy, ktoré budú všetky jej metódy dediť. Zároveň autori uvádzajú tvrdenie, že triedy s veľkým počtom metód sú skôr aplikačne špecifické, tým pádom aj ťažko znovupoužiteľné (Chidamber & Kemerer, 1994).

Hĺbka stromu dedenia

Každá trieda v objektovo-orientovanom návrhu je zaradená do stromovej hierarchie dedenia. Hĺbka stromu dedenia (angl. Depth of Inheritance Tree, DIT) vyjadruje najdlhšiu cestu od triedy ku koreňu stromu.

Hĺbka stromu dedenia vplýva na zložitosť návrhu a predpovedateľnosť správania triedy, keďže trieda dedí veľké množstvo metód od jej predchodcov. Na druhej strane, čím je trieda v hierarchii dedenia hlbšie, tým má väčší potenciál na znovupoužitie dedených metód.

Počet potomkov triedy

Metrika (angl. Number of Children, NOC) vyjadrujúca počet potomkov triedy (detí) v strome dedenia. Väčší počet potomkov triedy prispieva k jej znovupoužiteľnosti, no zároveň môže vyjadrovať aj chybu v abstrakcii triedy alebo nesprávnosti použitia dedenia.

Zviazanosť tried objektov

Metrika (angl. Coupling Between Object Classes, CBO) vyjadrujúca používanie metód alebo premenných inštancií tried vyhodnocovanou triedou. Výsledkom metriky je počet tried, s ktorými je trieda zviazaná. Vysoká zviazanosť medzi triedami totiž škodí modulárnemu návrhu a zabraňuje znovupoužitiu. Pre zvýšenie modulárnosti je vhodné udržať zviazanosť objektov rovnakej triedy na minime, čo uľahčí aj jej testovanie.

Odpoveď pre triedu

Metrika (angl. Response for a Class, RPC) vyjadruje množstvo metód, ktoré môžu byť volané metódami triedy. Ak je tento počet veľký, testovanie triedy si bude vyžadovať lepšie porozumenie pri testovaní. Počet metód, ktoré metódy triedy volajú hovorí aj o jej zložitosti.

Nedostatočná súdržnosť metód

Metrika (angl. Lack of Cohesion in Methods, LCOM) vyjadruje ako veľmi sú prepojené metódy triedy s jej premennými. Pre triedu C s n metódami $\mathrm{M}_1, \mathrm{M}_2, \dots, \mathrm{M}_\mathrm{n}$ a množinou $I = \{I_1, I_2, \dots, I_m\}$ inštancií tejto triedy, ktoré používa metóda M_i, potom hodnotu metriky určíme ako:

$$\mathrm{P} = \{(\mathrm{I_i}, \mathrm{I_j}) | \mathrm{I_i} \cap \mathrm{I_j} = \emptyset$$

$$\mathrm{Q} = \{(\mathrm{I_i}, \mathrm{I_j}) | \mathrm{I_i} \cap \mathrm{I_j} \neq \emptyset\}$$

$$LCOM = \begin{cases} |P| - |Q|, ak\ |P| > |Q| \\ 0, inak \end{cases}$$

Informácia o nedostatočnej súdržnosti napovedá, že trieda by mala byť rozdelená do podtried, alebo nie je správne navrhnutá. Nízka súdržnosť triedy zvyšuje jej zložitosť, tým pádom aj možnosť vzniku chýb v rámci procesu návrhu.

Príloha F

Príspevok na konferenciu IIT.SRC 2014

Výsledky našej práce sme publikovali a prezentovali v rámci študentskej vedeckej konferencie IIT.SRC 2014[18] na Fakulte informatiky a informačných technológií Slovenskej univerzity v Bratislave. V tejto prílohe uvádzame náš príspevok *Identifying Hidden Source Code Dependencies*, ktorý bol ocenený cenou dekana.

> Konôpka, M., 2014. Identifying Hidden Source Code Dependencies. In: *Proceedings of 10th Student Research Conference in Informatics and Information Technologies Bratislava (IIT.SRC 2014).* Bratislava: Nakladateľstvo STU, pp. 474-479.

[18] IIT.SRC 2014. `http://www.fiit.stuba.sk/iit-src`

Identifying Hidden Source Code Dependencies from Developer's Activity

Martin KONÔPKA*

Slovak University of Technology in Bratislava
Faculty of Informatics and Information Technologies
Ilkovičova 2, 842 16 Bratislava, Slovakia
xkonopkam@stuba.sk

Abstract. Traditional software metrics evaluate software product as a result of development process, we use them to identify problems in source code while not taking into account source of identified problems. Software and its source code is the result of developer's work and so its attributes depend on the developer's activity and context during the development process. In this paper we use developer's activity and interactions with source code files during development to identify hidden implicit dependencies in the source code. In our method we extend dependency graph of software components with implicit dependencies, which can be used for navigation in source code. We identify application of our method in the phase of source code development and review.

1 Introduction

Software project comprises of many phases with their own inputs and outputs which are important to monitor. The main output of software project is the resulting software product. Development of this product requires developers to understand requirements, design and work with source code. We may evaluate various attributes of the resulting software which are meaningful to both users and developers. Many software metrics have been proposed to evaluate these attributes, mostly based on syntactic analysis of the source code [7].

Developers of software project work on various tasks during the development process. Both types, i.e., adding new functionality and changing existing functionality, require developers to use or perform changes in dependencies in source code, e.g., existing structures, compositions or inheritance hierarchies. It is helpful for developers to know about these dependencies before making the change because it is the way to learn about how the existing source code works and how it is organized. As a dependency we understand oriented connection between two source code components of selected granularity, e.g., reference, inheritance or call.

Traditionally, dependencies reflect explicit statements in source code and are identified with syntactic analysis. Identified dependencies form dependency graph of software components which helps developers in study of software components and their attributes, in identifying problematic places and complexity of the web of software components.

---

* Master degree study programme in field: Software Engineering
Supervisor: Professor Mária Bieliková, Institute of Informatics and Software Engineering, Faculty of Informatics and Information Technologies STU in Bratislava

IIT.SRC 2014, Bratislava, April 29, 2014, pp. 474–479.

Developers' activities and their work environment affect resulting attributes of created source code [6]. We may look for connections between the attributes of software [3] and activities done
by the developers together with the context which they reside in. In our work we propose method for identification of dependencies between software components from developer's activity to broaden the space of dependencies for analysis in dependency graph.

This paper is structured as follows: in section 2 we discuss related work in our research area, in section 3 we introduce implicit dependencies in software source code, following with section 4 about integrating existing dependency graph with implicit dependencies and concluding with section 5 discussing application of our method, describing its evaluation and identifying further work.

2 Related work

Tools for navigation and techniques for discovery of dependencies in the source code are of interest in research because of the various scenarios developers may encounter during the development,
e.g., disruption of work, switching between multiple tasks to complete or taking over task after different developer. Most common tools are software product metrics [7] and dependency graph of software components [5]. Dependency graph can be used not only for studying the source code, but also for code smells detection [5]. In [10] authors applied network algorithms on dependency graph to predict problems in software design. Differently, in [4] authors propose method for inferring programming sessions from monitored activities during the development. Identified sessions are then used to describe tasks which developers had worked on.

In [1] authors shown that developers visit places in source code relevant for the task completion more often during the work on that task. Information about developers' tasks can then be used to recommend developers for next tasks, based on their knowledge about the source code parts [8]. Authors in [9] proposed method for identification of usage contexts from interactions with development environment. When compared to our method, they create and provide contexts to the developer to speed up navigation during development. Because of that, identification of relevant source code places is helpful for recommendation in source code search or for studying the solutions in the existing code base.

3 Dependencies between software components

Measuring attributes of software is of high interest in software engineering. Several software metrics have been proposed to measure the attributes in software development, e.g., complexity, modularity [3,6]. Additionally to the evaluation perspective of finished development, software metrics are also applicable during the development for developers. Example situation is identification of complex places in the source code by looking for dependencies of software component and how much possible change in it would affect the rest of the code.

Dependency between two software components is *oriented connection* of selected type, traditionally reference, call, inheritance or hierarchy membership [5]. These types of connections are explicitly stated in the source code, so we understand them as explicit dependencies. Explicit dependencies are identified from the contents of source code and are visualized in form of dependency matrix or oriented graph of vertices for software components (of selected granularity) and edges for explicit dependencies between components. Developers and code reviewers may use this graph to navigate in the source code space to understand existing connections.

3.1 Implicit dependencies in source code

Given the graph of explicit dependencies in source code we are not able to infer whether the developer got inspiration from other component during the development, whether the source code was copy-pasted or whether it follows some kind of good-practice solution for particular problem used elsewhere in the source code. At the same time, developers move in the source code space during the development, read it, edit it, they think about solutions for problems and bugs. This motivates us to identify implicit dependencies to underline connections in source code which are not explicitly stated, but still present in developer's activity, intents and decisions during development, or even during the runtime of the software.

Identification process of implicit dependencies relies on user's activities which we are able to record during the development process. In our work, we use these low-level logs of user activities with source code files:

- *Open*, *close* and *switch-to* operations on source code file – time-based activities of navigation in source code space of software project, result is change in currently opened file.
- *Copy-paste* content from one source code file into another.
- *Check-in* (or *commit*) a collection of source code files to source code revision control system.

These activities are recorded during the work within integrated development environment, e.g., Microsoft Visual Studio or Eclipse, with custom extension [2]. In case of *check-in* activities, we gather collections of files with wrapper around used revision control system [2], e.g., Microsoft Team Foundation Server or Git.

3.2 Identification of implicit dependencies from activity logs

Time-based activities are described with operation on source code file, file identifier, and timestamp when the activity occurred. When the developer switches to different file, we do not know the source and target of the *switch-to* operation, only the target file. Because of that, we use algorithm with state machine to emit new dependencies when switching between states (see Figure 1).

We distinguish between state when developer has opened file and state when we are not aware of the current file (unknown or none or none).

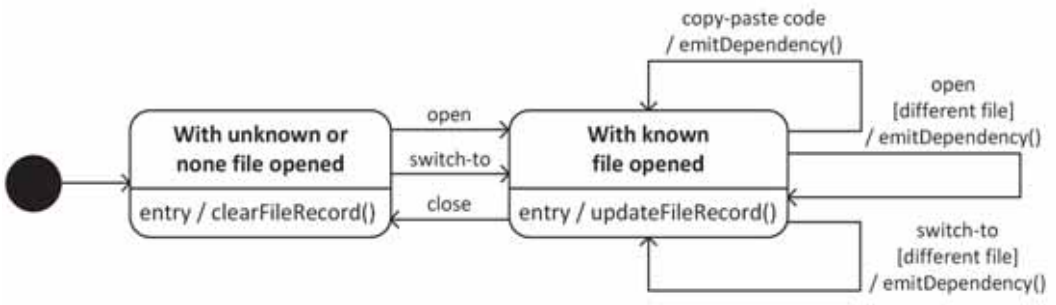


Figure 1. State machine of currently opened file in development environment used by algorithm for identification of implicit dependencies from activity logs.

We are aware of few downsides of our algorithm, although we see it as only option how to model developer's interactions with source code files because of the nature of activity logs. As an example of situation which we are not able to identify is when developer was viewing two source code files at once, i.e., docked side by side.

3.3 Weighing implicit dependencies

Explicit dependencies are weighed according to the number of existing connections of selected types. Weights describe significance of identified connections in the source code, so implicit dependencies have to be weighed as well, but differently according to the source which they were identified from.

Implicit dependencies identified from time-based activities describe developer's navigation in the source code space, when developer changes currently opened source code file (with open or switch-to action). These dependencies are weighed according to the time spent in opened file, i.e., significance of opening that file for developer on the closed interval from *0* to *1*.

Weight of dependencies identified from copy-pasting may correspond to amount of code copied. However, we chose to weigh every copy-paste operation with constant significance of *1*. Dependencies between files from one check-in action are identified between each pair of files in collection and are proportionally weighed, always summing to constant of *1*.

4 Dependency graph

Dependency graph with implicit dependencies is constructed with aggregation of identified dependencies into edges. We enhance definition of graph *G(V,E)* by adding new set of implicit edges E_{imp} to the explicit ones E_{exp}, so the set of edges equals to:

$$E = E_{exp} \cup E_{imp}$$

When constructing implicit edge for pair of selected components, we determine its weight by aggregating tuples *(timestamp, weight)* of all identified dependencies between these two components into single weighted edge. We identified these two possible versions of determining the weight:

- Sum of weights of dependencies regardless of timestamps,
- Sum of weights validated to selected point in time, i.e., timestamp.

Construction of explicit edges is based on aggregation of existing single explicit dependencies in the source code relevant to the selected time. It is important to differentiate between these two types of edges in the graph, not to aggregate them, because of the difference in their meaning:

- Explicit edges describe real connections in the source code, e.g., inheritance, references, calls.
- Implicit edges describe how developer interacted with source code during the development.

Implicit edges may reflect explicit edges in the graph, e.g., in scenario when developer references other class during writing his method, switches to the source code of that class, studies it and possibly implements the functionality. However, note that developers with knowledge about the source code do not need to study every class they reference to.

See Figure 2 for example of resulting dependency graph with implicit dependencies in environment of Microsoft Visual Studio. Existing *Code Map* functionality[1] provides generation of graph with explicit dependencies which can be later changed using its source file. We introduce implicit dependencies identified with our method into that source file, add new nodes for source code files and map their contained classes and interfaces. The result is that developers may interact with dependency graph directly in Microsoft Visual Studio, switch to the real source code files from the graph and see their contents. While this integration of our method with existing tool is very effective, it is currently available for projects in *C#* programming language only.

[1] Map dependencies in specific code using code maps in Visual Studio, Microsoft Developer Network, http://msdn.microsoft.com/en-us/library/jj739835.aspx

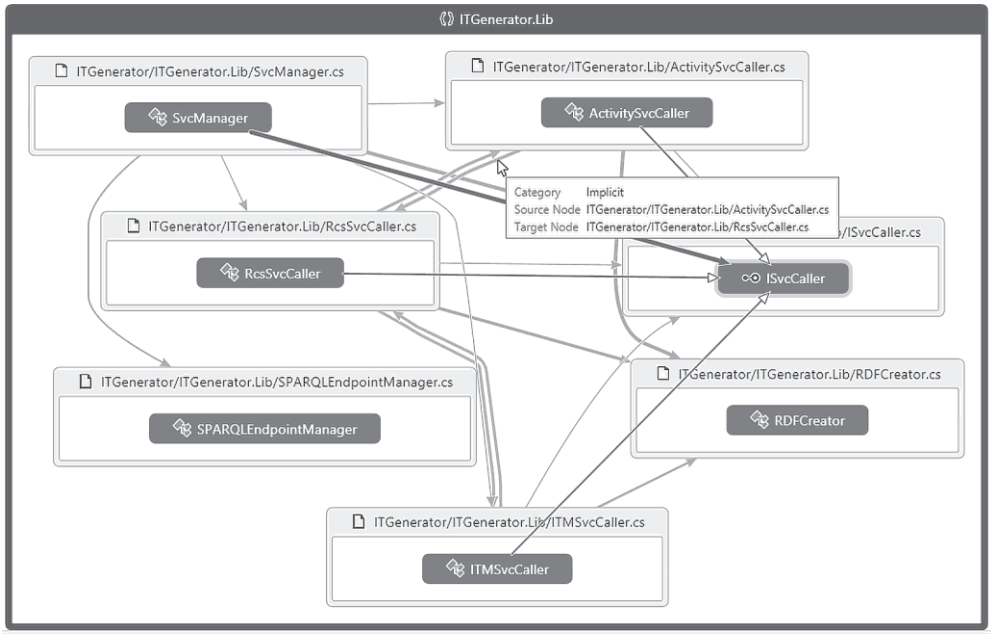


Figure 2. Visualization of dependency graph with explicit and implicit edges in Microsoft Visual Studio.

5 Conclusions and discussion

In this paper we introduced our method for identification of implicit dependencies in source code from developer's activity to enhance existing dependency graph, as we see its application in processes of development, code review or search. During the software development, developers may be affected by their context, e.g., personal, their knowledge, environment or context of tasks which they were working on [6]. Dependency graph with implicit edges may be used to discover relevant places in source code for particular task. We identify these scenarios to illustrate task context:

- Developer returning to work on his own source code searches for components which were relevant to the previous task. This includes ability to find relevant components which are loosely coupled or possibly not explicitly connected in the source code, e.g., when connections are defined in configuration files.
- Developer taking over another developer's task, or fixing bug, searches for related components that may be source of problem to be solved or may be affected by the solution.
- Senior developer searching for bugs and spread of problems across the source code. If we assume that developers were influenced by their context or environment during the development, we may discover bugs in time-related components during development as well as just locating one single problem.

Our work is part of research project PerConIK[2] – Personalized Conveying of Information and Knowledge, which applies Web engineering methods in software development domain [2]. We use PerConIK for provided tools and services, as well as a source for experimental evaluation of our method while it provides logs of activities on number of students' individual or team projects.

[2] PerConIK, `http://perconik.fiit.stuba.sk/`

To evaluate our method, we introductorily compared sets of explicit edges with identified implicit edges for selected projects. For 2 individual and 2 team projects we achieved average *55.7%* precision on of implicit dependencies (while not considering their orientation) reflecting explicit ones. Secondly, in evaluation of our method we look onto subset of implicit dependencies that were not matched with explicit ones whether they describe real hidden dependencies in the source code. This involves manual run through identified dependencies by developers of the selected projects.

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

References

[1] Antunes, B., Cordeiro, J., Gomez, P.: An Approach to Context-based Recommendation in Software Development. In: *Proceedings of the 6th ACM Conference on Recommender Systems (RecSys '12)*, ACM, (2012), pp. 171-178.

[2] Bieliková, M., Polášek, I., Barla, M., et al.: Platform Independent Software Development Monitoring: Design of an Architecture. In: *Proceedings of the 40th International Conference on Current Threads in Theory and Practice of Computer Society (SOFSEM '14)*, LNCS 8327, Springer Verlaag, (2014), pp. 126-137.

[3] Boehm, B.W., Brown, J.R., Lipow, M.: Quantitative Evaluation of Software Quality. In: *Proceedings of the 2nd International Conference on Software Engineering (ICSE'06)*, IEEE Computer Society Press, (1976), pp. 592-605.

[4] Coman, I.D., Sillitti, A.: Automated Identification of Tasks in Development Sessions. In: *Proceedings of the 16th IEEE International Conference on Program Comprehension (ICPC'08),* IEEE Computer Society Press, (2008), pp. 212-217.

[5] Counsell, S., Hassoun, Y, Loizou, G, et al.: Common Refactorings, a Dependency Graph and some Code Smells: An Empirical Study of Java OSS. In: *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering (ISESE '06)*, ACM, (2006), pp. 288-296.

[6] Eyolfson, J., Tan, L., Lam, P.: Do Time of Day and Developer Experience Affect Commit Bugginess?. In: *Proceedings of the 8th Working Conference on Mining Software Repositories (MSR '11)*, ACM, (2011), pp. 153-162.

[7] Fenton, N.E., Pfleeger, S.L.: Software Metrics: A Rigorous and Practical Approach. 2nd Edition, PWS Pub. Co., Boston, MA, USA, (1998).

[8] Fritz, T., Murphy, G.C., Hill, E.: Does a Programmer's Activity Indicate Knowledge of Code?. In: *Proceedings of 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on The Foundations of Software Engineering (ESEC-FSE '07),* ACM, (2007), pp. 341-350.

[9] Robillard, M.P., Murphy, G.C.: Automatically Inferring Concern Code from Program Investigation Activities. In: *Proceedings of 18th IEEE International Conference on Automated Software Engineering*, IEEE Computer Society Press, (2003), pp. 225-234.

[10] Zimmermann, T., Nagappan, N.: Predicting Defects Using Network Analysis on Dependency Graphs. In: *Proceedings of 30th International Conference on Software Engineering (ICSE '08)*, ACM, (2008), pp. 531-540.

Príloha G

Príspevok na konferenciu ESEM 2014

V tejto prílohe uvádzame návrh príspevku na konferenciu *Empirical Software Engineering and Measurement* (ESEM 2014)[19], ktorá sa koná 18.-19. septembra 2014 v Turíne, Taliansko.

[19] ESEM 2014. http://softeng.polito.it/ESEIW2014/ESEM/index.html

Developer Activity as a Source for Identifying Hidden Source Code Dependencies

Martin Konôpka, Mária Bieliková
Slovak University of Technology in Bratislava
Faculty of Informatics and Information Technologies
Ilkovičova 2, 842 16 Bratislava, Slovakia
{name.surname}@stuba.sk

ABSTRACT
Traditionally, evaluation of software product is done from the perspective of its source code or resulting software. Evaluating source code metrics and understanding connections in the software source code is based on its syntactic analysis. This may not be sufficient, or even possible, in case of dynamically typed programming languages or when looking for information about source of explicit statements in the source code. Software and its source code is the result of developer's work and so its attributes should depend on the developer's activity and context during the development process. In this paper we presents method of using developer's activity and interactions with source code files during development to identify hidden implicit dependencies in the source code. In our method we extend existing dependency graph of software components with implicit dependencies, which can be used for navigation in source code. We identify application of our method in the phase of source code development and review.

Categories and Subject Descriptors
D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement – *Restructuring, reverse engineering, and reengineering.*

General Terms
Measurement, Experimentation, Human Factors.

Keywords
Dependency Graph, Implicit Dependency, Implicit Feedback.

1. INTRODUCTION
Software project comprises of many phases with their own inputs and outputs which are important to monitor. The main output of software project is the resulting software product. Development of this product requires developers to understand requirements, design and work with source code. We may evaluate various attributes of the resulting software which are meaningful to both users and developers. Many software metrics have been proposed to evaluate these attributes, mostly based on syntactic analysis of the source code [7]

Developers of software project work on various tasks during the development process. Both types, i.e., adding new functionality and changing existing functionality, require developers to use or perform changes in dependencies in source code, e.g., existing structures, compositions or inheritance hierarchies. It is helpful for developers to know about these dependencies before making the change because it is the way to learn about how the existing source code works and how it is organized.

Developers' activities and their work environment affect resulting attributes of created source code [6]. We may look for connections between the attributes of software [3] and activities done by the developers together with the context which they reside in. In our work we propose method for identification of dependencies between software components from developer's activity to broaden the space of dependencies for analysis in dependency graph. Based on the source for identification, we distinguish between implicit and explicit dependencies.

Conference'10, Month 1–2, 2010, City, State, Country.

2. RELATED WORK
Tools for navigation and techniques for discovery of dependencies in the source code are of interest in research because of the various scenarios developers may encounter during the development, e.g., disruption of work, switching between multiple tasks to complete or taking over task after different developer. Most common tools are software product metrics [7] and dependency graph of software components [5]. Dependency graph can be used not only for studying the source code, but also for code smells detection [5]. In [10] authors applied network algorithms on dependency graph to predict problems in software design. Differently, in [4] authors propose a method for inferring programming sessions from monitored activities during the development. Identified sessions are then used to describe tasks which developers had worked on.

In [1] authors shown that developers visit places in source code relevant for the task completion more often during the work on that task. Information about developers' tasks can then be used to recommend developers for next tasks, based on their knowledge about the source code parts [8]. Authors in [9] proposed a method for identification of usage contexts from interactions with the development environment. When compared to our work, they create and provide contexts to the developer to speed up navigation during development. Because of that, identification of relevant source code places is helpful for recommendation in source code search or for studying the solutions in the existing source code base.

3. SOURCE CODE DEPENDENCIES
Source code dependencies traditionally reflect explicit statements in source code and are identified with syntactic analysis of source code file contents. As a dependency we understand oriented connection between two source code components of selected granularity, namely instance or type reference, inheritance relationship or call reference.

Identified dependencies are used to form dependency matrix or oriented graph of connected software components to study organization and hierarchy of software components and their attributes. Dependencies are also source for identifying problematic places, possibly code smells and complexity of the web of software components.

Dependencies identified with syntactic analysis of source code can be labelled as explicit, because they reflect explicit statements in the source code. However, we identify two main problems of explicit dependencies and their identification:

- Explicit dependencies do not reflect connections with configuration files, schema template files or runtime dependencies.
- Syntactic analysis of source code is not trivial or even possible for dynamically typed languages, e.g., JavaScript.
- Explicit dependencies do not reflect sources of solutions in source code, developer's inspiration and places required to be checked when particular component is changed.

These problems motivate us to identify dependencies in source code which are not explicitly stated, but still present in developer's activity, intents and decisions during development, or even during the runtime of the software.

3.1 Implicit Dependencies in Source Code

Implicit dependencies describe developer's own connections of software components during the development process. To be able to infer whether the developer got inspiration from good-practice solution in other component during the development or whether the source code was copy-pasted, we employ developer's activity as a source for identification of implicit source code dependencies.

Identification of implicit dependencies relies on developer's activities which we are able to record during the development process in development environments, e.g., Microsoft Visual Studio or Eclipse, with custom extension [2]. Another source for developer's activities are revision control systems, e.g., Microsoft Team Foundation Server or Git. In our work, we use these low-level logs of user activities with source code files:

- Open, close and switch-to operations on source code file – time-based activities of navigation in source code space of software project, result is change in currently opened file.
- Copy-paste content from one source code file into another.
- Check-in (or commit) a collection of source code files to source code revision control system.

Identification of implicit dependencies is based on task context, i.e., developer works with components that are related to each other in the task he or she works on. Thus switching between source code files, copy-pasting code or the check-ins should reflect implicit dependencies between these files.

3.2 Mining User Actions from Activity Logs

Our work is part of the research project PerConIK [2], which we use as a data source for our work. Implicit dependency of two software components is aggregation of all identified actions performed by developers, i.e., time-based and content-based actions.

Time-based actions of operations on source file are described with tuple *(file, operation, timestamp)*, describing the target file of the operation, type of performed operation (e.g., open, close or switch-to file) and timestamp when the activity occurred. When the developer switches to different file, we do not know the source and target of the *switch-to* operation, only the target file. Because of that, we reconstruct the stack of opened files in development environment and use algorithm with state machine to emit identified actions when switching between states. We distinguish between state when developer has opened file and state when we are not aware of the current file (unknown or none).

Activity logs of copying and pasting code are described with tuples *(file, code operation, code, timestamp)*, describing the target file where the code operation was performed (independently copy, paste, cut or selection of code) with *code* contents, and when the operation happened. Copy-paste operation is logged with at least two actions, i.e., copying the code from the source code file A and pasting it into the source code file B. Because of that we also reconstruct the clipboard stack of which we use to emit actions between source and target files.

3.3 Identification of Implicit Dependencies

Time-based and content-based actions serve as a source for implicit dependencies. Implicit dependency between selected source code documents is aggregation of actions between these documents. Similarly to explicit dependencies weighed according to the number of explicit connections in the source code, implicit dependencies are weighed as summation of actions used for their identification.

Time-based actions describe developer's navigation in the source code space, when developer changes currently opened source code file (with open or switch-to operation). These dependencies are weighed according to the time spent in opened file, i.e., significance of opening that file for developer on the closed interval from *0* to *1*.

Weight of dependencies identified from copy-pasting may correspond to amount of code copied. However, we chose to weigh every copy-paste operation with constant significance of *1*. Dependencies between files from one check-in action are identified between each pair of files in collection and are proportionally weighed, always summing to constant of *1*.

3.4 Validation of Implicit Dependencies

Changes of software components invoke changes of their explicit dependencies, although we are not able to explicitly validate implicit dependencies over time from contents of artifacts. To model relevance of implicit dependencies in selected time, we validate them to the selected time. Explicit and implicit dependencies are valid prior to the contents of artifacts. Developer interacts with source code files mostly based on their contents, thus when contents change over time, developer's interactions lose their validity. Because of that we define validation of explicit and implicit dependencies with functions $validity_{exp}$ and $validity_{imp}$:

$$validity_{exp} : D_{exp} \times T \rightarrow \{0, 1\}$$

$$validity_{imp} : D_{imp} \times T \rightarrow \langle 0, 1 \rangle$$

Explicit dependencies are valid or not according to existence of explicit statements to the selected time, hence the selected codomain for $validity_{exp}$. Implicit dependencies are valid according to forgetting function to the selected time.

3.5 Dependency Graph

When knowing dependencies of software components, we define dependency graph *G(V,E)* with *V* for set of vertices, i.e., software components (source code files), and *E* for edges, i.e., aggregated dependencies. It is important to note that we should not aggregate

explicit and implicit dependencies, thus we define edges with $E = E_{exp} \cup E_{imp}$. We differentiate between explicit and implicit edges in the graph because of differences in their meaning:

- Explicit dependencies represent defined connections in the source code, e.g., references, call hierarchy, inheritance.
- Implicit dependencies represent how developers interacted with source code during the development, e.g., their intents, inspirations.

4. EXPERIMENTAL EVALUATION

We propose implicit source code dependencies to be mainly applied during the software development, when developer searches for dependent components, e.g., after fixing a bug. To evaluate significance of implicit dependencies, we put following hypotheses for experimental evaluation:

H1: Implicit dependencies reflect explicit dependencies of components which developer worked on during the task.

H2: Implicit dependencies enrich dependency graph with new connections usable for maintenance of source code.

To validate these hypotheses we performed two experiments on data available from the PerConIK project, while discussing the contribution of identified implicit dependencies with developers who were monitored.

For our experiments we used data about 4 software projects of various sizes where developer roles were occupied by students, all projects were created using Microsoft technologies:

- Project A – development of web/desktop application, 3 developers implemented additional functionality during 3 months, generating 15,215 activity logs.
- Project B – development of desktop application, 1 developer implement functionality within time frame of 2 man-days, generating 8,584 activity logs of document operations.
- Project C – development of web/desktop application by the team of 7 developers during 6 months, generating 55,877 activity logs.
- Project D – development of web application by the team of 7 developers during 6 months, generating 48,717 activity logs.

Table 1 shows results with numbers of identified dependencies for selected weight thresholds from 0 to 3.

4.1 Reflection of Explicit Dependencies

In hypothesis *H1* we expect that implicit dependencies reflect explicit ones based on the existence of developer's task context. Thus our task is to compare existing explicit dependencies of these project with identified implicit dependencies. Because all projects were developed using Microsoft technologies, we employed Code Map functionality of Microsoft Visual Studio which generates explicit dependencies. Advantages of using this tool is that it uses built-in reference recognition.

With the generated explicit dependencies we were able to compare the overlap of identified implicit dependencies with weight thresholds *1* and *2*. Firstly, we determined how many of the identified implicit dependencies are as well explicit, i.e., $(E_{exp} \cap E_{imp}) / E_{imp}$. Secondly, we looked into how many of existing explicit dependencies of the files were identified with implicit dependencies, i.e., $(E_{exp} \cap E_{imp}) / E_{exp}$.

Table 1. Identified implicit dependencies for selected projects.

Project	Implicit dependencies count for thresholds			
	0	**1**	**2**	**3**
A	640	423	200	124
B	507	339	141	88
C	797	556	285	201
D	755	464	246	164

Table 2. Implicit dependencies reflecting explicit ones.

Project	$(E_{exp} \cap E_{imp}) / E_{imp}$		$(E_{exp} \cap E_{imp}) / E_{exp}$	
	1	**2**	**1**	**2**
A	40.9%	54.0%	74.6%	46.6%
B	41.3%	49.7%	51.1%	25.6%
C	34.4%	43.2%	70.7%	45.6%
D	32.1%	43.9%	78.84%	57.1%

With the results shown in Table 2 we confirm our hypothesis *H1* that implicit dependencies reflect explicit dependencies and thus may be used to partially compensate inability of identification of explicit dependencies. However, implicit dependencies are identified between source code files, not its contained software components, which the explicit dependencies are desired for.

4.2 Meaningfulness of Implicit Dependencies

In hypothesis H2 we expect that implicit dependencies provide new information about connected software components, e.g., when certain component relies on settings in configuration file. Our task was to discuss identified implicit dependencies with developers and decide whether they reflect some relationship in source code or not. As a relationship of source code files we understood:

If the components in the source code file A are changed, the contents of the file B should be checked or changed as well.

For this experiment we chose to validate implicit dependencies with weight threshold of *2*. Developers manually checked each one of dependencies and decided their significance. To simplify the process of validation we generated dependency graphs with implicit dependencies only in DGML format, accepted by Microsoft Visual Studio. Developers were able to switch to source code files included in the graph as vertices and thus ensure their decision whether the dependency should stay or be removed from the graph. Developers removed not significant dependencies and then we compared the results with original files. Table 3 shows results of this experiment.

Table 3. Evaluated meaningfulness for projects A, B and C.

Project	Meaningfulness success rate	
	Including non-existing files	**Non-existing files excluded**
A	75.1%	78.0%
B	90.4%	92.0%
C	89.4%	90.9%

After the experiment we received positive feedback from participated developers about ability to identify dependencies between source code files of the View layer with files of the Controller and Model layers, when the application was built upon Model-View-Controller model. Even more, when evaluating implicit dependencies, developers were able to remind the source problem solution in code of particular classes. However, because the implicit dependencies were identified from activities, some may indicated connections between non-existing files. While these dependencies seemed to be relevant, they were needless for already deleted files.

5. CONCLUSION AND FURTHER WORK

Knowledge about dependencies of software components is utilized mostly during the development process, helping developers with navigation and study of the existing source code. Identification of dependencies is important for easier maintainability. However, identification of implicit dependencies for dynamically typed languages is not trivial, or sometimes even possible, task. Recording developer's activity during the development work gives us insights of different kind of dependencies existing in the source code base. We propose identification of implicit source code dependencies to enrich existing dependency graph, or supplement it when it is not possible to be created.

While the straightforward application of implicit dependencies is to visualize them in form of graph, the graph may be used to simply provide prioritized list of software components to be checked for the selected component upon developer's request. This extension to development tools will utilize knowledge about explicit and implicit dependencies of software components. At the same time, this will provide us with another way how to evaluate identification of implicit dependencies, because we may serve developer's selection of components from the provided list as implicit feedback of correctly selected components.

6. ACKNOWLEDGEMENTS

This contribution/publication is the partial result of the Research & Development Operational Programme for the project Research of methods for acquisition, analysis and personalized conveying of information and knowledge, ITMS 26240220039, co-funded by the ERDF.

7. REFERENCES

[1] Antunes, B., Cordeiro, J., Gomez, P. 2012. An Approach to Context-based Recommendation in Software Development. In *Proc. of the 6th ACM Conference on Recom. Systems*. RecSys '12, ACM, New York, NY, USA, 171-178.

[2] Bieliková, M., Polášek, I., Barla, M., et al. 2014. Platform Independent Software Development Monitoring: Design of an Architecture. In *Proc. of the 40th International Conference on Current Threads in Theory and Practice of Computer Society*. SOFSEM '14, LNCS 8327, Springer Verlaag, Berlin, Germany, 126-137.

[3] Boehm, B.W., Brown, J.R., Lipow, M. 1976. Quantitative Evaluation of Software Quality. In *Proc. of the 2nd International Conference on Software Engineering*. ICSE'06, IEEE Computer Society Press, 592-605.

[4] Coman, I.D., Sillitti, A. 2008. Automated Identification of Tasks in Development Sessions. In *Proc. of the 16th IEEE International Conference on Program Comprehension*. ICPC'08, IEEE Computer Society Press, 212-217.

[5] Counsell, S., Hassoun, Y, Loizou, G, et al. 2006. Common Refactorings, a Dependency Graph and some Code Smells: An Empirical Study of Java OSS. In *Proc. of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*. ISESE '06, ACM, New York, NY, USA, 288-296.

[6] Eyolfson, J., Tan, L., Lam, P. 2011. Do Time of Day and Developer Experience Affect Commit Bugginess?. *In Proc. of the 8th Working Conference on Mining Software Repositories*. MSR '11, ACM, New York, NY, USA, 153-162.

[7] Fenton, N.E., Pfleeger, S.L. 1998. *Software Metrics: A Rigorous and Practical Approach*. 2nd Edition, PWS Pub. Co., Boston, MA, USA.

[8] Fritz, T., Murphy, G.C., Hill, E. 2007. Does a Programmer's Activity Indicate Knowledge of Code?. In *Proc. of 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on The Foundations of Software Engineering*. ESEC-FSE '07, ACM, New York, NY, USA, 341-350.

[9] Robillard, M.P., Murphy, G.C. 2003. Automatically Inferring Concern Code from Program Investigation Activities. In *Proc. of 18th IEEE International Conference on Automated Software Engineering*. IEEE Computer Society Press, 225-234.

[10] Zimmermann, T., Nagappan, N. 2008. Predicting Defects Using Network Analysis on Dependency Graphs. In *Proc. of 30th International Conference on Software Engineering*. ICSE '08, ACM, New York, NY, USA, 531-540.

Príloha H

Obsah elektronického média

/doc
- /Konopka-DP3.pdf – elektronická verzia dokumentu
- /Konopka-IITSRC.pdf – príspevok na konferenciu IIT.SRC 2014
- /Konopka-ESEM.pdf – návrh príspevku na konferenciu ESEM 2014

/exp – výsledky experimentov overenia práce

/src
- /DP.DG – zdrojové súbory implementácie riešenia
- /DP.DG.zip – archivované zdrojové súbory implementácie riešenia
- /Documentation.chm – vygenerovaná dokumentácia zdrojového kódu

/PerConIK – dokumentácia služieb infraštruktúry projektu PerConIK