Szegedi Tudományegyetem

Informatikai Tanszékcsoport

Antiminták, felismerésük és hatásuk

TDK dolgozat

Készítette:

Bán Dénes

MSc hallgató

Témavezetők:

Dr. Ferenc Rudolf

egyetemi adjunktus

Hegedűs Péter

doktorjelölt

Szeged

2014

Tartalomjegyzék

Tartalomjegyzék 2

ABSTRACT 4

BEVEZETÉS 5

1. AZ ANTIMINTÁKRÓL 7

1.1. Mik azok az antiminták? 7

1.2. Miért fontos a felismerésük? 8

1.3. A felhasznált antiminták rövid bemutatása 8

2. LÉTEZŐ FELISMERŐ ALGORITMUSOK BEMUTATÁSA 11

2.1. Általános mintafelismerési módszerek 11

2.1.1. Illeszkedés szerinti csoportosítás 11

2.1.2. A vizsgálat aspektusa szerinti csoportosítás 11

2.1.3. A rendszer reprezentálása szerinti csoportosítás 12

2.1.4. A minták reprezentálása szerinti csoportosítás 13

2.1.5. Az eredmények formátuma szerinti csoportosítás 13

2.2. Irodalmi áttekintés 14

2.3. A saját algoritmus 16

3. A LIM ALAPÚ ALGORITMUS IMPLEMENTÁCIÓJA 17

3.1. A program szerkezete 17

3.2. Használat 17

3.3. Függőségek 18

3.2.1. LIM 18

3.3.2. LIMMETRICS 19

3.3.3. GRAPH 19

3.4. Konfiguráció 20

3.5. Részletes bemutatás 20

3.6. Felismert antiminták 24

4. A HASZNÁLT ANTIMINTÁK TESZTELÉSE 26

4.1. Kisméretű, célzott tesztek 26

4.2. Tesztelés nagyobb, valós rendszereken 28

5. HATÁSVIZSGÁLAT 31

5.1. Minőségi modellek 31

5.2. PROMISE 32

5.3. Korreláció keresése 32

5.4. Gépi tanulási módszerek 34

6. JÖVŐBELI TERVEK 37

7. HELYESSÉGET VESZÉLYEZTETŐ TÉNYEZŐK 38

8. ÖSSZEFOGLALÁS 39

Irodalomjegyzék 40

Köszönetnyilvánítás 42

// ABSTRACT

A tervezési mintákhoz és a bennük hordozott, rendszertervezéssel kapcsolatos többletinformációkhoz hasonlóan az antiminták – vagy bad smell-ek – is nagyban befolyásolhatják a szoftverek minőségét. Habár széles körben elfogadott, hogy rontják a karbantarthatóságot, mégis aránylag kevés objektív eredmény támasztja ezt alá.

Ebben a dolgozatban egy felismerési módszer implementálása után a tesztelési eredményekből megpróbálok kapcsolatot kimutatni az antiminták és különböző, minőséghez kapcsolódó jellemző között. Ehhez a kinyert minták mellett egy korábban már bemutatott, valószínűségi minőségi modellt, és egy nyílt bug adatbázist is fel fogok használni.

Az elemzések alapján megállapíthatjuk, hogy statisztikailag szignifikáns, **0.55**-ös Spearman korreláció van a bugok és az antiminták száma között, illetve egy még erősebb fordított, **-0.62**-es Spearman korreláció található az antiminták és a karbantarthatóság esetében.

Ezeken felül még azt is sikerül empirikusan validálnom, hogy már ez a néhány megvalósított antiminta is elég jól tudja replikálni a forrásból kinyerhető metrikák bug-előrejelző képességét.

Kulcsszavak: Antiminák, Szoftver karbantarthatóság, Bug előrejelzés, Empirikus validáció

BEVEZETÉS

Egy átlagos szoftver életciklusának karbantartási fázisában a teljes költségvetésének akár 70%-át is felemésztheti [1]. Ez már önmagában is elég ok lehet arra, hogy minden olyan tevékenységet minél jobban próbáljunk támogatni, ami a szoftverek módosítását, bővítését, javítását vagy refaktorálását segíti elő. És akkor még nem vettük figyelembe a sebességgel, a skálázhatósággal és a megismételhetőséggel [2] kapcsolatos problémákat vagy a befektetett programozói munka nehézségét. Ezek a követelmények pedig már felvetik a fenti tevékenységek – fél vagy teljes – automatizálásának kérdését is, ami habár a közeljövőben nem fog az emberi megítélő képesség közelébe jutni, mégis lehetővé teszi akár milliós LLOC méretű rendszerek gyors elemzését is. Az outputként kapott illeszkedési jelöltek halmazából pedig azért jelentősen könnyebb a fals pozitívokat kiszűrni, mint a teljes rendszert manuálisan átvizsgálni.

Minden szoftver elkerülhetetlenül degradálódik az idő múlásával, mivel a fenntartási időszakban sem a hibákkal kapcsolatos javítási munkálatok, hanem a megváltozott követelmények és új elvárások miatt szükséges új funkcionalitás implementálása a hangsúlyosabb. Ilyen körülmények között pedig a módosítások szaporodásával egyre jobban előjönnek azok a tervezési hibák, amik az elején még megfelelő megoldásnak tűnhettek, de hosszú távon több baj van velük, mint amennyire hasznosak. Így előbb vagy utóbb a legtöbb szoftverrendszer eljut egy olyan pontra, ahol már elkerülhetetlen a refaktorálás, ha továbbra is biztosítani szeretnék a stabil és megbízható működést a bővülés mellett. A gond ezzel a menetrenddel csak az, hogy ilyenkor az esetleges átalakítások – főleg ha azok az architektúra alapvető pontjait is nagyban érintik – már csak sokszoros költséggel orvosolhatók a korai adaptálással szemben. Ha egy szoftver ilyen „gyengepontjait” már egy korai fázisban, gyorsan, egyszerűen és főleg automatikusan detektálni lehetne, az nagyban segíthetné a fejlesztők munkáját, felgyorsíthatná a későbbi fejlesztéseket és csökkenthetne a karbantartási költségeken.

Jelen dolgozat első fejezetének célja ezeknek a „gyengeségeknek” egy kicsit formálisabb definiálása és a később felhasznált antiminták ismertetése. A második fejezetben először általános, mintafelismeréssel kapcsolatos stratégiákat mutatok be, utána a – szorosabban az antimintákhoz kapcsolódó – szakirodalom áttekintésével adok képet a kutatási terület jelenlegi állapotáról, majd az addigiak fényében kategorizálom a saját algoritmust. A harmadik fejezet a konkrét algoritmus implementációjának összefoglalását, használatát,

függőségeit és részletes szerkezetét valamint a vele felismerhető antiminták probléma-specifikus ábrázolását tárgyalja. A negyedik fejezetben elsőször kisebb, célzott tesztekkel validálom a program helyes működését, majd nagyobb, „éles” rendszereken is futtatom.

Az ötödik fejezetben az előző szekció nagyobb rendszereiből kinyert eredményekkel korreláció és gépi tanulási módszerek segítségével megvizsgálom, hogy az antiminták milyen hatással vannak egy szoftver objektív minőségére. Pontosabban a következő három kérdésre keresek választ:

1. Hogyan függ össze az antiminták és a bugok száma?
2. Hogyan függ össze az antiminták száma és a karbantarthatóság?
3. Felhasználhatók-e az antiminták bug-előrejelzésre?

Ezek után a hatodik fejezetben a kutatás további lehetséges folytatási lehetőségeiről, a hetedik fejezetben pedig az elért eredmények helyességét veszélyeztető tényezőkről írok. Végül a nyolcadik fejezetben összefoglalom az addigiakat.

1. AZ ANTIMINTÁKRÓL

1.1. Mik azok az antiminták?

Egy antiminta egy probléma olyan, gyakran ismétlődő megoldása, ami tapasztalati úton bizonyítottan negatív következményekhez vezet. Lehet eredménye annak, hogy a fejlesztő nem rendelkezik elég tudással egy adott probléma helyes megoldásához, vagy helyes mintát használ ugyan, de rossz kontextusban. [3]

Egy antiminta leírásának legfontosabb részei:

- **A probléma leírása**: mi volt az az eredeti megoldandó feladat, ami kiváltotta az általános megoldást
- **Az általános megoldás**: maga az antiminta, vagyis az a megoldás, ami habár intuitív vagy elsőre jónak tűnik, mégsem optimális
- **A következmények**: milyen negatív utóhatásokra számíthatunk, ha mégis ezt a megoldást alkalmazzuk
- **A tünetek**: mik azok a jelek, amikből felismerhetjük, hogy esetleg a kódunkban egy ilyen antiminta szerepel
- **A refaktorált megoldás**: hogyan változtathatjuk meg az antimintát úgy, hogy megszabaduljunk a negatív következményektől

A refaktorálás lényege, hogy egy szoftver egy részének szerkezetét valamilyen előre meghatározott szempontból – karbantarthaóság, változtathatóság, tesztelhetőség – kedvezőbb állapotúra alakítjuk úgy, hogy közben a működése nem változik. [4]

Ezt természetesen manuálisan is megtehetnénk, de az antiminták esetében az előforduló „hiba" és annak szakértői vélemények és empirikus tapasztalatok alapján is kedvezőbb megoldása volt olyan gyakori, hogy a refaktorálást már megtették helyettünk, nekünk csak alkalmaznunk kell azt a megfelelő kontextusokban.

Jogosan felmerülhet, hogy az eddigiek nagyon hasonlítanak a tervezési mintákhoz [5] és azok katalogizálásához. Ez nem véletlen, az antiminták tulajdonképpen bizonyos értelemben azonos, bizonyos értelemben pedig épp ellentétes szerepet töltenek be. A fő különbség – nyilván a pozitív és negatív hatáson kívül – az, hogy míg a tervezési mintáknál elég annyit megadni, hogy milyen helyzetekben melyiket érdemes használni, itt szinte a legfontosabb a

„mikor?” és a „melyiket?” mellett az, hogy mire alakítsuk át, hogy a negatív következményeket eltűntethessük.

Szintén jogosan felmerülhet, hogy mi a különbség egy antiminta és egy egyszerű hiba vagy rossz szokás között? A válasz itt is abban rejlik, hogy míg egy hibánál csak azt tudjuk, hogy „így ne”, egy antiminta esetében azt is megadjuk, hogy „inkább így”.

1.2. Miért fontos a felismerésük?

Mint ahogy a bevezetésben is írtam, lehet, hogy egy fent említett típusú anitminta katalógus „támogatja” ugyan egy szoftver fejlesztését vagy karbantartását, a megoldás még mindig viszonylag nagy szakértelmet, sok befektetett munkát és minden alkalommal eseti vizsgálatot igényel, emiatt tehát nem gyors, egyszerű, megismételhető vagy skálázható. Sőt, néhány rendszernél még csak nem is kivitelezhető.

Ahhoz hogy ezt az ígéretes módszertant a gyakorlatban is sikeresen alkalmazhassuk, nagyon fontos a felismerés automatizálása. Legtöbbször még ezután is szükség lesz manuális átvizsgálásokra, de egy szűkített halmazon az már sokkal kisebb erőforrásból lehetséges.

1.3. A felhasznált antiminták rövid bemutatása

Ebben a részben csak a később ismertetett programmal jelenleg felismerhető antimintákra, azokon belül is inkább csak a felismerésükhöz szükséges tulajdonságaikra, tüneteikre térek ki, mert az automatizáláshoz csak ezeket használom fel. Az említett animintákról részletesebb leírás – többek között – a [4] könyben található.

Feature Envy (FE): Az objektum orientáltság egyik alappillére, hogy egybezárja az adatokat a rajtuk végezhető műveletekkel. Ebből eléggé nyilvánvalóan következik, hogy ha egy adott osztály valamely metódusa többet „foglalkozik” egy másik osztály attribútumaival, mint a saját osztályában lévőkkel, akkor ez utalhat arra, hogy a metódus valóban nem is oda tartozik.

Lazy Class (LC): Egy osztályt lustának nevezünk, ha ő magában igazából nem lát el fontosabb feladatot, csak minimális logika segítségével delegálja a neki küldött kéréseket másik osztályok felé. Vagyis nem komplex, de asszociáció vagy aggregáció útján több másik osztállyal is kapcsolatban van. Ha viszont ez a helyzet, akkor érdemes lehet megvizsgálni, hogy számít-e ez az indirekció és az osztálynak tényleg van-e létjogosultsága, vagy például az őt használó elemek és az általa továbbhívott elemek közvetlen összekapcsolásával egyszerűsíthető-e a rendszer szerkezete.

Large Class Code (LCC): A procedurális paradigmáról az objektum orientáltságra áttérők esetében szokott gyakran előfordulni, hogy egyetlen, vagy kevés osztályt használnak, mert alapelemeknek az azokon belüli metódusokat tekintik, nem pedig a modellezett objektumokat. Ez viszont egy nagyobb lélegzetvételű rendszer esetén hatalmas méretű osztályokhoz vezet. Ezzel szemben – általában – aki helyesen használja a paradigma által nyújtott lehetőségeket, annál jóval több, de egyenként pedig jóval kisebb osztály az eredmény. Ebből következtethetünk, hogy ha egy túl sok – logikai – sorból álló osztályt találunk, ott refaktorálni lenne érdemes.

Large Class Data (LCD): Ebben az esetben nem az osztály metódusainak sor- vagy utasításszáma a sok, hanem az adat, amit egységbe próbál zárni. Azért használom a „próbál" kifejezést, mert egy bizonyos korláton felül már egyre valószínűtlenebb, hogy az a sok adat tényleg csak egy egységbe tartozik. A legtöbb ilyen osztálynál az attribútumok egy vagy több részhalmaza és az azokhoz szorosabban kapcsolódó metódusok részegyedekké szervezhetők ki, amikből aggregáció vagy kompozíció segítségével állhat elő az eredeti funkcionalitást modellező entitás.

Long Function (LF): A Large Class Code mintához hasonlóan a Long Function is legtöbbször vagy a proceduralitásra vagy a viselkedés nem megfelelő feldarabolására utal. Megjegyezném még, hogy a „hosszúság" és a „nagyság" különböző értelmezései miatt érdemes lehet a sorok számán kívül a metódus más jellemzőit, például a csatolását, utasításait vagy komplexitását is figyelembe venni. Egy hosszú metódus az esetek túlnyomó többségében részfunkcionalitások külön metódusba történő kiemelésével rövidebbé és könnyebben érthetővé tehető.

Long Parameter List (LPL): A hosszú paraméter lista az egyik legkönnyebben felismerhető és az egyik leguniverzálisabban előnytelennek elfogadott antiminta. Ahogyan Alan Perlis is vélekedett: „If you have a procedure with 10 parameters, you probably missed some". Egy ilyen helyzet általában orvosolható, ha értékeket attribútumként veszünk fel – amennyiben azoknak a metódus futásán kívül, az osztály egész élettartama alatt van értelmük –, vagy egy külön „paraméter osztály"-ba szervezzük ki őket, amit aztán esetleg viselkedéssel is bővíthetünk és könnyebben átadhatunk.

Refused Bequest (RB): Ha egy osztályban egy attribútumot vagy metódust „protected" láthatósági módosítóval látunk el, azzal arra utalunk, hogy arra az osztály leszármazottainak is szüksége lesz. Ezt persze betartatni nem tudjuk, mert a leszármazott attól, hogy lát egy bizonyos mezőt, még nem biztos, hogy használni is fogja. Ha viszont nem használja, az általában egy tervezési, strukturális hibára hívhatja fel a figyelmünket.

Shotgun Surgery (SHS): Ezt a mintát onnan lehet felismerni, ha egy elem megváltoztatása után mindig több másik, helyileg is máshol található elemet szintén módosítani kell. Ha ezt minden esetben el kell végeznünk, akkor könnyen kihagyhatunk egy-egy fontos átalakítást, ami bugok forrása lehet. Optimális esetben a változtatásoknak lokalizáltan kellene hatniuk, ezért ilyenkor érdemes lehet attribútumok vagy metódusok áthelyezésével csoportosítani az „együtt mozgó” elemeket.

Temporary Field (TF): Ha egy osztály olyan attribútummal rendelkezik, amit nem, vagy csak néha használnak a metódusai, akkor a teljes figyelmen kívül hagyás esetében fölösleges is számára tárhelyet foglalni, de a ritka használat is jelentősen nehezíti a megértést. Az ilyen attribútumok eltűntetése, vagy egy külön osztályba történő kiemelése a velük szorosabban „foglalkozó” metódusokkal együtt, nagyban javíthatja a kód átláthatóságát.

2. LÉTEZŐ FELISMERŐ ALGORITMUSOK BEMUTATÁSA

2.1. Általános mintafelismerési módszerek

A tervezési minták felismerésével foglalkozó BSc-s szakdolgozatomban már összefoglaltam az általános, mintafelismeréssel kapcsolatos módszereket több szempont szerint csoportosítva. Mivel ezek az antimintákkal és azok kinyerésével foglalkozó algoritmusok esetében is ugyanolyan relevánsak, most itt is összefoglalnám őket. [6]

2.1.1. Illeszkedés szerinti csoportosítás

Az illesztő algoritmusok fontos ismertető jegye, hogy csak tökéletes egyezést detektál, vagy képes-e egy olyan értéket visszaadni, ami megmutatja, hogy mennyire jártunk közel. Az első gyakorlatilag egy *boolean* válasz, – van-e találat – a második pedig egy metrika, a hasonlóság mértéke.

A csak pontos egyezést elfogadó verzió – az algoritmus többi szempontjától függetlenül – általában egyszerűbben implementálható. Futási időben is jobban járunk, mert kihasználhatjuk, hogy amint találunk egy eltérést, azonnal befejezhetjük a keresést. Igaz, hogy az eredmények szempontjából nem elsődleges a programozók munkájának könnyítése, és az idő sem tűnhet olyan nagy előnynek, de bizonyos rendszerek rendelkezhetnek akkora méretekkel, illetve bizonyos illesztési módok rendelkezhetnek olyan kedvezőtlen skálázhatósággal, hogy nem mindig lesz választásunk.

Ha viszont a lehetőségek engedik, – a látszólagos paradoxon ellenére – pontosabb eredményeket kaphatunk, ha nem csak pontos találatokat keresünk. Ennek nyilván ára van komplexitás és futás terén is, de ha megfelelően értékeljük ki az egyezés mértékét, elkerülhetünk sok fals negatívot. Legtöbbször ezekhez a keresésekhez egy küszöbérték paraméter is tartozik, amely állításával megadhatjuk, hogy mekkora hasonlósági érték felett tekintünk egy potenciális illeszkedést találatnak. Így a vizsgálat rendszerenként finomhangolható, és ha a küszöböt a lehetséges maximális értékére állítjuk, akkor a pontos egyezés is szimulálható.

2.1.2. A vizsgálat aspektusa szerinti csoportosítás

Arra is több lehetőség van, hogy a rendszert milyen oldalról vizsgáljuk. Ez lehet:

- **Szerkezeti**: Ekkor csak a szoftver statikus felépítése releváns. Abból kinyerhetünk olyan, osztályokra vonatkozó információkat, mint például:
  - „Mely osztályok az ősei/leszármazottai?”
  - „Mely osztályokkal van kapcsolatban?” (asszociáció)
  - „Mely osztályok példányait tartalmazza?” (kompozíció, aggregáció)
  - „Absztrakt-e?”
  - „Tartalmaz-e másik osztályéval hasonló signature-ű metódust?”, stb.

  Ezek az információk – mint ahogy a csoport neve is mutatja – bőven elegendők a szerkezeti tervezési minták felismeréséhez, de lehetnek helyzetek, amikor másra is szükség van.
- **Viselkedési**: Itt már nem az a fontos, hogy egy adott osztály mikkel lép*het* kapcsolatba, hanem hogy ténylegesen mikkel *lép* kapcsolatba. Sok mintánál előfordulnak olyan gyakori, egymást követő eseménysorozatok, amelyek szűrésével pontosíthatjuk a keresést. Ezek elemzéséhez az egyik legelterjedtebb és legpontosabb módszer a kód futtatása és hívási gráfok építése. Emellett léteznek statikus alapú algoritmusok is, amik valamilyen statisztikai becsléssel következtetnek a valós dinamikus viselkedésre.
- **Szemantikai**: Az utolsó – és az NLP relatív fejletlensége miatt valószínűleg legkevésbé használt – aspektus a forrásban használt nevezéktanok figyelembe vétele. Ez magában foglalhatja az attribútumok/metódusok nevére történő reguláris kifejezés illesztéstől a hozzájuk tartozó kommentek átfésüléséig mindent. Önmagában talán nem a legjobb mintakeresési eljárás, de egy korábban kiválogatott találatlista szűkítésében, vagy éppen küszöbérték alatti fals negatívok jelzésében hatékony segítséget nyújthat.

2.1.3. A rendszer reprezentálása szerinti csoportosítás

Gyakran használt lehetőségek a vizsgált rendszer ábrázolására:

- **AST**: Absztrakt szintaxis fa (**A**bstract **S**yntax **T**ree), általában nyelvfüggő forma, ami az adott programozási nyelv elemeit hierarchikus szerkezetben tárolja. Csomópontként szerepelhet benne például egy osztály, aminek gyerekei az

attribútumai és metódusai – amelyeknek esetleg további gyerekei a hozzájuk tartozó típusinformációk, módosítók, stb.

- **Mátrix**: Ha a rendszert csak jól definiált tulajdonságok egy halmaza szerint elemezzük, akkor tárhely és feldolgozhatóság szempontjából is érdemes lehet a mátrixok használata. Például az előforduló asszociációkat ábrázolhatja egy – a gráfok szomszédsági mátrixaihoz nagyon hasonló – totálisan unimoduláris mátrix, ahol 1 jelezné, hogy a sor osztálya asszociált az oszlop osztályával, 0 pedig a kapcsolat hiányát. Ha a külön tulajdonságokhoz különböző prímeket rendelünk, és csak egy mátrixot használunk, – ami a releváns prímek szorzatait tárolja mezőnként – akkor még kevesebb hely szükséges.
- **ASG**: Absztrakt szemantikai gráf (**A**bstract **S**emantic **G**raph), két lehetséges eleme a csúcs, ami egy, a forrásban előforduló entitást jelöl, és az él, ami pedig az ezek közötti kapcsolatot és annak típusát mutatja.
- **XML**: A napjainkban olyan gyakran használt formátum éppen azért népszerű, mert tetszőleges *tag*-ek és szerkezetek definiálhatók benne, így alkalmas lehet többet között analizálandó szoftverek forrásának tárolására is.

2.1.4. A minták reprezentálása szerinti csoportosítás

Az előző pontban leírt módszerek a tervezési minták leírására is vonatkozhatnak. Nyilván logikus választás, ha a rendszert és a mintát azonos módon ábrázoljuk, de ez egyáltalán nem kötelező és nagyban függ az alkalmazott algoritmustól is. Persze a fentieken kívül még számtalan mód lehet az elvárt viselkedés reprezentálására. Például valamilyen szintaxis szerinti formális logikai specifikáció, amit a rendszerből előállított tényekre illesztünk, vagy akár egy vizuális nyelvtanban megadott pattern, amit képfeldolgozási módszerekkel vetünk össze a rendszer – például gráfként – ábrázolt változatával. A lehetőségek, és az azokon belüli variánsok száma gyakorlatilag végtelen.

2.1.5. Az eredmények formátuma szerinti csoportosítás

Az sem lényegtelen, hogy a keresést végző algoritmus mekkora információtartalmú válaszokat ad. Nagyban csökkentheti a komplexitást, ha csak a találatok darabszámát várjuk el. Viszont ha arra is szükség van, hogy melyik osztály melyik mintabeli szerepre illeszkedik, akkor már nem minden esetben annyira könnyű a helyzet.

Ezen felül a válasz tartalmazhatja a rendszerosztályok forrásfájlját – és akár az azon belüli sorszámot –, vagy például közelítő illesztésnél a hasonlóság mértékét is.

2.2. Irodalmi áttekintés

A jelen dolgozathoz legszorosabban kapcsolódó kutatást Radu Marinescu és társai végezték. 2001-es publikációjuk [2] fő célnak az adott típusú hibák keresésének szisztematikusságát, megismételhetőségét, skálázhatóságát és nyelvfüggetlenségét tűzi ki. Először egy egységes dokumentációs formátumot javasol antiminták metrika alapú felismerésére, illetve egy módszertant azok kiértékelésére Ezt konkrétan a GodClass és a DataClass elterjedt antimintákon be is mutatja, illetve rámutat, hogy ez a módszer hasonlóan más mintákra is alkalmazható. Az automatizálással kapcsolatban egy saját TableGen nevű, C++ nyelvű programokat statikusan elemezni képes programot használ, annak eredményeit pedig adatbázisba menti el. A metrikák maguk az Oracle adatbázison keresztül egyszerű lekérdezésekkel és PL/SQL scriptekkel vannak megvalósítva.

Ennek gyakorlatilag szellemi utódjaként tekinthető egyik 2004-es publikációjuk [7], amiben már a szerzők is inkább az automatizálásra fektették a hangsúlyt, és a minták deklarációját tették könnyebbé egy saját „detection strategy"-nek nevezett módszer specifikálásával, aminek keretében különböző, metrikákra vonatkozó filtereket adhatunk meg egyes forráskódbeli entitásokra. Egy filter lehet határ – az egyik fele nyitott – vagy intervallum – ami szimulálható két határfilterrel –, abszolút vagy relatív, illetve konkrét értékek megadása helyett lehet különböző statisztikai módszereket is alkalmazni, ami a teljes rendszerre vonatkozó, például átlag vagy szórás értékek alapján állapítja meg, hogy mely értékek a kiugróak. Az így kinyert, metrikánként értelmezett illeszkedési jelölteket ezután a hagyományos halmazműveletekkel a végső jelölthalmazzá kombinálták, majd a tesztrendszerükön az eredményeket manuálisan is átvizsgálták. A pontosságot strict – hagyományos – és loose „módban" is értelmezték, ahol az utóbbi azokat a találatokat nem vette hamisnak, amik hibásak voltak ugyan, csak valamilyen más ok miatt. Konklúziójuk egy 70%-os empirikus pontosságú eszköz, ami eredményesnek vélhető.

Az előbb bemutatott „detection strategy"-ket pontosítja tovább egy másik 2004-es publikációjuk [8], amiben nem csak a szoftver jelenlegi állapotát, hanem régebbi verzióit is górcső alá veszik. Egy példa szoftver rendszer – Jun 3D grafikus framework – revízióira egyenként futtatják az alap eljárást és így plusz két értéket nyernek ki minden elemre. A perzisztencia azt mutatja, hogy egy osztály vagy csomag élete hány százalékában volt hibás, a

stabilitás pedig azt, hogy élete hány százalékában változott. A módszer azt a gondolatmenetet veszi alapul, hogy ez a két új metrika birtokában informáltabb döntéseket hozhatunk arról, hogy egy adott találat valóban „rosszindulatú-e". Például a – be is mutatott – GodClass esetében egy illeszkedés karbantartási szempontból inkább akkor releváns, ha nem perzisztens, mert akkor a változások következtében létrejött hiba miatt észlelhettük, vagy ha nem stabil, mert akkor a változásai belezavarhattak a rendszer evolúciójába. A Jun-on futtatott tesztjeik alapján az eredeti 24 GodClass jelöltből 7-ről biztosan sikerült így megállapítani, hogy tényleg rosszindulatú, 5-ről pedig azt, hogy igazából nem is rosszindulatú. Így már csak a találatok felét kell kézzel validálni, ami nagy segítség.

Végül 2005-ös cikkükben [9] egy pontosított mintaleírási módszer javasolnak, ahol annyival mennek tovább, hogy az eddig tervezési hibaként kezelt mintákat – pl. Refused Bequest, GodClass, stb. – csak tünetekként kezeli, és ezeket kontextuális információkkal együtt tekinti egy magasabb absztrakciós szintű design hibának. Fő céljuk, hogy – orvosi hasonlattal élve – ne a tüneteket gyógyítsuk, hanem a betegséget, illetve az explicitség, hogy az antiminták javítása ne annyira művészet, mint jól bevett mérnöki gyakorlat legyen.

Egy másik megközelítésben Khomh és társai [10] 2009-es publikációjukban statisztikai módszerekkel próbálják az antimintákat – vagy „code smell"-eket – felismerni. Három fő kutatási kérdésük:

- Összefüggenek-e a „code smell"-ek és a változékonyság?
- Van-e szerepe a „code smell" mértékének?
- Számít-e, hogy melyik „code smell" érintett?

A kérdésekre saját – DECOR – programjukkal és nullhipotézisekkel nyerik ki a választ, és az empirikus bizonyíték erősen alátámasztja, hogy az antiminták valóban rossz hatással vannak a változékonyságra, így a karbantarthatóságra is. Megemlítik, hogy bizonyos körülmények között a metrika alapú felismerés pontosabbnak tűnik, de azzal is érvelnek, hogy az általuk előállított eredmények talán érthetőbbek a fejlesztők számára.

2007-es cikkükben [11] Lozano és társai a szélesebb körű irodalom áttekintésével egységesítésre hívják fel a területen munkálkodó kutatókat. Amellett, hogy egyes antiminták – vagy „bad smell"-ek – önmagukban is károsak lehetnek-e, arra is keresik a választ, hogy egy-egy ilyen minta mikortól számít hátrányosnak a karbantartás szemszögéből. Illetve a fentebbiekhez hasonlóan ők is azt vallják, hogy több kisebb antiminta gyakori közös előfordulása miatt magasabb absztrakciós szinten kellene tervezési hibákat definiálni, hogy a felismerő programok jobban segíthessék a fejlesztőket refaktorálási javaslatokkal.

Szintén másik szemszögből vizsgálták a témát 2004-es cikkükben Mäntylä és társai [12], akik kérdőívekkel a tervezési hibák jelenlétének és megítélt hatásának szubjektív oldalára koncentráltak. A fejlesztők eltérő véleményeire a demográfiai elemzés, illetve a vállalaton belül betöltött szerep sok esetben magyarázatot adott, de az objektív, metrika alapú eredményektől való nagy eltérés őket is meglepte. Ennek oka lehet a gépi és az emberi megítélés közti nagy különbség, de ők maguk is belátták, hogy az automatikus antimintakinyerő módszereik valószínűleg nem voltak annyira fejlettek és pontosak.

2.3. A saját algoritmus

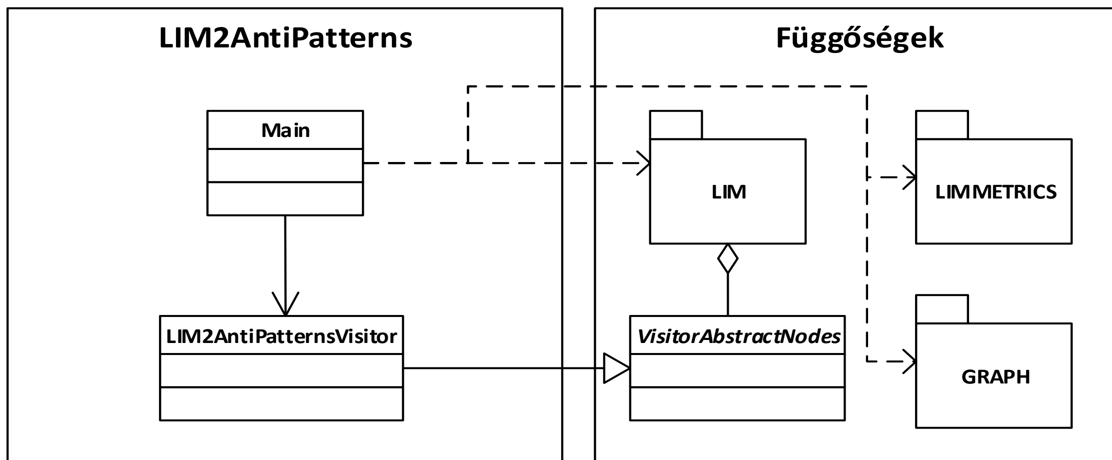
Az általam megvalósított algoritmus az előző csoportosítás tükrében:

- **Bináris**: egyelőre nincs hasonlósági mérték, csak igen-nem válasz
- **Szerkezeti**: csak a LIM által reprezentált adatok álnak rendelkezésre
- **ASG alapú**: a vizsgált rendszer LIM formátumban van, ami egy ASG
- **Kódolt mintaábrázolású**: a keresett minták egyelőre kötötten részei a program forráskódjának, de a paraméterei külső file-ban konfigurálhatók
- **Részletes**: találat esetén nem csak magát a találat tényét jelenti, hanem azt is, hogy melyik minta illeszkedett melyik forráskód elemre, ami melyik file hányadik sorában van, és mi volt az a metrika érték, ami miatt találatnak minősült

Az algoritmus metrikákra vezeti vissza, hogy egy forráskód elem egy antiminta osztályhoz tartozik-e. Működését részletesebben a harmadik fejezetben fejtem ki.

3. A LIM ALAPÚ ALGORITMUS IMPLEMENTÁCIÓJA

3.1. A program szerkezete



3.1. ábra: A program szerkezete

A program statikus szerkezete illetve osztályhierarchiája nagyon egyszerű, a teljes megvalósítás mindössze öt forrásfile-ból áll.

A **main.cpp** dolgozza fel a parancssori paramétereket, szemantikai ellenőrzéseket végez, a konfiguráció alapján engedélyez vagy letilt egyes antimintákat, majd betölti az input *.lim* file-t és az esetleges hozzá tartozó filtert. Ezután az input LIM-et graph formába is átkonvertálja és ezeken futtatja a limmetrics library metrikaszámoló vizitorát, ami a graph node-okhoz rendeli a mintakeresés szempontjából releváns metrika értékeket.

Ha minden előfeltétel adott, akkor – C++-ban megszokott módon – a **LIM2AntiPatterns.h**-ban deklarált és a **LIM2AntiPatterns.cpp**-ben definiált mintakereső vizitor fut le. Az eredményeket – a metrikákhoz hasonlóan – a graph node-jaihoz is rendelheti, de önálló, *.csv* formátumú output-ra is képes. A megvalósított antiminták felsorolása *enum* formában a **LIM2AntiPatterns_def.h** headerben találhatók, az esetleges hibaüzenetek pedig a **messages.h**-ban.

3.2. Használat

A programot a következő formában kell futtatni:

```
LIM2AntiPatterns [options] input file(s)
```

Ahol az *options* rész a következő kapcsolókat tartalmazhatja:

- **`-list:`**`fileName` – paramétereként megadható egy olyan szöveges file, ami sortörésekkel elválasztva felsorolja az input file-ok elérési útjait.
- **`-graph:`**`fileName` – ezzel a kapcsolóval adhatjuk meg, hogy a *.graph* output-ot milyen elérési út alá mentse a program.
- **`-warnings:`**`fileName` – a *.graph* kapcsolódó node-jai mellett a program a talált antiminta példányokat a paraméterként megadott file-ba is mentse ki, *.csv* formátumban.
- **`-metrics:`**`fileName` – lehetőséget ad, hogy az antiminta keresés miatt kiszámolt metrikákat külön is exportálhassuk.
- **`-rul:`**`fileName` – a futás során használt konfigurációs file-t módosíthatjuk vele, az alapértelmezés a program mappájában lévő **AP.rul** nevű file.
- **`-rulConfig:`**`name` – a futtatáshoz releváns rul-on belüli konfigurációt változtathatjuk meg vele, ami alapértelmezésben „Default”.
- **`-exportRul`** – a kimeneti *.graph*-ba beleintegrálja a *.rul* file tartalmát is, így az „standalone” lesz ebből a szempontból, illetve később a *.graph* file-ok szöveges „dump”-olását végző program is az ilyen beégetett strukturális információk alapján tudja, hogy mit és hova kell kiírnia.
- **`-all`** – megadható vele, hogy a program a konfigurációs file engedélyezési direktíváit felülírva, minden antimintát vizsgáljon.

Ezeken felül használható még a `-help` vagy a `-?` kapcsolók, amikkel a fentiekhez hasonló használati infót kaphatunk, illetve a `-ml:level` kapcsoló – úgy mint **M**essage **L**evel –, amivel `level` szintűre állíthatjuk a program „bőszavúságát”.

3.3. Függőségek

3.2.1. LIM

A LIM – vagyis **L**anguage **I**ndependent **M**odel – egy, a Szegedi Tudományegyetemen kifejlesztett nyelvfüggetlen rendszermodell. [13] Segítségével a forrásból kinyerhető információkat a kód lexikális elemzésénél absztraktabban, gráfszerű szemantikával járhatjuk be. Gyakorlatilag egy általánosított ASG, ami csomópontokkal reprezentálja az entitásokat – osztály, interfész, metódus, attribútum, stb. – és élekkel pedig az ezek közti kapcsolatokat.

Az általam készített antiminta felismerő fordításához szükségesek a LIM-hez tartozó fejlécek és könyvtárak, mert a program egy *.lim* kiterjesztésű fájlban tárolt gráfként kapja az input rendszert, amin az összes engedélyezett antimintából példányokat keres.

Itt jegyezném még meg, hogy a *.lim* file-ok mellé egy *.flim* kiterjesztésű filter file is tartozhat, ami a teljes gráf bizonyos csúcsairól tartalmaz annyi plusz információt, hogy az a node az adott elemzés során „érdekelt-e" minket. Ezt az elemző keretrendszer általában arra használja, hogy megkülönböztesse a valósan saját forráskód elemeket a beimportált függőségektől vagy például a java nyelv esetén az implicit nyelvi függésektől – Object, Serializable, stb.

3.3.2. LIMMETRICS

A limmetrics library egy, a LIM-hez készült speciális vizitort tartalmazó függvénykönyvtár, aminek segítségével egy LIM-ben ábrázolt rendszer különböző, statikus elemzéssel kinyerhető metrikáit lehet megállapítani minden típusú forráskód elemre – ez lehet file, komponens, csomag, osztály vagy metódus. A számolt metrikákat 5 csoportba sorolhatjuk:

- **Méret**: pl. LOC (Lines of Code), NM (Number of Methods), stb.
- **Dokumentáció**: pl. AD (Api Documentation), CD (Comment Density), stb.
- **Komplexitás**: pl. NL (Nesting Level), WMC (Weighted Methods per Class), stb.
- **Csatolás**: pl. CBO (Coupling Between Object classes), NII (Number of Incoming Invocations), RFC (Response set For Class), stb.
- **Öröklődés**: pl. NOP (Number of Parents), DIT (Depth of Inheritance Tree), stb.

3.3.3. GRAPH

A graph függvénykönyvtár egy, még a LIM-nél is sokkal általánosabb és absztraktabb gráf megvalósítást tartalmaz. Csak node-okat és a node-ok közötti éleket tartalmaz, illetve a node-oknak lehetnek különböző attribútumaik, amiknek a típusa is meghatározható.

Ez azért szükséges, mert a LIM modellben az egyes típusú forráskód elemekre nincsenek értelmezve az azokhoz számítható metrikák, így a fent említett limmetrics library az eredményeit nem tudja a *.lim* file-ba visszaírni. Kimenete ezért egy graph reprezentáció, amiről LIM node-ok és graph node-ok megfeleltetésével és néhány segédfüggvénnyel visszaolvashatók a metrika értékek.

3.4. Konfiguráció

Habár az egyes antimintákhoz tartozó felismerő algoritmusok egyelőre „bele vannak égetve" a program forráskódjába, az hozzájuk tartozó metrikákra vonatkozó határértékek, illetve a találat esetén a *.graph*-ban elhelyezett figyelmeztető szövegek és leírások egy küldő konfigurációs file-ban állíthatók.

Egy ilyen *.rul* file egy speciális szemantikájú, `Rul` gyökérelemmel rendelkező xml, ami a következő szekciókat tartalmazhatja:

- **ToolDescription**: Ahhoz a programhoz tartozó metainformációk gyűjteménye, amit a file konfigurál
- **Configurations**: Ebben a részben adhatók meg a különböző, nevekkel ellátott konfigurációk, amikhez később a metrikákat kötjük. Minden konfigurációhoz opcionálisan megadható, hogy melyik másik `config` bejegyzést definiálja felül, így ha változtatások szükségesek, akkor elég csak az érintett metrikáknál módosítanunk.
- **Metrics**: `Metric` tag-ekkel egy adott id-jű metrikánál – konfigurációnként – meghatározható, hogy:
  - Engedélyezett-e.
  - Milyen beállítások tartoznak hozzá – ez teljesen egyéni igényekre szabottan értelmezhető, jelen esetben az illeszkedési határértékek tartoznak ide.
  - Nyelvenként jelenlen-e meg magyarázó szöveg, vagy találat esetén figyelmeztető szöveg, és ha igen, mi.

3.5. Részletes bemutatás

Amikor a main megkapja a vezérlést, először is értelmezi és ellenőrzi a parancssori paramétereket. Például ha nem találat a pozícionális paraméterek végén valid input *.lim*-eket és nincs megadva a `-list` paraméter sem – vagy szerepel, de az általa jelölt file-ban nincsenek felsorolva file-ok –, akkor alapból hibát jelez.

Ezután header információkkal együtt – itt főleg a metódusokhoz tartozó override relációk a lényegesek – betölti a soron következő *.lim* file-t, inicializál hozzá egy üres filter-t – ami szerint minden node „számít" – majd ha a *.lim* file mellett szerepel egy azonos utolsó módosítási idejű *.flim* kiterjesztésű file, akkor abból frissíti a filtert. A betöltött modell már azt is tudja magáról, hogy milyen nyelvű forráskódból készült, így a RUL konfigurációnak itt

adunk kezdőértéket. Ha szerepel ezt az opciót specifikáló paraméter, akkor természetesen az fog számítani, de ha nem, akkor itt már egy nyelvtől függő alapértelmezést is választhatunk.

Ekkor következhet a *lim* modell *graph* formátumúvá alakítása. Itt még nem végzünk semmilyen szemantikai módosítást, csak egy absztraktabb reprezentációra váltunk, mert ekkor a létrejövő node-okhoz már – a lim sémafüggőségével szemben – dinamikusan csatolhatunk további attribútumokat.

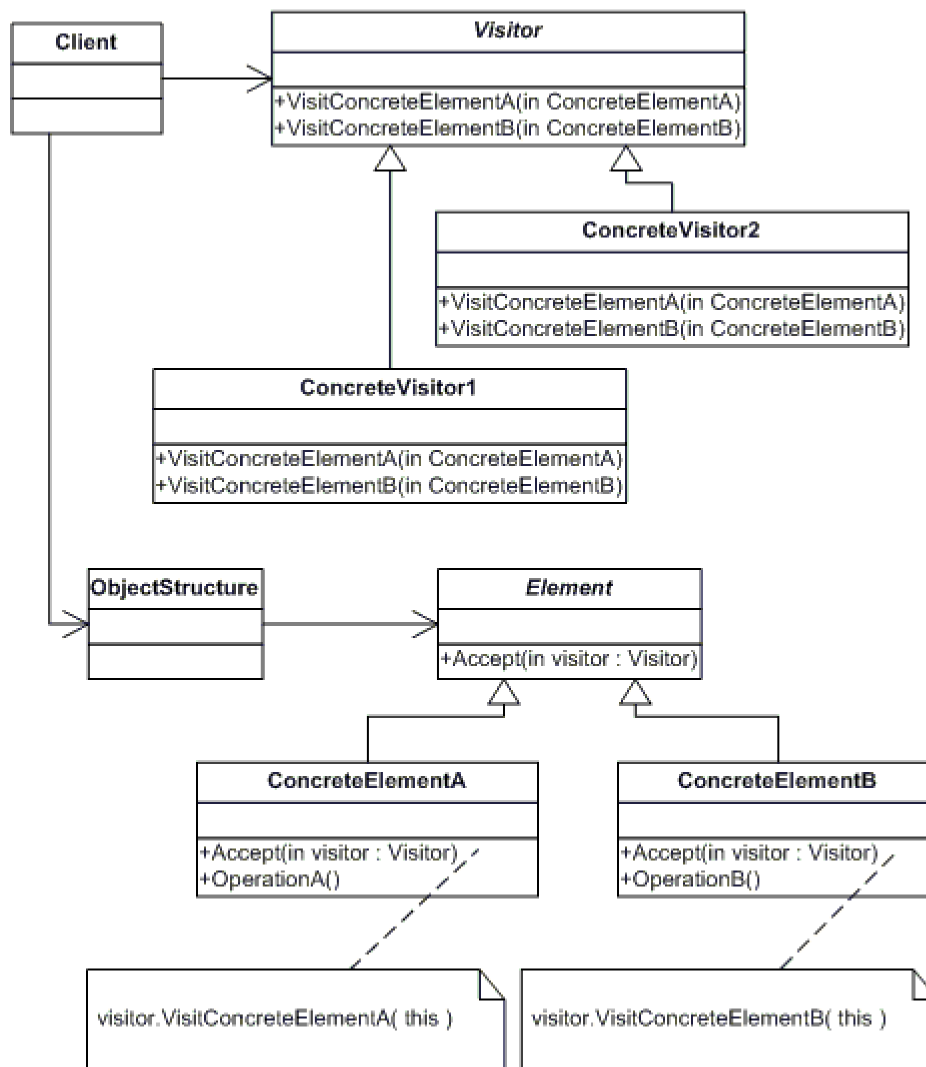
Szintén be kell még tölteni a konfigurációs információkat a `-rul` kapcsolóval megadott file-ból, vagy alapértelmezésben az aktuális könyvtárban található AP.rul-ból. Az xml formátumú file-ból ekkor egy `RulHandler` típusú objektum keletkezik, ami segít majd a későbbi értéklekérdezések során, és a különböző konfigurációk közti öröklődést is elrejti.

Ha már a *lim*, a *graph*, és a *rul* is készen áll, előkészíthetjük a limmetrics library által biztosított metrikaszámoló vizitort (A vizitorokról általában lásd később, a saját megvalósítás leírásánál). Ehhez egy string tömbben összegyűjtöm a kiszámítandó metrikák neveit. Ez kívülről nem paraméterezhető, hiszen csak az antimintáktól – azokon belül is csak az engedélyezett antimintáktól – függ. Ezek jelen esetben az NM, az NA, a CBO, a WMC és a NII metrikák, minden egyéb szükséges információ kinyerhető közvetlenül a *lim*-ből. Egy másik tömbbe ugyancsak össze kell állítani, hogy milyen forráskód elemeket vegyen egyáltalán figyelembe a vizitor, ami a fentiek tükrében szintén „beégethető” csak osztály és metódus szintre, mert – egyelőre – nem használok sem csomag, sem komponens, sem pedig file szintű értékeket. Egy így felparaméterezett vizitorral, valamint egy preorder algoritmust megvalósító osztály segítségével bejárhatjuk az input *lim*-et a gyökér node-jától kezdve.

Ez az a pont, ahol már minden szükséges előkészületet megtettünk, és kezdődhet az egész célja, az antiminták felismerése. Ehhez azonban először kitérnék egy kicsit az implementáció szerkezetére. Attól, hogy itt antimintákkal foglalkozunk, ne feledkezzünk meg az „ellentéteikről”, a tervezési mintákról sem.

A **Visitor** minta fő célja, hogy elválassza a műveletek az objektum struktúráktól, amikre azok hatással vannak. Ezt úgy éri el, hogy az objektum struktúra minden tagja egyszer deklarál egy `accept` műveletet, ami – legtöbbször absztrakt – `Visitor` típusú paramétert fogad, és meghívja annak megfelelő `visit` metódusát saját magát átadva paraméterként. Így leszármazással és felüldefiniálással tetszőleges számú új műveletet készíthetünk az elemek módosítása nélkül, valamint az objektum és művelet szétválasztásán felül még a struktúra felsorolásáért felelős algoritmus is külön szervezhető – lásd a fenti preorder bejárást.

A minta UML diagramja a 3.2-es ábrán látható.



3.2-es ábra: A Visitor tervezési minta szerkezete

Ezen az elven működik a korábban bemutatott *lim*-ből *graph*-fá alakítás és a metrikák kiszámítása is, természetes tehát, hogy én is így valósítottam meg a *lim* modellt bejáró antiminta felismerő programot.

A munka érdemi részét végző LIM2AntiPatternsVisitor a VisitorAbstractNodes-ból származik, ami annyiban egészíti ki a „sima” Visitor-t, hogy az adott séma hierarchiájában nem csak a példányosítható node-okat, hanem az absztraktakat is bejárja. Ez a jelenleg megvalósított antimintákkal nincs ugyan kihasználva, de későbbi – tervezett – bővítések esetén lehetőség nyílik majd általánosan `Member`-ek vagy `Scope`-ok vizsgálatára anélkül, hogy tudnánk, azok valójában osztályok, metódusok vagy attribútumok.

Az osztály létrehozásakor a main-től a forrás *lim*-et, a hibajelzések céljául szolgáló *graph*-ot, a kiszámítandó antiminták neveit és a konfigurációs file-t jelképező `RulHandler` objektumot kapja meg.

Amikor a main a *lim*-et – objektum struktúrát – járja be – például a látott AlgorithmPreorder segítségével – minden esetben a vizitorban felüldefiniált Method és Class node-okra vonatkozó `visit` és `visitEnd` metódusok hívódnak meg, amikor az algoritmus elér vagy elhagy egy megfelelő node-ot.

Osztályoknál a `visit` csak előkészítő műveleteket végez:

- Összegyűjti a node leszármazottjait
- A láthatósági információk alapján külön válogatja az osztály `protected` gyerekeit – azokat a `protected` módosítójú Member-eket, amikre az osztály `hasMember` éllel mutat
- Szétválogatja a gyerek Member-eket attribútumokra és metódusokra

Az osztályokhoz tartozó antiminták kiszámítása a `visitEnd` metódusban, a node elhagyásakor történik. Method node-ok esetén nincs szükség inicializálásra, ezért a hozzá tartozó minták már a `visit`-ben számíthatók, a `visitEnd` – egyelőre – üres.

Az egyes minták pontos értelmezéseiről a 3.6-os alfejezetben írok, itt csak összefoglalnám, hogy általánosságban mi is zajlik le egy ilyen metódusban. Először is a RUL alapján épített *boolean* tömb segítségével meghatározom, hogy az adott antiminta egyáltalán engedélyezett-e. Ha nem, a metódus azonnal vissza is térhet. Egyébként megkeresem a paraméterként kapott *lim* node-hoz tartozó *graph* node-ot egy segédfüggvénnyel és arról – általában – leolvasok egy vagy több metrika értéket.

Ezután ez(eke)t az érték(ek)et összevetem a RUL-ból beolvasott határokkal és egy megfelelő logikai kondíció teljesülése esetén átadom a node-okat és a „határszegő” értékeket az `addWarning` metódusnak, ami – szintén a RUL-ból beolvasott szöveges template-ek alapján – egy figyelmeztetést helyez el a *graph*-on. Egy ilyen warning tartalmazza a minta azonosítója és leírása mellett az illeszkedő elem forráskódbeli pozícióját is.

Van persze, amikor a metrikák kinyerése és a figyelmeztetések elhelyezése közé beékelődik még néhány, mintától függő lépés – például az élekkel reprezentált kapcsolatok szorosabb vizsgálata, vagy egy hányados kiszámítása –, de ezek a végrehajtás általános menetén nem változtatnak.

3.6. Felismert antiminták

Az 1.3-as alfejezetben bemutatott antimintákat a fentiek tükrében a következőképp értelmeztem:

- **Feature Envy**: A más osztályoktól irigyelt funkcionalitást úgy szimuláltam, hogy minden olyan metódus számít ide, ami legalább a RUL-ban meghatározott MinAccess darab különböző attribútumhoz hozzáfér, és ebből legalább MinPercent egy másik osztályhoz tartozik, mint maga a metódus.
- **Lazy Class**: Egy osztályt akkor tekintek lustának, ha legalább a RUL-ban meghatározott `CBO_Min`-nyi osztállyal kapcsolatban áll, de a saját komplexitása legfeljebb `WMC_Max`, mert ez azt jelentheti, hogy maga az osztály nem sok mindent csinál, csak továbbítja a feladatokat mások felé.
- **Large Class Code**: Egy osztályt túl nagynak tekintek, ha a logikai kódsorainak száma (LLOC) meghaladja a RUL-ban meghatározott `Min` értéket.
- **Large Class Data**: Egy osztály adattartalmát pedig akkor tekintem túl nagynak, ha az attribútumainak száma lépi át az ehhez a RUL-ban kijelölt `Min` értéket.
- **Long Function**: Egy függvény – vagy jelen esetben metódus – többféleképpen is lehet hosszú:
  - vagy a hosszú osztályhoz hasonlóan a logikai sorainak száma lép át egy határt (`LLOC_Min`)
  - vagy a benne lévő utasítások száma lép át egy másik határt (`NOS_Min`) – így kikerülhető, hogy az eltérő formázási szokások miatt az esetlegesen egy sorba kerülő több utasítással „elfedhető” legyen a nagy méret
  - vagy a metódus nem a szó szoros értelmében nagy, viszont a tartalma annyira „bonyolult”, hogy a ciklomatikus komplexitása átlép egy harmadik határt (`McCC_Min`)
- **Long Parameter List**: Egy metódusnak akkor van túl hosszú paraméterlistája, ha a formális paramétereinek száma átlépi a RUL-ban megadott `Min` értéket.
- **Refused Bequest**: Egy osztály akkor utasítja vissza az őseitől örökölt adatokat, ha van olyan „protected” láthatósági módosítóval megjelölt attribútuma, amihez egyik saját metódusa sem fér hozzá. Itt nincs szükség a *.rul* file-ban konfigurálható határra, mert csak az számít visszautasításnak, ha egyetlen hivatkozás sincs az adott attribútumra.

- **Shotgun Surgery**: Egy metódus módosítása valószínűleg akkor fog minden esetben sok másik, kisebb módosítással járni, ha túl sokan használják. Ez az én értelmezésemben akkor valósul meg, ha a metódust meghívó más kódelemek száma, vagy másként fogalmazva a metódus NII – Number of Incoming Invocations – metrikája a RUL-ban meghatározott `Min` értéket meghaladja.
- **Temporary Field**: Egy attribútumot ideiglenesnek tekintek, ha a tartalmazó osztály metódusainak legfeljebb a RUL-ban meghatározott `Max` százaléka használja azt.

4. A HASZNÁLT ANTIMINTÁK TESZTELÉSE

4.1. Kisméretű, célzott tesztek

Ellenőrzés gyanánt először néhány rövid, tömör tesztesetet készítettem, hogy biztosan tudhassam, a később elemzett nagy rendszerek esetén kinyert eredmények valóban csak olyan illeszkedésekből származnak, amiket szándékomban is állt illeszkedésnek tekinteni az antiminta értelmezése alapján. Ezek a tesztek antimintánként 1-2 file-ból álló java nyelvű forráskódot jelentenek, amikből egy python-ban megírt automatizáló szkript előbb a Columbus [13] CodeAnalyzer-ével *.lim* formátumú modellt készít, majd a LIM2AntiPatterns előállítja a warning-okat is tartalmazó *.graph*-ot és egy, csak a felismert minták nevét és helyét tartalmazó *.csv*-t. Az output-okból első futtatás esetén referenciákat is készít, minden későbbi futtatásnál pedig össze is veti az akkori output-ot a referenciával. Kezdetben tehát minden eredményt manuálisan validálnom kellett, viszont a további módosítások után elég volt azokkal a tesztekkel foglalkoznom, amiknél a python szkript eltérést jelzett.

A teszt források bizonyos esetekben csak egy osztály vagy metódus egy-egy metrikáját állítják be mesterségesen úgy, hogy azok pont a *.rul* file-ban beállított határérték alatt vagy felett legyenek. Ilyenek például:

- **LPL**: NUMPAR metrika, a határ a teszteknél **7** volt.
- **LCC**: LLOC metrika, a határ a teszteknél **500** volt.
- **LCD**: NA metrika, a határ a teszteknél **30** volt.

Néhol azért ezeknél komplexebb tesztekre volt szükség. Például a 4.1-es listázásban látható a FeatureEnvy minta tesztje, ami ezt ellenőrzi, hogy tényleg csak akkor lesz-e egyezés, ha egy metódus legalább **5** különböző attribútumhoz hozzáfér és azoknak legalább **80**%-a egy másik osztályhoz tartozik. Az `Envious` osztály `envy` metódusa pont ennek állít példát. A `NotEnvious` osztály ezzel szemben azt validálja, hogy sem határszám alatti százalékos érték – `underLimit` –, sem túl kevés attribútum hozzáférés – `notEnoughAttrAccess`, akár határszám feletti százalékkal – esetén ne legyen hibajelzés. Az elemzés eredménye a 4.2-es listázásban szerepel.

Ehhez hasonlóan kellett tesztelni a `LazyClass` mintát, ami csak két feltétel egyidejű teljesülése esetén érvényes, vagy a `LongFunction` mintát is, ami a 3. fejezetben kifejtettek alapján 3 különböző ok miatt is illeszkedhet.

```
class Beauty {

	public int a;
	public int b;
	public int c;
	public int d;
}

class Envious {

	private int inner;

	// 4/5 --> 80%
	public void envy() {
		Beauty b = new Beauty();
		int my = b.a;
		my = b.b;
		my = b.c;
		my = b.d;
		my = inner;
	}
}

class NotEnvious {

	private int innerA;
	private int innerB;

	// 3/5 --> 60%, under limit
	public void underLimit() {
		Beauty b = new Beauty();
		int my = b.a;
		my = b.b;
		my = b.c;
		my = innerA;
		my = innerB;
	}

	// 4/4 --> 100%, over limit but not relevant
	public void notEnoughAttrAccess() {
		Beauty b = new Beauty();
		int my = b.a;
		my = b.b;
		my = b.c;
		my = b.d;
	}
}
```

4.1-es listázás: A FeatureEnvy antiminta tesztje

UID	L127
Name	Envious.envy()V
ParentUID	L124
ParentName	Envious
Kind	Method
Path	FE.java
Line	13
Col	2
EndLine	20
EndCol	3
WarningID	FE

4.2-es listázás: A FeatureEnvy tesztjének eredménye

4.2. Tesztelés nagyobb, valós rendszereken

Miután megbizonyosodtam róla, hogy a program tényleg csak azoknál az elemeknél jelez egyezést, ahol terveztem, ideje volt a felismerőt éles környezetben is tesztelni. Így megmutatkozhatnak azok az esetleges, stabilitással vagy futásidővel kapcsolatos aspektusok, amik még javításra szorulnak.

Ehhez kapóra jött a Columbus [13] rendszer benchmark katalógusa, aminek segítségével **228** közepes vagy nagyobb méretű rendszer *.lim* modelljéhez volt hozzáférésem. Az ezek között szereplő programok vagy függvénykönyvtárak mind java nyelvű, nyílt forráskódú projektek, néhol több verzióban is. A futtatást a 4.3-as listázásban látható egyszerű batch szkripttel végeztem.

Elemzési sebesség szempontjából a LIM2AntiPatterns az elvárásoknak megfelelően teljesített, a futásidők tipikusan 1 másodperc és 1 perc közé estek. Szélsőérték az Eclipse fejlesztői környezet vizsgálata, ami majdnem 5 percig tartott ugyan, de itt azt is figyelembe kell venni, hogy ez a projekt közel másfél millió logikai sort tartalmaz (LLOC: 1483701, TNCL: 16934).

A stabilitással kapcsolatban szintén nem lehet panasz, a rendszer egyik input esetében sem ütközött váratlan kivételbe. Emellett a rendelkezésre álló erőforrásokba is bőven belefért és a

szúrópróba szerű manuális validálás – meg persze a 4.1-es alfejezetben taglalt kisebb tesztek és azok eredménye – arra enged következtetni, hogy még az outputja is helyes.

```
for /F %%l in ('dir /b /on lims\*.lim') do (
	call LIM2AntiPatterns.exe
		-ml:3
		-rulConfig:Default
		-exportRul
		-graph:result\%%~nl.graph
		-warnings:result\%%~nl.csv
		..\lims\%%l
	|| (goto :err)
)

echo "DONE"
goto :eof

: err
echo "ERROR"
```

4.3-as listázás: A nagy teszteket futtató batch szkript

Az így kapott eredményekből, illetve a *.graph*-ból kiexportált egyéb metrikákkal és származtatott értékekkel további elemzésre alkalmas táblázatokat készíthetünk. Egy ilyen például a 4.4-es táblázat, ami a 20 legtöbb antiminta találatot tartalmazó rendszert és a hozzájuk tartozó értékeket mutatja be.

Name	ALL	FE	LC	LCC	LCD	LF	LPL	RB	SHS	TF	TLLOC	TNCL	AP/Class	AP/Line
Eclipse	63754	6792	4193	469	2757	3998	1070	7825	5673	30977	1483701	16934	3.765	0.04297
apache-cloudstack-4,1,0	16623	371	539	151	793	2782	515	903	842	9727	62004	799	20.805	0.26810
GWT	16531	398	4782	159	223	1012	131	2052	2042	5732	680116	10877	1.520	0.02431
apache-camel-2,11,0	14189	404	5119	25	119	346	67	2731	1066	4312	47181	596	23.807	0.30074
CXF	11694	683	1101	102	81	815	142	2101	1359	5310	1373	14	835.286	8.51712
xalan-2,6	11115	224	175	82	256	312	36	493	266	9271	156247	1147	9.690	0.07114
jena	11105	379	1540	42	182	489	38	2115	1409	4911	327498	5087	2.183	0.03391
Spring-Framework	10156	112	1660	60	287	355	36	2618	1452	3576	412126	8656	1.173	0.02464
Rapidminer	9966	330	914	43	721	817	157	992	791	5201	321656	5069	1.966	0.03098
ApacheDS_Build_With_Dependencies	9219	391	956	121	53	591	14	474	1217	5402	34176	346	26.645	0.26975
jmol	9203	593	42	71	235	898	302	1457	309	5296	171400	746	12.336	0.05369
Hibernate	9185	255	1716	81	103	437	207	1806	1083	3497	471182	7377	1.245	0.01949
Tomcat	8264	242	496	72	262	729	92	1108	663	4600	173495	1549	5.335	0.04763
Jackrabbit	8090	702	472	58	772	573	85	1641	927	2860	325944	3770	2.146	0.02482
Weka	7643	421	175	97	75	832	107	1344	523	4069	261862	2418	3.161	0.02919
PDE_UI	7523	85	48	28	109	355	12	1324	71	5491	187323	3129	2.404	0.04016
openejb-4,5,2	7081	353	465	38	43	466	48	866	522	4280	321370	5495	1.289	0.02203
Silverpeas-Core-master	6402	212	366	70	47	344	143	530	912	3778	284652	2730	2.345	0.02249
musique-master	6071	297	191	53	91	346	81	368	163	4481	163425	1592	3.813	0.03715
Apache_ofbiz	5465	116	406	110	62	1161	288	352	591	2379	28704	502	10.886	0.19039

4.4-es táblázat: A nagy tesztek eredményeinek részlete

5. HATÁSVIZSGÁLAT

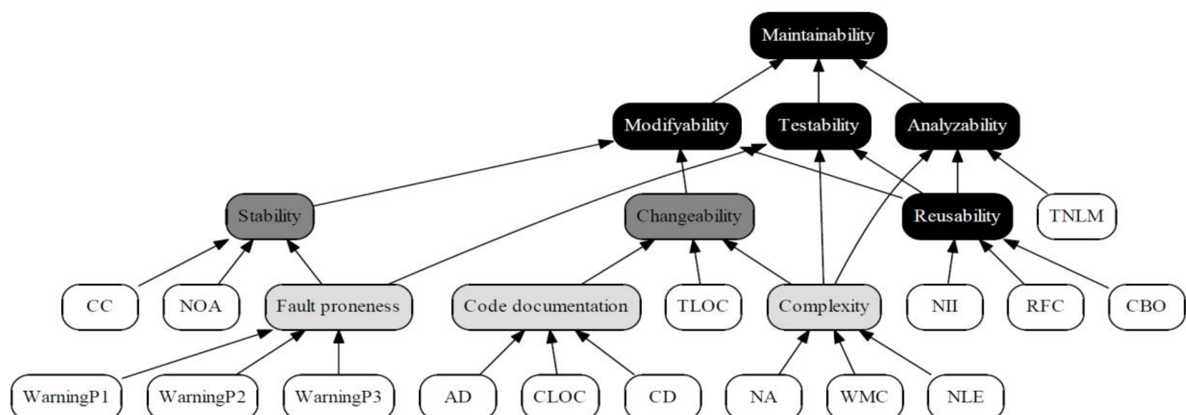
5.1. Minőségi modellek

Ha már megvannak az antimintákkal kapcsolatos információk, megkezdhetjük a bevezetésben felvetett kutatási kérdések megválaszolását. Szükség van még azonban a vizsgált rendszerek minőségi mutatóira, amiket a Szegedi Tudományegyetemen kifejlesztett valószínűség alapú ColumbusQM minőségi modellel nyertem ki [14].

A modell egy szoftver minőséggel kapcsolatos karakterisztikáit az ISO/IEC 25010-es szabványban definiáltak alapján számítja ki [15]. Az értékek meghatározásához egy irányított körmentes gráfot – DAG – használ, aminek a csúcsai különböző belső – alacsony szintű – vagy külső – magasabb absztrakciós szintű – minőségi mutatók lehetnek.

A belső tulajdonságok egy fejlesztő szemszögéből próbálják megragadni a rendszer milyenségét és általában a forráskódból kinyerhető metrikákkal jellemezhetők. Ezzel szemben a külső tulajdonságok – mint ahogy a nevük is mutatja – inkább egy külső, végfelhasználói szempontot jelképeznek. A képen fehérrel jelzett csomópontok úgynevezett *sensor node*-ok, amiket közvetlenül számíthatunk, a többi pedig *aggregate node*. A *sensor node*-ok egy több mint 100 rendszer elemzésével előállított „alapvonal”-tól való eltérések segítségeivel kapnak értéket, az ezek közti függést pedig a gráf élei mutatják. Az élek súlyozása kérdőíves és szakértői információkra épül.

Egy ilyen gráfot Attribute Dependency Graph-nak – Attribútum-függőségi gráf – vagy egyszerűbben csak ADG-nek nevezünk. Az kísérlet során felhasznált rendszerek minőségi vizsgálatához az 5.1-es ábrán látható ADG segítségével készült minőségi modellt használtam.



5.1-es ábra: A felhasznált minőségi modell szerkezete

Az elemzés során először minden belső node értéket kap a metrikák alapján, majd a külső node-ok is kiértékelődnek a függőségek és a súlyozások alapján, így végül egyetlen számot kapunk, a karbantarthatósági metrikát – „Maintainability” az ábrán –, ami a teljes rendszer minőségé jellemzi.

5.2. PROMISE

Ha az antimintákat a bugok számával is össze akarjuk vetni, akkor egy olyan adatbázist is keresnünk kell, ami ezt az információt nyilvántartja bizonyos rendszerekre. Egy ilyen a PROMISE [16] nyílt hozzáférésű bug-tár, aminek célja, hogy a szoftverminőséggel kapcsolatos kutatásokat megkönnyítse és megismételhetővé tegye. Innen összesen **34** java nyelvű rendszer bug-adatait sikerült kinyernem.

A bugok a szoftverek osztályaihoz voltak rendelve, ami az 5.4-es fejezetben leírt tanuláshoz pont megfelelő, de az általános korrelációk kereséséhez egy számra volt szükségem, ezért rendszer szinten összegeztem őket.

5.3. Korreláció keresése

Az adatok összegyűjtése után az 5.2-es táblázathoz hasonló alakú információ áll rendelkezésünkre.

Név	Antiminták	Karbantarthatóság	Bugok
jedit-4,3	2351	0.472396595	12
camel-1,0	685	0.62129318	14
forrest-0,7	53	0.733643936	15
ivy-1,4	709	0.494647716	18
pbeans-2	105	0.489089589	19
synapse-1,0	398	0.582020519	21
ant-1,3	933	0.515664609	33
ant-1,5	2069	0.41340424	35
poi-2,0	2025	0.363086478	39
ant-1,4	1218	0.45270205	47
…	…	…	…

5.2-es táblázat: A kiindulási adathalmaz

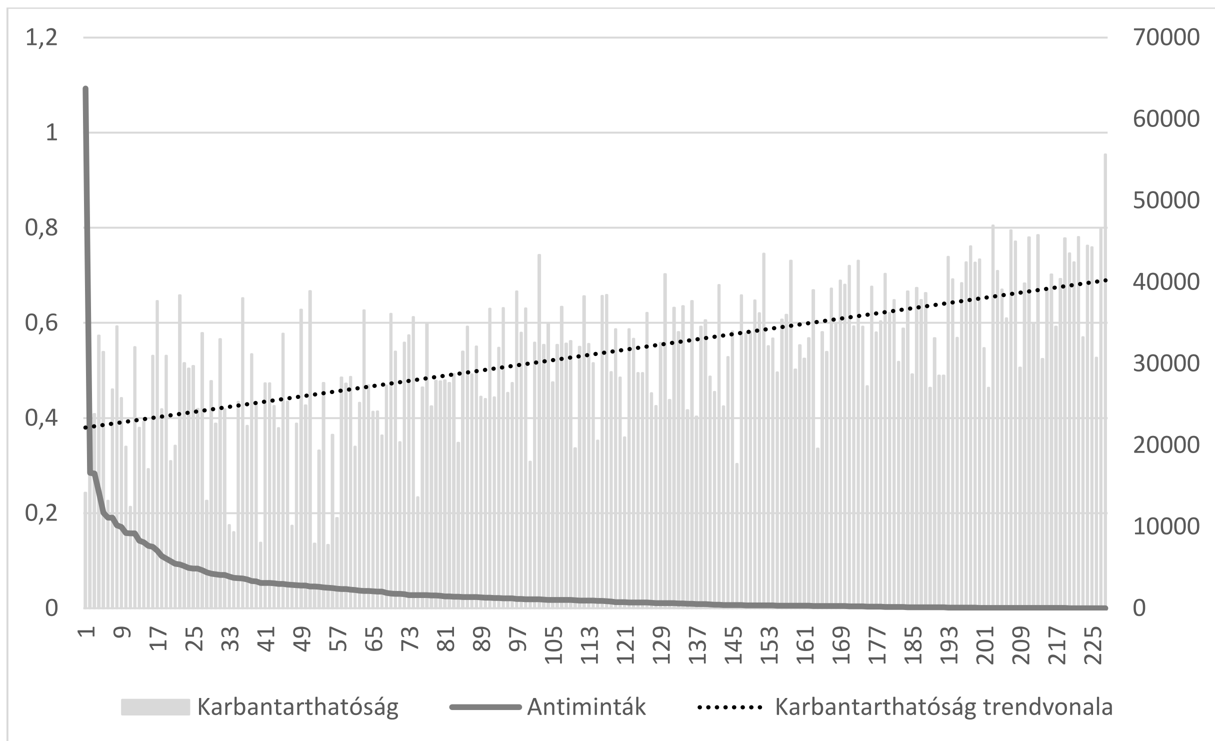
Ezután mind az antimintákat és a bugokat, mint az antimintákat és a karbantarthatóságot korrelációs elemzésnek vethetjük alá. Mivel nem feltételezünk mindenképp lineáris viszonyt az adatsorok között, csak a monotonitást várjuk el, így Spearman korrelációt érdemesebb

használni. A Spearman korreláció gyakorlatilag nem más, mint egy hagyományos Pearson korreláció, csak azt nem az eredeti adatsorokon, hanem az azokból előállított rangsorokon végezzük el. Ez azt mutatja meg, hogy a két adatsor „mennyire mozog együtt”, vagyis például ha az egyik csökken, akkor csökken-e a másik is. A feltételezett csökkenés mértékét elfedi ugyan a rangsorolás – tulajdonképpen információ-vesztésnek tekinthető – de ez ebben az esetben nem is annyira fontos, mert nem a kapcsolat jellegére, hanem a létezésére keresünk bizonyítékot.

Amennyiben csak a buginformációkkal is rendelkező rendszereket vesszük figyelembe, a bugok és az antiminták száma között egy **0.55**-ös Spearman korreláció van 0.001-nél kisebb szignifikancia értékkel. Ez a jelentősnek mondható összefüggés egyrészt igazolja az elvárt viselkedést – miszerint minél több antiminta található egy szoftver forráskódjában, valószínűleg annál több bugot is tartalmaz –, másrészt választ is ad az első kutatási kérdésünkre. Intuitíven ez is lenne az elvárható, de ezzel a kísérlettel objektíven is egy kicsit közelebb kerültünk ahhoz, hogy ezt tényként kezelhessük. A kapcsolat nyilvánvalóan nem mondható egy-az-egyhez megfeleltetésnek, de jól szemlélteti az antiminták forráskódra kifejtett negatív hatását.

Ha a bugokat nem vesszük figyelembe, viszont az összes elemzett rendszerre kiterjesztjük a vizsgálatot, akkor pedig azt tapasztalhatjuk, hogy az antiminták száma és a karbantarthatósági metrika között egy még erősebb, viszont fordított irányú, **-0.62**-es Spearman korreláció lelhető fel – szintén 0.001-nél kisebb p-value-val. Ez alapján megválaszolhatjuk a második kutatási kérdésünket is, miszerint minél több antimintát tartalmaz a kód, annál kisebb lesz a várható karbantarthatósága. Ez az eredmény szintén jól illeszkedik az antiminták és a karbantarthatóság definíciójához, mert ha az antiminták – habár elsőre jó megoldásnak tűnhetnek – hosszú távon tényleg negatív következményekhez vezetnek, akkor a belőlük adódó plusz javítási vagy refaktorálási munkálatok annál nehezebben karbantarthatóvá tették a teljes rendszert.

Az antiminták és a karbantarthatóság viszonyát szemlélteti az 5.3-as ábra is. A trendvonal jól látható javulást jelez előre az antiminták számának csökkenésével.



5.3-as ábra: Az antiminták és a karbantarthatóság viszonya

5.4. Gépi tanulási módszerek

Ahhoz, hogy a harmadik kutatási kérdésünkre is választ kaphassunk, már egy egyszerű korrelációnál komplexebb eszközt kell igénybe vennünk. Azt próbáljuk kideríteni, hogy a felismert antipatternek száma, illetve azok eloszlása rendelkezik-e valamilyen mögöttes szerkezettel, ami segíthet a jelenleg is ismert bugok alapján megjósolni, hogy egy adott elem hibás lesz-e, vagy sem. Ha itt egy elég pontos felismerő modellt sikerülne építeni, az azt jelenthetné, hogy a jövőbeli hibákat – illetve azok helyét – is nagy valószínűséggel előre tudnánk jelezni. Ez az információ pedig felbecsülhetetlen lenne code review-k és bármilyen más karbantartási munkálat során.

Pontosan ilyen és ehhez hasonló problémákkal foglalkozik a gépi tanulás. A fent adott feladat gyakorlatilag egy tankönyvi példa egy osztályozásra, amihez az empirikus tapasztalatok alapján legpontosabban működő algoritmust, a döntési fákat választottam. Egy ilyen fa köztes döntési csomópontokból épül fel, ahol a hátralévő osztályozatlan példányokat két vagy több további alcsoportba osztjuk valamely felhasznált attribútum értéke alapján. A fa levelei tartalmazzák azt az osztályt, amit választani érdemes, ha a gyökértől hozzá vezető úton választott feltételek konjunkciójából álló logikai kifejezés igaz – vagyis az adott

osztályozandó dokumentum esetén a megfelelő döntéseket követve ehhez a levélhez jutottunk.

Konkrét döntési fának a C4.5 algoritmust [17], illetve annak nyílt forrású, java alapú implementációját, a J48-at, a tanulás elvégzéséhez pedig a Weka [18] gépi tanulást segítő szoftvert választottam.

Az input adatokon a még pontosabb szemcsézettség érdekében további átalakításokat végeztem. Mivel jelen esetben csak a minőséggel kapcsolatos információk voltak rendszerszintűek, az antiminta és bug adatokból osztályszintű táblázatokat készítettem. A PROMISE adatbázisból kinyert bugszámok eleve ilyen formában is szerepeltek, az antimintákat pedig úgy csoportosítottam, hogy az osztályokhoz tartozó minták mellé társítottam az osztály metódusaihoz vagy attribútumaihoz tartozó, alacsonyabb szintű mintákat is.

Ez viszont olyan adathalmazt eredményezett, ami igen erősen torzíthatta volna a tanulást, mivel jóval több – nagyjából 80%-nyi – rekord esetében nem volt bug. Ezért a „bugtalan" osztályt alul-mintavételeztem, hogy a benne lévő egyedek száma megegyezzen a bugosakkal, így egyenletes eloszlást garantálva a döntési fának

Az így előállt adatokon három különböző konfigurációban végeztem el a tanulási feladatot:

- először csak a Columbus által kinyerhető metrikákból próbáltam a bugok számára következtetni
- másodszor csak az általam előállított antimintákból
- végül pedig mindkét csoport attribútumaiból

A fákat mindhárom esetben úgy próbáltam bekalibrálni, hogy nagyjából 50 levele legyen, és összesen 100 körüli node-ból álljon. Ez megközelítőleg egy jó kompromisszum a túlegyszerűsítés és a túltanulás között.

Az eredményeket az 5.4-es táblázat foglalja össze:

Method	TP Rate	FP Rate	Precision	Recall	F-Measure
Antipatterns	0.658	0.342	0.67	0.658	0.653
Metrics	0.711	0.289	0.712	0.711	0.711
Both	0.712	0.288	0.712	0.712	0.712

5.4-es táblázat: A gépi tanulás eredményi

Az értékekből jól látható, hogy habár az antiminták – egylőre – elmaradnak, mégis már kilenc, egy-egy entitáshoz rendelt antimintából elég jól megközelíthető a jelenlegi ötven osztályszintű metrika bug-előrejelző képessége. Megjegyezném még, hogy a két kategória együttes eredménye várhatóan nem javít a metrikák modelljén – pedig felhasználja az antiminták értékeit is –, mert az eddig megvalósított antiminták maguk is legtöbb esetben metrikák expliciten „beégetett" határaira építenek.

Ezzel tehát megválaszoltuk harmadik és egyben utolsó kutatási kérdésünket is, miszerint az antiminták már most ígéretes bug-prediktoroknak nevezhetők, és továbbiak – főleg több egyeden átívelő, egyéb strukturális viszonyokat ábrázoló minták – implementálásával akár a metrikákon túlmutató pontosság is elérhető.

6. JÖVŐBELI TERVEK

A kutatás és az annak keretében idáig elért eredmények nagy potenciált mutatnak ezért mindenképpen érdemesek a további fejlesztésre.

Elsődleges, remélhetőleg már a közeljövőben megvalósítható célom az eszköz integrálása a Columbus statikus elemző keretrendszerbe, így az nem csak az akadémiai vizsgálatokat, hanem a Columbus-t használó ipari projektek elemzését és fejlődését is segíthetné. Ez a lépés már nem jelentene nagy kihívást, mivel a rendszer eleve a keretrendszer egyik alapját jelentő nyelvfüggetlen modellre épül, és az ahhoz kapcsolódó, szabványos technológiákkal kommunikál a „külvilággal".

Egy másik lehetőség további, akár forráskódbeli entitásokon átívelő minták deklarálása, a *lim* modell kontextusára való értelmezése és az azokat felismerő metódusok implementálása.

Távlati tervem az absztrakciós szint jelentős növelése, egy saját mintaleíró nyelv definiálása illetve a forrásinformációk kibővítése viselkedési és szemantikai adatokkal.

7. HELYESSÉGET VESZÉLYEZTETŐ TÉNYEZŐK

A kinyert eredmények és a belőlük levonható következmények helyességét több feltételezés is veszélyeztetheti:

- **Nem azokat a mintákat ismerem fel, amiket terveztem**: Ezt a kételyt próbáltam a 4.1-es alfejezetben tárgyalt kisebb, célzott tesztekkel eloszlatni.
- **Nem a jó mintákat terveztem felismerni**: Ezt a buktatót úgy kerültem ki, hogy ismert, elfogadott és szakirodalomban dokumentált antiminták felismerésére használtam a programot. Az egyetlen, erre a kutatásra vonatkozó meghibásodási pont a minták LIM-re történő értelmezésében lehet.
- **Hibás a minőséget jelző modell**: A kapott karbantarthatósági értékeket alapértelmezetten jónak fogadtam el, így ha azok tévesek, akkor nyilvánvalóan az eredményeim is azok. Viszont ennek eléggé kicsi az esélye, mivel a felhasznált valószínűség alapú minőségi modellt korábban százas nagyságrendű tesztrendszer-adatbázison validálták és több hasonló témájú cikknek is alapja.
- **Hibás a bug adatbázis**: Az elemzett rendszerekben, illetve azok osztályaiban előforduló bugok számát szintén nem kérdőjeleztem meg, de ez a veszély is elhanyagolható és a szakirodalomban is sokan támaszkodnak a PROMISE adatbáisra.
- **A kimutatott korrelációk csak a véletlen eredményei**: Ezt a pontot azzal cáfolnám, hogy minden korrelációhoz 0.001-nél kisebb szignifikancia érték (p-value) tartozik.
- **Torzít az előállított tanuló adatbázis**: Valóban jóval több bug nélküli osztály volt, mint buggal rendelkező, de ezt a tanulás előtt az adatok előfeldolgozásánál a megfelelő osztály(ok) alul-mintavételezésével és egyenletes eloszlásúra hozásával orvosoltam.

8. ÖSSZEFOGLALÁS

Az antiminták ismerete, felismerése és kijavítása a szoftverek karbantartásának egyik fontos összetevője. Ezért is lényeges, hogy már a modellezési fázisban se kövessük el ezeket a „hibákat”, illetve hogy minél könnyebben, akár automatikusan képesek legyünk már kész forráskódból visszanyerni az ezekhez kapcsolódó információkat.

Dolgozatom első fejezetében bemutattam az antimintákat és kitértem a detektálásukból származó előnyökre is. A második fejezetben a szakirodalom áttekintésével összegeztem a mintafelismeréssel, és konkrétabban az antiminták felismerésével kapcsolatos módszereket és eredményeket. A harmadik fejezetben részletesen bemutattam egy antimintákat felismerő program implementációját és a hozzá kapcsolódó mintaértelmezéseket. A negyedik fejezetben mind kisebb, mind nagyobb rendszereken – sikeresen – leteszteltem a program működését.

Az ötödik fejezetben azt vizsgáltam, hogy kimutatható összefüggés van-e az antiminták száma, a bugok száma és a karbantarthatóság között. Az eredmények alapján levonhatjuk az informálisan egyértelműnek tűnő, mégis kis mértékben alátámasztott következtetést, hogy az antiminták összefüggenek a bugok számával és fordítottan arányosak a minőséggel.

A hatodik fejezetben szót ejtek a kutatással kapcsolatos jövőbeli terveimről is, végül a hetedik fejezetben felvázolom az eredmények helyességét veszélyeztető tényezőket.

Irodalomjegyzék

1. **Grubb, Penny és Takang, Armstrong A.** *Software Maintenance: Concepts and Practice.* 2nd. hely nélk. : World Scientific, 2003. ISBN: 981-238-426-X.

2. *Detecting Design Flaws via Metrics in Object-Oriented Systems.* **Marinescu, Radu.** hely nélk. : IEEE Computer Society, 2001. In Proceedings of TOOLS. old.: 173-182.

3. **Brown, William J., és mtsai.** *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis.* New York, NY, USA : John Wiley & Sons, Inc., 1998. ISBN: 0-471-19713-0.

4. **Fowler, M. és Beck, K.** *Refactoring: Improving the Design of Existing Code.* hely nélk. : Addison-Wesley, 1999. ISBN: 9780201485677.

5. **Gamma, Erich, és mtsai.** *Design Patterns: Elements of Reusable Object-oriented Software.* Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0-201-63361-2.

6. **Bán, Dénes.** *Tervezési minták felismerése objektum orientált programkódban.* University of Szeged. 2012. BSc Thesis.

7. *Detection strategies: Metrics-based rules for detecting design flaws.* **Marinescu, Radu.** 2004. In Proc. IEEE International Conference on Software Maintenance.

8. *Using history information to improve design flaws detection.* **Rapu, D., és mtsai.** 2004. Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings. Eighth European Conference on. old.: 223-232.

9. *Diagnosing Design Problems in Object Oriented Systems.* **Trifu, Adrian és Marinescu, Radu.** hely nélk. : IEEE Computer Society, 2005. Proceedings of the 12th Working Conference on Reverse Engineering. old.: 155-164. ISBN: 0-7695-2474-5.

10. *An Exploratory Study of the Impact of Code Smells on Software Change-proneness.* **Khomh, F., Di Penta, M. és Guéhéneuc, Y.** 2009. Reverse Engineering, 2009. WCRE '09. 16th Working Conference on. old.: 75-84.

11. *Assessing the Impact of Bad Smells Using Historical Information.* **Lozano, Angela, Wermelinger, Michel és Nuseibeh, Bashar.** hely nélk. : ACM, 2007. Ninth International Workshop on Principles of Software Evolution: In Conjunction with the 6th ESEC/FSE Joint Meeting. old.: 31-34. ISBN: 978-1-59593-722-3.

12. *Bad smells - humans as code critics.* **Mäntylä, M.V., Vanhanen, J. és Lassenius, C.** 2004. Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on. old.: 399-408.

13. *Columbus: A Reverse Engineering Approach.* **Beszédes, Árpád, Ferenc, Rudolf és Gyimóthy, Tibor.** 2005. Pre-Proceedings of IEEE Workshop on Software Technology and Engineering Practice (STEP 2005). old.: 93-96.

14. *A probabilistic software quality model.* **Bakota, T., és mtsai.** 2011. Software Maintenance (ICSM), 2011 27th IEEE International Conference on. old.: 243-252.

15. **ISO/IEC.** *ISO/IEC 25000:2005. Software Engineering -- Software product Quality Requirements and Evaluation (SQuaRE) -- Guide to SQuaRE.* hely nélk. : ISO/IEC, 2005.

16. **Menzies, Tim, és mtsai.** The PROMISE Repository of empirical software engineering data. *The PROMISE Repository of empirical software engineering data.* 2012. June.

17. **Quinlan, J. Ross.** *C4.5: Programs for Machine Learning.* San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1993. ISBN: 1-55860-238-0.

18. *The WEKA Data Mining Software: An Update.* **Hall, Mark, és mtsai.** 1, New York, NY, USA : ACM, 2009. #nov#, SIGKDD Explor. Newsl., 11. kötet.

Köszönetnyilvánítás

Ezúton szeretném megköszönni Ferenc Rudolfnak és Hegedűs Péternek a tanácsokat és a támogatást, valamint Ladányi Gergelynek a teszt inputok előkészítésében nyújtott segítségét és általános elérhetőségét!

A kutatás a TÁMOP-4.2.4.A/2-11/1-2012-0001 azonosító számú Nemzeti Kiválóság Program – Hazai hallgatói, illetve kutatói személyi támogatást biztosító rendszer kidolgozása és működtetése konvergencia program című kiemelt projekt keretében zajlott. A projekt az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósul meg.