Automatikus refaktoring eszközök alkalmazása a szoftver erózió csökkentése érdekében

Szőke Gábor

Szoftverfejlesztés Tanszék
Szegedi Tudományegyetem

Szeged, 2019

Témavezető:

Dr. Ferenc Rudolf

Ph.D. értekezés tézisei

Szegedi Tudományegyetem

Informatikai Doktori Iskola

Bevezetés

A karrierje egy pontján minden programozó szembesül egy olyan kódrészlettel aminek működését senki sem érti és senki sem szeretne hozzányúlni, nehogy véletlen elrontsa. A kérdés az, hogy hogyan keletkezett ez a siralmas kódrészlet. Feltételezhetően senki sem önszántából írta ilyenre. Sokkal valószínűbb, hogy a programunk a *szoftver erózió* áldozata, amely a szoftver egész életciklusára – legyen az fejlesztés vagy karbantartás – jellemző folyamatos hanyatlás.

A szoftver eróziója elkerülhetetlen. Egy szoftverrendszer folyamatosan fejlődik az idő múlásával: új funkciók kerülnek bele, korábbi funkciók módosulnak vagy tűnnek el; egyszóval igazodik az új igényekhez és környezethez. Ezek velejárója, hogy a forráskód általában bonyolultabb lesz és egyre inkább eltávolodik a kezdeti állapotától. Következményképp megnő a szoftver karbantartásának költsége. Ez is nagyrészt hozzájárul ahhoz, hogy a szoftverfejlesztési költségek nagyobb része (kb. 80%) a karbantartásra megy el [4]. A költségek csökkentéséhez redukálni kell a szoftvererózió okozta hatást, és növelni kell a karbantarthatóságot rendszeresen végrehajtott refaktoring műveletekkel.

A refaktoring kifejezés aztán vált népszerűvé, hogy Fowler publikálta katalógusát a refaktoring átalakításokról [3]. Ezen átalakítások célja az úgy nevezett "bűzlő" (vagy gyanús) kódok helyrehozása. Itt a "bűzlő" szó a nehezen karbantartható vagy rosszul megkonstruált kódrészekre utal. Ilyen például, ha egy metódus nagyon hosszú, vagy ha egy metódus szinte másolta egy másiknak. Az ilyen gyanús szerkezetek megértése segítséget nyújt, hogy felfedjünk hibákat és antimintákat, amik a valós problémákat jelentik a szoftverben. Ezek kiiktatása jobb minőségű szoftvert eredményez.

Mindenki közös érdeke, hogy a szoftver karbantarthatósága megmaradjon könnyűnek. A felhasználók így hamarabb kapnak új funkciókat, melyekben kevesebb hiba lesz. Emellett a fejlesztőknek is egyszerűbb dolguk lesz a kód módosításával, és a fejlesztőcégeknek is csökken a karbantartásra költött költségük. Jó karbantarthatóságot nagyon részletes specifikációval és alaposan kidolgozott tervekkel lehet a legjobban elérni. Azonban olyan helyzetek nagyon ritkán adódnak, ahol ezek a feltételek teljesülnek. Mivel a legtöbb szoftverre inkább a állandó fejlődés jellemző, ezért a gyakorlatban az időről-időre történő folyamatos refaktoring hatékonyabbnak bizonyult a könnyű karbantarthatóság szinten tartására. A tevékenység okán a kód "friss" marad és megnövekedik az élettartama.

A tézis célkitűzései

Jelen tanulmány célja, hogy elősegítse a szoftverrendszerek karbantartását automatizálással. Kiváltképpen olyan módszertanok, technológiák és eszközök kidolgozásával foglalkozik, amik az alábbi témakörökre terjednek ki: szoftverfejlesztők viselkedésének elemzése kézzel írott és gépi refaktoring tevékenységek közben; a refaktoring szoftver minőségre gyakorolt pozitív és negatív hatásának kiértékelése; valamint lokális-keresésre épülő antiminta felismerés átdolgozása modellalapú technológiára gráfillesztéses módszerekkel. A tanulmányt az alábbi kutatási kérdések motiválták:

K1: *Mit tesznek a szoftverfejlesztők, ha külön pénzt és időt kapnak refaktoring feladatok elvégzésére?*

K2: *Milyen fejlesztői igényeknek kell megfelelnie egy automatikus refaktoringokat támogató eszköznek?*

K3: *Milyen hatással vannak a szoftverminőség változására a kézzel írt és a gépi támogatással készített refaktoringok?*

K4: *Használhatjuk a gráfmintaillesztést olyan antiminták azonosítására, amelyek kezdőpontjai lehetnek egy refaktoring folyamatnak?*

Az elért eredményeinket 3 fő tézispontban foglaljuk össze, amelyeket 6 részre osztottunk az alábbiak szerint:

I Szoftverfejlesztők tevékenységeinek elemzése

(a) Szoftverfejlesztők viselkedésének elemzése kézzel írott refaktoring feladatok során

(b) Nagyméretű ipari rendszerek refaktorálásnak elemzése és a karbantarthatóságra gyakorolt hatása

II Automatikus refaktorálás előnyeinek és hátrányainak vizsgálata

(a) Kódolási szabálysértéseket automatán refaktoráló eszközkészlet kifejlesztése

(b) Géppel segített refaktoringok empirikus vizsgálata ipari környezetben

III Modell-alapú módszerek az antiminta detektálásban

(a) Antiminta-felismerés modell-lekérdezéssekkel

(b) Modell-lekérdezés alapú antiminta-felismerési technológiák teljesítményének összehasonlítása

Publikációk

A tézisben felhasznált publikációk jelentős része a szakma rangos, nemzetközi konferenciáinak kiadványaiban, valamint folyóirataiban került közlésre. A tézispontok és a publikációk kapcsolatát összegzi az 1. táblázat.

	Tézispont	Publikációk
I.	Szoftverfejlesztők tevékenységeinek elemzése	[7], [6], [11]
II.	Automatikus refaktorálás előnyeinek és hátrányainak vizsgálata	[8], [9], [10], [12]
III.	Modell-alapú módszerek az antiminta detektálásban	[13], [14]

1. táblázat. Tézispontok és a publikációk kapcsolatának összegzése.

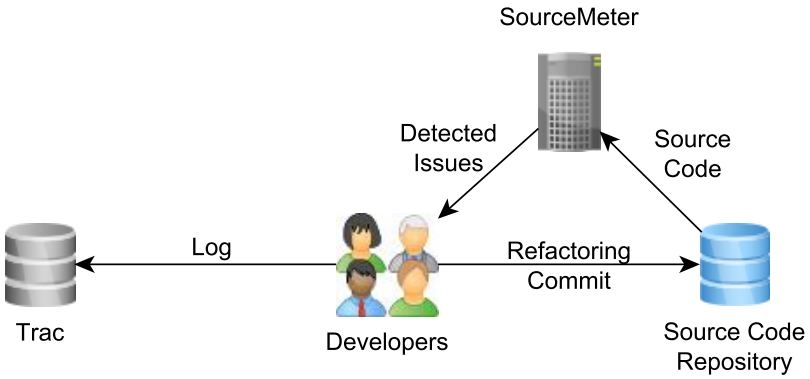
I. Szoftverfejlesztők tevékenységeinek elemzése

A forráskód refaktorálásából sok előnyünk származik (pl. robusztusabb lesz a szoftver, javul a forráskód minősége és ezáltal a karbantarthatóság), de ugyanakkor ezzel együtt jár az is, hogy kevesebb időt tudunk egyéb fejlesztési feladatokkal tölteni. Ennek sokszor az a következménye, hogy a fejlesztők elhanyagolják a refaktorálást, és inkább az előttük lévő feladatokat részesítik előnyben. De mi történik akkor, amikor tisztában vannak azzal, hogy a kódjuk karbantarthatóságára ráfér a javítás, és meg is kapják az ehhez szükséges erőforrásokat? Mihez kezdenek? Melyik problémákat veszik előre vagy hagyják legutoljára? Hogy javítják ki őket? Lehetséges kimutatni ezeket a változásokat egy egységes minőség skálán? Ha igen, akkor vajon összhangban van-e a mért javítás mértéke a fejlesztők erőfeszítéseivel?

Ebben a részben a fenti kérdéseket vizsgáljuk egy *in vivo* környezetben, ahol hat nagyméretű szoftverrendszer minőség mutatóit elemeztük. A kutatásunkat egy olyan periódusban folytattuk, amikor a rendszereken kézzel írt refaktoringokat végeztek. Az időszak alatt a refaktoringokon dolgozó fejlesztőkkel kérdőívet töltettünk ki, amellyel betekintést nyerhettünk a motivációjukba és szokásaikba.

Fejlesztők hozzáállása a kézzel írt refaktoringokhoz

Az itt tárgyalt kutatómunka alapját a *Refaktoring Projekt* képezte, ami egy az Európai Unió és Magyarország által támogatott kutatás-fejlesztési projekt. A projekt egyik fő célja volt olyan automatikus refaktoring eszközök kifejlesztése, amik segítik a fejlesztők mindennapi munkáját. A kutatásban több magyarországi szoftvercég is részt vett partnerként, akik abban a reményben csatlakoztak, hogy növelni tudják forráskódjuk karbantarthatóságát. Hogy olyan eszköz készülhessen el, ami *valóban* segíti a fejleszők munkáját, úgy döntöttünk, hogy kezdetben csak tanulmányozzuk őket normál körülmények között. Ezáltal többet megtudhatunk a fejlesztők igényeiről és hogy mit várnak el egy automatizált eszköztől. Következményképpen a Refaktoring Projekt első lépésében megkértük a partnercégeket, hogy hagyományos úton – azaz kézzel – refaktorálják a rendszereiket.



1. ábra. A refaktoring folyamat a kézzel írt időszakban.

Az 1. ábra egy rövid áttekintést ad a projekt eme fázisáról. Ebben a fázisban arra kértük a fejlesztőket, hogy adjanak egy részletes leírást az elvégzett refaktoringokról, amiben megmagyarázzák, hogy mit csináltak és miért az adott kódrészletet választották kijavításra érdemesnek. Segítségképpen, a saját fejlesztésű statikus forráskód elemző eszközünkkel [2] folyamatosan rámutattunk olyan kódrészletekre, melyek problémásak lehetnek a karbantarthatóság szempontjából. Miután kijavítottak és leteszteltek egy-egy problémát, ismét megkértük őket, hogy összegezzék tapasztaltaikat. Körülbelül 40 fejlesztő (cégenként 5-10 fő) vett részt a kérdőív kitöltésben és a refaktoringok kivitelezésében.

A kérdőívek eredménye rámutatott arra, hogy a fejlesztők sokkal gyakrabban javítottak kódolási szabálysértéseket, mint bármi egyéb mutató által jelzett problémát. Pontosabban, az esetek 85%-ban ezeket az indikátorokat választottak szemben a gyanús, a lemásolt, vagy a metrika határértékeket átlépő kódokkal szemben.

Az eredményekből az is kiderül, hogy a leginkább kihívást jelentő refaktoringok azok, amelyek a kódméret csökkentését célozzák meg, mint például a kódmásolatok kiiktatása. Hasonlóképpen, az üres kódok törlése is nehéznek bizonyult. Több fejlesztő is megjegyezte, hogy órákat töltöttek annak kiderítésével, hogy miért is maradt üresen az a bizonyos `if` utasítás. Egyéb időigényes refaktoringokat a biztonsági és az optimalizációs területekkel kapcsolatban találtunk.

Megfigyeltük, hogy a fejlesztők a legkockázatosabban refaktorálhatónak azokat a szabálysértéseket értékelték, amik alapvető Java funkcionalitásokkal álltak kapcsolatban, mint mondjuk azok amik érintik a `java.lang.Object` `equals`, `hashCode`, vagy `clone` metódusait. Ezen felül az is kiderült, hogy a fejlesztők optimalizálták a refaktoring folyamatot, és a javításokat a komolyabb problémákkal kezdték.

Esettanulmány a refaktoringok karbantarthatóságára gyakorolt hatásáról

Ebben a fejezetben arra keresetük a választ, hogy a refaktoringoknak milyen valós hatása van szoftverminőségre. A projektben részt vevő négy partnercég, hat kiválasztott rendszerén megmértük a karbantarthatósági mutatókat. A 2. táblázat egy rövid áttekintést nyújt a rendszerek méretéről, az elemzett verziókról és a refaktoringok számáról.

2. táblázat. A kiválasztott rendszerek főbb jellemzői.

Rendszer	Cég	kLOC	Elemzett verziók	Refaktoring kommitok	Refaktoringok
A rendszer	Cég I.	1.740	269	136	470
B rendszer	Cég II.	440	180	38	78
C rendszer	Cég III.	170	78	15	597
D rendszer	Cég IV.	38	37	16	18
E rendszer	Cég IV.	11	57	40	40
F rendszer	Cég IV.	50	111	70	70
	Összesen	**2.449**	**732**	**315**	**1.273**

Az elemzés során kiszámoltuk a minőség mutatót a refaktoring alkalmazása előtti és az azt követő verzión. Kiderítettük, hogy a fejlesztők mely szabálysértéseket javították és hogy azok mennyiben módosították a rendszer minőségét. A 2. ábra felső része az *A rendszer* karbantarthatóságának változását mutatja (kommitról-kommitra) a refaktoring időszak alatt. Az ábrán jól látható, hogy a karbantarthatósági érték növekszik az idő múlásával. Érdemes megjegyezni, hogy az ábra nem csak a refaktoring kommitokat mutatja, hanem a hagyományos fejlesztési kommitok is részei.

A 2. ábra alsó része a refaktoring időszak egy szeletét mutatja nagyobb részletességgel. Itt a vörössel kijelölt részek olyan verziókat jelölnek, ahol refaktorálás történt, míg a zöld részek a hagyományos fejlesztői kommitokat jelölik. A refaktoringnak jelölt módosítások észrevehetően javították a rendszer karbantarthatóságát, de volt néhány eset, ahol a javulás nem volt kimutatható szignifikánsan, ilyenkor a minőségmutató nem változott. Ezzel szemben a hagyományos fejlesztési kommitok hol javítják, hol rontják a karbantarthatóságot, viszonylag nagy szórással.

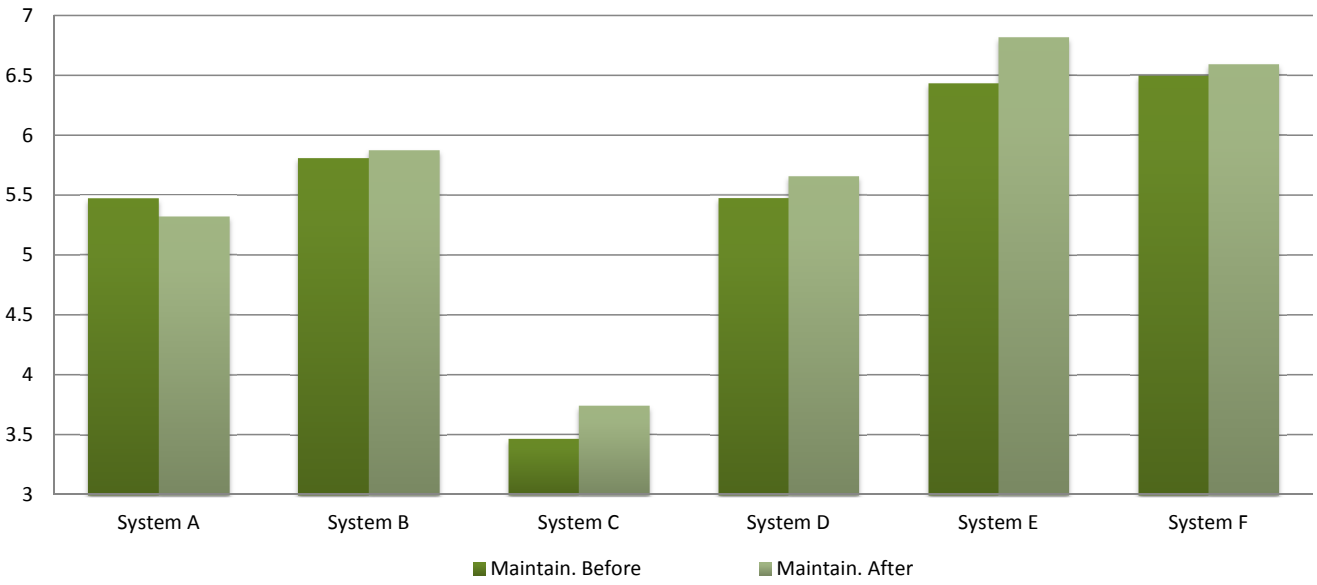
2. ábra. Az *A rendszer* karbantarthatósága a refaktoring időszak alatt, továbbá az időszak egy szeletének részletes megjelenítése, ahol a vörössel kiemelt részek egy-egy refaktoring kommit által elért minőségjavulást jeleznek.

A projektben részvevő partnercégek szabad kezet kaptak abban, hogy azt refaktorálhassák, amit akarnak. Akár nagyobb, rendszerszintű refaktorálásokat is végezhettek volna, de inkább úgy döntöttek, hogy csak kisebb átalakításokat eszközölnek. Nem akartak metrika értékeket vagy antimintákat javítani, egyszerűen a konkrét problémákra mentek rá. Az efféle kódolási szabálysértések jól körbehatárolhatók, konkrét helyük van a kódban és a leírásuk sokszor tartalmaz megoldási javaslatot is. Ilyen problémák javításához nincs szükség részleteiben megérteni a kódot, a fejlesztők könnyen megértik a feladatot és akár olyan kódot is ki tudnak hatékonyan javítani, amin nem is dolgoztak még azelőtt. Arról nem is beszélve, hogy az ilyen átalakítások tesztelése is egyszerűbbnek bizonyult.

Az eredményeink azt mutatják, hogy egy szimpla refaktoring hatását az egész szoftver karbantarthatóságára nehéz előrejelezni, főleg úgy, hogy ezek a hatások néha hátrányosak is lehetnek. Ugyanakkor az is látszik, hogy egy teljes refaktoring periódus az egész rendszer karbantarthatóságára szignifikáns pozitív hatással lehet (lásd: 3. ábra). Ez nem csak abban mutatkozik meg, hogy a fejlesztők kijavították a rossz kódot, hanem hogy eközben sokat tanultak és egyszerűen jobban odafigyelnek már arra, hogy jól karbantartható új kódot írjanak.

A szerző hozzájárulása az eredményekhez

A szerző kézzel írott refaktoringokról gyűjtött adatokat fejlesztőktől és egy kérdőívre adott válaszokat értékelt ki. Empirikus esettanulmányt végzett azon, hogy a kézzel írott refaktoringok miképp hatnak egy szoftverrendszer karbantarthatóságára. Olyan kommitokat azonosított be, amelyekben csak refaktoringok által történt módosítások szerepeltek és ezeken méréseket végzett. Kiértékelte az eredményeket és további megfigyeléseket tett a fejlesztők hozzáállásával és viselkedéseivel kapcsolatosan.



3. ábra. A rendszerek karbantarthatósága a kézzel írt refaktoring időszak előtt és után.

II. Automatikus refaktorálás előnyeinek és hátrányainak vizsgálata

A következőkben bemutatunk egy a fejlesztők segítését szolgáló refaktoráló eszközt. Megvizsgáljuk, hogy milyen minőség-módosító hatása van a géppel végzett refaktoringoknak, miközben azt is kutatjuk, hogy történik-e változás a fejlesztők mindennapi munkamódszereiben.

Egy automatikus refaktoring keretrendszer

A Refaktoring Projekt kézzel végzett refaktoring időszaka alatt tanultak alapján megalkottunk egy kódolási szabálysértéseket automatikusan refaktoráló eszközt, melyet *FaultBuster*nek neveztünk el. A FaultBuster tervezése közben a következő célok vezéreltek: nyújtson segítséget a fejlesztőknek refaktorálható szabálysértések azonosításában; szolgáljon megoldással a beazonosított problémára automatikus refaktoring algoritmusok segítségével; zavartalanul illeszkedjen be a fejlesztési folyamatokba, azaz könnyen integrálható legyen a népszerű fejlesztési környezetekbe (Eclipse, NetBeans, IntelliJ), hogy a fejlesztők könnyedén tudjanak kedvenc szerkesztőjükben refaktoringokat alkalmazni.

A FaultBuster egy kliens-szerver alkalmazás, ahol a fejlesztői környezetek (pontosabban azok beépülő moduljai) a kliensek. A fejlesztők ezekben a kliensekben kiválasztanak egy problémás kódrészt és ezt elküldik a szerver oldalra, ahol a szerver elvégzi rajta a refaktoring átalakítást és a különbségfájlt pedig visszaküldi a kliensnek átnézésre. A fejlesztők akár testre is szabhatják ezeket az átalakításokat. Szerverre küldés előtt lehetőségük van a kliensekben beállítani, hogy mely refaktoring algoritmust szeretnék alkalmazni a probléma megoldásaként, illetve több egyéb paraméterét is be lehet állítani a mögöttes transzformációnak. Ezeken felül a keretrendszer tartalmaz még egy feladatkezelő rendszert, ahol a fejlesztők nyomon követhetik a kódolási hibák állapotait (pl.: nyitott, refaktorálás alatt, teszteletlen, elkészült, visszautasított) és ahol minden ilyen hibához hozzá lehet rendelni egy fejlesztőt, ezáltal megakadályozva a egyidejű módosításokat.

A keretrendszer arra is lehetőséget nyújtott, hogy úgynevezett csoportos refaktoringot hajtsunk végre, azaz egy probléma típusnak több előfordulását is kijavítsuk egy lépésben. Ilyenkor a rendszer egy-

szerre hajtja végre a összes kijelölt refaktoringot és a fejlesztő egyben tekintheti meg a módosításokat.

A FaultBuster által végzett refaktoring átalakításokat szerver oldali refaktoring algoritmusok hajtják végre. Összesen 40 különböző szabálysértésre készítettünk algoritmusokat. Ezek jelenleg Java nyelvű kódokon képesek elvégezni a transzformációkat. A legtöbb algoritmus gyakori programozási problémákra ad megoldást, mint üres `catch` blokk, veremkivonat kiíratásának elkerülése, vagy boolean értékadás. Emellett, néhányuk olyan heurisztikákat valósít meg, amivel olyan antiminták is javíthatók, mint a hosszú függvények, a túl komplex metódusok vagy a kód duplikációk.

Egy ilyen refaktoring algoritmus bemenete egy kódolási probléma típusa és pozíciója, valamint a forráskódból a SourceMeter elemző által készített absztrakt szemantikus gráf (ASG). A kódolási hibák azonosítását egy külső eszköz, a PMD elemző végezte. Mivel a PMD elemző csak szöveges információt ad a hiba helyéről (fájl, sor és oszlop), szükség szerűen az a hiba javításához első lépésben lokalizálni kell a problémás kódelemet az ASG-ben. Ennek megvalósításához készítettünk egy algoritmust, ami képes megtalálni a szintaxis fában az elemet pusztán a szöveges információ alapján. Ez az algoritmus áttranszformálja a forráskódot egy kereshető geometriai térbe azáltal, hogy épít egy térbeli adatbázist. Megtalálása után ez a forráskód elemen lesz a refaktoring átalakítás eredője.

Az automatikus eszköz fejlesztése közben a fejlesztők több igényét is szem előtt kellett tartanunk. Számos kihívással szembesültünk, amikor feltártuk ezeket az elvárásokat. Fontos volt számukra többek között az eszköz teljesítménye, a generált forráskód helyes indentálása és formázása, az algoritmusok működésének érthetősége, a pontos probléma detektálás, és ennek szükségleteként a pontos szintaxis fa. Ezek nagy részének komoly befolyása lehet az ilyen eszközök használhatóságára, ezért fontos, hogy már a tervezés elején számoljunk velük. Az eszközünket alapos kiértékelésnek vetettük alá, ami igazolta, hogy a megközelítésünk valós szituációkban is képes életképes eredményeket produkálni.

Az automatikus refaktoringok és a karbantarthatóság kapcsolata

A fejlesztők saját projektjeiken használták a refaktoring keretrendszert, ezáltal egy valóéletbeli teszt környezetet teremtve. A kutatási projekt végére a partnercégek közel 5 millió sornyi kódot refaktorálták és ezzel együtt felszámoltak 11.000 szabálysértést. A FaultBuster egy olyan megoldást nyújtott számukra, ami segítségével növelni tudták kódminőséget és be tudták építeni a *folyamatos refaktoring* módszertanát a fejlesztési folyamataikba. A 3. táblázat egy áttekintést nyújt a tanulmányban résztvevő rendszerekről és a rajtuk végrehajtott refaktoringok számáról.

3. táblázat. Kiválasztott rendszerek

Cég	Rendszer	kLOC	Elemzett verziók	Refaktoring kommitok	Refaktoringok
Cég I	A rendszer	1.119	299	217	1.444
Cég II	B rendszer	962	868	449	1.306
Cég III	C rendszer	206	1.313	316	404
Cég IV	D rendszer	780	200	66	682
	Összesen	**3.067**	**2.680**	**1.048**	**3.836**

A QualityGate SourceAudit elemző segítségével megvizsgáltuk a különböző refaktoring műveletek karbantartásra gyakorolt hatását. Az elemzésünk kimutatta, hogy az összes támogatott szabálysértés refaktoring közül, egy kivételével mindegyik kimutathatóan pozitív hatással volt a szoftverrendszerek minőségére. A Refaktoring Projektben részt vevő négy cég közül három mérhetően jobb karbantarthatóságú rendszerre tett szert a refaktoring időszak végére. Megfigyeltük azonban, hogy az egyes cég

az alacsony költségű módosításokat preferálta, és ezért csak két fajta refaktoringot végeztek. Az egyik ezek közül a szükségtelen konstruktorok törlése, ami a minőségmodell szerint kétes hatást gyakorolt a teljes rendszer karbantarthatóságára. Továbbá azt is megfigyeltük, hogy olykor kontraproduktív vakon alkalmazni az automatikus refaktoringokat.Többször előfordult, hogy a gép felajánlotta a fejlesztőnek, hogy válassza ki a problémára a legalkalmasabb refaktorálási módszert, de a fejlesztő lustaságból figyelmen kívül hagyta és az alapértelmezett beállításokat használta. Ezek aztán új kódolási gondokhoz vezettek vagy csak továbbgörgették a problémát. Egyszóval az emberi tényező még mindig fontos, de összességében a cégeknek sikerült mérhető növekedést elérniük a karbantarthatóságukban csak automata refaktoringok alkalmazásával.

Végül, de nem utolsó sorban a tanulmány rávilágított a szoftverminőség mérés néhány érdekes aspektusára. A váratlan hatást kiváltott refaktoringok (pl.: felesleges konstruktor) az alkalmazott minőségmodell különlegességéből adódtak. Ezek orvosolhatók rendszerszintű mérések súlyozásával.

A szerző hozzájárulása az eredményekhez

A szerző munkája meghatározó volt a kifejlesztett eszközök megtervezésében és implementálásában is. A beépülő modulokat a partnercégek készítették, de a szerző fejlesztette le a keretrendszer Java Swing-es kliensét, ami alapján a modulok készültek. A szerző definiálta a refaktoring algoritmusok működésének folyamatát és integrálta őket a keretrendszerbe. A szerző beépített egy kérdőívet a Refaktoring Keretrendszerbe, majd begyűjtötte a válaszokat és megállapításokat vont le belőle. A szerző esettanulmányt végzett a gépi refaktoringokon és megmérte, hogy ezek milyen hatással vannak a szoftver karbantarthatóságra.

III. Modell-alapú módszerek az antiminta detektálásban

A FaultBusterben a PMD-t használtuk, mint harmadik féltől származó kódolási szabálysértés detektáló eszközt. A PMD széles körben használt, a Java közösség által elfogadott nyílt forráskódú statikus elemző. Azonban rendelkezik pár hátulütővel. Először, ahogy már korábban is említettük, ahhoz hogy használhassuk refaktoringra, szükséges egy algoritmus, ami a szöveges kimenetét hozzáköti egy AST elemhez. Másodszor, számos esetben a PMD nem nyújt pontos probléma kijelölést és találatokat, ami megnehezíti a használatát. Harmadszor, a fejlesztők folyamatosan olyan esetekről számoltak be, ahol a találat nem valós problémát tükrözött (hamis pozitívak) vagy pont hogy nem találta meg az igazi problémát (igaz negatívak). Ahhoz, hogy egy pontosabb probléma detektáló eszközt hozzunk létre, megvizsgáltuk az előnyeit és költségét az iparban gyakran használt Eclipse Modelling Keretrendszernek (EMF). Az így reprezentált forráskód gráfon kipróbáltunk négy különböző modell-lekérdező technológiát és kivizsgáltuk melyiket mikor érdemes használni.

Antiminta-felismerés modell-lekérdezésekkel

Ebben a tanulmányban az antiminta felismerésre koncentráltunk. Megvizsgáltunk kétféle módszert, melyekkel olyan kereséseket írhatunk, amik lehetséges refaktoringok kezdőpontjai lehetnek. Mindkét módszer életképesnek bizonyult: (1) az egyikben a kereséseket Java kódban írjuk melyek direktben az ASG-n dolgoznak; míg (2) a másik esetben az ASG-nek egy EMF reprezentációját hozzuk létre és itt általános modell-alapú eszközöket használunk.

Az első módszer szerepét a saját statikus kódelemzőnk, a SourceMeter töltötte be. Ahhoz, hogy a forráskód-modell feldolgozását elősegítse (pl.: keresés futtatása), a SourceMeter rendelkezik egy API-val, ami lehetőséget ad látogató mintával bejárni az ASG-t. Ezek a látogatók rendre minden elemet meglátogatnak a bejárás során, így a felhasználónak nem kell kézzel bejárnia a gráfot.

A második módszerhez létrehoztunk egy EMF metamodellt a SourceMeter ASG reprezentációja alapján. A metamodellből kigenerált kód nagyon hasonló a Java ASG által deklarált interfészekhez, ezért implementáltunk egy egységes réteget, amivel mindkettő módszer közösen használható. Így lehetőségünk van az EMF magasabb szintű funkcióinak használatára, mint a változásokról való értesítés vagy reflektív úton való hozzáférés a modellhez, ugyanakkor mégis kompatibilis marad a Columbus keretrendszer elemző algoritmusaival, mivel közös interfészt használ.

Miután már adott volt a forráskód EMF modellje, elkezdhettük alkalmazni rajta a *gráfminta-illesztést*. A gráfminták [1] lényegében speciális gráfok, amelyek egyfajta lekérdezőnyelvet alkotnak. Segítségükkel kifejezhetünk egy feltételt vagy megszorítást, amit összevethetünk egy gráf példányával. Ezt a leíró nyelvet aztán különféle modell-vezérelt fejlesztésekben használhatjuk, mint például modell átalakítási szabályok leírására vagy általános modell lekérdezések definíciójára. Itt antiminta-kereső lekérdezések írására használtuk.

Elsőnek lokális keresőalgoritmusokat hoztunk létre vele. A lokális keresés alapú mintaillesztést (LS) rendszeresen használják gráf transzformáló eszközökben, ahol a keresés egyetlen elemből indul ki, majd a halmaz fokozatosan bővül az élek mentén a szomszédos pontokkal, követve a keresési tervet.

Következőnek, a lokális keresés alapú mintaillesztés egy alternatíváját, az inkrementális keresést használtuk fel. Az inkrementális mintaillesztés eltárolja a konkrét találatokat, így a keresések eredménye bármikor előhívható, nem kell újra lefuttatni őket, viszont a gyorsítótárat minden modell változás alkalmával fokozatosan frissíteni kell.

Végül OCL (Object Constraint Language) nyelven készítettünk modell lekérdezéseket. Az OCL [5] egy szabványosított, tisztán funkcionális modell-validációs és lekérdező nyelv, melyben a metamodell kontextusán belül tudunk kifejezéseket definiálni. A nyelv maga rendkívül kifejező, meghaladja az elsőrendű logika kifejező erejét azáltal, hogy olyan konstrukciókat kínál, mint a gyűjteménygyűjtő műveletek. Az OCL kifejezéseket olyan modell keresésként lehet értelmezni, ahol a megfelelő keresési tervet maga a kifejezés tartalmazza.

A projektpartnerek visszajelzései alapján hatféle antiminta típust választottunk ki, és modell lekérdezésekként formalizáltuk őket. A legfontosabb kiválasztási kritériumnak a problémák sokféleségét tekintettük. Ez olyan lekérdezéseket eredményezett, amelyek mind komplexitásban, mind a programozási nyelv kontextusában eltértek, kezdve az egyszerű bejárási- és ellenőrzési lekérdezésektől, egészen a bonyolult navigációs lekérdezésekig, akár potenciálisan negatív feltételekkel. A lekérdezéseket mindegyik kereső módszerre lefejlesztettük és összevetettük a teljesítményüket és alkalmazhatóságukat.

Lekérdezésalapú antiminta-felismerő technológiák teljesítményének összehasonlítása

Több évvel ezelőtt arra jutottunk, hogy a hagyományos modellező eszközök csak közepes méretű programgráfokat tudnak kezelni [15]. Most újra elővettük ezt a kérdést és megnézzük, hogy az általános modellalapú megoldások fejlődtek-e annyit, hogy versenybe szálljanak a kézzel írt natív megoldásokkal. A teljesítmény leméréshez összeállítottunk egy teszthalmazt, ami 28 olyan nyílt forráskódú Java projektből áll, melyek különböznek méretben és komplexitásban. Ezen a teszthalmazon minden egyes

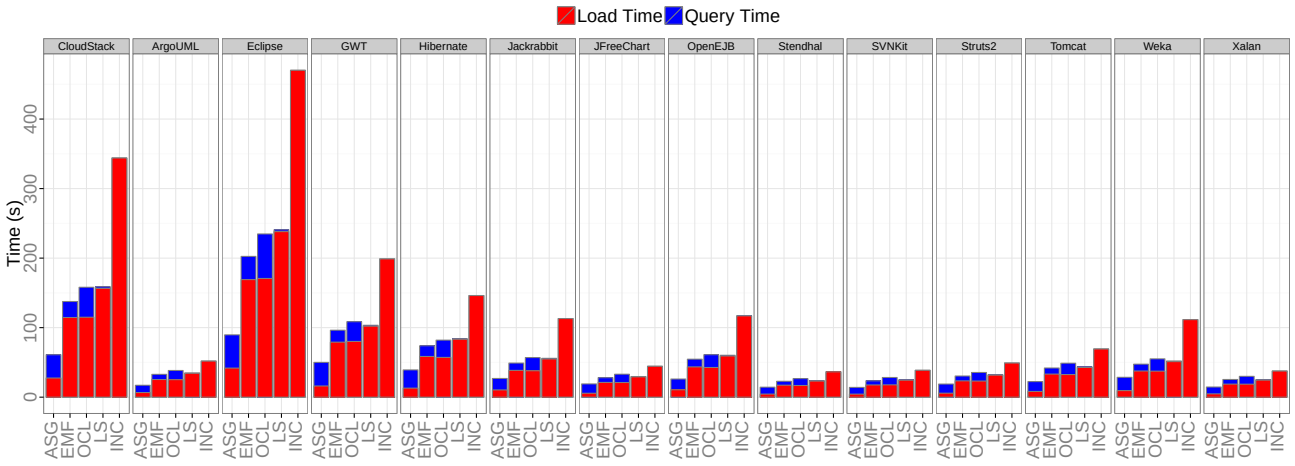
fenti lekérdezést lefuttatunk tíz alkalommal. Megmértük a modell betöltési és a lekérdezések futási idejét, mindezt három különböző használati esetben: egyszeri futásnál, többszöri keresésnél és folyamatos használat esetén.

A kísérleteink azt mutatták, hogy az ASG-hez képest a forráskód EMF-be való feltöltése 2-3 szoros memória növekedéssel és 3-4 szeres modell betöltési idővel jár, amíg az inkrementális lekérdezések jobb teljesítményt mutatnak a kézzel írt látogatókkal szemben. Ez körülbelül 2-3 nagyságrenddel gyorsabb futást jelentett, amit 10-15 szeres megnövekedett memória fogyasztás kísért. A futási idők összehasonlítása megtekinthető a 4. ábrán.

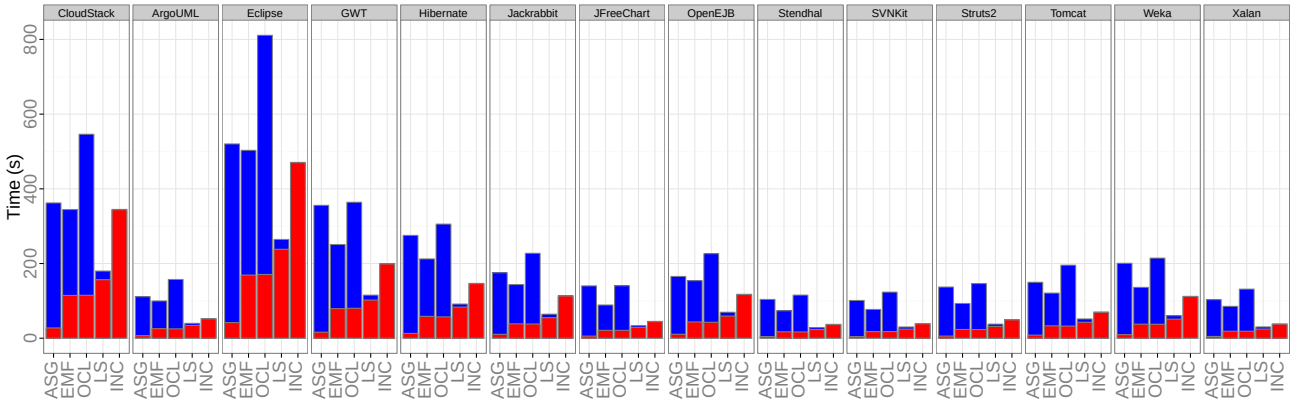
Ezt követően részletes összehasonlítást végeztünk a különböző megközelítésekről, és lehetővé tettük, hogy a kívánt felhasználási profil és a lekérdezések kifejező képességei alapján kiválasszuk megfelelő módszert. A mintaillesztési megoldások kifejezőképessége és tömör formalizmusa nagyobb teret biztosít arra, hogy kísérletezzünk a lekérdezésekkel és a különböző végrehajtási stratégiákkal, ugyanakkor, a felhasználási esettől függően, már 2 millió soros kódnál is nagy teljesítménykülönbség vehető észre.

A szerző hozzájárulása az eredményekhez

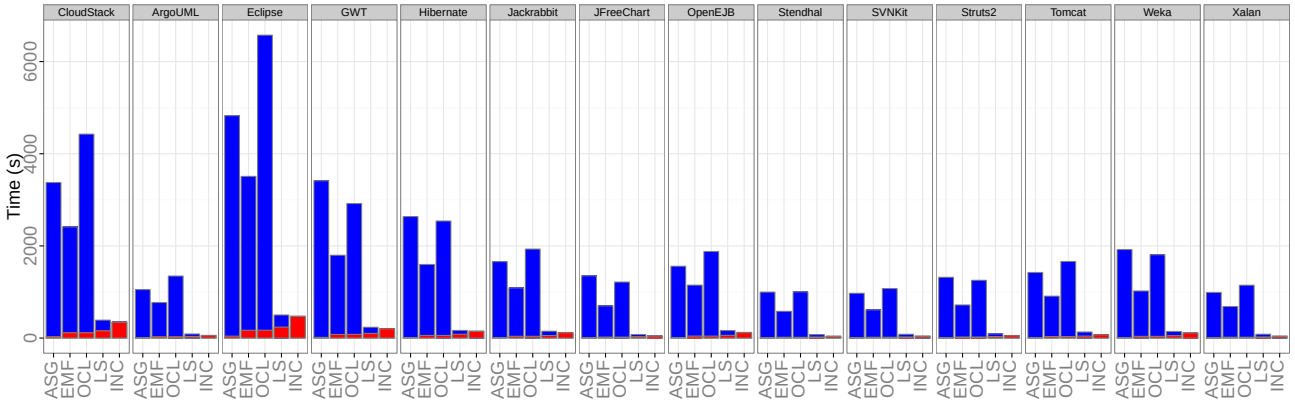
A szerző adaptálta a Columbus ASG-t Java és EMF platformokra és majd eszközöket készített használatukhoz. A szerző létrehozott egy teszthalmazt különböző méretű és komplexitású projektekből, amik alkalmasak a modell lekérdezések tulajdonságainak feltárására. Méréseket folytatott ezeken a projekteken és különböző jellemzőket állapított meg róluk. A szerző elvégezte a mérések empirikus validációját, majd kiértékelte az eredményeket. A szerző létrehozott egy döntési modellt a megállapítások alapján, amely segítségül szolgál a megfelelő módszerek kiválasztásában különböző szituációkhoz.



(a) Egyszeri futás



(b) Többszörös keresés



(c) Folyamatos használat

4. ábra. Futási idők projektenként és módszerenkként

Összefoglalás

Az eredményeink általánosságban azt mutatják, hogy a refaktorálást érdemes és lehetséges is lenne számítógépes segítséggel, automatikusan végezni. A fejlesztők igen fogékonyak olyan eszközökre, amik refaktoring javaslatokat kínálnak. Még inkább fogékonyak akkor, amikor ezek az eszközök a problémákra javítási javaslatokat is kínálnak. Azt is megmutattuk, hogy a refaktorálás jó gyakorlat, ha folyamatosan alkalmazzuk, akkor pozitív hatással van a mérhető szoftver minőségre. Továbbiakban elkészítettünk egy összehasonlítást különböző módszerekről, amelyekkel refaktorálásra alkalmas Java-s antimintákat keresünk. Emellett megvizsgáltuk azt is, hogy egy automatikus refaktoring eszköz fejlesztése milyen kihívásokból áll. Útmutatásként felvázoltunk javaslatokat, hogy mire kell odafigyelnie annak, aki olyan automata refaktoring eszközt fejleszt ami kielégíti a mai fejlesztők magas elvárásait.

Fontosnak tartjuk megjegyezni, hogy a bemutatott eredmények nagyban összefüggnek a Refaktoring Projekttel. A projekt jó lehetőséggel szolgált, hogy valós, ipari környezetben végezzünk kutatásokat, ami arra ösztönözött, hogy inkább gyakorlatias témákkal kapcsolatos tanulmányokat végezzünk.

A projekt az évek során rengeteg kutatási munkát eredményezett. Ezek közül sokat nemzetközi konferenciákon mutattak be, köztük azt a tanulmányt is ([13]), amely megnyerte a legkiemelkedőbb címet az IEEE CSMR-WCRE 2014 konferencia szoftverfejlesztési héten, Antwerpenben.

Következőkben újra feltesszük a kutatási kérdéseinket és egyesével megválaszoljuk őket.

K1: Mit tesznek a szoftverfejlesztők, ha külön pénzt és időt kapnak refaktoring feladatok elvégzésére?

Tanulmányaink értékes betekintést nyújtanak a fejlesztők tevékenységeibe. A kutatásunk során 1.273 darab kézzel írott és körülbelül 6.000 gépi refaktoringot gyűjtöttünk össze nagyméretű, éles ipari rendszerekből. Azt találtuk, hogy a fejleszők sokkal inkább kódolási szabálysértéseket javítanak szemben az antimintákkal vagy metrika-alapú szabálysértésekkel. Azt is megfigyeltük, hogy optimalizálják a refaktoring folyamatot és a súlyosabb hibákkal kezdik a javítást. Mindemellett azért igyekeznek inkább a könnyen javítható hibákat választani feladatként. A fejlesztőkkel való interjúink és az analízisünk is mind arra engedtek következtetni, hogy a projekt végére megtanultak jobban karbantartható kódot írni.

A kutatásunk folytatásaként egy automatikus refaktoring eszközzel láttuk el a fejlesztőket. Kiderült, hogy optimisták az automatizációval kapcsolatban és hogy egy automata eszköz növelheti hatékonyságukat. Úgy gondolják, hogy a legtöbb kódolási szabálysértés egyszerűen javítható automatikus transzformációkkal. Ez a bizalom meg is mutatkozott, amikor sokszor vakon elfogadták az automatikus eszköz refaktoring javaslatait, anélkül, hogy átnézték volna a javasolt változtatást. Több esetben, amikor az eszköz felkínálta a lehetőséget a legjobb refaktoring módszer kiválasztására, ők azt az alapbeállításon hagyták, mert az kényelmesebb volt. A partnerek alapvetően elégedettek voltak az automata refaktoring megoldással és többször kérték hogy bővítsük ki a repertoárunkat egy-egy új szabálysértésre adott megoldással. Azt találtuk, hogy a legjobban kedvelt funkciójuk a csoportos refaktoring volt, ahol több hasonló problémát tudtak egyszerre javítani, mert ez nagyban növelte a produktivitásukat.

K2: Milyen fejlesztői igényeknek kell megfelelnie egy automatikus refaktoringokat támogató eszköznek?

A kézzel írott refaktoring időszakban azt láttuk, hogy a fejlesztők leginkább szabálysértéseket javítanak refaktoring gyanánt. Az is kiderült, hogy nem szeretnek átváltani egy refaktorálásért a normál fejlesztői környezetükből egy külön refaktoring eszközre, ezért fontos az eszköz integrálása a fejlesztői környezetbe. Eredményeink azt mutatják, hogy egy refaktoring eszköz legfontosabb tulajdonsága, hogy refaktoring javaslatokat adjon, azaz megmutassa mit és hogyan lehet javítani. Ehhez pontos probléma detektálásra van szükségünk, hogy elkerülhessük a hamis ajánlásokat. Úgy találtuk, hogy a refaktoring átalakításoknak könnyen áttekinthetőnek és jól dokumentáltnak kell lenniük. A fejlesztők a megértés hiányából fakadóan az egyszerűbb átalakításokat választották, szemben az olyan komplexebb refaktoringokkal, mint a kód klónok kiemelése. Érdekes felfedezés volt, hogy a különböző cégek eltérő megoldásokat kértek ugyanarra a probléma típusra. Ezzel párhuzamban a fejlesztők pedig azt kérték, hogy a refaktoringok ne teljesen automatán viselkedjenek, hanem futtatás előtt testre lehessen szabni az algoritmusok néhány paraméterét. Egy teljesen automata megoldás nem felelt meg nekik, beleszólást kértek a folyamatba, ezért fél-automata megoldás a javallott. A döntés lehetőségét a refaktoring paraméterek beállításán kívül – a folyamat végén – a refaktoring eredményének elfogadásánál is igényelték a fejlesztők. Ezen a ponton még áttekintik a módosításokat, lefuttatják a unit és integrációs teszteket, és csak ez alapján hoznak döntést a változtatás elfogadásáról. A fejlesztők azokat a refaktoring algoritmusokat tekintették jónak, amik az átalakítás során helyesen formázták a kódot, figyeltek a indentálásra, és csak minimális változtatásokat végeztek a kódbázison. Emellett azt is kérték, hogy a kommentekre is figyeljünk oda, például egy metódus törlésénél a hozzá tartozó kommentet is szüntessük meg.

A fenti irányelvek alapján megalkottuk a saját automatikus refaktoring eszközünket, a Faultbustert. A FaultBuster két olyan tulajdonsággal is rendelkezik, amelyek különbbé teszik a hasonló eszközökhöz képest. Az egyik, hogy kliens-szerver architektúrával készült, így rendelkezik egy feladatkezelő modullal is, ami megakadályozza, hogy ugyanazt a problémát véletlen kétszer kijavítsuk. Ugyanakkor, a másik lehetővé teszi a programozóknak több ugyanolyan típusú probléma egyszerre történő kijavítását. A FaultBuster főleg a kódolási szabálysértések és gyanús kódok javításával foglalkozik. A belsejében egy jól definiált automatikus refaktoring folyamat működik, ami a program modelljén végez transzformációkat. Ennek a folyamatnak a része az az algoritmus is, amik képes a talált hibákat ráilleszteni a forrásból épített szintaxisfára.

K3: Milyen hatással vannak a szoftverminőség változására a kézzel írt és a gépi támogatással készített refaktoringok?

A kutatási projekt során sok refaktoring kommitot azonosítottunk. Elsőnek 315-öt a kézzel írt fázisban, majd 1048-at a gépi támogatásos időszakban. A QualityGate SourceAudit program segítségével leelemeztük ezeket kommitokat és megállapítottuk az egyes refaktoringokból származó módosítások mennyiben befolyásolták a karbantarthatóságot. Azáltal, hogy megnéztük a minőségmutatót az adott rendszerre – a refaktoring alkalmazása előtt és után – értékes információkat nyertünk nagyméretű, ipari projektek refaktorálásának hatásairól.

Megtanultuk, hogy egy szimpla refaktoring kimenetele a teljes szoftverrendszer karbantarthatóságára nehezen előjelezhető, sőt, olykor negatív eredménye is lehet. Általánosságban azonban, egy

teljes refaktoring folyamat jelentősen tudja növelni a mérhető karbantarthatóságot. Azt találtuk, hogy antiminták eliminálásával lehet legjobban növelni a karbantarthatóságot, melyet a kódolási szabálysértések és metrika mutatók javítása követ. Emellett a tanulmány rávilágított a szoftverminőség mérés néhány érdekes aspektusára is, például, hogy a váratlan hatást kiváltott refaktoringok (pl.: felesleges konstruktor) az alkalmazott minőségmodell különlegességéből adódtak.

Az eredményeink nem sugallnak szignifikáns különbséget a kézzel írott és a gépi refaktoring tevékenységek szoftverminőségre gyakorolt hatása szemszögéből. Amennyiben a refaktoring egy út a szoftvermennyországba vagy a szoftverpokolba, akkor az automatikus refaktoring csak egy gyorsabb út ugyanoda.

K4: Használhatjuk a gráfmintaillesztést olyan antiminták azonosítására, amelyek kezdőpontjai lehetnek egy refaktoring folyamatnak?

Megvizsgáltuk milyen költségekkel és előnyökkel jár, ha a népszerű EMF keretrendszert használjuk a forráskód modelljének reprezentációjaként. Négyféle általános lekérdező módszert teszteltünk ezen, melyek natív Java kódon, OCL kiértékelésen és (inkrementális) gráfmintaillesztésen alapultak. Összehasonlítottuk ezeket a módszereket egy 28 darab nyílt forráskódú Java projektből álló teszthalmazon. Mindegyik módszerrel többfajta antiminta-kereső lekérdezést valósítottunk meg, majd lefuttattuk és lemértük őket különböző felhasználási esetekben.

A fő megállapításunk az, hogy a fejlettebb modell-lekérdezések EMF felett akár több nagyságrenddel gyorsabban is futhatnak, mint a hagyományos, kézzel írott módszerek. Ugyanakkor ez a teljesítménynövekedés egy 10-15-szörös memóriahasználatbeli emelkedéssel járhat, és a modellek betöltési ideje is 3-4-szeresére növekedhet. Ezért érdemes előre eltervezni, hogy milyen szituációkban szeretnénk használni a lekérdezéseinket, hány alkalommal és milyen gyakran lesznek futtatva a betöltés után. A négy módszer közül bármelyik alkalmas lehet olyan antiminták felismerésére, amelyek alkalmasak refaktoring javaslatok azonosítására.

Köszönetnyilvánítás

Mindenek előtt szeretném megköszönni témavezetőmnek, Dr. Ferenc Rudolfnak, az érdekes kutatási témákat és célokat, a biztos hátteret és a vezetését éveken keresztül. Külön köszönettel tartozok szerzőtársamnak, Dr. Nagy Csabának, aki mint mentorom irányította a munkámat és segített az évek alatt. Meg szeretném köszönni Dr. Fülöp Lajosnak, aki bátorított, hogy a kutatói pályára lépjek. Szintén szeretném megköszönni Dr. Gyimóthy Tibornak a kutatómunkám során nyújtott folyamatos támogatását. Köszönöm továbbá David P. Curleynek a munka nyelvi helyességének ellenőrzését és javítását. Köszönettel tartozom még kollégáimnak és társszerzőimnek, Dr. Vidács Lászlónak, Dr. Hegedűs Péternek, Antal Gábornak, Csiszár Norbertnek, Dr. Ujhelyi Zoltánnak, Dr. Horváth Ákosnak, Dr. Varró Dánielnek, Sógor Zoltánnak, Siket Péternek, Dr. Siket Istvánnak, Balogh Gergőnek és Ladányi Gergelynek. Külön köszönettel tartozok Dr. Beszédes Árpádnak és Dr. Gombás Évának, akik segítettek a szingapúri ösztöndíjam megszervezésében. Köszönet továbbá a tanszék minden dolgozójának az évek során nyújtott támogatásért.

A disszertáció egy fontos része foglalkozik a Refaktoring Projekttel. Ez a munka nem jöhetett volna létre a pályázati partnerek együttműködése nélkül. Külön köszönöm ezért a projektben részvevő tanszékes kollégáknak és a cégek munkatársainak az együttműködését.

Végül, de nem utolsó sorban, köszönöm családomnak, szüleimnek és barátaimnak, hogy mindenben támogattak és biztattak a munkám során.

Szőke Gábor, 2019. június

Hivatkozások

[1] *Gábor Bergmann, Zoltán Ujhelyi, István Ráth, and Dániel Varró. A graph query language for EMF models. In* Theory and Practice of Model Transformations, *volume 6707 of* Lecture Notes in Computer Science, *pages 167–182. Springer Berlin / Heidelberg, 2011.*

[2] *Rudolf Ferenc, Árpád Beszédes, Mikko Tarkiainen, and Tibor Gyimóthy. Columbus – Reverse Engineering Tool and Schema for C++. In* Proceedings of the 18th International Conference on Software Maintenance (ICSM 2002), *pages 172–181. IEEE Computer Society, October 2002.*

[3] *Martin Fowler.* Refactoring: Improving the Design of Existing Code. *Addison-Wesley Longman Publishing Co., Inc., 1999.*

[4] *Bennet P Lientz, E. Burton Swanson, and Gail E Tompkins. Characteristics of application software maintenance.* Communications of the ACM, *21(6):466–471, 1978.*

[5] *Object Management Group.* Object Constraint Language Specification (Version 2.3.1), *May 2012.* `http://www.omg.org/spec/OCL/2.3.1/`.

[6] *Gábor Szőke, Gabor Antal, Csaba Nagy, Rudolf Ferenc, and Tibor Gyimóthy. Bulk fixing coding issues and its effects on software quality: Is it worth refactoring? In* 14th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2014, Victoria, BC, Canada, September 28-29, 2014, *pages 95–104, 2014.*

[7] *Gábor Szőke, Csaba Nagy, Rudolf Ferenc, and Tibor Gyimóthy. A case study of refactoring large-scale industrial systems to efficiently improve source code quality. In* Computational Science and Its Applications - ICCSA 2014 - 14th International Conference, Guimarães, Portugal, June 30 - July 3, 2014, Proceedings, Part V, *pages 524–540, 2014.*

[8] *Gábor Szőke, Csaba Nagy, Lajos Jeno Fülöp, Rudolf Ferenc, and Tibor Gyimóthy. Faultbuster: An automatic code smell refactoring toolset. In* 15th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2015, Bremen, Germany, September 27-28, 2015, *pages 253–258, 2015.*

[9] *Gábor Szőke, Csaba Nagy, Péter Hegedűs, Rudolf Ferenc, and Tibor Gyimóthy. Do automatic refactorings improve maintainability? an industrial case study. In* 2015 IEEE International Conference on Software Maintenance and Evolution, ICSME 2015, Bremen, Germany, September 29 - October 1, 2015, *pages 429–438, 2015.*

[10] *Gábor Szőke. Automating the refactoring process.* Acta Cybernetica, *23(2):715–735, Jun. 2017.*

[11] *Gábor Szőke, Gábor Antal, Csaba Nagy, Rudolf Ferenc, and Tibor Gyimóthy. Empirical study on refactoring large-scale industrial systems and its effects on maintainability.* Journal of Systems and Software, *2016.*

[12] *Gábor Szoke, Csaba Nagy, Rudolf Ferenc, and Tibor Gyimóthy. Designing and developing automated refactoring transformations: An experience report. In* IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016 - Volume 1, *pages 693–697, 2016.*

[13] *Zoltán Ujhelyi, Ákos Horváth, Dániel Varró, Norbert Istvan Csiszár, Gábor Szőke, László Vidács, and Rudolf Ferenc. Anti-pattern detection with model queries: A comparison of approaches. In* 2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering, CSMR-WCRE 2014, Antwerp, Belgium, February 3-6, 2014, *pages 293–302, 2014.*

[14] *Zoltán Ujhelyi, Gábor Szőke, Ákos Horváth, Norbert Istvan Csiszár, László Vidács, Dániel Varró, and Rudolf Ferenc. Performance comparison of query-based techniques for anti-pattern detection.* Information & Software Technology, *65:147–165, 2015.*

[15] *László Vidács. Refactoring of C/C++ Preprocessor constructs at the model level. In* Proceedings of the 4th International Conference on Software and Data Technologies (ICSOFT 2009), *pages 232–237, July 2009.*