

TDK Dolgozat

Antal Gábor

Kódolási szabálysértések refaktoringja és a minőségváltozás vizsgálata

Készítette:

Antal Gábor

programtervező informatikus

I. évf. MSc hallgató

Témavezetők:

Dr. Nagy Csaba

tanársegéd

Dr. Ferenc Rudolf

adjunktus

Szegedi Tudományegyetem Természettudományi és Informatikai Kar

Szoftverfejlesztés Tanszék

Szeged

2014

Absztrakt

Ipari szoftverek fejlesztése során gyakran szűkösek a határidők, és a funkcionális megfelelés fontosabb, mint a jó minőségű, karbantartható kód írása, vagy az alapvető kódolási szabályok betartása. Emiatt gyakran előfordul, hogy a forráskódban olyan kódrészletek maradnak, melyek az adott esetben jól működnek ugyan, de egy-egy módosítás során könnyen hibára vezethetik a fejlesztőt. Az ilyen kódrészletek működése nem feltétlen rossz, de mindenképpen ellenőrzésre, szükség esetén pedig javításra, azaz „refaktoringra” szorulnak. A refaktoring fogalma Martin Fowler könyvének megjelenésével vált elterjedté, amiben a fogalmat a következőképpen definiálja: „A szoftver belső szerkezetén végzett változtatás, amelynek célja, hogy a szoftverkód jobban áttekinthető és egyszerűbben módosítható legyen a megfigyelhető viselkedés megváltoztatása nélkül.”

A dolgozatban több nagyvállalati rendszer refaktoring időszakát vizsgáljuk. Ezen időszak alatt a rendszerekben számos refaktoringot hajtottak végre, a kódminőség javításának reményében. A fejlesztők célja egyértelműen a minőség javítása volt, de vajon tényleg sikerült-e nekik? Objektívan is mérhető-e egy-egy kódolási szabálysértés javításának minőségre gyakorolt hatása? Egyáltalán gyakorol-e bármilyen hatást a kódminőségre? Elképzelhető esetleg, hogy javítás helyett pont rontunk a minőségen? És vajon, ha ezeket a refaktoringokat tömegesen hajtjuk végre egy-egy rendszerben, akkor látható-e javulás?

A dolgozatom ezekre, és hasonló kérdésekre próbál meg választ adni, a valószínűségi minőségmodell, és az ezen alapuló forráskód minőségmérő eszközök segítségével, valós környezetben, ipari rendszerek adatainak felhasználásával, és kiderül, hogy a refaktoring mennyire számít „csodafegyvernek” a forráskód-minőség javítására.

Tartalomjegyzék

Absztrakt	3
Tartalomjegyzék	5
Bevezetés	6
1. Szakirodalom bemutatása	8
2. Kódminőség és refaktoring	10
2.1 Kódminőség	10
2.1.1 A kódminőségről általában	10
2.1.2 Elemzési módszerek	11
2.1.3 Probabilisztikus minőségmodell	11
2.2 Refaktoring	12
2.2.1 A refaktoringról általában	12
2.2.2 A refaktoring művelete	14
2.2.3 Automatikus refaktoring eszközök	15
2.2.4 Refaktoring végrehajtása egy példán szemléltetve	15
3. Refaktoring műveletek vizsgálata, motivációs háttér	18
3.1 A refaktoring időszak vizsgálata	18
3.2 Ticket rendszer	19
3.3 Verziókövető rendszer	19
3.4 Forráskódminősítő rendszer	20
3.5 Nehézségek	20
3.6 A rendszerek vizsgálata, és az elvégzett refaktoringok száma	21

3.6.1	„A” rendszer	21
3.6.2	„B” rendszer	22
4.	Szabálysértések, metrikák	24
4.1	Szabálysértések	24
4.1.1	„Bad smell”	24
4.1.2	PMD	25
5.	Eredmények, következtetés	29
5.1	Az eredmények bemutatásánál használt fogalmak, metrikák leírása	29
5.2	„A” rendszer	30
5.2.1	Becsült és tényleges ráfordítás a refaktoringok elvégzésére	30
5.2.2	Egyes szabálysértés típusok refaktoringjának becsült és tényleges hatása a minőségre	32
5.2.3	Az elvégzett refaktoringok teljes rendszer minőségére gyakorolt hatása	34
5.2.4	Magas szintű minőségmutatók változása a refaktoring során	36
5.2.5	Alacsony szintű minőségmutatók változása a refaktoring során	37
5.3	„B” rendszer	37
5.3.1	Becsült és tényleges ráfordítás a refaktoringok elvégzésére	38
5.3.2	Egyes szabálysértés típusok refaktoringjának becsült és tényleges hatása a minőségre	39
5.3.3	Az elvégzett refaktoringok teljes rendszer minőségére gyakorolt hatása	39
5.3.4	Magas szintű minőségmutatók változása a refaktoring során	42
5.3.5	Alacsony szintű minőségmutatók változása a refaktoring során	43
6.	Konklúzió és jövőbeli tervek	44
A.	Függelék	46
A.1	A „B” rendszerben lévő szabálysértések hatása a minőségre	46

Bevezetés

A szoftverek minőségének meghatározására számos módszer létezik, mégis a forráskód az, ami a legtöbb információval tud szolgálni. A fejlesztés során azonban előfordul, hogy a programunk működésének helyességét fontosabbnak tartjuk, mint forráskódunk minőségét. Ebből kifolyólag érdemes egy-egy „húzóssabb” fejlesztési időszak végén időt szánni a kódunk alapos áttekintésére, és – amennyiben szükséges ítéljük – a minőségének javítására. Ilyenkor a célunk, hogy a működés megtartása mellett javítsunk a forráskódon, úgy, hogy az könnyebben karbantartható legyen. Ez a refactoring művelete. Ezen művelet célja, hogy a forráskód minősége javuljon, de a belső struktúra letisztultabbá, érthetőbbé váljon, és lecsökkentsük a hiba megjelenésének valószínűségét. Mindeközben ügyelve arra, hogy a funkcionalitás megmaradjon, a működés ne változzon.

Dolgozatom célja megvizsgálni a kódolási szabálysértések javításával jelentkező kódminőségbeli változásokat, a probabilisztikus minőségmodell segítségével. A vizsgálathoz statikus elemző eszközök eredményét, valamint a rendelkezésünkre álló ticketing rendszer, és verziókövető rendszer információit használok, továbbá a valószínűségi minőségmodellen alapuló forráskód minősítő eszközt (QualityGate - SourceAudit modul).

Ezen adatok feldolgozásával, és összehangolásával többféle szabálysértést vizsgálok meg két ipari, Java nyelven íródott rendszeren, valamint azt, hogy a fejlesztők egy-egy szabálysértésre milyen becsléseket adtak, mennyi idő lesz a javítás, és a javítás hogyan befolyásolja a kódminőséget. Módosítják-e egyáltalán? Ha módosítják, pozitív, vagy negatív irányba? Érezhető egyáltalán ez a minőségváltozás? A dolgozat további része ezekre, és hasonló kérdésekre keres válaszokat, és kiderül az is, hogy a probabilisztikus minőségmodell segítségével kimutatható-e egyáltalán ez a változás.

Az eredmények tanulmányozásához több cég manuális refactoring időszakában előálló adatok álltak rendelkezésemre, amiket a dolgozatban vizsgálok. A dolgozat elkészítéséhez a rendkívül

nagy méretű és gyakran rendezetlen adathalmaz rendszerezése mellett a probabilisztikus minőségmodell alkalmazása és az elemzések végigvitele is mind a saját munkám eredményeiként kerülnek bemutatásra. Az itt bemutatott eredmények publikálva lettek a „*14th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2014)*” konferencia kiadványában „*Bulk Fixing Coding Issues and Its Effects on Software Quality: Is It Worth Refactoring?*” című cikkben [22] is.

1. fejezet

Szakirodalom bemutatása

A refaktoring fogalmát Opdyke vezette be, aki a doktori tézisében [18] használta először ezt a fogalmat. Később Martin Fowler könyvében [16] lett pontosabban definiálva a napjainkban is elterjedt refaktoring fogalmat. Fowler a könyvben „bad-smell”-eknek nevezi azokat a kódrészleteket, amelyek adott esetben jól funkcionálnak, de a jövőbeli fejlesztések folyamán komoly meghibásodásokat okozhatnak. Ezeknek a javítását célozza meg a refaktoring segítségével. A bad smell fogalma szintén Fowler könyvével terjedt el a köztudatban. Az ilyen, problémás kódrészleteket egy később megjelent könyv kapcsán gyakran „AntiPattern”-eknek is nevezik [14].

A refaktoringgal számos kutató foglalkozik, mint a szoftver karbantarthatóságának javítására szolgáló eljárás. Mára valóban kiderült, hogy a megfelelően, körültekintően alkalmazott refaktoringok képesek a forráskód karbantarthatóságán javítani, amely segítségével a kód könnyebben értelmezhető, és egyszerűbben módosíthatóvá válik, továbbá segítséget nyújthat különböző programhibák megtalálásában. Mindezekén túl a késztermék gyorsabban is futhat [16, 15]. Az eddig megjelent kutatási eredményeket egy összefoglaló tanulmányban mutatja be Tom Mens szerzőtársaival [17].

Számos publikáció foglalkozik a refaktoringgal, mint a szoftver karbantarthatóságának javításával, de alig néhány publikáció elemzi a refaktoring hatását a forráskód kódminőségére.

Sahraoui és a szerzőtársai által publikált cikkben [19] egy minőség becslési modellt alkalmaznak, amely segítségével párhuzamot próbálnak vonni az objektum-orientáltságra vonatkozó metrikák és a refaktoring alkalmazásával az ezeken bekövetkező változások között. Azonban eredményeiket nem támasztották alá ipari rendszereken végzett esettanulmányok.

Stroulia és Kapoor [21] cikkükben a méret és csatolást reprezentáló metrikák viselkedését tanulmányozzák refactoring alkalmazása előtt, valamint azután.

Demeyer [15] publikációja bemutatja, hogy a refactoring nem csak a forráskódra, s annak minőségére van hatással. Publikációjában a szoftverek teljesítményét vizsgálja, melyből kiderül, hogy a refactoring technika alkalmazása pozitívan befolyásolja a performanciát. Ennek egyik oka, hogy a fordítóprogramok a polimorf hívásokat képesek jobban optimalizálni, mint az egyszerű, statikus szerkezetekben lévő metódushívásokat.

Stroggylos [20] társaival népszerű, nyílt forrású (open source) rendszereket vizsgált meg, oly módon, hogy a verziókövető rendszerben lévő, refactoringnak megjelölt verziójú forráskódok logjait vizsgálta meg, és azt, hogy ezek hogyan befolyásolják a különböző metrikákat.

Természetesen az itt említett publikációkon kívül még számos kutatás foglalkozik a refactoringgal, és rengeteg kutató vizsgálja ezen műveletek forráskód minőségre gyakorolt hatását. A fejezetben a dolgozathoz szorosan kapcsolódó munkákra koncentráltunk.

2. fejezet

Kódminőség és refaktoring

A refaktoring műveletek kódminőségre gyakorolt hatásainak vizsgálatához elengedhetetlen az alapfogalmakat ismertetni, valamint az elméleti háttérrel is. Ahogyan azt is, hogy milyen módszerek szerint, hogyan mérhetjük egy szoftver forráskódjának minőségét. Ebben a fejezetben kerülnek bevezetésre a dolgozatban később használt kódminőség és refaktoring fogalmak. Mindezen túlmutatóan, szemléletesen mutatom be a refaktoringot egy gyakorlati példán keresztül.

2.1. Kódminőség

2.1.1. A kódminőségről általában

Bármilyen rendszerről is legyen szó, a szoftverfejlesztőknek minden esetben a helyesen működő program a cél. A fejlesztők és felhasználók közös érdeke, hogy a elkészült szoftvertermék minél magasabb minőségű legyen, és a folyamatos fejlesztés alatt se veszítsen a minőségéből. A minőség fogalma ugyan nem egyértelmű, hiszen a szoftvernek egyszerre kell megfelelnie a specifikációnak (azaz a helyes, elvárt működéshoz), és egyszerre kell megfelelnie a fejlesztők, illetve fejlesztő cége különböző, fejlesztéshez kapcsolódó (belső) igényeinek is. Ilyen igények például: legyen könnyen karbantartható a szoftver, ritkán lépjen fel hiba, adott esetben könnyen és gyorsan lehessen a problémákat javítani.

Ezen okok összességéből adódóan a szoftverek minőségének objektív mérése nem triviális feladat. Létezik néhány módszer és eszköz, amely objektíven képes mérni a forráskód minőségét (pl.:

SQALE [10], QualityGate [9]), ezen kívül számos nemzetközi szabvány [4] is létezik. A legtöbb ilyen módszer az ún. metrikák segítségével jellemzi a kódunk minőségét. A metrikák a forráskód egyes számszerűsíthető adatait mérik, és általuk a kódunk egy-egy jellemzőjét kapjuk meg. (Ilyen jellemző például az osztály sorainak száma, vagy éppen a forráskódban található klónok száma, egymásba ágyazott „if” utasítások száma, stb.).

2.1.2. Elemzési módszerek

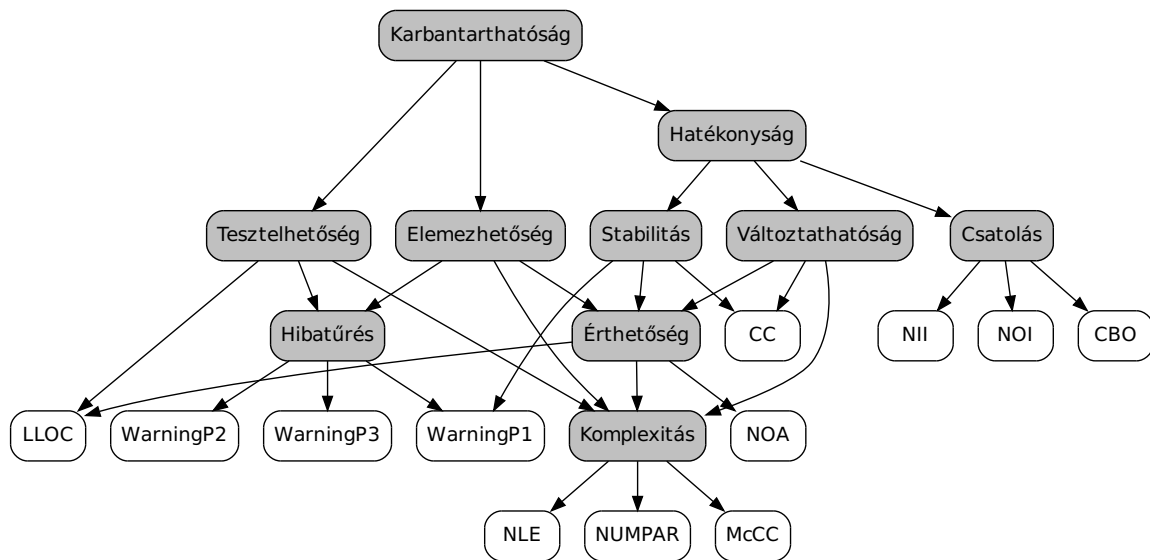
A szoftver minőségmutatóinak vizsgálatára számos különböző módszer létezik. Ezek jellemzően statikus vagy dinamikus elemzési módszerek. A statikus módszerek a forráskódot elemzik, így a programot nem szükséges futtatni, míg a dinamikus módszerek az adott futást, annak körülményeit, valamint a futás közben szerzett információkat együttesen vizsgáljuk meg. Az általunk használt eszközök a Columbus technológia statikus elemző eszközeire épülnek. Ezek az eszközök a probabilisztikus minőségmodell [13] alapján képesek meghatározni a szoftver minőségét, annak problémás részeit.

2.1.3. Probabilisztikus minőségmodell

A forráskód elemzése statikus eszközökkel történik, amely a szoftver kódjának egy adott verzióját használja. Az elemzés során a kódból ASG (*Abstract Semantic Graph – Absztrakt Szemantikus Gráf*) készül. Majd a kódelemző ezen a gráfon végzi el a metrikák számítását, valamint ezen a gráfon kerülnek meghatározásra a kódolási szabálysértések, kód duplikációk. A forráskód-minősítő eljárás az *ISO/IEC 25010* [5] szabvány által definiált magas szintű minőségi mutatókat egy körmentes, irányított gráf alapján határozza meg. A gráf minden egyes csomópontja egy minőségi mutatónak felel meg.

A 2.1. ábra a Java rendszerek minőségének kiértékelésére kidolgozott, valószínűségi minőségmodell alapján készült gráfot mutatja. A forráskód-minősítő eszköz célja, hogy a magas szintű minőségmutatókat objektíven határozza meg, az alacsony szintű – rendszer belső minőségét leíró – minőségmutatókkal. Ezek a mutatók konkrét, számszerűsíthető adatok, melyeket a forráskódból olvasunk ki. A minősítési eljárás során minden mutatót megnézünk, hogy az mennyire jó, vagy éppen rossz. Ez egy 0 és 1 közötti szám, ezt jósági értéknek nevezzük. A jellemző jóság értéke

nem határozható meg egyértelműen, ezért a minőségi mutatókat nem számokkal reprezentáljuk, hanem egy jóság függvénnyel. A jóság egy relatív fogalom, ezért a kapott értékeket összeveti a minősítési eljárás más rendszerek azonos értékeivel. Ehhez elengedhetetlen egy referencia adatbázis, mely nagyszámú rendszer elemzési eredményeit tartalmazza. A kiértékelt rendszert minden referencia rendszerrel összevetjük, és az előálló értékeket jóság függvénnyé alakítjuk, majd a megfelelő súlyozással előállítjuk a magas szintű minőségi mutatókat.



2.1. ábra. A probabilisztikus minőségmodell [13]

2.2. Refaktoring

2.2.1. A refaktoringról általában

Mindennapos, hogy egy szoftver hibátlan funkcionalitást produkál, de mindeközben találhatók olyan részek a kódban, amelyek már minimális módosítása esetén hibák, úgynevezett bugok jelennek meg a programuban, ami meghibásodáshoz vezethet. Az ilyen jellegű problémák megoldására is szolgál a refaktoring művelete. Martin Fowler így definiálta a refaktoring fogalmát: „A szoftver belső szerkezetén végzett változtatás, amelynek célja, hogy a szoftverkód jobban áttekinthető

és egyszerűbben módosítható legyen a megfigyelhető viselkedés megváltoztatása nélkül” [16]. Tehát a művelet elvégzése után egy átlagos felhasználó semmit sem vehet észre, viszont a rendszer belső strukturáltsága megváltozik (pozitív értelemben), így a kód letisztultabbá, érthetőbbé válik. Mindeközben ügyelve arra, hogy a lehető legkisebb legyen egy új hiba vagy bug bekövetkezésének lehetősége.

A refaktoringok viszonylag egyszerű műveletek, mint például az adattagok áthelyezése egyik osztályból a másikba, vagy az adott módszer felbontása több, különböző módszerre. A refaktoring felfogható egy speciális reengineeringnek (a reengineering alatt újratervezést értünk. Az újratervezés folyamán a már elkészült forráskódot tervezzük újra, javítjuk, így létrehozva egy jól megtervezett, implementált kódot.)

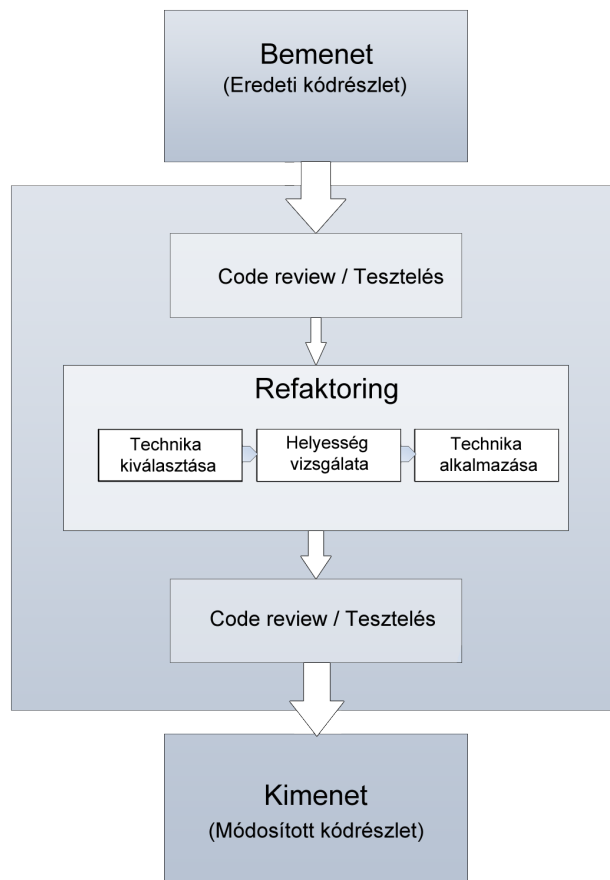
De a refaktoring folyamata nem triviális. A célunk mindig az, hogy a forráskód minőségén javítsunk, de könnyedén előfordulhat egy olyan eset is, hogy egy-egy refaktoring művelettel újabb problémákat hozunk létre. Egy egyszerű példa: túl komplex egy adott módszerünk, amelyet úgy szeretnénk refaktorálni, hogy a komplex módszert kisebb, önálló módszerekre bontunk. Ilyenkor azonban előfordulhat, hogy az osztályban túl sok lesz a módszer, vagy egy módszernek túl sok paramétere lesz, melyet valamilyen módon helyre kell hozni. Adott esetben helyes refaktoringok segítségével a kód minőségén sokat javíthatunk. A forráskódunk könnyebben értelmezhető lesz, karbantarthatóbb, kevesebbszer fordulhatnak benne elő hibák, működése gyorsabb lehet, valamint optimálisabb (pl. kevesebb memóriát foglal, gyorsabban felszabadítja a foglalt memóriát). A helyesen elvégzett refaktoringok mellett érdemes használni valamilyen tervezési mintát is, amely segítségével elkerülhetőek különböző tervezési hibák, hiányosságok is.

A refaktoringnak rengeteg technikája van, amelyeket többféle kategóriákba sorolhatunk. Az egyik ismert, Fowler [16] által definiált csoportosítás:

- Adatok elrendezése
- Függvények szerkesztése
- Tulajdonságok mozgatása objektumok között
- Feltételes kifejezések leegyszerűsítése
- Függvényhívások leegyszerűsítése

- Nagymértékű újratervezés
- Általánosítási feladatok

2.2.2. A refaktoring művelete



2.2. ábra. A refaktoring folyamata

A refaktoring műveletek végrehajtásának folyamatát szemlélteti a 2.2. ábra. Egy refaktoring folyamat során meg kell határoznunk az(oka)t a hibá(ka)t, amely a szoftver minőségének romlásáért felelősek. Ezeket a hibákat különböző statikus és dinamikus elemzőeszközökkel kereshetjük meg (pl.: PMD, Chord). A talált problémák javításához illeszkedő, megfelelő refaktoring technikát válasszunk. Ezek után kijavítjuk a hibát a megfelelő technika alkalmazásával, valamint a járulékos módosításokat is elvégezzük. A folyamat végén szükséges a módosított kódrészletet ellenőrizni, hogy a működése továbbra is az elvártak szerint történik-e. Végül, érdemes odafigyelni, hogy a

javítással nem hoztunk-e létre újabb hibákat.

2.2.3. Automatikus refaktoring eszközök

Az előbbiekben láttuk, hogy a refaktoring műveletek végrehajtása nem feltétlen bonyolult művelet. Ezt rengeteg IDE (Integrated Development Environment) fejlesztő is észrevette, ahogyan azt is, hogy sok esetben automatizálható a művelet. Így az alábbiakban néhány olyan eszközt szeretnék felsorolni, amelyek rendelkeznek automatikus refaktoring-mechanizmussal, melyek segítségével javíthatunk kódunk minőségén.

- Eclipse - Minden támogatott nyelv [2]
- NetBeans - Java [7]
- IntelliJ IDEA - Java [3]
- WebStorm - JavaScript [12]
- Microsoft Visual Studio - .NET [6]

2.2.4. Refaktoring végrehajtása egy példán szemlélítve

A következőekben a metódus kibontás (extract method) segítségével szeretném bemutatni a refaktoring műveletet. A két használt osztályunkat mutatja be a 2.1. és a 2.3. kódrészlet. A leegyszerűsített példánkban egy `Dolgozat` osztályt használunk, amely tárolja a dolgozatot író ember nevét, pontszámát, az elérhető maximális pontot, illetve az érdemjegyet. A `Kiirat.java` fájlban egyetlen, statikus metódus található, amely egy `Dolgozat` típusú objektumot vár paraméterül, majd ezt használva egy egyszerű szöveggel tér vissza.

```
public class Dolgozat {  
    public String nev;  
    public int pontszam;  
    public int maxPont;  
    public int erdemjegy;
```

```
@Override
```

```

    public String toString() {
        return "Dolgozat_[" + nev + ",_pontszam=" + pontszam + ",_
            maxPont=" + maxPont + ",_erdemjegy=" + erdemjegy + " ]";
    }
}

```

Kódrészlet 2.1. Dolgozat.java

Láthatjuk, hogy a `Kiirat` osztályban lévő `adatKiir` metódus túl hosszú, valamint tartalmaz egy logikailag teljesen különálló egységet (a switch szerkezetet). A switch szerkezet kiszervezhető egy külön metódusba, amely egy számot vár, és egy szöveggel tér vissza.

```

public class Kiirat {
    public static String adatKiir(Dolgozat dolgozat) {
        int tmpPont = dolgozat.pontszam;
        int tmpMaxPont = dolgozat.maxPont;
        int tmpErdemjegy = dolgozat.erdemjegy;

        double elertSzazalek = ((double) tmpPont / tmpMaxPont) * 100;

        String jegySzoveg;
        switch (tmpErdemjegy) {
            case 1: jegySzoveg = "elégtelen"; break;
            case 2: jegySzoveg = "elégséges"; break;
            case 3: jegySzoveg = "közepes"; break;
            case 4: jegySzoveg = "jó"; break;
            case 5: jegySzoveg = "jeles"; break;
            default: jegySzoveg = "ismeretlen"; break;
        }

        return dolgozat.nev + "_pontszáma:_ " + tmpPont + "_(" +
            elertSzazalek + "%)_érdemjegye:_ " + jegySzoveg;
    }
}

```

Kódrészlet 2.2. Kiirat.java

Ezért kiszervezzük a switch szerkezetet egy (például) „jegyAtvalto” nevű metódusba, amely egy egész számot vár, és egy szöveggel tér vissza. Ezen refaktoringot alkalmazva a metódusunk lerövidül, valamint egy másik létrejön, és így a logikailag elkülönülő egységek külön metódusba kerülnek. A művelet elvégzése után így fog kinézni a Kiirat osztályunk:

```
public class Kiirat {
    public static String jegyAtvalt(int ejegy) {
        String osztalyzat;
        switch (ejegy) {
            case 1: osztalyzat = "elégtelen"; break;
            case 2: osztalyzat = "elégséges"; break;
            case 3 : osztalyzat = "közepes"; break;
            case 4 : osztalyzat = "jó"; break;
            case 5 : osztalyzat = "jeles"; break;
            default : osztalyzat = "ismeretlen"; break;
        }
        return osztalyzat;
    }

    public static String adatKiir(Dolgozat dolgozat) {
        int tmpPont = dolgozat.pontszam;
        int tmpMaxPont = dolgozat.maxPont;
        int tmpErdemjegy = dolgozat.erdemjegy;

        double elertSzazalek = ((double) tmpPont / tmpMaxPont) * 100;
        String jegySzoveg = jegyAtvalt(tmpErdemjegy);
        return dolgozat.nev + "_pontszáma:_" + tmpPont + "_" +
            elertSzazalek + "%)_érdemjegye:_" + jegySzoveg;
    }
}
```

Kódrészlet 2.3. Kiirat.java (refaktoring után)

A fentiekből is látszik, hogy valóban nem bonyolult egy-egy ilyen művelet elvégzése, viszont a forráskód átláthatóbb, logikusabban felépített lett. Ilyen, és ehhez hasonló műveletek alkalmazásával adott szoftverrendszer kódjának minősége javítható, és ez a javítás nincs kihatással a program külső viselkedésére.

3. fejezet

Refaktoring műveletek vizsgálata, motivációs háttér

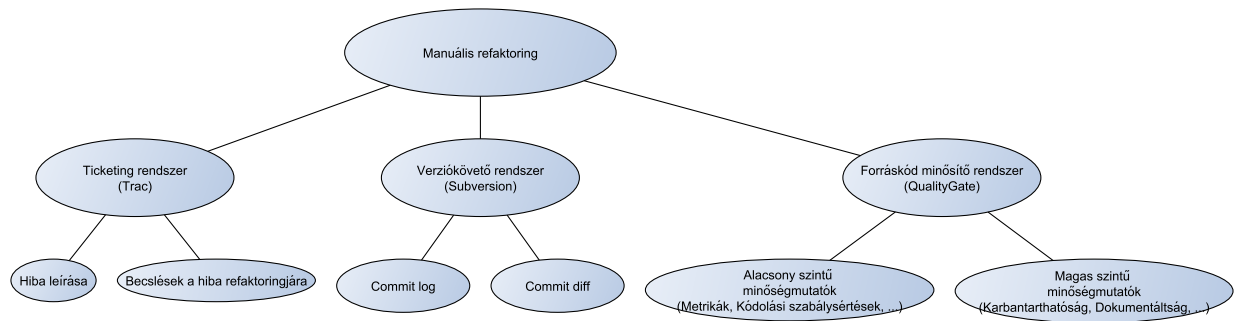
Jelen dolgozatban két szoftverfejlesztő cég refaktoring időszakát tanulmányozom. A cégek ezen időszak alatt manuálisan, azaz kézzel valósítottak meg refaktoringokat, bármely automatizált eszköz segítségével nélkül.

3.1. A refaktoring időszak vizsgálata

A refaktoring időszakban a cégek a Columbus statikus kódelemző eszközei segítségével elemezték a fejlesztett rendszerüket, és az eszközök által talált bad smell-eket, szabálysértéseket megvizsgálták, majd – amennyiben jogosnak találták – refaktorálták azokat, egy megfelelő refaktoring technikával. Mind a hiba felderítését (refaktoring okát), mind pedig a refaktoring műveletet is dokumentálták és különböző becsléseket adtak az elvégzett refaktoringgal kapcsolatban.

Egy-egy refaktoring művelettel kapcsolatban így több adatot is begyűjthettünk, amelyeket összegeztünk a 3.1. ábra, melynek egyes elemeit a továbbiakban ismertetem.

Dolgozatom célja, hogy az így összegyűjtött adatokat megvizsgálva következtetést vonhassunk le, hogy a szabálysértések javítása milyen hatást gyakorol a forráskódminőségre. Ezen időszak során több száz refaktoring műveletről gyűjthettünk adatokat, különböző cégek által fejlesztett, ipari rendszeren, ami jó alapot ad a refaktoring időszak tanulmányozására, és a bevezetőben is feltett kérdések megválaszolására.



3.1. ábra. Elemzési folyamat

3.2. Ticket rendszer

A Trac [11] egy nyílt forráskódú, web alapú projektmenedzselő eszköz, hibakövető rendszer. A partnercégek feladata volt, hogy a jogosnak ítélt szabálysértésekhez a Trac rendszerén belül hozzanak létre egy „hibajegyet”, azaz ticketet, amely tartalmazza a szabálysértés adatait: melyik fájlban találták meg, mi a hiba leírása, állapota, tulajdonosa, stb. Ezeken kívül a hiba bejelentőjének tennie kellett egy becslést is arra vonatkozóan, hogy a hiba javítása mennyi időbe telik, és milyen minőségváltozást eredményez. Ezek után a fejlesztők a bejelentett szabálysértéseket refaktorálták, majd az ehhez tartozó Trac ticketet lezárták.

3.3. Verziókövető rendszer

A kijavított hibákat ezek után az SVN (Subversion) [1] rendszerébe kellett felvinni (commitálni). A Subversion egy verziókövető rendszer, amely minden fájl minden verzióját tárolja egy központi szerveren (minden verzióhoz egy egyedi azonosítót rendel, melyet revízióknak hívunk), így folyamatosan nyomon követhető a forráskód-változás a fejlesztés egyes szakaszaiban. Egy-egy kijavított hiba commit üzenetébe a fejlesztők beleírták a Trac-beli ticket számot, vagyis beleírták, hogy melyik számú szabálysértést javították ki az adott feltöltéssel.

3.4. Forráskódminősítő rendszer

A forráskód elemzéséhez, és minősítéséhez a QualityGate [9] rendszert használtuk. A QualityGate egy forráskódminőség-menedzselő eszköz, amelynek két modulja van: SourceAudit, és CloneManager. Jelenleg, minden szükséges információt megszerezhetünk a SourceAudit modulból. A SourceAudit modul egy kódminősítő eszköz, amely az ISO/IEC-25010 [5] szabvány szerinti, a valószínűségi minőségmodell által definiált modellt alkalmazva képes a kódelemzésre. Az eszköz használatával magas-, és alacsonyszintű minőségmutatókat is elemezhetünk, több mint 60 metrika áll rendelkezésünkre, több mint 200 kódolási szabálysértést vizsgálhatunk a projektünkben, valamint folyamatosan monitorozhatjuk a fejlesztésben bekövetkező változások forráskódminőségre gyakorolt hatását.

A QualityGate segítségével minden egyes commitot (minden revíziót) leelemeztünk és minősítettünk. A kutatás során feladatom volt, hogy egy automatizált eszközt készítsek, amely automatikusan vezérli az elemzést, azaz a forráskód egyes revízióit letölti a központi szerverről, majd leelemzi a SourceAudit modul segítségével. Ehhez Bash és AWK nyelven írt szkripteket készítettem.

3.5. Nehézségek

Habár a partnercégekkel előzetesen egyeztettünk, mégsem sikerült egyik cégnek sem maradéktalanul teljesítenie a megbeszélteket, amelyekből kifolyólag a rendelkezésemre álló adatok feldolgozása is bonyolultabb lett.

A partnercégekkel történt megállapodás értelmében, a Trac rendszerébe minden egyes hibát külön ticketbe kellett volna felvenniük, de ez valami miatt nem sikerült teljes mértékben. Sok esetben előfordult, hogy egy tickethez több, sajnos néha teljesen különböző szabálysértést vettek fel, így ezeket az adatokat nem használhatjuk úgy, mintha csak az egyik szabálysértésre vonatkozna az adott commit/trac ticket.

Gyakran előfordult, hogy commitáláskor a commit üzenetbe nem írták bele a szabálysértés Trac-beli ticket számát, holott ezt külön kértük tőlük, s ők ezt tudomásul vették. Sőt, még olyan eset is volt, hogy a szabálysértés nevét sem írták be a commit üzenetbe, sőt semmi olyat, amelyből

kitalálható lett volna egyértelműen a refaktorálás oka. Az ilyen extrém esetekben kézzel kellett végigböngészni az adatokat, forráskód-beli módosításokat, és amennyiben megtaláltam, hogy az adott commit melyik tickethez tartozik, kézzel, táblázatba írtam azt be. És ebből az összekapcsolásból tudtam a munkát folytatni.

Előfordult az is, hogy a Trac-ben a tickethez nem írtak be szabálysértést a fejlesztők, így ebben az esetben is kézzel kellett a forráskód-beli módosításokat átvizsgálnom. Mindezek miatt az adatok feldolgozása hosszadalmas, időigényes folyamat volt.

3.6. A rendszerek vizsgálata, és az elvégzett refaktoringok száma

A fejlesztő cégek rendszerei (a dolgozatban A, B rendszer néven fogok rájuk hivatkozni, ahol az A rendszer jelöli az A cég által fejlesztett rendszert, a B rendszer pedig a B cég által fejlesztett rendszert) kivétel nélkül Java alapúak. Minden esetben az volt a cél, hogy javítsanak a rendszerük karbantarthatóságán, s ezzel együtt a lehető legtöbb minőségmutatón.

3.6.1. „A” rendszer

Metrika	Metrika értéke
TLOC	436 895
TLLOC	348 607
TNCL	4 370
TNM	40 220
TNPK	888

3.1. táblázat. Az „A” rendszer néhány jellemzője

Az „A” rendszer egy méretesnek mondható, kb. 348 ezer kódsoros Java ERP rendszer, aminek néhány alapvető méret metrikáit mutatja be a 3.1. táblázat. A táblázatban látható rövidítések jelentése, felülről lefelé haladva: Total Lines of Code (Összes kódsor), Total Logical Lines of Code (Összes tényleges kódsor, ebbe nem számít bele a komment- vagy üres sor), Total Number of Classes (Összes osztály száma), Total Number of Methods (Összes metódus száma), Total Number of Package (Összes csomag száma).

Szabálysértés	Refaktoringok száma
LF	29
LPL	10
PMD_SDTE	8
PMD_USBFSA	34
PMD_OBEAH	23
PMD_MRJA	14
PMD_AISD	2
PMD_CLA	1
PMD_ATNFS	1
PMD_ATNPE	1
PMD_CCOM	1
Összesen	124

3.2. táblázat. Elvégzett refaktoringok száma

A 3.2. táblázatban látható a rendszeren elvégzett refaktoringok száma. Ha összegezzük, akkor több, mint 100 refaktoring műveletet hajtottak végre a kézi refaktoring időszak alatt a fejlesztők. A refaktoringok egy részét a rendszer, alrendszer komolyabb méretű architektúrális átstrukturálása vezérelte (ilyenek a Long Function (LF), Long Parameter List (LPL) bad smellek javításai), egy másik részét pedig a PMD statikus kódelemző által jelzett szabálysértések eltávolítása. A táblázatban is látható bad smellek és szabálysértések részletes ismertetésével a 4. fejezet foglalkozik.

Itt megjegyezzük, hogy a dolgozatban a PMD szabálysértés javítását is refaktoringoknak tekintjük, ugyanakkor előfordulhat, hogy egy PMD szabálysértés javítása során a szoftver viselkedése kis mértékben megváltozik (pl. a szoftver futása korábban null-pointer hibával megállt, ugyanakkor a javítás után a hiba már nem fordul elő). A klasszikus, Fowler-féle refaktoring definíció szerint a refaktoring művelet célja, a kód minőségének javítása, a program külső viselkedésének megtartása mellett. A definíció viszont nem tér ki arra, hogy a külső viselkedés alatt pontosan mit értünk, egy programozói hibát pedig gyakran inkább minőséghibának tekintünk, mint specifikáció szerinti, elvárt viselkedésnek.

3.6.2. „B” rendszer

A „B” rendszer egy közepes méretű, kb. 172 ezer soros J2EE technológián alapuló keretrendszer. Ezen rendszer néhány metrikáját szemlélteti a 3.3. táblázat. A táblázatban található rövidítések:

Metrika	Metrika értéke
TLOC	172 003
TLLOC	134 007
TNCL	1057
TNM	9949
TNPK	172

3.3. táblázat. A „B” rendszer néhány jellemzője

Total Lines of Code (Összes kódsor), Total Logical Lines of Code (Összes tényleges kódsor, ebbe nem számít bele a komment- vagy üres sor), Total Number of Classes (Összes osztály száma), Total Number of Methods (Összes metódus száma), Total Number of Package (Összes csomag száma).

Szabálysértés	Refaktoringok száma
PMD_ECB	2
PMD_ALOC	2
PMD_ADL	2
PMD_LC	4
PMD_CC	3
PMD_CT	2
PMD_ULV	2
PMD_ULBR	2
PMD_SSSHD	3
PMD_MRIA	2
PMD_PLFIC	4
PMD_PCI	5
Egyéb PMD	27
Egyéb	10
Összesen	70

3.4. táblázat. Elvégzett refaktoringhoz tartozó ticketek száma

A rendszerben 70 darab ticket volt, amely jóval több elvégzett refaktoringra utal, melyeket a manuális refaktoring időszak alatt. Ezek egy része a kód nagyobb lélegzetvételű átstrukturálása volt, valamint viszonylag sok PMD szabálysértést is sikerült kijavítaniuk a fejlesztőknek. Jelen táblázatban olvasható „Egyéb PMD” sor, amelyben az összes többi olyan javított PMD szabálysértés van, amelyhez csak 1-1 ticket (és az ahhoz tartozó commit) létezik, így azok nem lettek feltüntetve a táblázatban.

4. fejezet

Szabálysértések, metrikák

Ebben a fejezetben egy leírást készítettem a cégek által javított főbb szabálysértésekről, melyik mire hívja fel a figyelmet, valamint említést teszek a leggyakoribb javítási lehetőségekről is.

4.1. Szabálysértések

4.1.1. „Bad smell”

A „bad smell”, avagy code smell a program forráskódjában található olyan jelenség, amely nem okoz közvetlenül meghibásodást, mint pl. a bug-ok, viszont a későbbi fejlesztés során problémákat okozhatnak. Ezek a kódrészletek az adott helyzetben jól működnek ugyan, de egy-egy apró módosítás is igen komoly, költséges hibákat idézhet elő.

Long Function *LF*

A probléma: egy metódus túl hosszú, és/vagy több dolgot is megvalósít. Minél hosszabb egy metódus annál nehezebben érthető, karbantartható. A javításnak két tipikus módja létezik: az összefüggő részeket kiemeljük egy új függvénybe, és a régi függvényből meghívjuk ezt a függvényt, vagy pedig a hosszú függvény algoritmusát leegyszerűsítjük, ezáltal lerövidítjük a kódot.

Long Parameter List *LPL*

Ez a bad smell akkor áll fent, amikor egy függvény túl sok paramétert vár. Gyakran ezek az adattagok együtt fordulnak elő, különböző függvényhívásokban. Ilyen esetben úgy javíthatjuk a problémát, hogy az együtt előforduló adattagok számára létrehozunk egy új osztályt, és a függvényünknek egy ilyen típusú objektumot adunk át.

4.1.2. PMD

A PMD[8] egy statikus forráskódelemző eszköz, amely különböző problémaforrásokat talál meg, mint például: lehetséges bug - üres catch block, használaton kívüli kódrészlet - nem használt változók, privát metódusok, paraméterek, túl bonyolult kifejezések, túl komplex metódusok, duplikált kódrészlet. A PMD a forráskódból egy AST-t (*Abstract Syntax Tree – Absztrakt Szintaxis Fa*) épít fel. Ezek után az AST-n keres különböző szabálysértéseket XPath, valamint előre elkészített, szabálysértést felismerő Java nyelven implementált algoritmusok segítségével.

Override Both Equals and Hashcode *PMD_OBEAH*

Ez a szabály kimondja, hogyha egy adott osztály felülírja az equals() vagy hashCode() metódusok bármelyikét, akkor mindkét függvényt érdemes felülnünk. Erre azért van szükség, mert definíciójuk kiegészíti egymást, így csak mindkettő megléte és összehangoltsága esetén van értelme használnunk őket. Ezen szabálysértés leggyakoribb javítása a meglévő metódus mellé a hiányzó metódus felülírása, vagy pedig a nem használt, de felülírt metódus törlése.

Array is Stored Directly *PMD_AISD*

Ez a probléma akkor áll fent, amikor egy tömböt nem másolunk, hanem referencia szerint adunk értékül. Ez azért veszélyes, mert ezáltal a hívó számára továbbra is lehetőség van az objektum állapotának megváltoztatására, korrupt helyzet kialakítására. Ezt a szabálysértést legtöbbször úgy javítunk, hogy nem referenciát másolunk, hanem a tömb tartalmát másoljuk egy új tömbbe, a System.arraycopy() metódus segítségével.

Method Returns Internal Array *PMD_MRIA*

Ez a szabálysértés akkor áll fent, ha egy függvény visszatérési értéke tömb, és az egy tömböt direktben ad vissza a hívófél számára. Azokon a helyeken, ahol ezt a tömböt felhasználjuk, előfordulhat, hogy a tömb tartalmát valaki módosítja a saját igényeinek megfelelően. Ez veszélyes, hiszen így az eredeti tömb tartalmát is módosítjuk. A probléma javítása legtöbbször úgy történik, hogy egy új tömböt adunk vissza, melyet az eredeti tömbről másolunk, és ezt tesszük elérhetővé.

Avoid Throwing Null Pointer Exception *PMD_ATNPE*

A szabály szerint érdemes kerülni a NullPointerException dobását, mivel ezzel nem lehet pontosan meghatározni a hiba eredetét. Ennek a problémának nincsen jellemző javítási módja, többféleképpen megoldható a NullPointerException eltüntetése.

Use StringBuffer for String Appends *PMD_USBFSA*

Ez a probléma akkor áll fent, amikor egy String-et += operátorral bővítenek valahol, többszöri alkalommal (pl.: for ciklus). Ez azért nem helyes, mert a += operátor használatával ideiglenesen létrejön egy StringBuffer objektum, amely konkatenálja a Stringeket. Ezután meghívjuk az osztály toString() metódusát, melynek hatására új String objektum jön létre. Ezért ez az operátor több memóriát használ, és lassabb, mintha kézzel készítenénk a kódunkba egy StringBuffer-t. A legtöbbször ezt a szabálysértést úgy oldjuk meg, hogy a kódban a += konkatenálást eltávolítjuk, és létrehozunk egy StringBuffer/StringBuilder objektumot, és ezt használjuk.

Signature Declare Throws Exception *PMD_SDTE*

Ennek a szabálynak az értelmében kerülni kell a függvények fejlécében az általános hibadobást (pl.: public void foo() throws Exception). Ez túlságosan általános hibakezelés, ami gondot okozhat, hiszem függvényhívásnál, ha kivétel dobódik, akkor nem fogjuk tudni a konkrét hibatípust, így azt speciálisan lekezelni sem tudjuk. A javítás leggyakoribb módja az általános helyett speciális hibadobás-deklaráció, valamint, adott esetekben törölhető is a deklarációból ez a hibadobás-jelzés.

Assignment To Non Final Static *PMD_ATNFS*

Ez a probléma olyan esetekben fordul elő, amikor egy nem final, de statikus adattagnak értéket adunk. Úgy javíthatunk egy ilyen szabálysértést, ha a statikus mező helyett sima adattagot használunk.

Proper Clone Implementation *PMD_PCI*

A szabálysértés értelmében a clone() metódust a super.clone() segítségével kell implementálni, ügyelve a kivételekre.

Position Literals First In Comparisons *PMD_PLFIC*

Ez a szabály arra figyelmeztet, hogy összehasonlításoknál mindig a literált hasonlítsuk össze a változó értékével, mivel így ha egy Null objektummal hasonlítjuk össze a literált, az simán false-szal tér vissza, míg ha Null objektum metódusát próbáljuk meghívni, akkor NullPointerException-t dob a program.

Unused Local Before Return *PMD_ULBR*

Ez a jelenség akkor áll fent, amikor (közvetlenül) egy függvény visszatérése előtt hozunk létre egy változót, majd ezzel a frissen létrehozott változóval térünk vissza. A változó létrehozása teljesen felesleges, hiszen a változónak értékül adott kifejezéssel is visszatérhetünk.

Unused Local Variable *PMD_ULV*

Ez a probléma akkor áll fent, amikor egy függvényen belül létrehozunk egy változót, esetleg inicializáljuk, de azt sehol sem használjuk fel a későbbiekben. Ezen változó létrehozása teljesen felesleges, hiszen a későbbiekben sehol nem használjuk, nem befolyásolja a program működését.

Switch Statements Should Have Default *PMD_SSSHD*

Ezen szabálysértés értelmében minden switch-bloknak kell, hogy legyen egy default ága. Abban az esetben, ha egy olyan értéket kap a switch-blokk, ami nincsen egyetlen ággal sem lefedve, a switch-blokkban semmi sem fut le, és feltehetően nem ez volt a programozó célja, hiszen nem

véletlen van ott a switch blokk. Minden esetben érdemes default ágot készíteni, még akkor is, ha jelenleg csak olyan érték vehet fel a változó, amikre már van eset, hiszen a jövőben előfordulhatnak új esetek is, melyek így nem kerülnének lekezelésre.

5. fejezet

Eredmények, következtetés

Ebben a fejezetben a rendszer összes revíziójának minősítése után vizsgáljuk meg a metrikákat, valamint a minőségváltozást. A minőségváltozás minden esetben a probabilsztikus minőségmodellen alapuló minőségjellemző, így itt kaphatunk választ a bevezetőben feltett kérdésekre is.

5.1. Az eredmények bemutatásánál használt fogalmak, metrikák leírása

Az eredmények ismertetése előtt, fontos, hogy megismerjünk néhány alapvető fogalmat, amelyeket az alábbiakban ismertetek.

A magas szintű minőségmutatók bemutatása:

- *Analyzability (Elemezhetőség)*: A szoftver hiányosságai és különböző hibák okait milyen könnyű beazonosítani.
- *Changeability (Változtathatóság)*: Előre meghatározott módosításokat mennyire egyszerű elvégezni.
- *CodeComplexity (Kód komplexitás)*: A forráskód általános komplexitása.
- *CodeFaultProneness (Hibatűrés)*: Mekkora az esély arra, hogy az adott rendszerelemben hiba van?

- *Comprehensibility (Érthetőség)*: Mennyire könnyű a rendszer kódját megérteni, átlátni (módosítások elvégzéséhez, új funkciók bevezetéséhez, hibajavításhoz).
- *Documentation (Dokumentáltság)*: A rendszer dokumentáltsága.
- *Maintainability (Karbantarthatóság)*: A rendszer mennyire tartja be a karbantarthatósághoz kapcsolódó szabványokat, konvenciókat.
- *Modifiability (Módosíthatóság)*: A rendszer egyes elemeit mennyire könnyű átalakítani.
- *Stability (Stabilitás)*: A rendszer mennyire képes elkerülni a módosítások következtében felépő előre nem látható hatásokat.
- *Testability (Tesztelhetőség)*: Mennyire könnyű a rendszer adott elemeit ellenőrizni, tesztelni.

Az alacsony szintű minőségmutatók bemutatása:

- *CC*: Klónlefedettség, azaz az adott rendszeren belüli klón kódsorok száma.
- *CriticalRuleViolations*: A kritikus szabálysértések számából számolt minőségmutató.
- *MajorRuleViolations*: A jelentős, de nem kritikus szabálysértések számából számolt minőségmutató.
- *LLOC (Logical Lines of Code)*: A LLOC metrikából számolt jellemző.
- *McC* (*McCabe's Cyclomatic Complexity*): Az *McC* metrikából számolt minőségmutató.
- *NUMPAR (Number of Parameters)*: A *NUMPAR* metrikából számolt minőségmutató.

5.2. „A” rendszer

5.2.1. Becsült és tényleges ráfordítás a refaktoringok elvégzésére

Az elemzett kódbázis és a Trac adatbázis segítségével összegyűjtöttem, hogy az akkor még elvégzendő kódolási hibák javítására átlagosan mennyi időt becsültek a fejlesztők.

A refaktoring művelet elvégzésére ténylegesen ráfordított időről nincsen egzakt adatunk, ugyanakkor az elvégzett refaktoringok adataiból láthatjuk, átlagosan hány sort érintett 1-1 módosítás. Az 5.1. táblázatban látható adatok összegyűjtött adatok, melyeket szabálysértés típusonként kategorizáltam, így láthatóak a típusonkénti becsült átlagidő, valamint az egyes szabálysértéstípusok javításával átlagosan hány kódsor módosult, azaz egy darab refaktoring az adott típusból átlagosan hány sor módosításával járt.

A táblázatban szereplő átlagos időt úgy számoltam ki, hogy megnéztem, a fejlesztők egyesével mennyi időt becsültek 1-1 refaktoring kijavítására (az időt egy előre megadott menüből választhatták ki: <1 óra, 1-2 óra, 3-8 óra). Ezeket az értékeket átlagoltam, így kaptam meg ezeket az időintervallumokat.

Azonban az egy probléma javítására szánt idő nem mérhető egzaktan, így a módosult kódsorokat hozom összefüggésbe az előre megbecsült idővel. Valóban, már pusztán egy-egy kódsor módosítására szánt idő is teljesen eltérhet a probléma jellegétől függően, ezért nem igaz, hogy ha két különböző típusú probléma azonos mennyiségű sor módosításával jár, akkor az azokra ráfordított idő is azonos lesz. Általában, az eredmények mégis azt mutatják, hogy a több kódsort érintő módosításokhoz a fejlesztők is nagyobb idő ráfordítást becsültek.

Szabálysértés	Módosult sorok száma (átlagosan)	Javításhoz szükséges becsült idő (átlagosan)
LF	230,24	1- 2.5 óra
LPL	94,90	1.3 - 3.1 óra
PMD_CLA	33	0 - 1 óra
PMD_SDTE	17,13	0 - 1 óra
PMD_USBFSA	7,21	0.3 - 1.3 óra
PMD_OBEAH	6,09	0 - 1 óra
PMD_ATNFS	6	0 - 1 óra
PMD_MRIA	4,71	0 - 1 óra
PMD_AISD	3	0 - 1 óra
PMD_ATNPE	1	0 - 1 óra
PMD_CCOM	1	0 - 1 óra

5.1. táblázat. Átlagosan módosult kódsorok száma, valamint a becsült idő szabálysértéstípusonként

A táblázatból láthatjuk, hogy a *Long Function* bad smell javítása érinti a legtöbb kódsort. Ez érthető, hiszen itt arról van szó, hogy egy függvény túl hosszú. Ugyanakkor elsőnek talán érthetetlen, hogy egy ennyi módosítással járó javítást miért becsültek csak ennyi időre? A válasz a bad

smell jellegéből adódik: a forráskód már kész van. A javítás jellemzően úgy történik, hogy az erősen összefüggő forráskódrészleteket kiszervezzük egy külön metódusba, majd az eddigi függvényünkéből meghívjuk azt. Így valóban egyszerűbb dolgunk van, mintha külön osztályokat, vagy új metódusokat kellene írni.

Ugyanakkor látszik, hogy a második, *Long Parameter List* bad smell javítására már átlagosan is több időt becsültek, noha a módosult kódsorok száma itt jóval kevesebb. A válasz itt is a bad smell jellegében keresendő: túl hosszú a paraméterlista. Itt már jóval nehezebb lehet a javítás. Legtöbbször, ha a sok paraméter egy objektumból származik, egyszerűbb az egész objektumot átadni, majd lekérni az attribútumait. De nem mindig ilyen egyszerű a javítás, ugyanis előfordulhat, hogy az összes paramétert különböző objektumtól kapjuk, ilyenkor már átfogóbb, architekturális szintű javításra van szükség.

A további szabálysértéseknél láthatjuk, hogy jóval kevesebb kódsorváltozás is történik, és szinte mindenhol 0-1 óra közé becsülhető a javítás ideje. Ezek a szabálysértések láthatóan kisebb lélegzetvételűek, nem igényelnek architekturális szintű módosítást, átírást.

5.2.2. Egyes szabálysértés típusok refaktoringjának becsült és tényleges hatása a minőségre

A refaktoring elvégzése előtt a fejlesztőknek meg kellett becsülniük, hogy az adott probléma javítása milyen mértékben javítja a rendszer minőségét. Ezt a becslést egy ötfokú skálán kellett megtenniük, melynek értékei: 1 – semmilyen, 2 – alacsony, 3 – mérsékelt, 4 – magas, 5 – nagyon magas. Noha a refaktoring célja egyértelműen a forráskód minőségének javítása, előfordulhat, hogy egy beazonosított hibát csak úgy tudunk javítani, ha ezáltal egy új, hasonló súlyosságú hibát hozunk létre. Az ilyen, és ehhez hasonló esetek miatt kellett bevezetni az „1 – semmilyen” értéket. Ez az adat összevethető a QualityGate által mért forráskódminőség mutatóval, amit a QualityGate a 0-10 közötti jóság értékkel szemléltet.

Az 5.2. táblázatban látható értékek az egyes hiba típusok szerinti átlagolt értékek. A becsült értékek a fejlesztők által adott értékek átlaga, a tényleges minőségjavulást pedig a probabilisztikus minőségmodellen alapuló minőségvizsgáló eszköz, a QualityGate segítségével vizsgáltam meg, és táblázatban lévő minőségjavulási értékek a rendszer teljes egészére értendő, általános minőségmu-

tató.

Szabálysértés	Becsült minőségjavulás	Mért minőségjavulás
PMD_ATNPE	2	0,000162
PMD_SDTE	2	0,000139
LF	2,5	0,000126
PMD_MRIA	2	0,000105
PMD_OBEAH	3	0,0000744
PMD_USBFSA	2	0,0000518
PMD_AISD	3	0,0000351
LPL	2,6	0,00002
PMD_CCOM	3	-0,0000231
PMD_CLA	2	-0,000076
PMD_ATNFS	2	-0,000255

5.2. táblázat. Módosítások becsült és mért minőségjavulásának összevetése

A táblázatból jól látható, hogy az ráfordításban is kisebb volumenűnek mondható, PMD szabálysértések javításának a minőségre gyakorolt hatását a fejlesztők is kisebbre becsülték és a ténylegesen mért minőségjavulás is rendszerint alig volt mérhető. A legnagyobb időráfordításra becsült LPL és LF bad smell javításától közepes minőségjavulást vártak a fejlesztők, ugyanakkor az LF bad smell javítása jobban mérhető minőségjavulást eredményezett, mint az LPL probléma javítása.

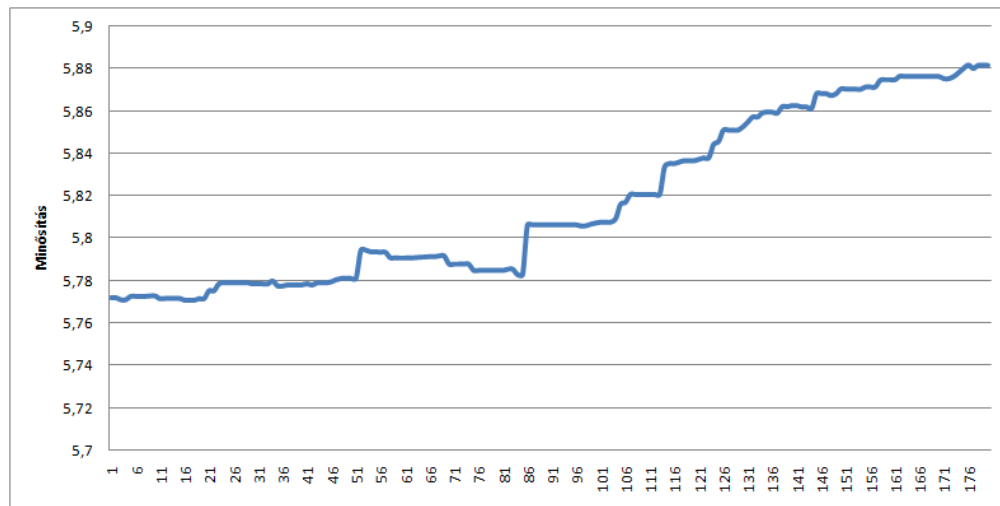
A táblázatból az is látható, hogy egyes refaktoringok egyenesen negatív minőségjavulást eredményeztek. Ezek általában olyan hibák, ahol a javításhoz egy új osztály, metódus bevezetése lehetett szükséges, így a szabálysértések száma ugyan csökkenhetett a javítás során, az egyéb szenzorok változása mégis negatív irányba változtatta a szoftver minőségét.

Az eredmények alapján a sejtésünk az, hogy önmagában, egy kisebb hiba (pl. PMD szabálysértés) okozta refaktoring, minőségre gyakorolt hatásának mérése nehezen kimutatható, csak olyan esetben mérhető ez, ahol a rendszer nagyobb részét átfogóan érinti a refaktoring.

A sejtéshez megmértük, a refaktoringok minőségre gyakorolt hatásának átlagos mértékét, amely alapján egy darab szabálysértés javítása a minőséget átlagosan 0,000061 ponttal javította. Ez egy nagyon alacsony értékű javulás, amely szinte érezhetetlen, a rendszer teljes egészére tekintve. Forráskód-szinten a javított hiba környezetében természetesen észrevehető ez a változás is. A minőségváltozás értéke a [-0,000387; 0,002335] intervallumból került ki, minden esetben. Tehát jelen esetben bármelyik refaktoringot is nézzük meg, önmagában nem elegendőek a kódminőség jelen-

tős javítására. Azonban mi a helyzet abban az esetben, ha nem csak egy-egy refaktoringot nézünk meg, hanem az összeset?

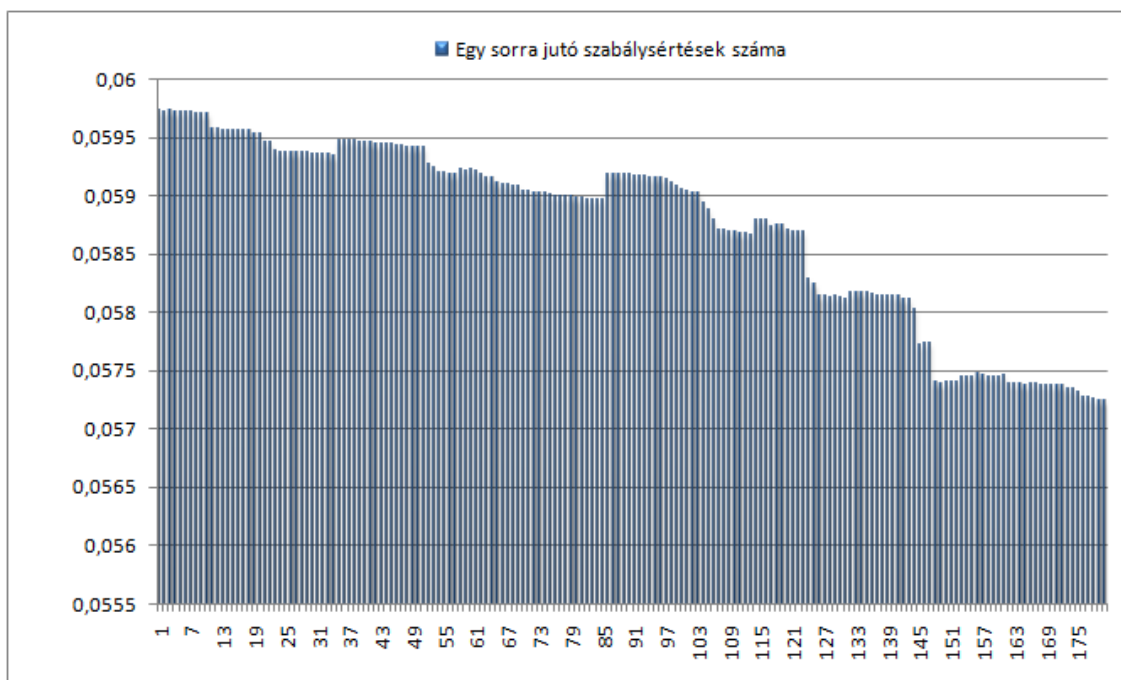
5.2.3. Az elvégzett refaktoringok teljes rendszer minőségére gyakorolt hatása



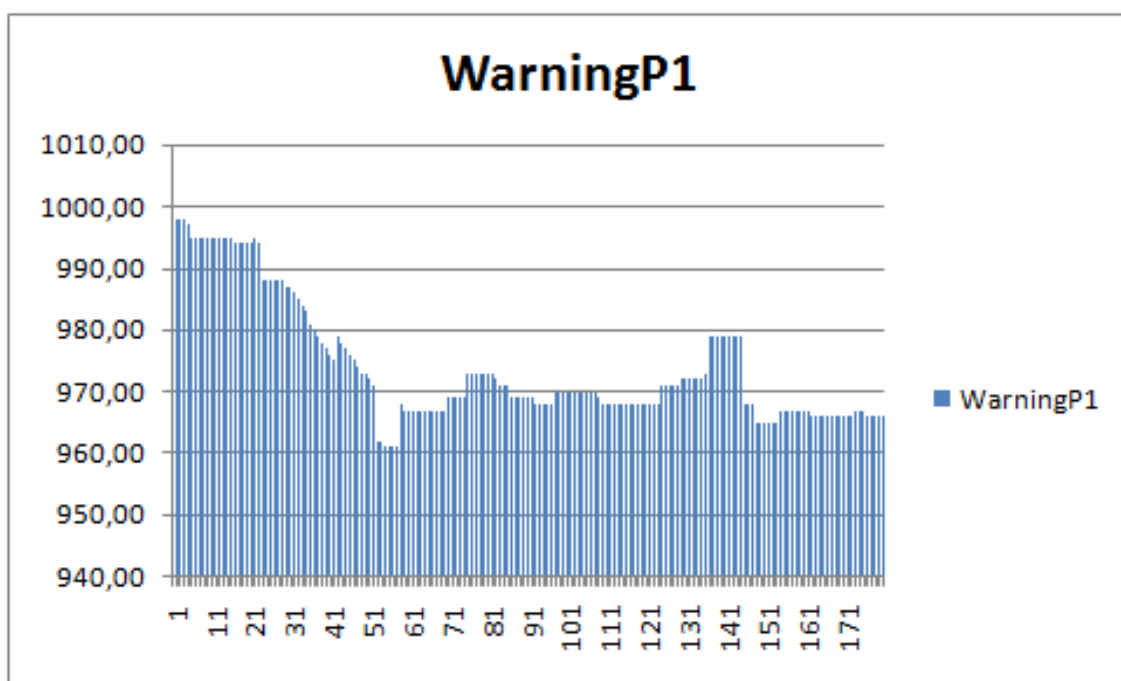
5.1. ábra. Revíziókénti minőségértékek

A refaktoringok elvégzésének ideje alatti minőségváltozást láthatjuk az 5.1. ábrán. Az ábra alapján is látható, és az adatok is alátámasztják (a minőség 5,77-ről 5,88-ra módosult), hogy a minőségben történő változás kimutatható, a teljes rendszer távlatában. A bekövetkező változás pozitív, tehát a refaktoring folyamata célravezető volt. Az ábrán is látható, hogy a javulás nem folyamatos, előfordul – még ha kevés esetben is – hogy a minőség csökken. Ennek az az oka, hogy a vizsgált időszakban nem tisztán refaktoringok történtek, hanem emellett folyamatosan fejlesztettek is, s ezen a fejlesztések során commitálhattak olyan kódot, amely a minőségmutatókat negatív irányba befolyásolta.

Az 5.2. ábrán láthatjuk, adott revízió esetében egy kódsorra hány szabálysértés jut. Ezt a következőképp számoltam ki: $EgySorraJutoWarningokSzama(Rev) = \frac{WarningAll(Rev)}{TLLOC(Rev)}$ Ahol a *Rev* az adott revíziót jelöli, míg a *WarningAll* az összes szabálysértés (*WarningP1* + *WarningP2* + *WarningP3* metrika) száma, a TLLOC pedig a *Total Logical Lines Of Code* metrikát jelenti. A TLLOC az összes tényleges kódsort jelölő metrika, amibe komment sor, vagy üres sor nem kerül beszámításra.



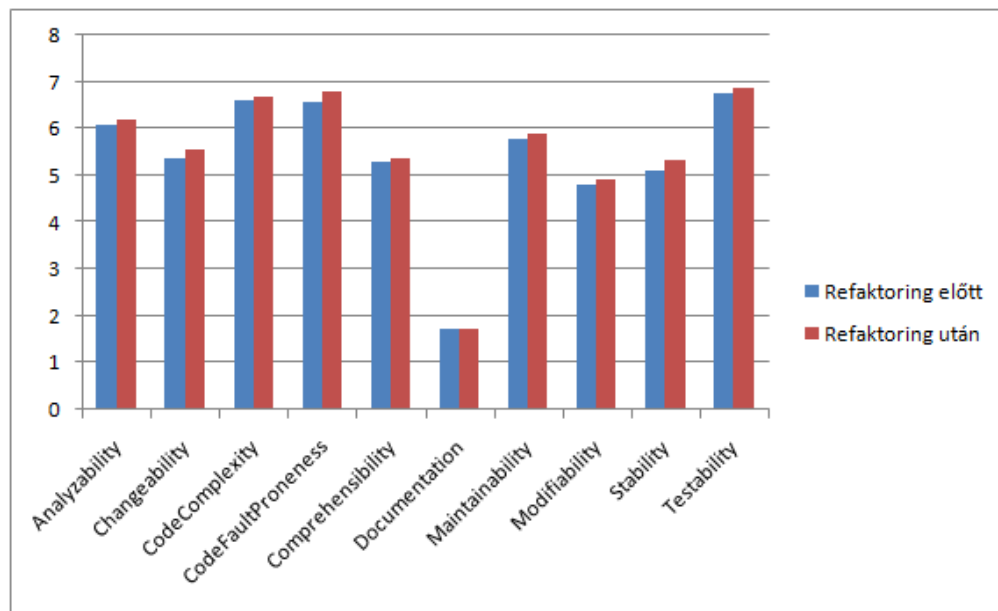
5.2. ábra. Szabálysértések száma soronként



5.3. ábra. P1-es prioritású (súlyos) szabálysértések száma

A refaktoring időszakban elsődlegesen a magas prioritású szabálysértések javításán volt a hangsúly. Az 5.3. ábrán láthatjuk, hogy az idő múlásával hogy csökkent az egyes prioritású szabálysértések száma. Itt is jól kivehető, hogy a refaktoringok mellett fejlesztés is történt a kódbázisba, de ettől függetlenül is látható a refaktoring hatása a projekt egészére tekintve.

5.2.4. Magas szintű minőségmutatók változása a refaktoring során

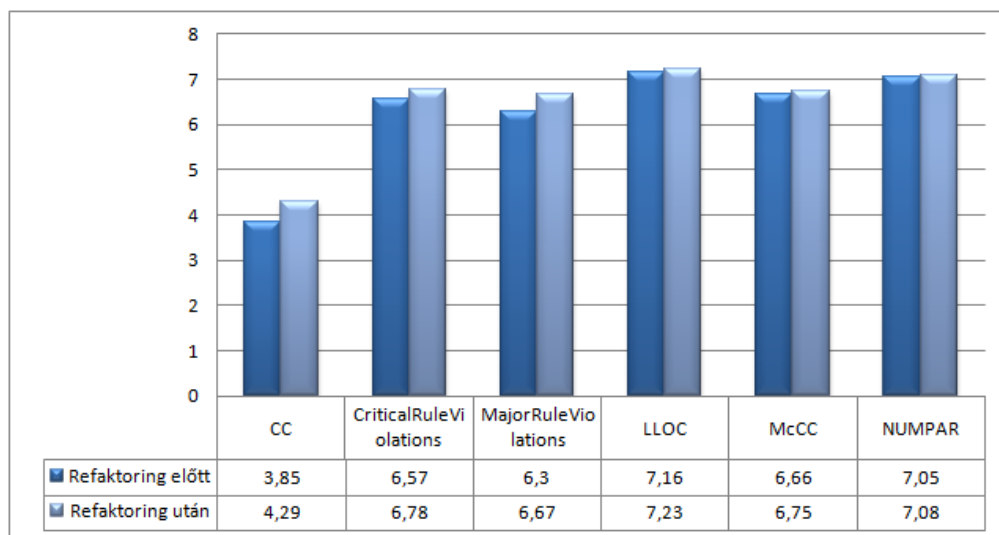


5.4. ábra. Magas szintű mutatók változása

Az 5.4. ábrán láthatóak az *ISO/IEC 25010* [5] szabvány által definiált magas szintű minőségi mutatók értékei a refaktoring időszak előtt, valamint azután. Ezeket a minőségmutatókat a fejezet bevezető szekciójában már ismertettük.

Láthatjuk, hogy a dokumentáltsági mutatón kívül az összes mutató értéke nőtt. Ez valószínűsíthetően azért van, mert kijavított szabálysértések egyike sem volt kommenteléshez köthető szabálysértés, ebből kifolyólag érthető, hogy ez a minőségmutató nem is változott.

A magas szintű minőségmutatók javulásán is látható, hogy a valószínűségi minőségmodellen alapuló minőségértékeken látható a refaktoring hatása.



5.5. ábra. Néhány alacsony szintű mutató változása

5.2.5. Alacsony szintű minőségmutatók változása a refaktoring során

Míg a magas szintű minőségjellemzők több, különböző alacsony szintű mutató súlyozott aggregációja, addig az alacsony szintű mutatók közvetlenül a forráskódból kiolvasható értékek, melyek a rendszer egy-egy konkrét jellemzőjét írják le. Az 5.5. ábrán láthatunk néhány alacsony szintű minőségjellemző értéket a refaktoring időszak előtt, valamint utána. Néhány információ ezekről a mutatókról:

Az alacsony szintű minőségmutatók értékén is látszódik a refaktoring időszak, a vizsgált mutatók mindegyike pozitívan változott, ami azt jelenti, hogy az elvégzett refaktoringok csökkentették a klónlefedettséget, sokkal kevesebb a magas prioritású szabálysértés, valamint egy-egy metódus átlagosan kevésbé komplex, és kevesebb paraméterrel is működnek, mint a refaktoring időszak előtt.

5.3. „B” rendszer

A „B” rendszer vizsgálata során az adatok feldolgozását nehezítette, hogy a fejlesztők olykor összevont refaktoringokat alkalmaztak, vagy éppen a szabálysértés javításakor megláttak egy kódrészletet, melyet szintén módosítottak, így az adott commitban nem csak a szabálysértés javítása volt megtalálható, hiába kértük őket, hogy ezen időszakban kerüljék az ilyen, és ehhez hasonló javítá-

sokat.

Az itt bemutatott táblázatok ezért csak a releváns mennyiségben feldolgozható refaktoringokat tartalmazzák. A teljesség érdekében, az összes refaktoringot bemutató táblázatot a függelékben mellékeljük (A.1. táblázat).

5.3.1. Becsült és tényleges ráfordítás a refaktoringok elvégzésére

A fejlesztők a refaktoring elvégzése előtt becslést tettek arra, hogy az adott refaktoring hogy fogja befolyásolni a kódminőséget. Természetesen – ahogy az „A” rendszer esetében sem lehetett – a refaktoringok tényleges megvalósításának idejét nem lehet pontosan lemérni, viszont láthatjuk, hogy az adott ticketnek megfelelő commit esetében hány sor módosult. A függelék A.1. táblázatában látható minden javított szabálysértés minden adata, melyek rendelkezésünkre álltak. A táblázatból egy rövid részletet mutat be az 5.3. táblázat.

Szabálysértés	Módosult sorok száma (átlagosan)	Javításhoz szükséges becsült idő (átlagosan)
PMD_OBEAH	82	0-1 óra
PMD_PCI	66	1-2 óra
PMD_PLFIC	23	0.75-1.75 óra
PMD_SSSHD	214	1-2 óra
Nem PMD (metrika, klón)	378	0.5-2 óra

5.3. táblázat. Átlagosan módosult kódsorok száma, valamint a becsült idő szabálysértéstípusonként

Láthatjuk, hogy a becslések nem teljesen tükrözik a valódi eredményt, de ez érthető is, hiszen egy probléma megoldása függ a probléma nehézségétől, és ezt megbecsülni nem triviális feladat. A táblázatból azt is láthatjuk, hogy a legtöbb kódsorváltozást azok a szabálysértések hozták, melyek nem PMD szabálysértések voltak, hanem klónok, metrika alapú szabálysértések, és ezek javítása jóval komolyabb feladat, mivel ezek bonyolultabb szerkezeti átalakítást igényelhetnek.

Azt is leolvashatjuk a táblázatról, hogy a legkevesebb módosítást a PLFIC (Position Literals First In Comparisons) szabály javítása igényelte. Ez a szabály jellegéből adódóan (4. fejezet) következik, hiszen kevés kódsor módosításával eltüntethetjük a rendszerből az ilyen jellegű szabálysértéseket.

5.3.2. Egyes szabálysértés típusok refaktoringjának becsült és tényleges hatása a minőségre

A fejlesztők ezen rendszer esetén is egy ötös skálán osztályozhatták, hogy szerintük milyen hatással lesz az adott refaktoring a rendszer minőségére. Mint említettem korábban ez az adat összevethető a QualityGate rendszer karbantarthatósági mutatójával. Az 5.4. táblázatban láthatunk néhány adatot, melyek az egyes szabálysértés-típus szerint kerültek átlagolásra. A becslések a fejlesztők által adott értékek átlaga, a tényleges javulást pedig a QualityGate SourceAudit modul segítségével vizsgáltam meg. A táblázatban található javulás a rendszer teljes egészére kiterjedő karbantarthatósági minőségmutató. A teljes táblázat a függelékben található (A.1. táblázat).

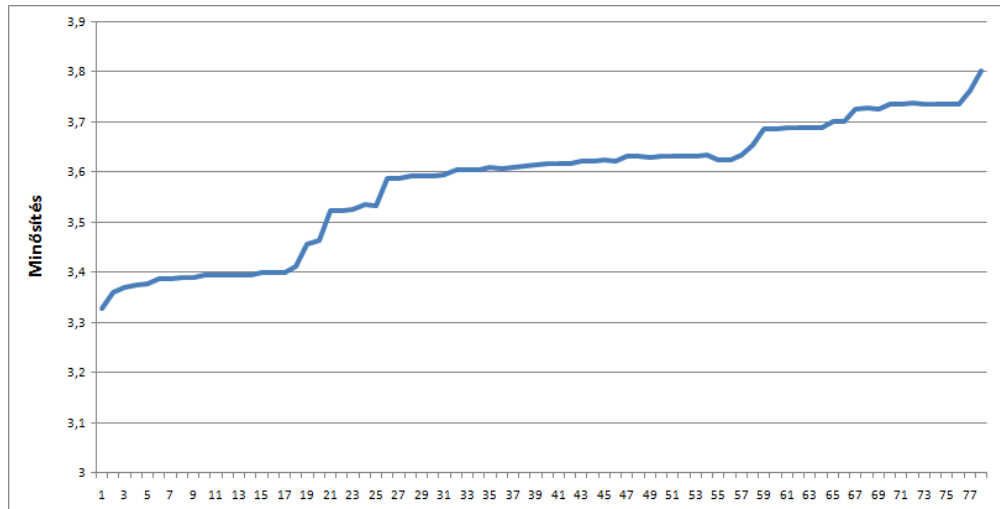
Szabálysértés	Becsült minőségjavulás	Mért minőségjavulás
PMD_OBEAH	0.169014336	3
PMD_PCI	-0.021941623	3
PMD_PLFIC	0.000990774	2
PMD_SSSHD	-0.048011665	2
Nem PMD (metrika, klón)	0.001493855	2

5.4. táblázat. Módosítások becsült és mért minőségjavulásának összevetése

A táblázatban két negatív érték is szerepel. Ezek oka részben a szekció bevezetőjében említett okok lehetnek (ilyen például, hogy a fejlesztők refaktoringon kívül mást is kommitáltak a szabálysért javításával), továbbá előfordulhat, hogy a fejlesztők figyelmetlenségéből adódóan új szabálysértéseket csináltak, amikor javították az adott szabálysértéseket. Valamint az is elképzelhető, hogy a végrehajtott kevés refaktoring nem volt elegendő ahhoz, hogy ezen szabálysértések minőségjavító hatása kimutatható legyen.

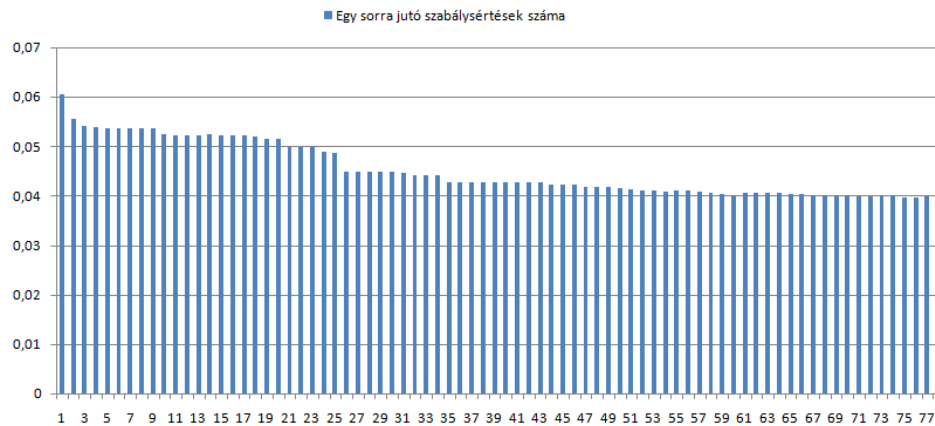
5.3.3. Az elvégzett refaktoringok teljes rendszer minőségére gyakorolt hatása

Az 5.6. ábrán láthatjuk, hogy a rendszer minősítése hogyan változott a refaktoring időszak múlásával. Látható, hogy a refaktoring időszak itt is célravezető volt, hiszen a rendszer karbantarthatósága 3,32-ről 3,80-ra módosult, ami körülbelül 14,5 %-os javulást jelent a forráskód minőségében. Azonban láthatjuk, hogy a rendszer minősítése itt sem mindig javult. De természetesen, ahogy az



5.6. ábra. Revíziónkénti minőségértékek

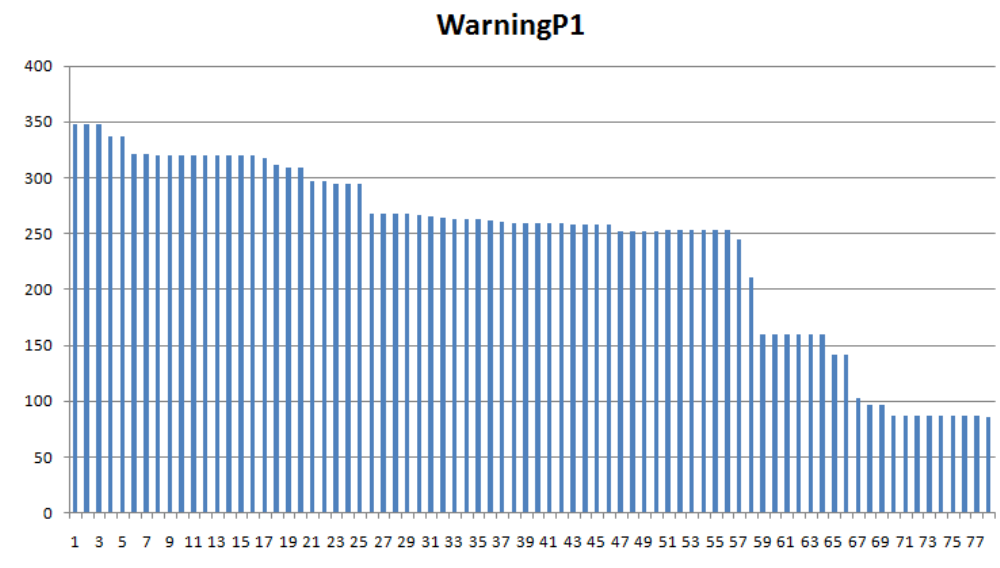
„A” rendszerénél is láttuk, előfordulhat, hogy egy-egy refactoring időlegesen ront is kicsit a minőségen, ám a műveleteket tömegesen alkalmazva ez már nem mondható el.



5.7. ábra. Szabálysértések száma soronként

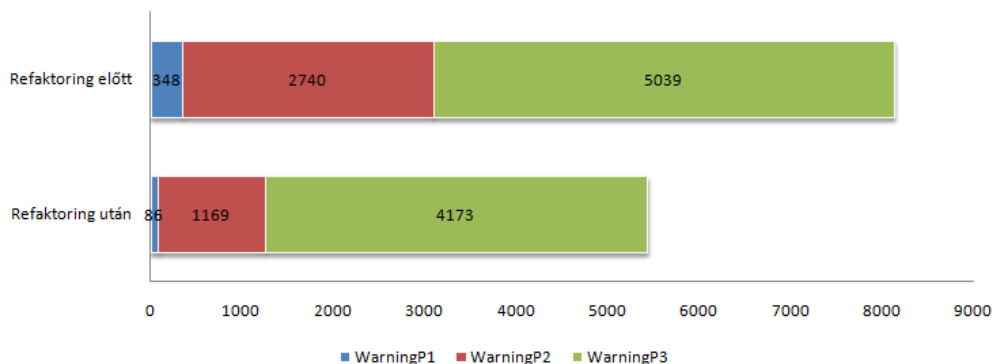
Az 5.7. ábrán láthatjuk, hogy egy kódsorra átlagosan hány szabálysértés jut. Ezt az alábbi módon számoltam ki: $EgySorraJutoWarningokSzama(Rev) = \frac{WarningAll(Rev)}{TLLOC(Rev)}$ Ahol a Rev az adott revíziót jelöli, míg a $WarningAll$ az összes szabálysértés ($WarningP1 + WarningP2 + WarningP3$ metrika) darabszáma, a $TLLOC$ pedig a *Total Logical Lines Of Code* metrikát jelenti. Itt is láthatjuk, hogy a kezdeti, kiemelkedően magas érték után, a szabálysértések száma legtöbbször csökken. Természetesen itt is lehetnek kisebb romlások, amelyek - a már említett - fejlesztések mi-

att történhettek. Amennyiben a köztes adatokat nem vizsgáljuk, csak a refaktoring időszak előtti, és utáni adatokat vizsgáljuk, jelentős csökkenést láthatunk, ugyanis 0,0606-ről 0,04010-re csökkent.



5.8. ábra. P1-es prioritású (súlyos) szabálysértések száma

Az 5.8. ábrán láthatjuk, hogyan változott a rendszerben lévő súlyos prioritású szabálysértések száma. Eltekintve az időleges, kisebb romlásoktól itt is azt láthatjuk, hogy a szabálysértések száma folyamatosan csökken. A kezdeti 348 darab szabálysértésből a refaktoring időszak végére mindösszesen 86 darab szabálysértés maradt. Ez azt jelenti, hogy a P1-es prioritású szabálysértések száma közel a negyedére csökkent, ami hatalmas javulásnak mondható egy ekkora rendszer esetén.



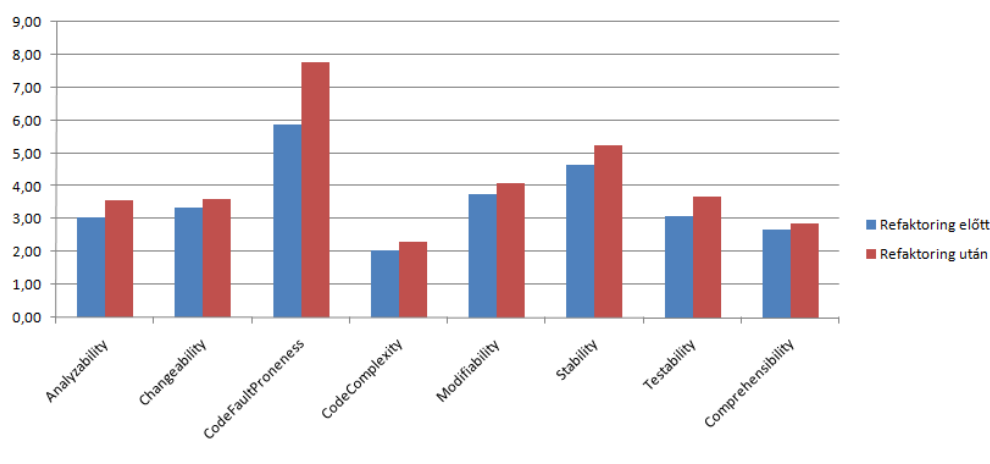
5.9. ábra. Szabálysértések számának változása

Itt szeretném megjegyezni, hogy a szabálysértések javításakor elsőbbséget élveztek a P1-es prioritású szabálysértések, de természetesen P2-es, P3-as szabálysértésekből is rengeteget javítottak a fejlesztők. Konkrétan ez azt jelenti, hogy a kezdeti 2740 P2-es prioritású szabálysértésből 1571 darabot, míg a kezdeti 5039 P3-as prioritású szabálysértésekből 866 darabot javítottak a refaktoring időszak folyamán.

Ahogy az 5.9. ábrán is látható, minden típusú szabálysértésből refaktoráltak, ugyanakkor a P1-es prioritású szabálysértések szinte eltűntek a rendszerből, és a P2-es prioritású szabálysértések száma is majdnem a felére csökkent, valamint jó néhány P3-as prioritású szabálysértés is kikerült a rendszerből.

Ezen adatok és ábrák mind alátámasztják azt a feltételezést, hogy a refaktoring időszak alatt a teljes rendszer minőségében kimutatható a változás, amely pozitívan befolyásolja a karbantartathóságot, és egyéb minőségmutatókat.

5.3.4. Magas szintű minőségmutatók változása a refaktoring során



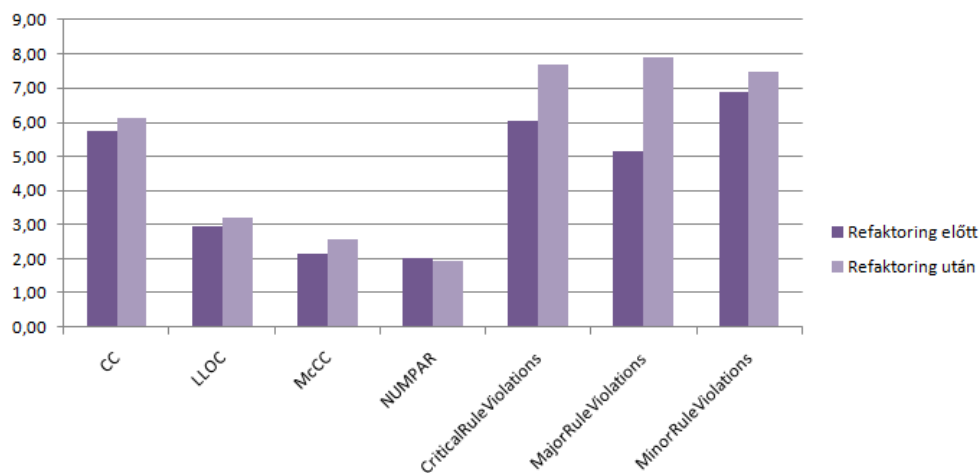
5.10. ábra. Magas szintű mutatók változása

Az 5.10. ábrán láthatóak az *ISO/IEC 25010* [5] szabvány által definiált magas szintű mutatók értékei a refaktoring időszak előtt, valamint a refaktoring időszak után. Ezeket a magas szintű mutatókat már korábban ismertettük.

Az ábrán látható összes minőségmutatót pozitívan befolyásolta a refaktoring időszak, mindegyik minőségmutató értéke növekedett. Azonban a CodeFaultProneness (Hibatűrés) minőségmu-

tató kimagaslóan sokat javult. Ez köszönhető annak is, hogy a fejlesztők az új fejlesztések során már próbáltak minél kevesebb új szabálysértést létrehozni, valamint természetesen annak, hogy a refaktoring időszak folyamán rengeteg különböző prioritású szabálysértést javítottak ki (előző szekció), amelyek erre minőségmutatóra vannak közvetlen hatással (2.1. ábra).

5.3.5. Alacsony szintű minőségmutatók változása a refaktoring során



5.11. ábra. Alacsony szintű mutatók változása

Az alacsony szintű minőségmutatók – mint említettem – a forráskódból kiolvasható számadataiból képzett minőségi jellemzők. Néhány ilyen jellemző értéke látható refaktoring előtt, illetve után, az 5.11. ábrán.

Jól látszik, hogy a legjelentősebb mértékben a szabálysértések reprezentáló Critical Rule Violations (Kritikus szabálysértések), Major Rule Violations (Fontos szabálysértések), valamint Minor Rule Violations (E) mutatók értékei nőttek. Ennek triviális oka, hogy a fejlesztők rengeteg szabálysértést (konkrétan definiáltakat) javítottak ki, melyek így a rendszerből eltűntek.

6. fejezet

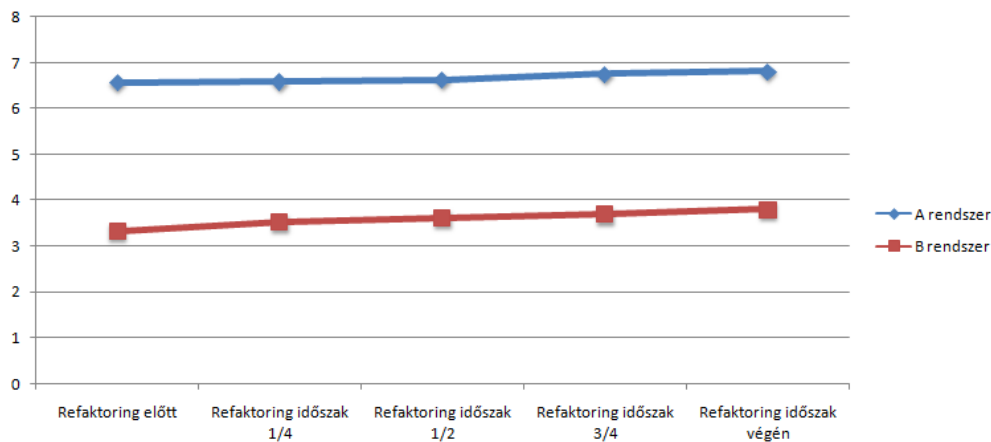
Konklúzió és jövőbeli tervek

Dolgozatomban kódolási szabálysértések refaktoringját vizsgáltam, és azt, hogy ezen szabálysértések javítása hogyan befolyásolja a kódminőséget. Ehhez egy több cég részvételével zajló refaktoring projekt során mért adatok álltak a rendelkezésemre. Dolgozatomban két ipari rendszer refaktoring időszakát tanulmányoztam. Több, mint 200 refaktoring commit állt rendelkezésemre, egy kb. 340 ezer kódsoros Java ERP, valamint egy kb 170 ezer kódsoros J2EE technológián alapuló keretrendszerből. Mindkét rendszert évek óta aktívan fejlesztik, és mindkét rendszernek aktív felhasználatára van.

A dolgozat bevezetőjében több kutatási kérdést tettem fel, amelyekre a korábbi fejezetekben adtam válaszokat. Láthattuk, hogy egy-egy refaktoring művelet mennyire befolyásolja a kódminőség alakulását, valamint, hogy a refaktoring hatása kimutatható-e egyáltalán, és azt is, hogy kimutatható-e a valószínűségi minőségmutató alkalmazásával, és ezekből milyen következtetéseket vonhatunk le.

Mindkét rendszer esetében láthattuk, hogy a becslések nem minden esetben helytállóak, de azért az mindkét esetben látható, hogy a komolyabb módosítást érintő problémák (Long Function szabálysértés, klónok kiszervezése) kivitelezéséhez több kódsor módosítása szükséges, mint az egyszerűbb, gyorsabban javítható szabálysértések esetében (ilyenek pl.: Position Literals First In Comparisons). Mindazonáltal általánosságban igaz, hogy a nagyobb javítási időre becsült szabálysértések esetében valóban több kódsor is módosult.

Ahogy láttuk mindkét rendszer esetében, hogy egy-egy szabálysértés javítása (függetlenül a típusától) csak nagyon kis mértékben befolyásolja a forráskód minőségét. Azonban, ha ezeket a



6.1. ábra. Minőségváltozás a refactoring időszak folyamán

refactoringokat tömegesen alkalmazzuk, akkor látható, hogy a forráskód minősége javul. Mindkét rendszer refactoring időszakát 4 részre osztottam, majd ezekben az időszakokban megnéztem a karbantarthatósági mutatót. Láthatjuk, hogy az idő elteltével a forráskód minősége egyre jobb értékeket vesz fel. Ezt a változást szemlélteti a 6.1. ábra, melyen mindkét rendszer minőségváltozását láthatjuk.

Általános konklúzióként elmondhatjuk, hogy az általam vizsgált adathalmazból egyértelműen kiderül, hogy a kódolási szabálysértések refaktorálása a forráskód minőségét hogyan befolyásolja, és ezt a hatást kimutathatjuk a probabilisztikus minőségmodell segítségével. Habár egy-egy refactoring végrehajtása a minőséget csak nagyon kis mértékben befolyásolja, tömegesen alkalmazva már egyértelműen látható a pozitív irányú a változás, s ez nem csak adott osztályon, metóduson belül vehető észre, hanem a teljes rendszer egészét tekintve is.

Jövőbeli terveim között szerepel a módszer alkalmazása további ipari szoftverek refactoring időszakáról, ahol több, különböző refactoring műveletet is vizsgálhatunk. Emellett, a fejlesztők becslést tettek az érintett fájlok, metódusok, osztályok számára is. Ezek az adatok ebben a dolgozatban nem kerültek feldolgozásra, mivel ezek feldolgozásához szükség lenne egy olyan programra, amely a verziókövető adatai alapján képes visszakeresni egy-egy sor tartalmazó metódusát, osztályát is. Egy ilyen eszköz segítségével lehetőség nyílna a fent említett adatok feldolgozására, valamint összehasonlítására a fejlesztői becslésekkel. Ám jelenleg nem létezik ilyen forráskód-visszafejtő eszköz, így ezen adatok feldolgozása még várat magára.

A. Függelék

Függelék

A.1. A „B” rendszerben lévő szabálysértések hatása a minőségre

Szabályértés	Szabálysér- tések száma	Mért minőség- változás	Becsült javulás	Becsült idő a javításra	Átlagosan módo- sult kódsorok
PMD_ACNPE	1	0.006295869	3	1-2	68
PMD_ACT	1	0	2	1-2	15
PMD_ADL	1	0.055150355	3	1-2	4037
PMD_{ ADL ECB SSSHD }	1	0.000720343	2	0-1	53
PMD_{ ADNIS ALOC CC CIS CT ECB }	1	0.060985047	3	3-8	4180
PMD_AES	1	0.000676777	3	1-2	76
PMD_AICICC	1	0.001735073	2	1-2	68
PMD_{ AISD MRJA }	1	0.006482705	4	3-8	662
PMD_ALOC	1	0.005727039	2	1-2	293
PMD_ATRET	1	0.001447104	2	0-1	25
PMD_{ CC CT LoC NMC NPC }	1	0.010422994	3	8-16	208
PMD_{ CC LC LoC }	1	0.002646725	4	3-8	133
PMD_CCOM	1	0.010914621	2	1-2	109
PMD_CLA	1	0.000187529	2	1-2	31
PMD_ECB	1	0.120150729	3	0-1	60
PMD_{ LC LoC }	1	0.000526858	3	1-2	53
PMD_{ LC LoC RVWL UALIOV }	1	0.031824911	3	8-16	2721
PMD_OBEAH	2	0.169014336	3	0-1	82
PMD_PCI	5	-0.021941623	3	1-2	66
PMD_PLFIC	4	0.000990774	2	0.75-1.75	23
PMD_SMN	1	0.02514288	2	1-2	359
PMD_SSSHD	2	-0.048011665	2	1-2	214
PMD_UFP	1	-0.01635989	3	1-2	157
PMD_UIOC	1	0	2	1-2	16
PMD_{ ULBR ULV }	2	0.00003291	3	1-2	36
PMD_ULV	1	0.009461304	2	1-2	1479
PMD_ULWCC	1	0.031355741	2	3	
PMD_UnI	1	0	2	1-2	12
PMD_UPF	1	0.004591189	1	1-2	403
PMD_USBFA	1	0.00163073	2	1-2	229
PMD_USDF	1	0.008044125	3	1-2	38
PMD_UWOC	1	0.009756997	2	1-2	304
Nem PMD (metrika. klón)	13	0.001493855	2	0.5-2	378

A.1. táblázat. A B rendszerben javított szabályértések

Irodalomjegyzék

- [1] Apache Subversion – <http://subversion.apache.org/>.
- [2] Eclipse Foundation – <http://www.eclipse.org/>.
- [3] IntelliJ IDEA – <http://www.jetbrains.com/idea/>.
- [4] ISO/IEC 25000:2005 – Software product Quality Requirements and Evaluation (SQuaRE).
- [5] ISO/IEC 25010:2011 - Systems and software engineering – Systems and software Quality Requirements and evaluation (SQuaRE).
- [6] Microsoft Visual Studio – <http://www.visualstudio.com/>.
- [7] NetBeans – <https://netbeans.org/>.
- [8] PMD – <http://pmd.sourceforge.net/>.
- [9] QualityGate – „The ultimate software quality management solution” – <http://qualitygate.com/>.
- [10] SQALE – Software Quality assessment based on Lifecycle Expectations – <http://www.sqale.org/>.
- [11] Trac – <http://trac.edgewall.org/>.
- [12] WebStorm – <http://www.jetbrains.com/webstorm/>.
- [13] Tibor Bakota, Péter Hegedűs, Péter Körtvélyesi, Rudolf Ferenc, and Tibor Gyimothy. A probabilistic software quality model. In *Proceedings of the 2011 27th IEEE International*

- Conference on Software Maintenance*, ICSM '11, pages 243–252. IEEE Computer Society, 2011.
- [14] William J. Brown, Raphael C. Malveau, Hays W. McCormick, III, and Thomas J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons, Inc., New York, NY, USA, 1998.
 - [15] Serge Demeyer. Refactor conditionals into polymorphism: what's the performance cost of introducing virtual calls? In *Proceedings of the 21st IEEE International Conference on Software Maintenance, 2005. ICSM'05.*, pages 627–630. IEEE, 2005.
 - [16] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
 - [17] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Trans. Softw. Eng.*, 30(2):126–139, February 2004.
 - [18] William F Opdyke. *Refactoring object-oriented frameworks*. PhD thesis, University of Illinois, 1992.
 - [19] Houari A Sahraoui, Robert Godin, and Thierry Miceli. Can metrics help to bridge the gap between the improvement of oo design quality and its automation? In *Proceedings of International Conference on Software Maintenance*, pages 154–162. IEEE, 2000.
 - [20] Konstantinos Stroggylos and Diomidis Spinellis. Refactoring—does it improve software quality? In *Proceedings of the 5th International Workshop on Software Quality, WoSQ '07*, pages 10–, Washington, DC, USA, 2007. IEEE Computer Society.
 - [21] Eleni Stroulia and Rohit Kapoor. Metrics of refactoring-based development: An experience report. In *OOTS 2001*, pages 113–122. Springer, 2001.
 - [22] Gábor Szőke, Gábor Antal, Csaba Nagy, Rudolf Ferenc, and Tibor Gyimóthy. Bulk fixing coding issues and its effects on software quality: Is it worth refactoring? In *Proceedings of the 14th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2014)*, SCAM 2014, Victoria, British Columbia, Canada, 2014. IEEE Computer Society.