Szoftver termék metrikák alkalmazása a szoftverkarbantartás területén

Siket István

SZTE Szoftverfejlesztés Tanszék

Szeged, 2010.

Témavezető:

Dr. Gyimóthy Tibor

ÉRTEKEZÉS DOKTORI FOKOZAT

MEGSZERZÉSÉHEZ

Szegedi Tudományegyetem

Informatika Doktori Iskola

Előszó

Az értekezésről

Az értekezés célja az objektum-orientált metrikák átfogó vizsgálata. Ennek részeként az objektum-orientált metrikák kiszámításának bemutatása után a metrikák hibaelőrejelző képességét vizsgáltuk. Ezt először néhány metrikára, majd metrika-kategóriákra végeztük el. A bemutatott vizsgálatok alapján meghatározhatjuk azokat a metrikákat illetve metrika-kategóriákat, amelyek a szoftverfejlesztés során a legalkalmasabbak a szoftver minőségének jellemzésére. A módszer jelentősége nem csak abban rejlett, hogy ezeket a metrikákat meghatároztuk, hiszen számos korábbi kísérlet hasonló eredményt adott, hanem hogy ipari méretű rendszerek esetében korábban senki sem igazolt hasonló összefüggéseket.

Ahhoz, hogy a metrikákat a mindennapi fejlesztésben is hatékonyan használhassuk, nem csak eszközt kell erre biztosítanunk, de ismernünk kell azt, hogy egy mai szoftverfejlesztő hogyan képes felhasználni a metrikák által nyújtott információkat a munkájában. Ennek a kérdéskörnek a vizsgálatával zárjuk az értekezést.

Köszönetnyilvánítás

Először is szeretnék köszönetet mondani témavezetőmnek, Gyimóthy Tibornak, aki megismertetett ezzel az érdekes témával, és hosszú évek óta irányítja a kutatásaimat. Továbbá szeretnék külön köszönetet mondani Ferenc Rudolfnak, Fülöp Lajosnak, Jász Juditnak, Siket Péternek, Sógor Zoltánnak, valamint Szokody Fedornak a sok segítségért és támogatásért, amit a munkám során tőlük kaptam.

Táblázatok jegyzéke

Ábrák jegyzéke

Tartalomjegyzék

1. fejezet

Bevezetés

Napjainkban a szoftverekkel szembeni elvárások egyre nagyobb kihívást jelentenek a szoftveripar számára. Egyre nagyobb és komplexebb rendszereket kell kifejleszteni rövid idő alatt, majd az elkészült termékeket karban kell tartani, miközben a felhasználók gyorsan változó igényeit is ki kell elégíteni. Ebből következik, hogy a szoftver „átadása" nem azonos a szoftverfejlesztési folyamat végével, csak egy mérföldkő a folyamatban, amely után az adott szoftver karbantartása és továbbfejlesztése következik. Így míg a felhasználók számára a szoftver *megbízhatósága*, *használhatósága* vagy éppen a *hordozhatósága* a legfontosabb szempont, addig a fejlesztés és a költségek szempontjából figyelembe kell vennünk az adott szoftver *karbantarthatóságát* is (ezen minőségi jellemzők leírása megtalálható az ISO/IEC 9126-1 szabványban [34]). Egy hiba megtalálásának és javításának, vagy egy új funkció hozzáadásának a költsége nagy mértékben függ a rendszer karbantarthatóságától. Ez azt jelenti, hogy az „olcsó" megoldások csak rövid távon számítanak olcsónak, ha hosszabb távon a megtakarításnak a többszörösét kell visszafizetnünk. Ezért a többi minőségi jellemző mellett szükséges a karbantarthatóság folyamatos mérése és nyomon követése is a szoftverfejlesztés teljes életciklusa alatt. Ennek segítségével időben felfedezhetjük azt, ha nem a megfelelő irányba változik a szoftver minősége, valamint használhatjuk arra is, hogy becslést adjunk a változások költségére.

A szoftver karbantartás egyik fontos része a szoftverben található hibák felfedezése és azok kijavítása. Mivel a tesztelés során lehetetlen az összes hibát megtalálni, ezért a tesztelés célja az, hogy minél több hibát találjunk meg minél kevesebb költséggel. Bármilyen segítség, ami növeli a tesztelés hatékonyságát egyben növeli a szoftver minőségét is, illetve csökkenti a költségeket. A szoftver minőség növelésére egyrészt lehetőség nyílik a szoftverfejlesztési folyamat javításával, ami az úgynevezett *szoftver folyamat metrikák* ellenőrzésével valósulhat meg. Másik lehetőség az úgynevezett *szoftver termék metrikák* folyamatos ellenőrzésén és javításain alapuló szoftver minőség növelő módszerek. Az értekezésben az utóbbi módszerekkel foglalkozunk, vagyis a szoftver termék metrikákon alapuló minőség ellenőrző és javító modelleket mutatunk be. Az értekezésben igazoljuk, hogy az objektum-orientált metrikák és az osztályok hibára való hajlamossága között létezik kapcsolat. Így az objektum-orientált metrikák segítségével növelhetjük a tesztelés hatékonyságát, illetve az objektum-orientált metrikák folyamatos mérésével és vizsgálatával javíthatjuk a szoftver minőségét és karbantarthatóságát.

1.1. Az objektum-orientált metrikák áttekintése

A metrikák lényege, hogy a szoftver (vagy a szoftver egyes részeinek) bizonyos tulajdonságait numerikus formában fejezik ki. A legismertebb példa a metrikákra a *sorok száma* (Lines of Code, vagy röviden LOC) metrika, ami az adott elem kódsorainak a számát adja meg. Ennek a metrikának az előnye, hogy bármely programozási nyelvre definiálható, komolyabb programozói ismeret nélkül is megérthető, és a kiszámítása is könnyű, vagy legalábbis könnyebb, mint az értekezésben bemutatásra kerülő metrikák legtöbbje. Ezen tulajdonságoknak köszönhető, hogy a LOC metrika széles körben elterjedt, és számos területen alkalmazzák.

Annak ellenére, hogy a LOC metrika további változatait is definiálták (például az LLOC metrikát, amelynél csak a nem üres és nem komment sorok számítanak), ez a metrika még így is túl „egyszerűnek" bizonyult, amelyet bizonyos területen nem, vagy csak korlátozottan tudtak használni, ezért újabb metrikákat definiáltak. Például a Halstead féle komplexitás metrikát [29], amely a rendszer komplexitását méri, vagy a McCabe-féle ciklomatikus komplexitást [40], amely szintén a program komplexitását méri a program vezérlési folyam gráfját használva. A McCabe-féle ciklomatikus komplexitás felülről korlátozza az elágazás szintű lefedettséghez szükséges tesztesetek számát, és alulról korlátozza az útvonal szintű lefedettséghez szükséges tesztesetek számát. Ebből következik, hogy sokkal alkalmasabb a tesztelési erőforrások becslésére, mint a LOC metrika.

Az objektum-orientált nyelvek megjelenésével egyidőben szükségessé vált újabb metrikák definiálása is, amelyek a programok objektum-orientált tulajdonságait mérik. Több próbálkozás is volt olyan metrikák definiálására, amelyek a rendszer objektum-orientáltságát mérik, azonban nem mind volt sikeres. Az első jelentősebb munka Chidamber és Kemerer [12] nevéhez fűződik, akik hat objektum-orientált metrikát definiáltak. Ezek a metrikák az osztályok komplexitását, kohézióját és öröklődését, valamint az osztályok közötti csatolást mérik (a metrikák bővebb leírásai megtalálhatók a 3. fejezetben). Folytathatnánk a sort például a Brito e Abreu és Melo által definiált hat MOOD (Method for Object-Oriented Design) metrikával [17], azonban nem célunk az összes azóta publikált metrikát ismertetni [3, 38]. Helyette ismertetünk néhány munkát, amelyekben az objektum-orientált metrikák és az osztályokban található hibák kapcsolatát vizsgálták.

Basili és munkatársai [4] a Chidamber és Kemerer által definiált hat metrika és az osztályokban található hibák száma közti kapcsolatot vizsgálták nyolc kis, illetve közepes méretű rendszeren. Öt esetben találtak szignifikáns kapcsolatot a metrika és a hibák száma között, és csak egy metrika bizonyult alkalmatlannak a hibák előrejelzésére.

A Basili és munkatársai által vizsgált rendszert használta Fioravanti és Nesi [24] is. A kísérletükben 226 metrikát vizsgáltak, és azt tűzték ki célul, hogy egy olyan modellt alakítsanak ki, amelynek a pontossága legalább 90% legyen (azaz az osztályok legalább 90%-át helyesen osztályozzák). A kitűzött célt elérték, mert 42 metrika felhasználásával 97,35%-os pontosságú modellt alakítottak ki. A metrikák számának redukálásával egy 12 metrikából álló modellt kaptak, amelynek a pontossága még mindig 84,96% volt[1].

Olague és munkatársai [30] a Rhino projekt[2] [45] hat verzióján vizsgálták a Chi-

[1] Valószínű ez az igen magas pontosság azért volt elérhető, mert egy kis méretű, kevés osztályt tartalmazó rendszert vizsgáltak.

[2] A Rhino a JavaScript Java nyelvű, nyílt forráskódú implementációja.

damber és Kemerer metrikák [12], a Brito e Abreu féle MOOD metrikák [17] és a Bansiya and Davis féle QMOOD metrikák [3] hiba-előrejelző képességeit. Azt tapasztalták, hogy a hat Chidamber és Kemerer által definiált metrika bizonyult a legjobb prediktornak, de a QMOOD metrikák sem sokkal maradnak el mögöttük, míg a MOOD metrikákat nem találták hasznosnak.

A 4.2. alfejezetben részletesen foglalkozunk az itt röviden bemutatott metrikavizsgálatokkal, valamint további munkákat is megemlítünk majd [49, 56].

1.2. Az értekezés eredményei

Az objektum-orientált metrikák vizsgálatával foglalkozik az értekezés, amely a metrikák előállításával kezdődik, és a gyakorlati alkalmazhatóságuk feltételeinek vizsgálatával zárul. Első lépésként a metrikák előállításának nehézségeit vizsgáljuk, majd egy megoldást mutatunk, aminek a segítségével automatikusan és hatékonyan ki tudjuk számolni a metrikákat ipari méretű rendszerek esetében is. Ez elengedhetetlen a további vizsgálatok szempontjából, valamint szükséges a gyakorlati alkalmazhatóságuk miatt is.

Ezután a Mozilla[3] [41, 44] esetében különböző módszerek segítségével megvizsgáljuk, hogy a Chidamber és Kemerer által definiált metrikák és az LLOC metrika mennyire alkalmas a hibák előrejelzésére, azaz mennyire alkalmas a szoftver minőségének mérésére. Mivel az eredmények bíztatóak voltak, ezért a kísérletet kiterjesztettük úgy, hogy a metrika-kategóriák hiba-előrejelző képességeit vizsgáltuk. Mindkét esetben bebizonyosodott, hogy bizonyos metrikák, illetve metrika-kategóriák alkalmasak a hibák előrejelzésére, így alkalmasak lehetnek a szoftverfejlesztés segítésére.

Végezetül a szoftverfejlesztők metrikákról kialakult véleményét vizsgáltuk, amely elengedhetetlen a metrikák gyakorlati alkalmazásának bevezetése előtt. A vizsgálatok eredményei azt mutatták, hogy nem elegendő a fejlesztők kezébe adni az „eszközt" (jelen esetben a metrikákat), meg is kell őket tanítani a helyes használatára. Továbbá ezeket az eredményeket felhasználhatjuk arra is, hogy a kialakított hiba-előrejelző modelljeinket javítsuk a fejlesztők véleményének figyelembe vételével.

Az alábbiakban ismertetjük az értekezés eredményeit tézisekbe foglalva.

I. **C++ rendszerek elemzésének problémái és megoldásai, egy nyelvfüggetlen modell kidolgozása objektum-orientált rendszerek magas szintű ábrázolására, és az objektum-orientált metrikák kiszámítása ennek segítségével. Heurisztika kidolgozása a kijavított hibák osztályokhoz rendelésére.**

   A C++ nyelven íródott rendszerek nagyon bonyolultak és összetettek lehetnek, melyeknek már a fordítása sem egyszerű feladat. A rendszerek megvizsgálásával, és a fordítási folyamat tanulmányozásával kidolgoztunk egy olyan általános módszert, aminek a segítségével képesek vagyunk tetszőleges C++ rendszert elemezni anélkül, hogy bármit változtatnánk rajta. A kidolgozott technológia Windows és Linux operációs rendszeren egyaránt működik, valamint a különböző fordítóprogramok (például Microsoft Visual Studio vagy GCC) sem okoznak problémát. A technológia erejét mutatja, hogy számos nyílt forráskódú (például Mozilla [41] vagy OpenOffice.org [42]), illetve több komplex ipari rendszer esetében alkalmaztuk sikeresen. A legnagyobb elemzett rendszer 30 millió programsorból állt.

[3] A Mozilla egy C/C++ nyelven írt, nyílt forráskódú web böngésző és levelező alkalmazás.

Kidolgoztunk egy nyelvfüggetlen modellt, amely alkalmas egy objektum-orientált rendszer magas szintű ábrázolására, valamint megvalósítottuk a C++, Java és C# nyelv ábrázolásának konverzióját erre a magas szintű nyelvfüggetlen reprezentációra. Ezen nyelvfüggetlen modellt használjuk arra, hogy tetszőleges C++, Java és C# rendszer esetében kiszámoljuk az objektum-orientált metrikákat. Jelenleg 16 rendszer szintű, 65 osztály szintű és 17 metódus szintű metrikát tudunk meghatározni.

Kidolgoztunk továbbá egy heurisztikát, melynek segítségével képesek vagyunk egy adott szoftver hibakövető rendszeréből automatikusan összegyűjteni a bejelentett és kijavított hibákat, és azokat a forráskódelemekhez (például osztályokhoz) rendelni. Így ezen technológia segítségével egyszerre állnak rendelkezésünkre az adott osztályok metrika értékei, valamint a bennük megtalált és kijavított hibák száma.

A kidolgozott technológia bemutatása, valamint a Mozilla különböző verzióinak elemzése megtalálható a [21] publikációban. A [28] publikáció többek közt röviden összefoglalja a Mozilla elemzését, valamint bemutatja a hibák osztályokhoz rendelésére kidolgozott heurisztikát is.

II. **A Chidamber és Kemerer által definiált metrikák és a tradicionális sorok száma metrika felhasználása hiba-előrejelző modellek kialakítására.**

Korábban már számos tanulmány vizsgálta az objektum-orientált metrikák és a hibák száma (vagy a hibára való hajlamosság) közti kapcsolatokat, és több esetben is igazolták, hogy van összefüggés a metrikák és a szoftver minősége között. Azonban ezek a kutatások kis méretű, kevés osztályból álló rendszereken történtek, így nem bizonyították be, hogy valós projektek esetében hasonló eredményeket lehetne elérni a metrikák segítségével.

Az általunk kidolgozott módszer segítségével a Mozilla különböző verzióira kiszámoltuk a Chidamber és Kemerer által definiált metrikákat, valamint a tradicionális LLOC metrikát, és meghatároztuk az osztályokban talált és kijavított hibák számát. Az eredményeket felhasználva lineáris és logisztikus regresszió, valamint döntési fa és neuron háló segítségével vizsgáltuk az osztályok metrikái és a hibaszámai, valamint a hibára való hajlamosságuk közti összefüggéseket. A különböző módszerek közel azonos eredményt adtak, amellyel igazoltuk, hogy van kapcsolat a metrikák és a hibák száma között, azaz a metrikák alkalmasak a szoftver minőségének meghatározására. Az eredmények értékét növelte, hogy ezt az összefüggést az elsők között igazoltunk ipari méretű projektek esetében [28].

III. **Az objektum-orientált metrika-kategóriák hiba-előrejelző képességeinek vizsgálata.**

A korábbi kutatásunk folytatásaként a kísérletet kiterjesztettük, melyben már nem az egyes metrikák alkalmasságára koncentráltunk, hanem azt vizsgáltuk meg, hogy a különböző metrika-kategóriák mennyire alkalmasak az osztályok hibára való hajlamosságának előrejelzésére. A vizsgálatokhoz 58 metrikát használtunk, amelyeket öt kategóriába (méret, komplexitás, csatolás, öröklődés és kohézió) soroltunk. A kategóriák hiba-előrejelző képességeit a kategóriákba sorolt metrikák hiba-előrejelző képességei alapján határoztuk meg.

Az első vizsgálatok során bebizonyítottuk, hogy néhány metrika alkalmas a hibák előrejelzésére, míg mások nem. Az újabb vizsgálatokkal [47] általánosítottuk a korábbi eredményeket, így a különböző metrika-kategóriák hiba-előrejelző képességeit is felhasználhatjuk a szoftver karbantartásban.

IV. **A szoftverfejlesztők metrikákkal kapcsolatos ismereteinek felmérése, és a minőségi modellek javítása a tapasztalatok figyelembe vételével.**

Kísérletekkel igazoltuk, hogy van kapcsolat az objektum-orientált metrikák és a hibák száma között, így a metrikák megfelelő gyakorlati alkalmazása javíthatja a szoftver minőségét. Bemutattuk az általunk kidolgozott technológiát a metrikák kiszámítására, amit sikeresen alkalmaztuk ipari méretű projektek esetében is. Továbbá kifejlesztettünk két olyan eszközt is, amelyek a bemutatott technológiára épülnek, de grafikus felülettel rendelkeznek, és így jelentős támogatást nyújtanak mind a metrikák előállításához, mind az eredmények hatékony kezeléséhez. Így azt gondolhatnánk, hogy minden adott ahhoz, hogy a fejlesztők munkáját segítsük a metrikák gyakorlati bevezetésével, és javítsuk a szoftver minőségét. Mielőtt ezt megtehetnénk, meg kell vizsgálnunk, hogy a szoftverfejlesztők mennyire ismerik a metrikákat, és vajon a véleményük megegyezik-e azzal, amit a kísérleteink során kaptunk. Ha a fejlesztők nincsenek tisztában azzal, hogy a metrikákat hogyan kell hatékonyan alkalmazni, akkor a metrikák bevezetése csak hátráltatja a fejlesztést ahelyett, hogy javítaná a szoftver minőségét.

Ezért készítettünk egy felmérést [48], amiben a szoftverfejlesztők véleményét vizsgáltuk aszerint, hogy adott három objektum-orientált metrika és egy kód duplikációkkal kapcsolatos metrika milyen kapcsolatban áll a program megértéssel és teszteléssel. Kérdéseket tettünk fel, amelyekkel azt vizsgáltuk, hogy a résztvevők hogyan alkalmaznák a metrikákat különböző konkrét gyakorlati problémák megoldásában. Így egy átfogó képet kaptunk arról, hogy szerintük mely területeken lehet hatékonyan alkalmazni a metrikákat, és melyeken nem. Emellett statisztikai módszerekkel vizsgáltuk a különböző tapasztalattal rendelkező résztvevők válaszai közti eltéréseket, és számos esetben szignifikáns különbséget találtunk, ami azt jelenti, hogy a különböző területeken szerzett tapasztalat jelentősen befolyásolja a metrikákról kialakult véleményt.

Ezzel a kísérlettel rávilágítottunk arra, hogy nem elegendő a fejlesztők kezébe adni az eszközt, meg is kell tanítani őket az eszköz megfelelő használatára, mert hiába áll rendelkezésükre, ha nem tudják azt megfelelően használni.

Az eredmények rámutattak arra is, hogy nem minden esetben lehet a metrika értékek alapján megítélni a kód minőségét. Például a fejlesztők nagyobb része gondolta úgy, hogy a generált kód esetében a rossz metrika értékek nem jelentik azt, hogy a kód minősége is rossz lenne. Ezen eredmények felhasználásával a minőségi modelljeink pontossága is javítható.

Az áttekinthetőség kedvéért a következő táblázat összefoglalja a kapcsolódó publikációk és a tézisek kapcsolatát:

	[21]	[28]	[47]	[48]
I.	•	•		
II.		•		
III.			•	
IV.				•

Megjegyezzük, hogy a disszertációban ismertetett téma iránt nagy nemzetközi érdeklődés mutatkozik, amit az is jelez, hogy a [21] cikkre a dolgozat beadásáig 34 független hivatkozás, míg a [28] cikkre már több mint 100 független hivatkozás történt.

1.3. Az értekezés felépítése

Az értekezés az objektum-orientált metrikák kiszámításával (2. fejezet), azok hiba-előrejelző képességeinek vizsgálatával (3. és 4. fejezet), gyakorlati alkalmazásával (5. fejezet) és a fejlesztők metrikákról kialakult véleményének megismerésével (6. fejezet) foglalkozik.

A Columbus technológia bemutatása

Az értekezés az objektum-orientált metrikák kiszámításával és az osztályokban található hibák számának meghatározásával kezdődik. A 2. fejezetben ismertettük azokat a problémákat, amelyekkel akkor találkozhatunk, ha egy ipari méretű rendszerre szeretnénk meghatározni a metrika értékeket. Bemutattuk a Columbus technológiát, amellyel egyszerűen, az eredeti kód megváltoztatása nélkül képesek vagyunk tetszőleges C++ rendszert elemezni. Röviden ismertettük az általunk kialakított nyelvfüggetlen modellt, melynek segítségével ki tudtuk számolni az objektum-orientált metrikákat, illetve egy példát is mutattunk a metrikák kiszámítására.

Emellett ismertettük az általunk kidolgozott heurisztikát, melynek segítségével a vizsgált szoftverben megtalált és kijavított hibákat az osztályokhoz rendeltük (ami elengedhetetlen a metrikák hiba-előrejelző képességeinek vizsgálatához). A bemutatott technológia segítségével a Mozilla több verzióját elemeztük, és minden osztályra kiszámoltuk a metrika értékeket, valamint meghatároztuk az osztályokban található hibák számát.

A objektum-orientált metrikák vizsgálata

A 3. fejezetben a Chidamber és Kemerer által definiált hat objektum-orientált metrika, valamint a hagyományos LLOC metrika és az osztályokban található hibák (illetve az osztályok hibára való hajlamosága) közti kapcsolatot vizsgáltuk a Mozilla esetében. Először a metrikákat külön-külön vizsgáltuk különböző statisztikai és gépi tanulási módszerekkel, majd az eredményeket összehasonlítottuk, hogy megtudjuk, melyik metrika bizonyult a legjobbnak. Emellett megvizsgáltuk, hogy több metrika együttes használatával képesek vagyunk-e jobb modelleket kialakítani annál, mint amit a metrikák önmagukban adnak.

A 4. fejezetben a kísérlet folytatásaként 58 objektum-orientált metrikát vizsgáltunk meg, de a célunk nem az egyes metrikák validálása volt, hanem a metrika-kategóriák (méret, komplexitás, csatolás, öröklődés és kohézió) hiba-előrejelző képességeinek megismerése. Az 58 metrikát az 5 metrika-kategória valamelyikébe soroltuk, és a metrika-kategória minősítését a bennük található metrikák eredményei alapján határoztuk meg.

Az eredmények ismertetése után összehasonlítottuk azokat további, hasonló kísérletek eredményeivel, hogy kiderüljön, új eredményekről van-e szó, vagy a korábbi eredményeket erősítettük meg. Végezetül egy rövid értékelés következik, melyben elmagyaráztuk, hogy a kapott eredmények „mit jelentenek" és hogyan kell értelmezni azokat.

A metrikák gyakorlati alkalmazásai

Az 5. fejezetben bemutattunk két olyan eszközt, amelyek a Columbus technológiára épülnek, de a használatuk sokkal egyszerűbb, mint a 2. fejezetben bemutatott parancssori eszközöké. Emellett támogatást nyújtanak a metrikák „vizsgálatára" is, így segíthetik mind a fejlesztők, mind a menedzserek mindennapi munkáját. A fejezet végén megemlítettünk néhány példát is a metrikák ipari alkalmazásaira, melyek alátámasztották, hogy nem csak elméleti eredményekről beszéltünk, hanem az iparban is alkalmazható módszert mutattunk be.

A szoftverfejlesztők véleményének vizsgálata

A metrikák gyakorlati alkalmazásához elengedhetetlen, hogy megismerjük a szoftverfejlesztők metrikákról kialakult véleményét. Ezért a 6. fejezetben bemutattuk az általunk készített online kérdőívet, amit 50 szoftverfejlesztő töltött ki. Ennek segítségével megvizsgáltuk, hogy a résztvevők a három kiválasztott objektum-orientált metrikát és egy kód duplikáció metrikát hogyan alkalmaznák a program megértés és tesztelés segítésére. A metrikákat vizsgáltuk külön-külön is, de voltak olyan kérdések is, amelyekben a metrikákat kellett összehasonlítani. A kérdésekre adott válaszokból általános képet kaptunk a fejlesztők metrikákról kialakult nézeteiről, ami önmagában segítheti a metrika-alapú szakértői rendszerek kifejlesztését, illetve a metrika alapú hiba-előrejelző modellek javítását. Emellett megvizsgáltuk azt is, hogy a különböző területeken szerzett tapasztalat hogyan befolyásolja a metrikák gyakorlati alkalmazását. Számos olyan példát találtunk, ahol a tapasztalat szignifikánsan befolyásolta a metrikák alkalmazhatóságának megítélését.

Az eredmények összegzése

Az utolsó fejezetben az értekezés eredményeinek rövid összefoglalása található.

I. rész

A Columbus technológia

2. fejezet

A metrika értékek kiszámítása és a hibák osztályokhoz rendelése

Ahhoz, hogy vizsgálni tudjuk az objektum-orientált metrikák és a forráskód elemekben található hibák száma közötti kapcsolatot, először meg kell határoznunk ezeket az értékeket. Ebben a fejezetben bemutatjuk a Columbus technológiát, melynek segítségével tetszőleges C++ rendszert tudunk elemezni és a metrika értékeket kiszámolni. Ezután ismertetjük azt a heurisztikák, amelynek segítségével meghatároztuk a Mozilla osztályaiban található hibák számát.

2.1. A Columbus technológia

A metrika értékek kiszámításához először elemeznünk kell a vizsgált rendszert. Ez a feladat kis méretű és „egyszerű" programok esetében sem triviális, azonban nagy méretű ipari projektek vizsgálatakor komoly nehézségekbe ütközhetünk. Nehéz lenne az összes felmerülő problémát és azok megoldásait ismertetni, azonban néhányat megemlítünk, hogy szemléltessük, milyen jellegű nehézségeket kellett megoldanunk.

A programok forráskódja fájlokba van osztva, és ezek a fájlok könyvtárakba és alkönyvtárakba vannak szervezve. Az az információ, hogy ezek a fájlok hogyan kapcsolódnak egymáshoz és milyen további információ szükséges a fordításukhoz, általában *makefile*-okban vagy különböző projekt fájlokban van tárolva. Ezekben számos egyéb, a program fordításához szükséges információ is található (például különböző külső programok meghívása), amit szintén fel kellene dolgozni, ha valóban minden, a fordításhoz szükséges információt össze akarunk gyűjteni. Egy másik probléma lehet, hogy vannak olyan forrásfájlok, amik nincsenek eltárolva a verziókövető rendszerekben, csak fordítás közben generálódnak (például IDL-ből vagy különböző nyelvtan fájlokból), és csak a generálásukhoz szükséges információ áll rendelkezésre. Az ilyen fájlok elemzése, illetve a kigenerált fájlokat használó további fájlok elemzése lehetetlenné válna a generálás nélkül. Folytathatnánk a sort további nehézségek felsorolásával (például forrás fájlok mozgatása, linkek létrehozása, ...), de már ezekből is látszik, hogy szinte lehetetlen az elemzéshez szükséges információk összegyűjtése pusztán a fájlok vizsgálatával.

A folytatásban bemutatjuk, hogyan oldottuk meg ezeket a problémákat, valamint ismertetjük, hogy hogyan tudunk nagy rendszereket elemezni, és hogyan számoljuk ki az adott rendszer metrikáit.

2.1.1. A Columbus keretrendszer bemutatása

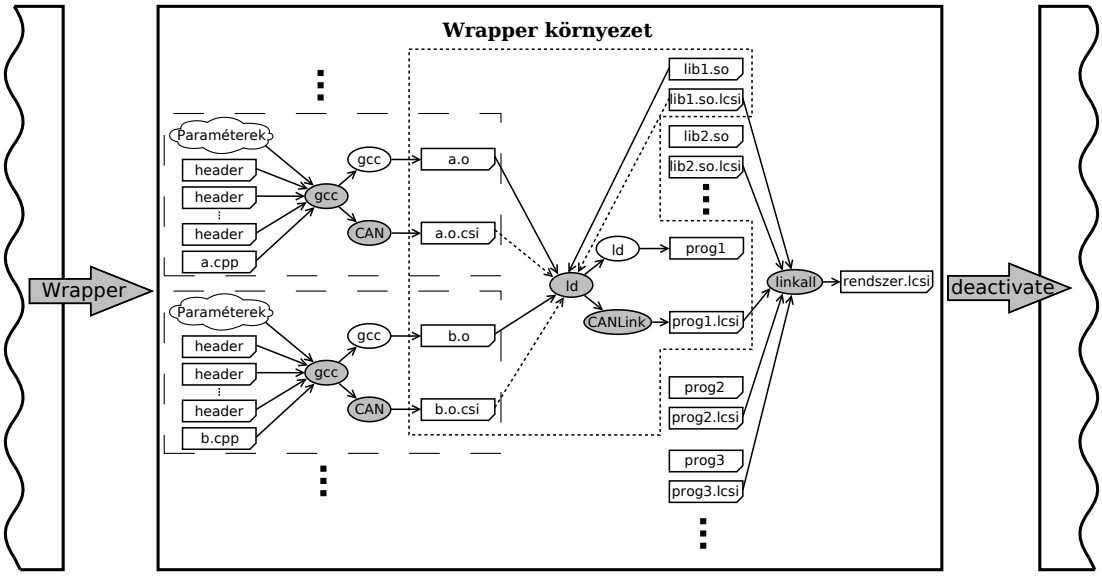
A Columbus keretrendszer egy visszatervező eszköz, amelyet a Szegedi Tudományegyetem, a Nokia Research Center és a FrontEndART Kft. [26] közösen fejlesztettek ki. A kifejlesztésének fő motivációja az volt, hogy készítsünk egy olyan keretrendszert, amely támogatja az automatikus elemzést, valamint egy közös interfészt biztosítsunk a különböző visszatervezési feladatok megvalósításához. A Columbus keretrendszer alkalmas arra, hogy tetszőleges C/C++ rendszert elemezzünk (2.1. ábra), és a kinyert információt különböző célokra felhasználjuk (2.2. ábra).

Egy rendszer elemzése

Az elemzés folyamatát a 2.1. ábrán láthatjuk. Az elemzéshez szükséges két programot, a *CAN*-t és a *CANLink*-et, külön ismertetjük, míg a további programok feladatait és működéseit az elemzési folyamat elmagyarázása közben adjuk meg a könnyebb érthetőség végett (bővebb leírás található néhány korábbi munkában [20, 21, 46]).

A *CAN* (C++ ANalyzer) egy parancssori program, melynek a bemenete egy C++ forrás fájl, valamint a fordításához szükséges további információk, mint például makró definíciók vagy include útvonalak. A kimenet a Columbus C++ séma [19, 20, 22, 46, 52] egy példánya (egy .csi fájl), amely tartalmazza az elemzett fájlban található összes információt. Az ANSI C++ mellett a CAN támogatja a Microsoft dialektust (amit a Visual C++-ban használnak), a Borland dialektust (amit a C++Builder-ben alkalmaznak), valamint a GCC dialektust is (amit a g++ használ).

A *CANLink* (CAN Linker) egy C++ séma linker. Működése hasonlít a fordítók linkeréhez, azaz különböző Columbus séma példányokat egyesít egy nagyobb példánnyá (aminek a kiterjesztése már .lcsi). Így azok a C++ elemek, amelyek logikailag összetartoznak, de különböző fájlokban vannak, egy példánnyá lesznek egyesítve (mint például amikor több object fájlt linkelünk össze egy futtatható programmá). Ennek az újabb példánynak ugyanaz a formátuma, mint az eredetinek, azaz ezt is lehet tovább linkelni, míg végül eljutunk egy olyan példányhoz, ami egymaga reprezentálja a teljes rendszert.



2.1. ábra. Az elemzés folyamata a Columbus keretrendszer segítségével

Az automatikus elemzéshez először aktiválni kell a *wrapper környezetet*, amihez a *Wrapper* programot kell elindítani (a 2.1. ábrán a szürke *Wrapper* nyíl). Ez összegyűjti a fordítóprogram előre definiált beállításait (például az előre definiált makrókat vagy include útvonalakat), valamint beállítja az elemzési folyamathoz szükséges környezetet, azaz inicializálja, vagy módosítja a szükséges környezeti változókat. Ezek közül a legfontosabb, hogy a `PATH` környezeti változó elejére beszúrjuk a *wrapper szkripteket* tartalmazó könyvtár útvonalát. A wrapper szkriptek nevei megegyeznek az eredeti fordító neveivel (például *gcc* vagy *ld*, amelyek szürkével szerepelnek az ábrán). Ezért ha a wrapper környezet aktív, és meghívjuk a fordító valamely programját (például a *gcc*-t), akkor a mi *gcc szkriptünk* fog meghívódni az eredeti gcc helyett, mert ez található meg hamarabb a `PATH` környezeti változóban megadott útvonalakon történő keresésben. Ahhoz, hogy a fordítás során semmi se változzon, a szkriptünk először meghívja az eredeti programot (a fehér *gcc* és *ld*) az eredeti paraméterezéssel az eredeti környezetben. Így a fordítás szempontjából minden ugyanúgy történik majd, mintha nem is lenne a wrapper. Ezután a szkript meghívja a megfelelő programot (gcc esetében a CAN-t, míg mondjuk az ld esetében a CANLink-et), így a fordítással párhuzamosan megtörténik az elemzés is. Hogy jobban lássuk, mi történik egy fordítási vagy linkelési lépés során, az ábrán szaggatott vonallal összefogtuk az adott lépéshez tartozó bemeneteket, programokat és kimeneteket.

Az elemzés befejezése után a *linkall* parancsot kiadva a teljes rendszert egyetlen C++ sémában ábrázolva kapjuk meg[1] (`rendszer.lcsi`). A továbbiakban nincs szükségünk a *wrapper környezetre*, amit a *deactivate* program segítségével „kikapcsolhatunk”, azaz visszakapjuk az eredeti állapotot.

Az elemzési eredmény lehetséges felhasználásai

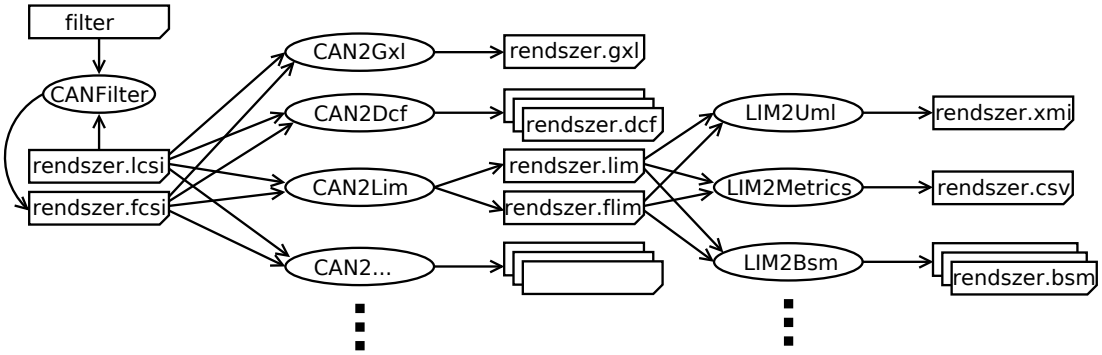
A következőkben bemutatjuk, hogy a Columbus keretrendszer segítségével hogyan tudjuk felhasználni az elemzés eredményét (`rendszer.lcsi`), illetve milyen programok és milyen lépések szükségesek ehhez. A folyamatot a 2.2. ábra szemlélteti.

Az elemzés eredményeként előállt `rendszer.lcsi` minden információt tartalmaz a rendszerről, azonban előfordulhat, hogy bizonyos részeire nem vagyunk kíváncsiak (például mert generált fájlokból származnak). Kézenfekvő megoldás lenne ezen részek elemzésének kihagyása, vagy az elemzés után ezeknek a részeknek a törlése (amire van lehetőség), azonban így fontos információkat veszíthetünk el (például olyan típusdefiníciókat, amelyeket a nem törölt részen használunk). Ezért a törlés helyett csak kiszűrjük ezeket az elemeket, ami azt jelenti, hogy a kiszűrt elemek továbbra is megmaradnak (még csak nem is változnak a `rendszer.lcsi` fájlban), viszont a további felhasználás során nem vesszük figyelembe azokat, de szükség esetén mégis elérhetőek. A megfelelő filter (szűrő) előállítását a *CANFilter* (CAN Filter) végzi, amely egy létező C++ séma példányhoz készít filter fájlt (`rendszer.fcsi`) a séma fájl (`rendszer.lcsi`) mellé anélkül, hogy magát a C++ séma példányt megváltoztatná. A sémában szereplő elemeket azok útvonala alapján lehet kiszűrni vagy megtartani, ahol ezek az útvonalak (pontosabban az ezeket tartalmazó fájl) lesznek a *CANFilter* bemenetei a séma példányt tartalmazó fájl mellett.

Annak ellenére, hogy a 2.2. ábrán a különböző programok bemeneteként szerepelnek

[1] Erre a lépésre azért van szükség, mert annak ellenére, hogy az elemzés során a teljes rendszerről összegyűjtöttük az információkat, ezek általában különböző fájlokban találhatók, amiket még egy közös C++ sémává kell összegyűjteni.

a filter fájlok is (`rendszer.lcsi` és `rendszer.flim`), ezek megadása nem kötelező, és bármelyik program képes ezek nélkül is működni. Ebben az esetben a programok úgy működnek, mintha semmi sem lenne kiszűrve.



2.2. ábra. Az elemzés eredményének felhasználása

A (filterezett) séma példány (`rendszer.lcsi`) feldolgozását a különböző *CAN2** parancssori programok végzik. Néhány közülük a meglévő formátumot transzformálja más formátumra, hogy támogassuk a különböző formátumok együttműködését[2], míg mások különböző számításokat vagy ellenőrzéseket végeznek[3]. A metrikák kiszámolása szempontjából a *CAN2Lim* programot kell külön megemlítenünk, amely a C++ sémát egy nyelvfüggetlen modellre (*Language Independent Model* vagy röviden *LIM*) alakítja át. A nyelvfüggetlen modellt a 2.2. alfejezetben ismertetjük bővebben.

A dolgozatnak nem célja, hogy az összes programot bemutassa, amelyek a C++ sémára épülnek, azonban megjegyezzük, hogy az itt bemutatott néhány példán kívül a Columbus keretrendszer több tíz olyan programot tartalmaz, amelyek különböző szempontok alapján vizsgálják a rendszert, vagy más formátumokra alakítják az elemzés eredményét. Ezekről részletesen olvashatunk például Ferenc Rudolf doktori értekezésében [46]. Azonban ha ezek közt nem találjuk meg azt, amire szükségünk lenne, akkor a nyilvános API segítségével mi magunk is megvalósíthatjuk a szükséges programot.

A metrikák kiszámítása a nyelvfüggetlen modellen történik. A *LIM2Metrics* program minden névtérre, osztályra, metódusra és függvényre kiszámolja a metrika értékeket, és kiírja azokat egy `csv` (pontosvesszővel tagolt szöveges) fájlba. A *LIM2Metrics* programnak megadhatjuk azt is, hogy melyik metrika legyen kiszámolva, és melyik ne.

2.1.2. A Columbus technológia fejlődése

A bemutatott technológiát először a Mozilla elemzéséhez fejlesztettük ki, ami azt jelenti, hogy C/C++ rendszereket tudtunk elemezni Linux alatt, amelyeket GCC-vel fordítottak. A technológia eleinte még nem volt annyira általános, hogy bármely más rendszert könnyedén tudtunk volna elemezni, azonban folyamatosan fejlesztettük azt az újabb és újabb problémákat kiküszöbölve. A felmerülő problémák közt szerepelt például a fájlok másolása, a linkek létrehozása, a fordítónak szóló paramétereknek a

[2] A *CAN2Gxl* a C++ sémát Graph Exchange Language (gxl) [32] formátumra konvertálja.

[3] A *CAN2Dcf* a forráskódban található klón-példányokat találja meg, valamint a forráskód-elemekre számol különböző klón-metrikákat.

megfelelő értelmezése és feldolgozása, vagy amikor az általunk készített, és az eredeti kimeneti fájlok „mellé" tett fájljaink megzavarták a fordítás menetét[4].

A módszer sikerét mutatja, hogy az elmúlt időszakban több nagy ipari rendszert, valamint számos nyílt forráskódú rendszert (például OpenOffice.org [42] vagy WebKit [53]) elemeztünk sikeresen. Emellett ezt a technológiát kiterjesztettük más nyelvekre is, így már Java és C# nyelveken írt rendszereket (például az Eclipse-et [18]) is tudunk elemezni, és a legtöbb dolgot, amit C++ esetében ki tudtunk számolni, azt tudjuk a többi nyelv esetében is.

Végezetül meg kell említenünk, hogy akármennyire is általános a jelenlegi rendszerünk, mindig lehet olyan rendszert készíteni, amelynek az elemzésére nem lesz alkalmas az éppen aktuális formájában. Eddig azonban nem találkoztunk olyan rendszerrel, amelyikhez ne tudtuk volna hozzáigazítani. Ezért azt mondhatjuk, hogy a jelenlegi technológia és az elv, amelyre épül, elég általános ahhoz, hogy szinte tetszőleges C/C++, Java vagy C# rendszert elemezni tudjunk.

2.2. Nyelvfüggetlen modell

Mint azt korábban már említettük, kezdetben csak C/C++ nyelvet tudtunk elemezni, és minden program, ami az adott forráskódra „számolt valamit" (például egy UML osztály diagramot állított elő), az a C++ sémát használta, és azon dolgozott. Ennek megfelelően létezett egy *CAN2Metrics* nevű program is, amely a metrikákat számolta a C++ nyelvre. Mivel a C++ séma célja az volt, hogy a C++ forráskódot pontosan reprezentálja, így minden apró részletet pontosan ábrázolt. Ennek következtében nagy rendszereknél nem tudtuk egyetlen nagy sémában reprezentálni a vizsgált rendszert a túl nagy memória használat miatt, és így nem tudtunk kiszámolni bizonyos metrikákat[5].

A célunk a memória használatának csökkentése volt, ezért kidolgoztunk egy olyan absztrakt modellt, amely csak azokat az elemeket, tulajdonságokat és kapcsolatokat tartalmazta, amelyek szükségesek voltak a metrikák kiszámításához. Ezzel elvesztettük az alacsony szintű információkat, mint például a metóduson belüli utasításokat, de ezekre nem is volt szükség a metrikák kiszámolásánál. Az előny ezzel szemben, hogy a metrika számolás szempontjából fontos kapcsolatok könnyebben elérhetők (mivel azok már metódus illetve osztály szintre vannak megadva), illetve az egyszerűbb sémának köszönhetően sokkal kevesebb memória elegendő. A kevesebb memória használata önmagában nem jelentett volna megoldást, mivel már a különálló részeket sem lehet összelinkelni egyetlen C++ séma fájlba (2.1. ábra *linkall* program nem tudta elkészíteni a `rendszer.lcsi` fájlt). Ezért a *CAN2Lim* programot úgy alakítottuk ki, hogy képes legyen több C++ séma fájlt is beolvasni, és azokat egyesével átkonvertálni az absztrakt modellbe úgy, hogy közben el tudja végezni a CANLink feladatát is, azaz tudjon „linkelni" is az átépítés közben. Ezzel megoldódott a memória problémánk, és a legnagyobb C++ rendszerek esetében is ki tudtuk számolni a metrika értékeket.

[4] Ez akkor fordult elő, amikor az egyik program a konfigurációs fázisban úgy tesztelte, hogy sikerült-e lefordítani egy fájlt, hogy megvizsgálta a keletkezett fájlokat. Azonban az általunk generált és az eredeti kimenet mellé tett .csi fájl megzavarta ezt az ellenőrzést, ezért azóta nem közvetlen a kimenetek mellé tesszük a fájlokat, hanem egy rejtett alkönyvtárba.

[5] A metrikák közül sokat ki lehetett számolni a különálló részeken is (például az ősosztályok számát), és az eredményt összegyűjtve ugyanazt az eredményt kaptuk, mintha az teljes rendszert ábrázoló sémán dolgoztunk volna. Azonban voltak olyan metrikák is, amelyeket nem lehetett ezzel a módszerrel összeszámolni. Például egy metódus bejövő hívásainak a száma az nem egyenlő a különálló részekben található bejövő metódushívások számának összegével, mert lehetnek átfedések.

A Columbus keretrendszer fejlődésével egyre több nyelvet támogattuk (például Java és C#), amelyekre ugyanazokat a kimeneteket kell előállítani, mint a C++ nyelv esetében. Ahelyett, hogy mindent megvalósítottunk volna a különböző nyelvekre is, ezt a modellt – amit innentől már nyelvfüggetlen modellnek (**L**anguage **I**ndependent **M**odel, röviden LIM) hívtuk – használtuk azoknak az eredményeknek az előállítására, amelyek már meg voltak valósítva LIM-en. Ez azt jelenti, hogy minden nyelvhez készítettünk egy programot, amely az adott nyelv sémáját átkonvertálja LIM-re (és közben a CAN2Lim programhoz hasonlóan linkeli is), így szinte azonnal megkaptuk a LIM-ből előállítható kimeneteket (például a metrikákat, bad smell-eket, UML osztály diagramot, hívási gráfot, ...).

2.2.1. A nyelvfüggetlen modell ismertetése

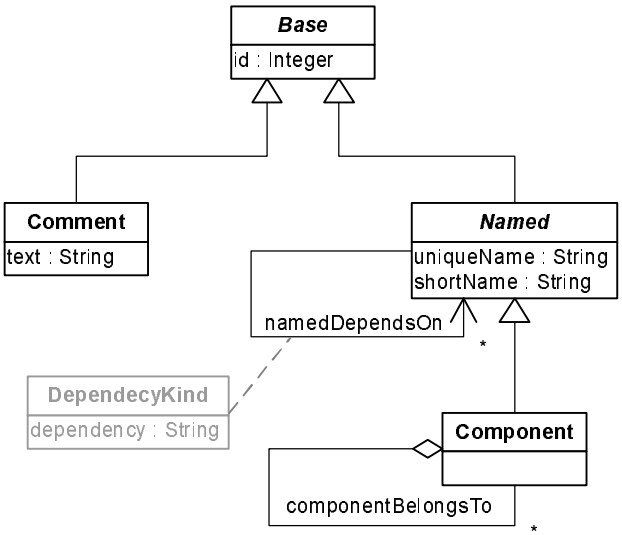
A metrikák kiszámításának részletes ismertetéséhez be kellene mutatnunk azt, hogy a forráskódot hogyan tudjuk ábrázolni az adott nyelvi sémák segítségével, el kellene magyaráznunk, hogy a LIM mit hogyan ábrázol, majd definiálnunk kellene, hogy hogyan építjük át a nyelvi sémát LIM-re, és végül a metrikák LIM-en történő kiszámítását is pontosan le kellene írni. Azonban ez meghaladná az értekezés kereteit, ezért csak a LIM-et mutatjuk be röviden, majd a metrikák kiszámítását egy példa segítségével, ami magában foglalja a példa LIM-en történő ábrázolását is.

A LIM négy csomagból áll. Az *alap* csomag (2.3. ábra) tartalmazza a közös ősosztályt, valamint két általános osztályt is, a *fizikai* csomag (2.4. ábra) a rendszer fizikai, a *logikai* csomag (2.5. ábra) a rendszer logikai nézetét írja le, míg a *típus* csomag (2.6. ábra) a rendszerben ábrázolható típusok felépítéséhez szükséges osztályokat tartalmazza.

Az alap csomag

Az *alap* csomag tartalmazza a *Base* osztályt, amely absztrakt ősosztálya a sémában található összes többi osztálynak. Az *id* egy egyedi azonosító, amely alapján megkülönböztetjük a példányosított osztályokat. A *Comment* osztály reprezentálja a forráskódban található „fontosabb" kommenteket, mint például az osztályok vagy metódusok előtt található kommenteket, ellenben a metóduson belül található kommentek már nem tartoznak ide[6]. A *text* attribútuma tárolja a komment szövegét. A *Named* osztály a *Base* osztályt egészíti ki egy egyedi névvel (*UniqueName*), és egy rövid névvel (*ShortName*). Ez az osztály lesz a közös őse az összes névvel rendelkező osztálynak. Később látni fogjuk, hogy nagyon sokféle speciális függőséget tudunk ábrázolni, azonban előfordulhat, hogy valamiért egy olyan függőséget szeretnénk feltüntetni, amely ezek közül egyikkel sem írható le. A *namedDependsOn* élt használhatjuk arra, hogy tetszőleges *Named* osztályok közti általános függőséget ábrázoljunk, ahol a *DependenceKind* segítségével további információkat adhatunk meg a kapcsolathoz. A *Component* osztály a rendszer komponenseinek ábrázolására szolgál, ahol komponens alatt a különböző könyvtárakat vagy futtatható programokat értjük. A köztük lévő hierarchiát (például egy program használ bizonyos könyvtárakat) a *componentBelongsTo* éllel tudjuk ábrázolni.

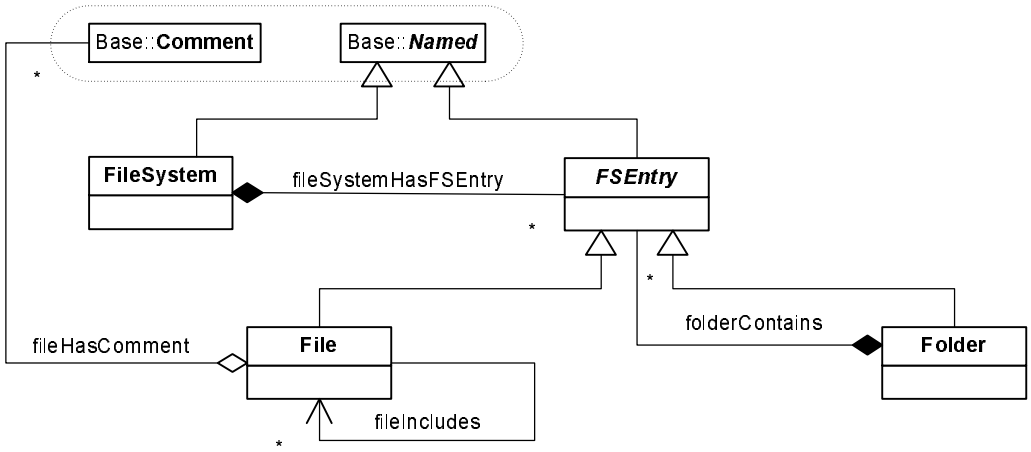
[6] Ezekre a kommentekre ezért van szükségünk, mert a LIM-ből forráskód dokumentációt is előállítunk, amelyhez kellenek a fontosabb kommentek.



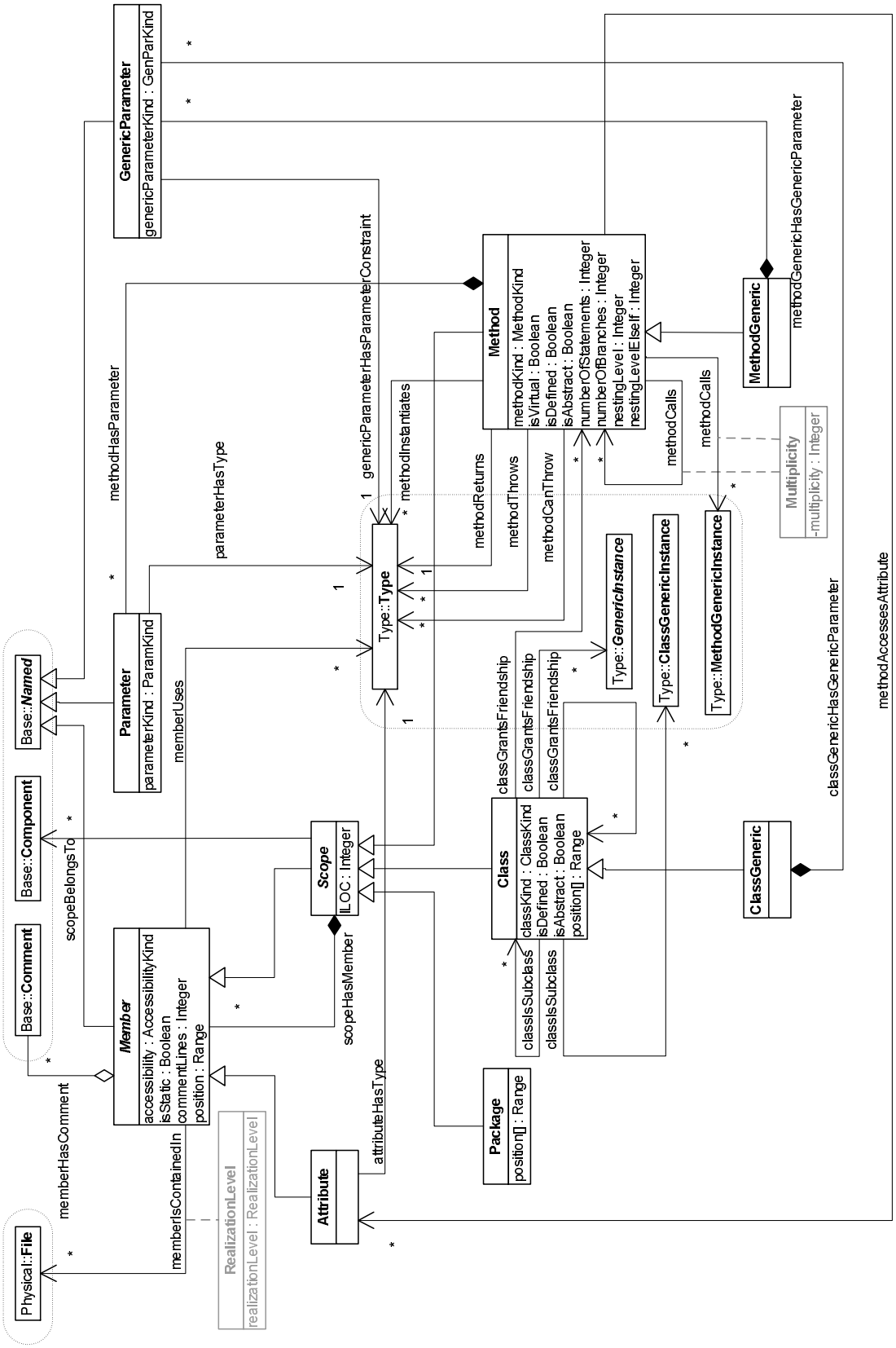
2.3. ábra. Az *alap* csomag UML osztály diagramja

A fizikai csomag

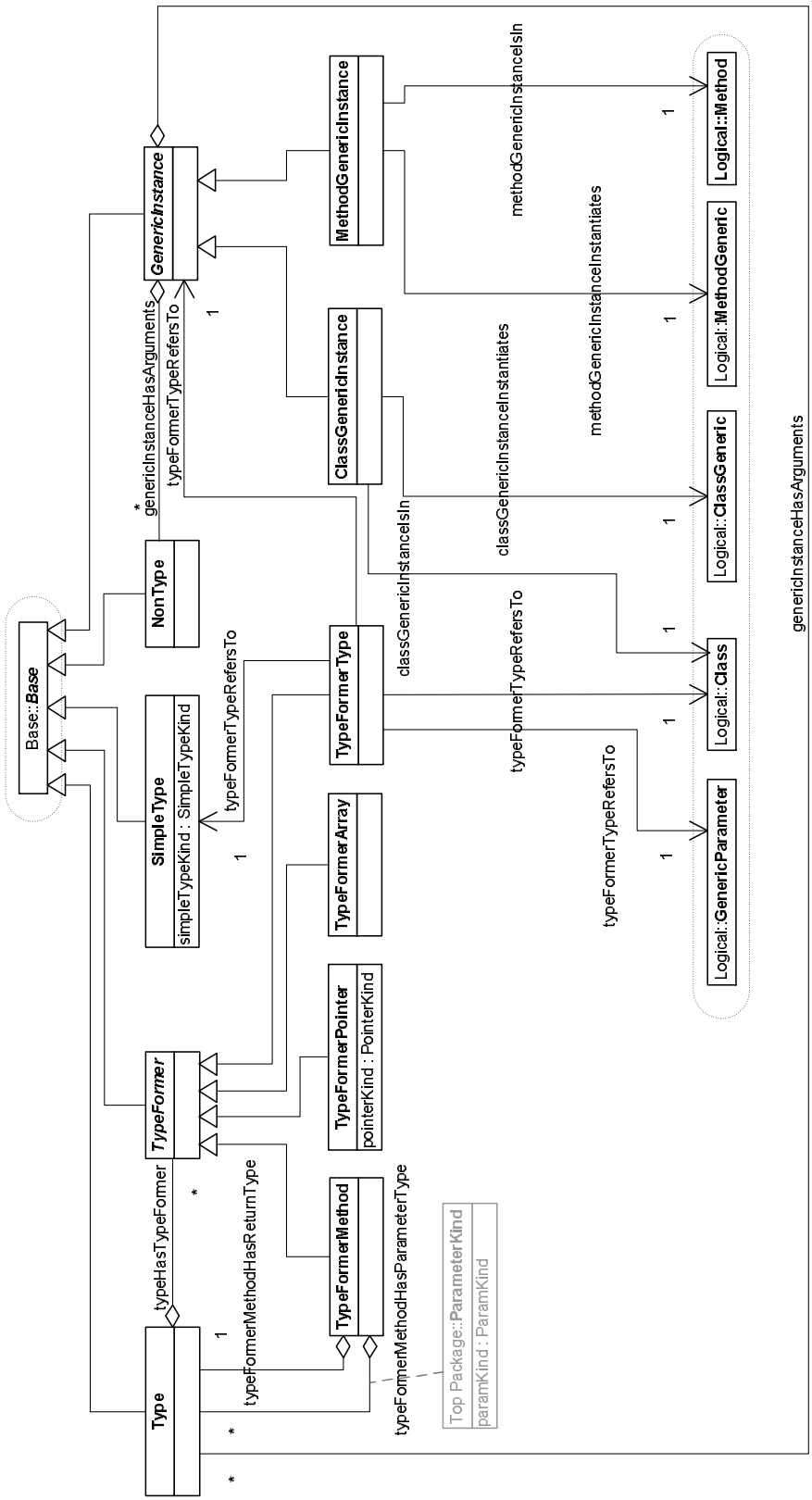
Az elemzett rendszer fizikai struktúráját a fizikai csomag írja le. Ebben található a *FileSystem* osztály, ami magát a fájlrendszert reprezentálja, és amiből csak egy példány lehet, ami már a séma példányosításakor rendelkezésünkre áll. Ez alatt találhatók a fájlrendszer elemei, amit a *FSEntry* absztrakt osztály reprezentál. Kétféle fájlrendszer elemet különböztetünk meg: a könyvtárat (*Folder* osztály), amely tartalmazhat további fájlrendszer elemeket (*folderContains*), és a fájlt (*File* osztály), ami a forráskódot (vagy egyéb fontos információt) tartalmazza. A „legmagasabb szintű" könyvtárakat (például Windows alatt a meghajtókat) nem másik könyvtárak (*Folder*) tartalmazzák, hanem a *FileSystem* tartalmazza azokat, és a *fileSystemHasFSEntry* éllel fejezzük ki ezt a kapcsolatot. A forrásfájlok elején található kommentet szintén eltároljuk, és a fájlhoz tartozó kommenteket a *fileHasComment* éllel kötjük a megfelelő fájlhoz. Emellett a fájlok egymásra hivatkozását (például C++ esetében az *include* vagy Java esetében *import*) a *fileIncludes* éllel adhatjuk meg.



2.4. ábra. A *fizikai* csomag UML osztály diagramja



2.5. ábra. A *logikai* csomag UML osztály diagramja



2.6. ábra. A *típus* csomag UML osztály diagramja

A logikai csomag

A *logikai* csomag a rendszer logikai felépítését, azaz a forráskódban található elemeket, azok tulajdonságait és kapcsolatait írja le. A *Member* osztály a *Named* osztályból származik, és közös absztrakt ősosztálya mindennek, ami egy szkópban megtalálható. Az *accessibility* attribútuma a *Member* láthatóságát tárolja, az *isStatic* azt mondja meg, hogy statikus-e vagy sem, a *commentLines* a *Member* belsejében található komment sorok számát tárolja[7], míg a *Range* a *Member* fájlon belüli elhelyezkedését reprezentálja. Az *Attribute* osztály a *Member* osztályból származik, és a forráskódban található osztályok, struktúrák, unionok és enumok attribútumait és a globális változókat reprezentálja. A típusát a megfelelő *Type* osztályra mutató *attributeHasType* éllel adhatjuk meg. A *Member* osztályból származó *Scope* osztály közös absztrakt ősosztálya a szkópoknak. A *Scope* tartalmazhat *Member* osztályokat (*scopeHasMember*), így a *Scope* osztályok rekurzívan, tetszőleges mélységben tartalmazhatják egymást. Így tudjuk reprezentálni azt, hogy egy csomag osztályokat és további csomagokat tartalmaz, vagy hogy egy osztály további osztályokat és metódusokat tartalmaz. Mivel az alacsony szintű elemeket nem ábrázoljuk a LIM-ben, ezért számos fontos metrikát nem tudnánk kiszámolni LIM-en. Ezt a problémát úgy oldottuk meg, hogy az ilyen értékeket a nyelvi sémán kiszámoljuk, és az eredményt eltároljuk a megfelelő osztályok attribútumaként. A *Scope* nem üres és nem komment sorainak a számát is hasonló módon ábrázoljuk a *LLOC* attribútum segítségével. A *Package* osztály a névterek (C++ és C# esetében) vagy a csomagok (Java esetében) reprezentációjára szolgál. Mivel egy *Package* több fájlban is lehet, ezért a *Member* osztályban található *Range* „helyett" a *Package* esetében *Ranges*-t használunk, amely csak annyiban tér el a *Range*-től, hogy tetszőleges számú intervallumot adhatunk meg. A *Class* osztály reprezentálja az osztályokat, struktúrákat, unionokat, interfészeket és enumerátorokat. Hogy ezek közül melyiket reprezentálja az adott példány, a *classKind* mutatja meg. A deklarációk és definíciók megkülönböztetésére az *isDefined* attribútum használható, míg az absztrakt osztályok megjelölésére az *isAbstract* attribútum szolgál. Mivel a C# nyelv megengedi, hogy egy osztály több részből álljon (partial class), ezért a *Scope* osztályhoz hasonlóan a *Class* osztálynál is a *Ranges*-t használjuk a pozíció tárolására. Az osztályok (és a generikus osztályok) közti öröklődést a *classIsSubclass* kapcsolat ábrázolja. A C++ nyelv esetében lehetőségünk van az osztályokban barát (friend) osztályokat és metódusokat megadni, amelyet a *classGrantsFriendship* kapcsolattal írhatunk le. A *Class* osztályból származik a *ClassGeneric* osztály, ami a generikus (vagy template) osztályokat reprezentálja, és ami annyiban tér el a nem generikus osztálytól, hogy vannak generikus paraméterei (*classGenericHasGenericParameter*). A metódusokat és a globális függvényeket a *Method* osztály ábrázolja. Az általános metódusok mellett megkülönböztetjük a konstruktorokat, destruktorokat, operátorokat, valamint a beállító és lekérdező metódusokat, amit a *MethodKind* attribútum tárol. Megadhatjuk, hogy egy metódus virtuális-e vagy sem (*isVirual*), hogy definiált-e vagy sem (*isDefined*), vagy hogy absztrakt-e vagy sem (*isAbstract*). A metódusban található utasítások számát a *numberOfStatements* attribútum, az elágazások számát a *numberOfBranches*, míg az elágazások egymásba ágyazásának a maximumát, illetve annak kétféle változatát a *nestingLevel* és a *nestingLevelElseIf* attribútumok tárolják. A metódusok (és a generikus metódusok) közti hívásokat a *methodCalls*

[7]Korábban említettük, hogy a *Comment* osztályt nem használjuk az összes komment tárolására, de bizonyos metrikák kiszámításához szükségünk van a komment sorok számára, így ezt el kell tárolnunk.

kapcsolattal ábrázoljuk azok multiplicitásaival együtt, amit a *Multiplicity* asszociációs osztály tárol. Minden metódushoz eltároljuk, hogy mely attribútomokat használja (*methodAccessAttribute*), mi a visszatérési értéke (*methodReturns*), milyen típusokat példányosít (*methodInstantiates*), hogy milyen kivételeket dobhat (*methodCanThrow*) és hogy miket dob valójában (*methodThrows*). Továbbá minden metódus esetében feltüntetjük annak paramétereit is (*methodHasParameter*). A *MethodGeneric* osztály a generikus metódusokat reprezentálja, amely a *Method* osztálytól csak a generikus paraméterben tér el (*methodGenericHasGenericParameter*). A *Parameter* osztály a függvények paramétereit ábrázolja. A *parameterKind* segítségével megadhatjuk, hogy a paraméter bemenő, kimenő vagy be- és kimenő. A paraméter és a típusa közti kapcsolatot a *paraméterHasType* él realizálja. A csomag utolsó osztálya a *GenericParameter*, ami a generikus osztályok és generikus metódusok generikus paramétereit ábrázolja. A generikus paraméter *genericParameterKind* attribútuma határozza meg, hogy a generikus paraméter típus paraméter, nem típus paraméter, template paraméter vagy megszorítás. Továbbá a generikus paraméter megszorítását egy típus segítségével reprezentáljuk (*genericParameterHasParameterConstraint*).

A típus csomag

A metrikák kiszámításához szükséges kapcsolatokat a logikai csomag segítségével is ábrázolhatnánk, és nem lenne szükségünk bonyolult típus-ábrázolásra. Azonban a LIM-et felhasználjuk UML osztálydiagramm és forráskód dokumentáció előállítására is, amihez már szükséges típusok minél pontosabb ábrázolása. Ezért dolgoztuk ki a *típus* csomagot, amely az objektum-orientált nyelvek típus-ábrázolását ötvözi egy közös sémában. A csomag egyik fele a generikus példányokat ábrázolja, míg a másik része a típusok leírását végzi. Míg a többi csomag esetében a csomagban található osztályok és kapcsolataik maguktól érthetőek voltak, addig a típus csomag elemeit nem lehet egyszerűen és röviden leírni. Ezért itt nem is kíséreljük meg ennek ismertetését, a LIM szerepe enélkül is megérthető[8].

2.2.2. A metrikák kiszámítása a nyelvfüggetlen modell segítségével

A 2.7. ábrán látható egyszerű példa segítségével mutatjuk be, hogy a különböző elemek és kapcsolatok hogyan jelennek LIM-ben, valamint hogy LIM-ből hogyan számoljuk ki a metrikákat. A 2.8. ábrán látható a példához tartozó LIM reprezentáció. Mivel az ábra már így is túl sok elemet tartalmazott, és a metrikák kiszámítása szempontjából a fizikai csomag elemei nem érdekesek, ezért azokat nem tüntettük fel. Hasonló megfontolásból az elemek pozícióját ábrázoló Range attribútumot sem jelenítettük meg.

Az A melléklet tartalmazza az értekezésben használt metrikák definícióit, amelyek közül kettő kiszámítását bemutatjuk. Az első a `Child` nevű osztály NOA metrikájának kiszámítása, amihez meg kell számolnunk, hogy hány *ClassIsSubclass* éle van, és ez adja meg a metrika értéket, ami a jelen esetben 1. A másik példa a `User` osztály CBO metrikája, amit a következő módon számolunk. Egy halmazba összegyűjtjük, hogy az osztály, illetve annak metódusai és attribútumai „mit használnak”, és a metrika értéke

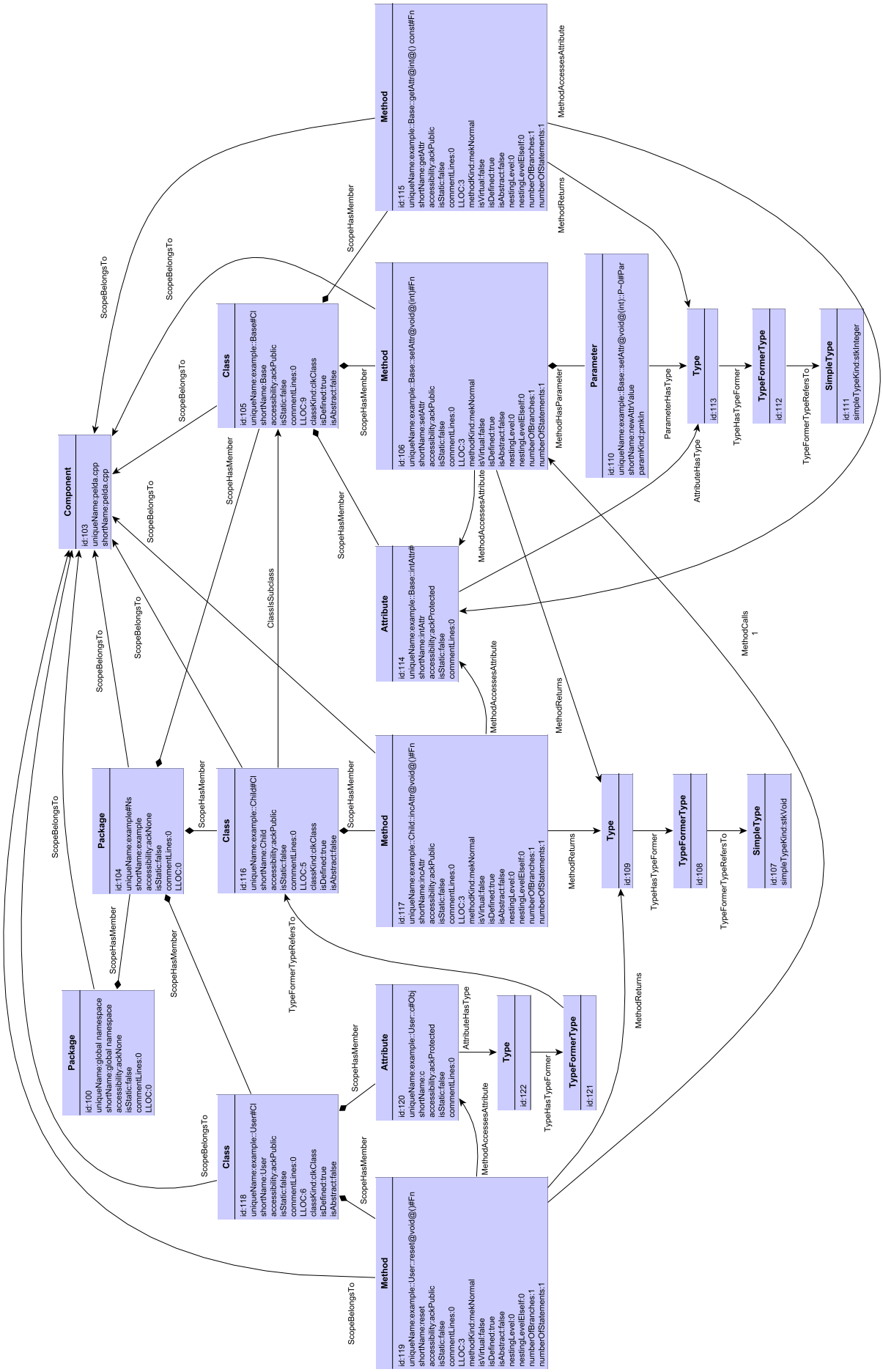
[8] Az érdeklődőknek Ferenc Rudolf doktori disszertációját [46] ajánljuk, ahol részletesen be van mutatva a C++ típus-ábrázolása, amely nagyon hasonlít erre.

```
namespace example {
  class Base {
  public:
    void setAttr(int newAttrValue) {
      intAttr = newAttrValue;
    }
    int getAttr() const {
      return intAttr;
    }
  protected:
    int intAttr;
  };

  class Child : public Base {
  public:
    void incAttr() {
      ++intAttr;
    }
  };

  class User {
  public:
    void reset() {
      c.setAttr(0);
    }
  protected:
    Child c;
  };
}
```

2.7. ábra. C++ nyelvű példa a metrikák kiszámításának szemléltetéséhez

a halmaz elemszáma lesz. Maga az osztály nem használ további osztályokat. A metódusának a *MethodAccessesAttribute* éle saját osztálybeli elemre mutat, így nem növeli a csatolás értékét, a *MethodReturns* éle olyan típusra mutat, amelyet „feloldva" egy primitív típushoz, a *Void*-hoz jutunk, így ez sem növeli a csatolást. A harmadik kapcsolata egy olyan metódust (`setAttr`) hív, amely a `Base` osztályhoz tartozik (*scopeHasMember*), így ez az osztály növeli a csatolás értékét. Végezetül megnézzük az osztály `c` attribútumát, amelynek az *AttributehasType* éle egy olyan típusra mutat, amelyet „feloldva" a `Child` osztályhoz jutunk, így ez is növeli az osztály csatolását. Végül azt kaptuk, hogy a `User` osztály használja a `Base` és a `Child` osztályokat, azaz a CBO metrikájának értéke 2.

További példákkal is szemléltethetnénk a LIM-en történő metrikaszámolást, azonban már ebből is látszik, a magas szintű ábrázolásnak köszönhetően egyszerüen dolgozhatunk rajta. Ennek következtében sokkal egyszerűbben számolhatjuk ki a metrika értékeket, mintha a nyelvi sémákat használnánk.



2.8. ábra. A példa LIM-en történő ábrázolása

2.3. A Mozilla elemzése

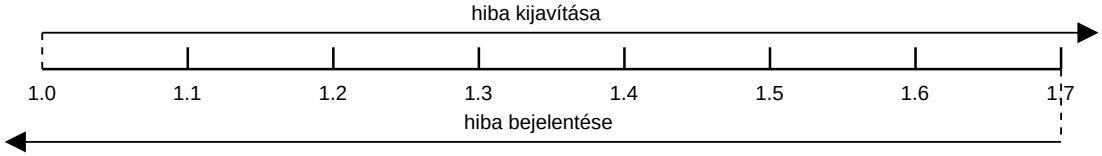
A Columbus keretrendszer segítségével az első vizsgálatunk során [28] a Mozilla [41] hét különböző verzióját elemeztük az 1.0-tól (amit 2002. augusztusában adtak ki) az 1.6-ig (amit 2004. januárjában adtak ki), és minden osztályra kiszámítottuk az általunk vizsgálat objektum-orientált metrikákat. A 2.1. táblázat mutatja a hét verzió néhány méret alapú metrikáját. Mind a hét verzió esetében a forráskód több mint egymillió nem üres és nem komment sorból áll (amit a **T**otal **L**ogical **L**ines **o**f **C**ode, azaz *TLLOC* metrika mutat). Továbbá közel 4 700 osztályt (**T**otal **N**umber of **CL**asses, röviden *TNCL*), 70 ezer metódust (**T**otal **N**umber of **M**ethods, röviden *TNM*) és 47 ezer attribútumot (**T**otal **N**umber of **A**ttributes, röviden *TNA*) tartalmaznak.

Verzió	TLLOC	TNCL	TNM	TNA
1.0	1 127 391	4 770	69 474	47 428
1.1	1 145 470	4 823	70 247	48 070
1.2	1 154 685	4 686	70 803	46 695
1.3	1 151 525	4 730	70 805	47 012
1.4	1 171 503	4 967	70 096	48 389
1.5	1 169 537	5 007	72 458	47 346
1.6	1 165 768	4 991	72 314	47 608

2.1. táblázat. A vizsgált Mozilla verziók fontosabb méret-alapú metrikái

2.3.1. A hibák osztályokhoz rendelése

Ahhoz, hogy a metrikák és a hibák száma közti kapcsolatot vizsgálni tudjuk, a metrikák mellett az osztályokban talált és kijavított hibákat is össze kellett gyűjtenünk. Mivel 2004-ben – az első vizsgálatok idején [28] – a legfrissebb verzió az 1.7-es volt, ezért csak 7 verziót vizsgáltuk, az 1.0-tól az 1.6-ig (az 1.7-et azért nem, mert a Bugzilla még nem tartalmazott kijavított hibákat az 1.7-es verzióra).



2.9. ábra. A kijavított hibák szűrése a bejelentési és a kijavítási idejük alapján

A hibákat a következő módon gyűjtöttük össze. A Mozilla közösség biztosította számunkra a Bugzilla [9] adatbázist, amely tartalmazza az összes bejelentett hibát a szoftver fejlesztésének kezdete óta. A Bugzilla adatbázis 256 613 különböző hibabejegyzésére azonban nem volt szükségünk (lásd a 2.2. táblázatot). Először kiszűrtük azokat, amelyek a Mozilla más termékeire (például a Bonsai vagy a Tinderbox) vonatkoztak, így 231 021 maradt. Továbbá csak azokat a hibákat vizsgáltuk, amelyek ki lettek javítva (57 151 maradt), valamint a hibabejelentés tartalmazta a hiba kijavításához szükséges

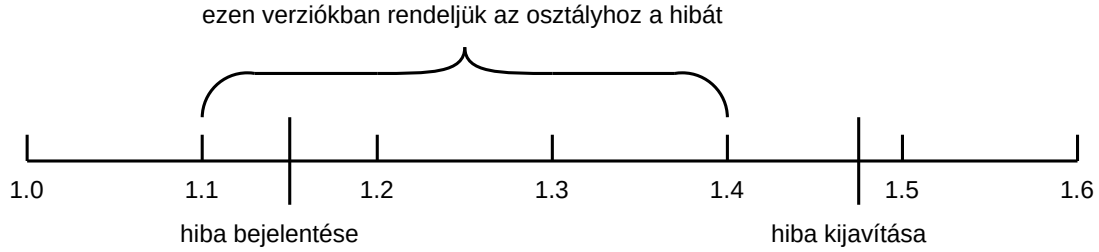
patch[9] fájlt is (22 553 maradt). Az 1.0-ás Mozilla verzió 2002. májusában jelent meg, így nem vizsgáltuk a korábban kijavított hibákat (9 539 maradt). Utolsó lépésként kiszűrtük azokat a hibákat, amelyek az 1.7-es verzió megjelenése, azaz 2004. júniusa után lettek bejelentve (lásd 2.9. ábra), mivel ezeket már az 1.7-es verzióhoz kellett volna rendelnünk, de azt nem vizsgáltuk. Végül 8 936 hibabejelentés maradt, amely teljesítette az összes feltételt.

Szűrési szempont	Hibák száma
Összes hibabejelentés	256 613
Termékek	231 021
Kijavított hibák	57 151
Patch	22 553
Dátum szerint (2002. május után)	9 539
Dátum szerint (2004. június előtt)	8 936

2.2. táblázat. A Mozilla bejelentett hibáinak szűrési lépései

A célunk az volt, hogy a hibákat az osztályokhoz rendeljük, azonban a hibajavításoknál nem volt odaírva, hogy a hiba kijavításához mely osztályokat kellett módosítani. Ennek ellenére a következő heurisztika segítségével nagy pontossággal megmondhatjuk, hogy az adott hiba kijavítása során mely osztályok módosultak, azaz mely osztályok tartalmazták a hibát.

Az általunk vizsgált, kijavított hibák mindegyikéhez tartozott legalább egy patch fájl. Egy patch fájl leírja, hogy a hiba kijavításához milyen változásokra van szükség a forráskódban, illetve tartalmazza azoknak a forrás fájloknak a neveit, amik változtak, valamint leírja, hogy mennyi sor lett törölve egy adott pozíciótól kezdve, és mennyi sor lett beszúrva egy adott pozíciótól kezdve. Ezekből az információkból meghatároztuk azokat a fájlokat, valamint a fájlokban azokat az intervallumokat, amelyek változtak a hiba kijavítása során.



2.10. ábra. Egy konkrét hiba kijavítását érintő Mozilla verziók

Az szintén nem szerepelt a hibajavítás mellett, hogy melyik verziókban létezett a hiba. A hibabejelentés tartalmazza azt az információt is, hogy mikor jelentették be a hibát, valamint, hogy mikor javították ki azt. Ezek alapján azt feltételeztük, hogy a hiba létezett attól a verziótól, amelyikre bejelentették, addig a verzióig, amelyiknek a fejlesztésében ki lett javítva (2.10. ábra). Minden egyes hibára megvizsgáltuk, hogy a patch fájlok segítségével kialakított intervallumoknak és a javításhoz tartozó verzióban

[9] A patch fájl tartalmazza egy fájl két egymást követő verziója közti különbséget. Megadja, hogy pontosan melyik fájlról van szó, melyik sorok változtak, és tartalmazza a változást is.

szereplő osztályok forráskód pozícióinak van-e átfedése. Ha találtunk átfedést, akkor az adott hibát hozzárendeltük az adott osztályhoz, továbbá minden, a hiba által érintett verzióban is hozzárendeltük az adott osztályhoz a hibát. Abban az esetben, ha egy hiba több osztályt is érintett, akkor az összes osztályhoz hozzárendeltük a hibát.

Ezzel a módszerrel a 8 936 bejelentett és kijavított hibának közel a felét, 4 429 hibát sikerült osztályokhoz rendelnünk. Figyelembe véve, hogy a hibákat csak osztályokhoz rendeltük (ami objektum-orientált konstrukció, így csak a C++ forrásokban található), és a Mozilla több mint 1 500 C forrás fájlt is tartalmazott, az eredményt elfogadhatónak ítéltük.

Utolsó lépésként a további vizsgálatból kizártuk azokat az osztályokat, amelyek a fordítás során generálódnak, mert ezekhez nem lehet hibákat rendelni. Továbbá a vizsgálatoknál nem vettük figyelembe azokat sem, amelyek léteztek mind a hét vizsgált Mozilla verzióban, nem tartalmaztak hibát, és a metrika értékük sem változott. Végül a Mozilla 1.6-os verziója esetében 3 192 osztály maradt a 3 667 osztályból. A további vizsgálatokhoz is ezeket az osztályokat használtuk fel. A 2.3. táblázat foglalja össze az eddigi eredményeket. Az osztályoknak több mint a fele (1 850 darab) nem tartalmazott hibát, és közel ötöde (666 darab) csak egy hibát tartalmazott.

Az osztályok száma	%	A hozzárendelt hibák száma
1 850	57,96	0
666	20,86	1
236	7,39	2
136	4,26	3
78	2,44	4
49	1,54	5
44	1,38	6
26	0,81	7
28	0,88	8
18	0,56	9
14	0,44	10
4	0,13	11
7	0,22	12
2	0,06	13
5	0,16	14
9	0,28	15-19
8	0,25	20-24
8	0,25	25-29
4	0,13	30-34
3 192	100,00	3 961

2.3. táblázat. A hibák eloszlása az 1.6-os verzióban

A bemutatott módszer alkalmas volt arra, hogy a Bugzilla adatbázisából egyszer összegyűjtsük és osztályokhoz rendeljük a hibákat, majd a megfelelő vizsgálatokat elvégezzük. Azonban ha meg akarjuk ismételni az egész folyamatot később (mint ahogy az meg is történt [47]), vagy folyamatosan (például heti vagy havi rendszerességgel) nézni szeretnénk a bejelentett és kijavított hibákat, akkor ismét el kellene kérnünk az aktuális Bugzilla adatbázist, ami körülményessé és szinte használhatatlanná tenné a

folyamatot. Hogy kiküszöböljük eme gyengeséget, kifejlesztettünk egy olyan programot, amely az interneten keresztül közvetlenül a Bugzillából gyűjti össze a hibabejelentéseket. Ezt egy olyan URL segítségével tudtuk megtenni, amely tartalmazta a Bugzilla címét és az összes szűrési feltételt (például, hogy a hiba a Mozillára vonatkozzon, legyen kijavítva és a megfelelő időintervallumba essen). Egy további paramétert is hozzátettünk az URL-hez, hogy az eredményt XML formában kapjuk meg. A visszakapott XML fájl feldolgozásával megkaptuk az összes hibabejelentés azonosítóját, amelyek megfeleltek a kritériumoknak. Ezután az egyes hibabejelentésekhez tartozó információkat is letöltöttük XML formában, majd ezek elemzésével megkaptunk minden szükséges információt a hibákról, beleértve azt is, hogy mely patch fájlok tartoznak hozzájuk. A patch fájlokat az előzőekhez hasonlóan letöltve megkaphatunk minden szükséges információt, amit korábban a Bugzilla adatbázisából nyertünk ki.

Igaz, hogy ezzel a megoldással függetlenítettük magunkat a Bugzilla tulajdonosától, azonban ennek a módszernek is van egy hátránya. Ha nagyon sok hibabejelentés felel meg a kritériumoknak (a mi esetünkben ez fennállt), akkor a folyamat lassú, és nagyon megterheli a Bugzillát és az azt kiszolgáló infrastuktúrát (például az SQL adatbázist és a Bugzillát kiszolgáló szervert). Így például a később megismételt vizsgálatokhoz [47] a Mozilla ismételt elemzéséhez tartozó adatok ilyen módon történő összegyűjtése napokig tartott[10] (igaz, hogy valamivel több hibáról volt szó, de a hibák száma nagyságrendileg nem változott). Viszont ha már egyszer összegyűjtöttük a hibákat ezzel a módszerrel, akkor archiválhatjuk azokat, és ha később meg akarjuk ismételni a gyűjtögetést, vagy ki akarjuk bővíteni a már meglévő anyagot, akkor nem kell megismételni a teljes folyamatot. Ilyenkor elegendő csak a „különbséget” (az új vagy módosult hibabejelentéseket és a hozzájuk tartozó információt) letölteni, ami sokkal gyorsabb és nem terheli a Bugzillát. Ezt a módszer sikeresen alkalmaztuk a Mozilla későbbi vizsgálatakor [47], illetve azóta már más rendszerek (például az OpenOffice.org [42]) esetében is.

A 3. fejezetben a Mozilla 1.6-os verzióján elvégzett kísérletünket írjuk le, ahol nyolc kiválasztott metrika és a hibaszámok közötti kapcsolatot vizsgáltuk. A 4. fejezetben szintén a Mozilla 1.6-os verzióján kiszámolt metrika értékeket és hibaszámokat használjuk a vizsgálatokhoz, azonban a két kísérlet között eltelt időben a Columbus keretrendszer fejlődött, továbbá újabb hibákat is kijavítottak a Mozillában, így a 4. fejezet elején röviden ismertetjük majd az újabb elemzés eredményeit, amely kissé eltér a most ismertetettől.

2.4. Kapcsolódó munkák

A Columbus technológia kidolgozásakor a fő célunk a technológia kifejlesztése és a vizsgálatokhoz szükséges adatok előállítása volt. Annak ellenére, hogy a bemutatott parancssori programokat könnyű megérteni és használni, ezeket mégsem nevezhetjük olyan felhasználóbarát programoknak, amelyeket a fejlesztők szívesen használnának minden nap. Az pedig végképp elképzelhetetlen, hogy a menedzserek az eredményeket abban a formában használják fel, ahogy azt a programok előállítják. Az 5. fejezetben két olyan programot mutatunk be, amelyek a Columbus technológiára épülnek, de grafikus felülettel rendelkeznek, és támogatást nyújtanak az eredmények felhasználásához is. Ezért a metrika számításhoz kapcsolódó munkákat is annak a fejezetnek a

[10] A hibák összegyűjtése azért is tartott ilyen sokáig, mert két letöltés közé egy pár másodperces várakozást iktattunk be, hogy ne terheljük meg nagyon a Bugzillát kiszolgáló infrastuktúrát.

végén fogjuk ismertetni.

Korábban Godfrey és Lee [27] is vizsgálta a Mozilla Milestone-9 verziójú[11] architektúráját. Az adatok kigyűjtésére és vizualizálására a PBS [23] és Acacia [11] visszatervező eszközöket használták. Kialakították a Mozilla alrendszereinek hierarchiáját, majd vizsgálták a köztük lévő kapcsolatokat. 11 magas szintű komponenst sikerült beazonosítaniuk, amelyeket további alrendszerekre lehetett osztani. Az alrendszerek hierarchiáját a forráskód könyvtárszerkezetét és a fellelhető dokumentációkat felhasználva határozták meg. Az általuk végzett vizsgálatokból kiderült, hogy a magas szintű komponensek függőségi gráfja majdnem teljes gráf, azaz majdnem minden magas szintű komponens használja az összes többit.

Ayari és munkatársai [1] azt vizsálták, hogy vajon a Bugzilla hibakövető rendszere és a Mozilla verziókövető rendszere által szolgáltatott adatok mennyire összevethetőek, illetve a kidolgozott heurisztikák mennyire képesek összerendelni ezek bejegyzéseit egymással, illetve a forráskóddal. Az egyik fő cél volt az, hogy megnézzék mennyi a pontosság (precision) és fedés (recall) értéke a hozzárendeléseknek. Rámutattak arra, hogy a vizsgált rendszerből összegyűjtött adatokat nehéz összerendelni. A fedés általában 34-42% között változott, vagyis körülbelül ennyi elemet tudtak a Bugzilla adatai alapján visszakeresni a verziókövető rendszer log bejegyzéseiből (ahol az volt a feltételezés, hogy minden log értelmes). A vizsgálat során kiderült, hogy több a megoldott hiba – azaz amit javítottak, de még nem ellenőrizték annak jóságát – mint amennyit ellenőriztek is, valamint megállapították, hogy a bejegyzéseknek körülbelül 58%-a volt hibajavítás.

2.5. Az eredmények összegzése

Ebben a fejezetben bemutattuk, hogy a Columbus technológia segítségével hogyan lehet elemezni egy rendszert. Ismertettük azt a nyelvfüggetlen modellt, amelynek segítségével több különböző nyelvre ki tudjuk számolni a metrikákat. Leírtuk azt a heurisztikát, amelynek segítségével képesek voltunk a Mozilla osztályaihoz meghatározni a bennük található kijavított hibákat.

A Columbus keretrendszer elemző környezetének kidolgozása, valamint a Mozilla elemzése nem a szerző eredménye. A nyelvfüggetlen modell ötlete, kidolgozása, a C++ séma nyelvfüggetlen modellre történő konvertálása, illetve a nyelvfüggetlen modellen történő metrika számolás a szerző saját eredménye. A hibáknak a hibakövető rendszerből történő összegyűjtése, illetve a hibák osztályokhoz rendelésének heurisztikája szintén a szerző önálló eredménye.

[11] A kezdeti időszakban a Milestone-X (ahol X egy számot jelöl) jelölte a különböző Mozilla verziókat, és csak később tértek át a ma is használatos jelölésre.

II. rész

Objektum-orientált metrikán alapuló hiba-előrejelző modellek

3. fejezet

Metrika alapú hiba-előrejelző modellek

Az előző fejezetben bemutattuk, hogyan számoltuk ki a metrika értékeket a Mozilla több verziójának összes osztályára, valamint hogyan határoztuk meg, hogy a vizsgált időszakban mennyi hiba lett kijavítva bennük. Most azt fogjuk megvizsgálni, hogy van-e kapcsolat a metrikák és az osztályokban található hibák között.

3.1. A metrikák bemutatása

A kísérleteinkhez nyolc metrikát választottunk ki, és ezek segítségével vizsgáltuk azt, hogy van-e bármilyen kapcsolat a metrikák és az osztályokban található hibák száma között, illetve használhatók-e a metrikák az osztályokban található hibák számának becslésére vagy az osztályok hibára való hajlamosságának előrejelzésére. A nyolc metrikából hat a Chidamber és Kemerer által definiált és publikált metrika [12]. Ezt kiegészítettük egy objektum-orientált metrikával, az LCOMN-nel, amely a hat metrika közül az egyiknek egy módosítása. A nyolcadik metrika a nem üres és nem komment sorok száma metrika, amelyet azért választottunk ki, hogy összehasonlítsuk az objektum-orientált metrikákat és a tradicionális méret alapú metrikát.

A Chidamber és Kemerer által adott metrika definíciók nem egy konkrét programozási nyelvhez vannak megfogalmazva, hanem objektum-orientált fogalmak segítségével általánosan írják le azokat. Voltak olyan metrikák, amelyek szándékosan általánosabban voltak megfogalmazva, hogy a metrikák minél szélesebb körben legyenek alkalmazhatóak. Ennek következtében többféleképpen lehet értelmezni és kiszámolni az egyes metrikákat, így ugyanannak a metrikának különböző változatait kaphatjuk. Annak ellenére, hogy a Columbus keretrendszer segítségével ki tudtuk számolni az adott metrika több különböző verzióját is, először minden metrika esetében csak egy változatot vizsgáltunk meg. A metrikák további változatait más metrikákkal együtt egy későbbi munkában [47] vizsgáltuk, és az eredményeket a 4. fejezetben ismertetjük. A következőkben megadjuk az általunk használt metrikák pontos definícióit, amelyek figyelembe veszik a C++ nyelv sajátosságait, valamint felhasználjuk Basili és munkatársai [4] észrevételeit, mert ők szintén ezt a hat metrikát vizsgálták C++ rendszerekre. A metrikák különböző értelmezéseinek más és más jelentése lehet, így bizonyos esetekben a jelentéseket figyelembe véve találóbb nevet lehet adni a metrikának. Ennek köszönhetően néhány esetben a metrika elnevezések el fognak térni a Chidamber és Kemerer által adottól. Erre azért is szükség van, hogy konzisztens elnevezéseket használjunk a későbbi munkánkkal, valamint a dolgozat második felével.

- **Number of Methods Local (NML)**: Adott egy C osztály, amelynek n darab metódusa van, és a metódusok komplexitása $c_1, c_2, \ldots, c_n$. Ekkor a komplexitását a

$$WMC(C) = \sum_{i=1}^{n} c_i$$

képlettel számolhatjuk ki. Az eredeti definícióban szándékosan nem határozták meg, hogy mit használjunk a metódus komplexitásának. Ha a metódusokat egyformán komplexnek tekintjük, és a súlyukat egynek választjuk, akkor a WMC metrika valójában a metódusok számával egyenlő (azaz $WMC = n$), amit a **Number of Methods Local (NML)** metrikával definiálunk. A vizsgálatok során mi is ezt a változatot alkalmaztuk, így nem a WMC jelölést alkalmazzuk, hanem az NML rövidítést.
A WMC egy másik változata az, ha a metódusok komplexitásának az azokban található elágazások számát választjuk (amit a *McCabe Cyclomatic Complexity (McCC)* metrika számol [40]). Mi ezt a változatot jelöljük WMC rövidítéssel, és a 4. fejezetben foglalkozunk majd a vizsgálatával.

- **Number of Ancestors (NOA)**: Az eredeti definícióban a *Depth of Inheritance Tree (DIT)* metrika szerepelt, ami egy adott osztályra azt adja meg, hogy az osztály mennyire mélyen helyezkedik el az öröklődési hierarchiában. Abban az esetben, ha az adott nyelv nem támogatja a többszörös öröklődést, ez az érték megegyezik az osztály ősosztályainak számával (azaz a **Number of Ancestors (NOA)** metrikával). Azoknál a nyelveknél, ahol a C++ nyelvhez hasonlóan lehetséges a többszörös öröklődés, ott ez a két metrika eltérhet. A vizsgálatainkhoz a NOA metrikát választottuk, mert ez jobban figyelembe veszi a C++ nyelv sajátosságait, azaz a többszörös öröklődést, továbbá ez a választás egybeesik Basili és munkatársai értelmezésével is.

- **Number of Children (NOC)**: A NOC metrika az adott osztályból közvetlenül származó osztályok száma.

- **Coupling Between Object classes (CBO)**: Egy osztály közvetlenül kapcsolódik egy másikhoz, ha használja annak metódusait és/vagy attribútumait. A CBO azon osztályok száma, amelyekhez a vizsgált osztály közvetlenül kapcsolódik.

- **Response set For a Class (RFC)**: Az RFC metrika egy C osztályra az

$$RFC(C) = \left| M \cup \bigcup_{i=1}^{n} R_i \right|$$

képlettel számolható ki, ahol M az n metódussal rendelkező C osztály metódusainak a halmaza, míg R_i az osztály i-edik metódusa által közvetlenül hívott metódusok halmaza. Azaz az RFC metrika azon metódusok számát jelöli, amelyek végrehajtódhatnak válaszul egy, az adott osztályból példányosított objektum felé irányuló hívásnál.

- **Lack of Cohesion on Methods (LCOM)**: Adott egy C osztály, és jelölje $M_1, M_2, \ldots, M_n$ az osztály n darab metódusát, továbbá jelölje I_i az osztály azon

attribútumait, amelyeket az M_i metódus használ ($i = 1, 2, \ldots, n$). Legyen $P = \{(I_i, I_j) | I_i \cap I_j = \emptyset\}$ és $N = \{(I_i, I_j) | I_i \cap I_j \neq \emptyset\}$. Ekkor

$$LCOM(C) = \begin{cases} |P| - |N| & \text{ha } |N| \leq |P| \\ 0 & \text{különben} \end{cases}.$$

Azaz az LCOM metrika azon metóduspárok száma, amelyek nem használnak közös attribútumot, mínusz azon metóduspárok száma, amelyek használnak közös attribútumot.

- **Lack of Cohesion on Methods allowing Negative value (LCOMN)**: Az LCOMN definíciója megegyezik az LCOM definíciójával, kivéve hogy megengedjük a negatív értékeket is, azaz a fenti jelöléseket felhasználva $LCOMN(C) = |P| - |Q|$.
- **Logical Lines Of Code (LLOC)**: Egy osztály LLOC metrikája az osztály és annak metódusainak a nem üres és nem komment sorainak a száma.

A 3.1. táblázat mutatja a Mozilla 1.6-os verziójában található osztályok metrikáinak néhány alap statisztikáját. A *minimum* jelöli az adott metrika értékek közül a legkisebbet. Ezek közül egyedül az LCOMN kimagasló negatív értéke volt meglepetés, mert ez azt jelenti, hogy van egy nagyon koherens osztály. A *maximum* értékek már szélesebb skálán mozognak. Van olyan osztály, aminek 337 lokális metódusa van (NML), a legnagyobb osztály 9 371 sort tartalmaz (LLOC), míg a legnagyobb NOA érték 33. Annak ellenére, hogy a Mozilla egy nagy rendszer, ezek az értékek még így is extrém nagynak számítanak. A Mozillában közel háromezer osztály van, így elsőre meglepő lehet az, hogy egy osztálynak 1 214 közvetlen leszármazottja legyen, de mivel a második legnagyobb érték csak 115, és a harmadik 37, így azt feltételeztük, hogy az adott osztály valamiféle közös ősosztály a Mozilla forráskódjában, amiből a legtöbb osztály származik. A forráskód megvizsgálásával a feltevésünk beigazolódott, így a kiugró NOC érték elfogadható. Az osztályok átlagosan 183 sort (LLOC) és 17 metódust tartalmaznak, amely azt mutatja, hogy a Mozillában sok nagy osztály van. A NOA 3,1-es átlaga elsőre magasnak tűnhet, de ha figyelembe vesszük, hogy van egy közös ősosztály a Mozillában, akkor már elfogadható az érték. A nagy *szórás* értékek azt mutatják, hogy az osztályok igen változatosak, ami nem meglepő egy olyan rendszer esetében, amely több mint 3 000 osztályt tartalmaz.

	NML	NOA	NOC	CBO	RFC	LCOM	LCOMN	LLOC
Minimum	0,0	0,0	0,0	0,0	0,0	0,0	-54 614,0	0,0
Maximum	337,0	33,0	1 214,0	70,0	1 103,0	55 198,0	55 198,0	9 371,0
Median	9,0	2,0	0,0	6,0	30,0	21,0	21,0	57,0
Átlag	17,4	3,1	0,9	7,8	66,7	364,7	344,2	183,3
Szórás	25,3	3,4	21,7	8,4	97,0	1 875,4	2 113,6	425,3

3.1. táblázat. A Mozilla 1.6 osztályainak alap statisztikái

Annak ellenére, hogy ezek a metrikák az osztályok különböző tulajdonságait fejezik ki, megnéztük, hogy van-e kapcsolat köztük. A 3.2. táblázat tartalmazza a metrikák közti Pearson-féle lineáris korrelációs együtthatókat. Azt tapasztaltuk, hogy a NOC

kivételével az összes korreláció szignifikáns volt 0,01 szignifikancia szint mellett, ami azt jelenti, hogy a NOC és a többi metrika között nincs lineáris kapcsolat, viszont a többi metrika nem független. A NOA csak az RFC metrikával korrelál jelentősebben, míg a többi korrelációs értéke alacsony. Az LCOM és LCOMN közti magas korreláció várható volt, mivel a nem negatív értékekre ez a két metrika megegyezik. Ennek ellenére az LCOM metrikának magasabbak a korrelációs értékei, mint az LCOMN-nek. Azt tapasztaltuk, hogy az NML, a CBO, az RFC, az LCOM és az LLOC metrikák között jelentősebb korreláció figyelhető meg. Ezek alapján azt mondhatjuk, hogy ezek a metrikák nem teljesen függetlenek.

Mozilla	NOA	NOC	CBO	RFC	LCOM	LCOMN	LLOC
NML	0,16		**0,43**	**0,54**	**0,64**	**0,37**	**0,56**
NOA	1,00		0,17	**0,52**	0,09	0,07	0,08
NOC		1,00					
CBO			1,00	**0,48**	0,19	0,15	**0,58**
RFC				1,00	**0,33**	0,21	**0,40**
LCOM					1,00	**0,79**	**0,46**
LCOMN						1,00	**0,36**
LLOC							1,00

3.2. táblázat. A metrikák közti korrelációk

3.2. A metrikák hiba-előrejelző képességeinek vizsgálata

A vizsgálatok első lépéseként minden metrikához felállítottunk egy null-hipotézist és a hozzá tartozó alternatív hipotézist, amelyek a metrikák és az osztályok hibára való hajlamossága közti összefüggést írják le. A továbbiakban azt fogjuk vizsgálni, hogy mely esetekben fogadhatjuk el a null-hipotézist, és mikor kell elvetnünk azt, elfogadva helyette az alternatív hipotézist helyesnek. A nyolc metrikához tartozó null-hipotézis és a hozzájuk tartozó alternatív hipotézis a következő:

- **NML hipotézis**: Azok az osztályok, amelyeknek több metódusuk van, hajlamosabbak a hibákra, mint azok az osztályok, amelyeknek kevesebb metódusuk van. (Null-hipotézis: Azok az osztályok, amelyeknek több metódusuk van, nem hajlamosabbak jobban a hibákra, mint azok az osztályok, amelyeknek kevesebb metódusuk van.)

- **NOA hipotézis**: Azok az osztályok, amelyeknek több ősosztályuk van, hajlamosabbak a hibákra, mint azok az osztályok, amelyeknek kevesebb ősosztályuk van. (Null-hipotézis: Azok az osztályok, amelyeknek több ősosztályuk van, nem hajlamosabbak jobban a hibákra, mint azok az osztályok, amelyeknek kevesebb ősosztályuk van.)

- **NOC hipotézis**: Azok az osztályok, amelyek több gyerekosztállyal rendelkeznek, kevésbé hajlamosak a hibákra, mint azok, amelyek kevesebb gyerekosztállyal ren-

delkeznek[1]. (Null-hipotézis: Azok az osztályok, amelyek több gyerekosztállyal rendelkeznek, nem hajlamosak kevésbé a hibákra, mint azok, amelyek kevesebb gyerekosztállyal rendelkeznek.)

- **CBO hipotézis**: Azok az osztályok, amelyek több másik osztályhoz kapcsolódnak, hajlamosabbak a hibákra, mint azok, amelyek kevesebb osztályhoz kapcsolódnak. (Null-hipotézis: Azok az osztályok, amelyek több másik osztályhoz kapcsolódnak, nem hajlamosabbak jobban a hibákra, mint azok, amelyek kevesebb osztályhoz kapcsolódnak.)

- **RFC hipotézis**: Azok az osztályok, amelyeknél több metódus hajtódhat végre egy kérésre válaszul, hajlamosabbak a hibákra, mint azok, amelyeknél kevesebb metódus hajtódhat végre. (Null-hipotézis: Azok az osztályok, amelyeknél több metódus hajtódhat végre egy kérésre válaszul, nem hajlamosabbak jobban a hibákra, mint azok, amelyeknél kevesebb metódus hajtódhat végre.)

- **LCOM és LCOMN hipotézisek**: Azok az osztályok, amelyek metódusai nem koherensek, hajlamosabbak a hibákra, mint azok, amelyek metódusai koherensek. (Null-hipotézis: Azok az osztályok, amelyek metódusai nem koherensek, nem hajlamosabbak jobban a hibákra, mint azok, amelyek metódusai koherensek.)

- **LLOC hipotézis**: Azok az osztályok, amelyek több programsort tartalmaznak, hajlamosabbak a hibákra, mint azok, amelyek kevesebbet tartalmaznak. (Null-hipotézis: Azok az osztályok, amelyek több programsort tartalmaznak, nem hajlamosabbak jobban a hibákra, mint azok, amelyek kevesebbet tartalmaznak.)

A következőkben ismertetjük azokat a statisztikai és gépi tanulási módszereket, amelyek segítségével vizsgáltuk a metrikák és az osztályok hibára való hajlamossága közti kapcsolatot. Először *regressziós analízist* alkalmaztunk, amit gyakran használnak olyan esetekben, amikor egy ismeretlen változót kell egy vagy több ismert változó segítségével becsülni. A *logisztikus regresszió* segítségével vizsgáltuk a metrikák és az osztályok hibára való hajlamossága közti kapcsolatot, pontosabban hogy az adott osztály tartalmazott-e hibát vagy sem, míg a *lineáris regresszióval* már az osztályokban található hibaszámokat becsültük. Ezek után *döntési fát* és *neuron-hálót* alkalmaztunk, melyek segítségével ismét a metrikák és az osztályok hibára való hajlamosságának kapcsolatát elemeztük. A következő alfejezetekben ezeket a vizsgálatokat fogjuk részletesen ismertetni.

3.2.1. Logisztikus regresszió

A logisztikus regresszió esetében az ismeretlen változó, vagy más néven a *függő változó* csak két különböző értéket vehet fel, ezért az osztályokat két csoportra osztottuk aszerint, hogy tartalmazott-e hibát vagy sem. Így az osztályokban található hibák száma helyett az osztályok hibára való hajlamosságát vizsgáltuk. A metrikák[2] voltak az ismert változók, vagy más néven a *magyarázó változók*.

[1] Basili és munkatársai [4] azt tapasztalták, hogy minél több leszármazottja van egy osztálynak, annál kevesebb benne a hiba, így a hipotézist ennek megfelelően mondtuk ki.

[2] Korábban már láttuk (3.1. ábra), hogy a metrikák nagyon különböző skálán változtak, ezért normalizáltuk azokat, hogy össze lehessen hasonlítani a kapott eredményeket.

A többváltozós logisztikus regresszió a

$$\pi(X_1, X_2, \ldots, X_n) = \frac{e^{C_0 + C_1 \cdot X_{i_1} + \cdots + C_n \cdot X_{i_n}}}{1 + e^{C_0 + C_1 \cdot X_{i_1} + \cdots + C_n \cdot X_{i_n}}}$$

képlettel írható le, ahol az X_i-k a magyarázó változók, π pedig annak a valószínűsége, hogy az adott osztály hibás. Az egyváltozós logisztikus regresszió a többváltozós azon speciális esete, amikor csak egy magyarázó változó van. Mivel a logisztikus regresszió egy széles körben alkalmazott módszer, ezért nem fogjuk részletesebben ismertetni, további részletek megtalálhatók Hosmer és Lemeshow [33] könyvében.

Első lépésként egyváltozós logisztikus regressziót alkalmaztunk minden metrikára külön-külön. Az eredményeket a 3.3. táblázat mutatja. Az *együttható* a becsült regressziós együttható. Minél nagyobb ennek az együtthatónak az abszolút értéke, annál nagyobb a magyarázó változó befolyása arra, hogy mekkora valószínűséggel találunk az osztályban hibát. Ha az együttható előjele negatív, akkor a kapcsolat fordított irányú, azaz minél nagyobb lesz a magyarázó változó értéke, annál kisebb lesz a valószínűsége, hogy az osztály tartalmaz hibát. A *p-érték* annak eldöntésében segít, hogy az adott összefüggés szignifikáns-e vagy sem. Ha ez a p-érték egy előre választott α *szignifikancia szintnél* nem kisebb ($\alpha \leq$ p-érték), akkor elfogadjuk a null-hipotézist, viszont ha a p-érték kisebb a választott α értéknél ($\alpha >$ p-érték), akkor elvetjük a null-hipotézist és helyette az alternatív hipotézist fogadjuk el. Az általunk választott α küszöbérték 0,05 volt. Az R^2 érték azt mutatja meg, hogy a függő változó változásai mennyire magyarázhatók a független változó(k) által. Minél nagyobb az R^2 értéke, annál nagyobb a függő változó befolyása a független változó változására.

	NML	NOA	NOC	CBO	RFC	LCOM	LCOMN	LLOC
Együttható	1,069	0,598	-0,041	1,082	0,868	2,120	1,400	1,642
Konstans	-0,252	-0,316	-0,321	-0,283	-0,278	-0,135	-0,248	-0,162
p-érték	0,000	0,000	0,551	0,000	0,000	0,000	0,000	0,000
R^2	0,114	0,067	0,000	0,152	0,108	0,076	0,049	0,128

3.3. táblázat. Az egyváltozós logisztikus regresszió eredménye

Ezeket az eredményeket figyelembe véve (3.3. táblázat) azt mondhatjuk, hogy a nyolc vizsgált metrikából hét esetben szignifikáns kapcsolatot találtunk. Egyedül a NOC metrika esetében kaptunk nem szignifikáns összefüggést (p-érték = 0,551). A CBO metrikához tartozik a legnagyobb R^2 érték (nagyobb, mint az LLOC metrikáé), ami azt sugallja, hogy a CBO metrika a legjobb prediktor. A NOA, LCOM és LCOMN metrikákhoz tartozó R^2 értékek lényegesen kisebbek a többinél, ezért ezek a metrikák kevésbé hasznosak[3].

Korábban láttuk, hogy a metrikák nem teljesen függetlenek (3.2. táblázat), így redundás információkat reprezentálnak. Ezért nem kell az összes metrika a többváltozós regresszióhoz, így *lépésenkénti kiválasztást* (stepwise selection) alkalmaztunk, hogy kiválasszuk a szükségeseket. A CBO, NOA, NML és LLOC metrikák lettek kiválasztva 0,05 szignifikancia szint mellett ebben a sorrendben. Az R^2 értéke a többváltozós logisztikus regresszionak 0,175, ami csak egy picit nagyobb, mint az egyváltozós vizsgálatnál a CBO metrika esetében kapott érték.

[3]Megjegyezzük, hogy a logisztikus regresszió esetében a magas R^2 értékek ritkák, szemben például a legkisebb nyégyzetek módszerével.

	konstans	CBO	NOA	NML	LLOC
Együttható	-0,229	0,631	0,310	0,246	0,461
p-érték	0,003	0,000	0,000	0,000	0,002

3.4. táblázat. A többváltozós logisztikus regresszió eredménye

A logisztikus regresszió nem csak azt mondja meg, hogy van-e összefüggés a metrikák és a hibára való hajlamosság között, hanem minden egyes vizsgálathoz megad egy modellt is. Ezeket a modelleket alkalmaztuk 0,5 küszöbértékkel, ami azt jelenti, hogy ha $0,5 < \pi$, akkor az osztályt hibásnak osztályozzuk, különben hibamentesnek. 3.5. táblázat tartalmazza a többváltozós logisztikus regresszióhoz tartozó modell osztályozási eredményét. A zárójelben lévő számok azon hibák száma, amelyek az adott kategóriába sorolt osztályokban találhatóak.

	Előrejelzett	
Tapasztalt	Nem hibás	Hibás
Nem hibás	1 624	226
Hibás	744 (1 377)	598 (2 584)

3.5. táblázat. A többváltozós logisztikus regressziós modell eredménye

A táblázatból leolvasható, hogy a 3 192 osztályból a modell 2 222 (1 624+598) osztályt osztályozott helyesen, ami azt jelenti, hogy a modell *helyessége* (correctness) 69,61% (2 222/3 192).

A modell helyessége is fontos, azonban van két másik mennyiség, ami a tesztelés szempontjából sokkal fontosabb. Az első a *pontosság* (precision), ami azt mondja meg, hogy a hibásan osztályozott osztályok hány százaléka hibás valójában, azaz a hibásnak jelzett és valóban hibás osztályok számát el kell osztani az összes hibásnak jelzett osztályok számával. Minél nagyobb a modell *pontossága*, annál kevesebb hibamentes osztályt jelez a modell hibásnak, így kevesebb hibamentes osztályt kell tesztelni, ami növeli a tesztelés hatékonyságát. Ebben az esetben 824 (226+598) osztályt jelzett a modell hibásnak, de ezekből csak 598 volt valójában hibás, azaz a modell *pontossága* 72,57%.

A másik fontos kérdés, hogy a hibáknak hány százalékát találjuk meg az adott módszerrel. Ezt az értéket a *teljesség* (completeness) mutatja meg, amit úgy számolunk ki, hogy a hibásnak jelzett osztályokban talált hibák számát elosztjuk az összes osztályban található összes hiba számával. Minél nagyobb ez az érték, annál több hibát találunk meg. Habár a többváltozós regressziós modell a hibás osztályoknak kevesebb, mint a felét találta meg (598-at az 1 342-ből (744+598)), mégis ez a kisebb csoport tartalmazott 2 584 hibát a 3 961-ből, így a modell teljesség értéke 65,24%.

A 3.6. táblázat tartalmazza az egyváltozós logisztikus regressziós modellek helyesség, pontosság és teljesség értékeit, míg az utolsó sorban a többváltozós modell eredményei szerepelnek. Mivel a NOC metrika esetében a modell minden osztályt hibamentesnek osztályzott, így nem volt értelme kiszámolni a pontosság és teljesség értékeket. Az LCOM és LCOMN metrikáknak nagyon magas pontosság értékei vannak (81,34% és 85,02%), ami azt jelenti, hogy a hibamentes osztályoknak csak egy kis része (18,66% és 14,98%) lett hibásnak osztályozva. Ez nagyon jó, viszont az alacsony teljesség értékek

Metrika	Helyesség	Pontosság	Teljesség
NML	65,38%	68,84%	55,24%
NOA	64,04%	65,06%	45,17%
NOC	57,96%	–	–
CBO	69,77%	70,38%	69,12%
RFC	66,01%	71,89%	53,60%
LCOM	64,69%	81,34%	43,68%
LCOMN	63,82%	85,02%	39,01%
LLOC	66,85%	72,98%	54,58%
Többv. regr.	69,61%	72,57%	65,24%

3.6. táblázat. Helyesség, pontosság és teljesség értékek a Mozilla 1.6 esetében

(43,68% és 39,01%) azt jelentik, hogy a hibáknak csak egy kisebb részét lehet megtalálni ezen metrikákon alapuló modellek segítségével. A NOA metrika teljesség értéke szintén alacsony (45,17%), amihez gyengébb helyesség (64,04%) és pontosság (65,06%) értékek párosulnak. A NOA, LCOM és LCOMN metrikák gyengébb értékei összhangban állnak az alacsonyabb R^2 értékekkel (3.3. táblázat). Az NML, RFC és LLOC metrikák értékei többé-kevésbé azonosak és jónak mondhatók. A CBO metrikához tartozó modellnek voltak a legnagyobb helyesség és teljesség értékei – nagyobbak, mint a többváltozós modellé – míg a pontosság értéke szintén jó, ami megerősíti, hogy a CBO metrika a legjobb prediktor.

Mozilla 1.0	Helyesség	Pontosság	Teljesség
Többv. regr.	57,96%	83,76%	55,94%

3.7. táblázat. Helyesség, pontosság és teljesség értékek a Mozilla 1.0 esetében

Megnéztük, hogy a Mozilla 1.6-os verzióján kialakított modell milyen eredményt ad a Mozilla 1.0-ás verzióján. Azért ezt a verziót választottuk, mert a vizsgált verziók közül ez volt időben legtávolabb az általunk kiválasztott 1.6-os verziótól. A 3.7. táblázat mutatja a többváltozós logisztikus regressziós modell eredményét. Annak ellenére, hogy a helyesség és a teljesség értékek romlottak, a modell még mindig használható.

3.2.2. Lineáris regresszió

A Mozilla esetében sok olyan osztály van, amely több hibát tartalmazott, és a hibák száma is széles skálán mozgott (lásd 2.3. táblázat). Ezt az információt a logisztikus regresszió segítségével nem tudtuk modellezni, ezért lineáris regressziót is alkalmaztunk. A lineáris regresszió esetében a magyarázó változók szintén a normalizált metrikák voltak, míg a függő változó az osztályokban található hibák száma volt.

Az egyváltozós lineáris regresszió eredményét a 3.8. táblázat mutatja. Hasonlóan a logisztikus regresszió eredményéhez, itt is hét metrika bizonyult szignifikánsnak, és egyedül a NOC metrika nem volt az. A CBO metrikának van a legnagyobb R^2 értéke, igaz csak egy kicsit jobb, mint az LLOC metrikáé. Hasonlóan az egyváltozós logisztikus regresszió eredményéhez, itt is a NOA, az LCOM és az LCOMN metrikák rendelkeznek a legkisebb R^2 értékekkel.

	NML	NOA	NOC	CBO	RFC	LCOM	LCOMN	LLOC
Együttható	1,641	1,082	-0,018	1,712	1,533	1,328	1,147	1,694
Konstans	1,241	1,241	1,241	1,241	1,241	1,241	1,241	1,241
p-érték	0,000	0,000	0,728	0,000	0,000	0,000	0,000	0,000
R^2	0,321	0,139	0,000	0,349	0,280	0,210	0,157	0,342

3.8. táblázat. Az egyváltozós lineáris regresszió eredménye

	konstans	CBO	NML	LLOC	NOA	RFC
Együttható	1,241	0,744	0,597	0,678	0,505	-0,214
p-érték	0,000	0,000	0,000	0,000	0,000	0,012

3.9. táblázat. A többváltozós lineáris regresszió eredménye

A többváltozós lineáris regresszió eredményét a 3.9. táblázat mutatja. Az öt kiválasztott metrikából négy megegyezik a logisztikus regressziónál kiválasztottal, továbbá itt is a CBO metrika lett először kiválasztva, igaz a többi metrikának a kiválasztási sorrendje eltér. Az R^2 érték 0,43, ami azt jelenti, hogy a modell a változások 43%-át magyarázza.

3.2.3. Gépi tanulási módszerek

A regressziós analízis után a gépi tanulási módszerek eredményeit ismertetjük. Kétféle gépi tanulási módszert alkalmaztunk (C4.5 algoritmust a döntési fa építésére [43], illetve neuron-hálót [5]), hogy metrikákon alapuló modelleket készítsünk, amelyek előrejelzik a várható hibák számát az osztályokban. Két különböző megközelítést választottunk az osztályok minőségének meghatározására. Először a logisztikus regressziónál ismertetett besorolást alkalmaztuk, azaz az osztályokat két csoportra osztottuk, míg másodjára négy kategóriába soroltuk az osztályokat hibaszámuk alapján. A két felosztás pontos definíciója a következő:

1. *Az osztályokat két csoportra osztottuk*: az egyik kategóriába a hibamentes osztályok tartoztak, a másikba azok, amelyek tartalmaztak legalább egy hibát. A <0,1-34> jelölést fogjuk használni ezen csoportosításra.

2. *Az osztályokat négy csoportra osztottuk*: az első csoportba tartoztak a hibamentes osztályok, a másodikba azok, amelyek csak egy hibát tartalmaztak, a harmadikba azok, amelyeknek a hibaszáma 2 és 12 közé esett, míg a negyedik kategóriát a legalább 13 hibát tartalmazó osztályok alkották. A <0,1,2-12,13-34> jelölést fogjuk használni ezen csoportosításra.

A vizsgálatok során a *ten-fold cross-validation* módszert alkalmaztuk, ami azt jelenti, hogy a vizsgált osztályokat tíz egyenlő részre osztottuk, majd a tanulást tízszer megismételtük úgy, hogy minden alkalommal kiválasztottunk egy részt tesztelésre, és a maradék kilenc-tized részen végeztük a tanítást. Ezután kiszámoltuk a tíz különböző teszteredmény átlagát és szórását. Ezt a módszert alkalmaztunk mind a két gépi tanulás esetében.

A 3.10. táblázat mutatja az átlagos helyesség értékeket és zárójelben a szórásokat azon gépi tanulások esetében, amikor az összes metrikát felhasználtuk. Megvizsgáltuk a modelleket, és azt tapasztaltuk, hogy azokban az esetben követik el a legtöbb hibát,

Osztályozás	Döntési fa	Neuron háló
<0,1-34>	69,58%(4,92%)	68,77%(4,93%)
<0,1,2-12,13-34>	62,91%(4,43%)	63,75%(4,25%)

3.10. táblázat. A gépi tanulási modellek átlagos helyesség értékei és szórásai

ha az osztály egy vagy két hibát tartalmazott, és sokkal megbízhatóbbak azokban az esetekben, ha az osztály hibamentes, vagy legalább három hibát tartalmazott.

Metrikák	Döntési fa	Neuron háló
NML	66,51%(4,32%)	66,20%(4,33%)
NOA	63,66%(3,77%)	63,47%(3,40%)
NOC	57,95%(6,00%)	57,96%(6,00%)
CBO	69,77%(4,63%)	69,46%(4,60%)
RFC	66,45%(5,06%)	66,76%(4,60%)
LCOM	66,67%(4,10%)	66,17%(3,71%)
LCOMN	66,67%(4,10%)	67,17%(3,90%)
LLOC	67,98%(4,93%)	67,58%(4,58%)

3.11. táblázat. A tanuláshoz tartozó helyesség értékek (<0,1-34>)

Annak eldöntésére, hogy melyik metrikák képesek a hibákat hatékonyan előrejelezni, a gépi tanulási módszerek hatékonyságát megvizsgáltuk a metrikákra külön-külön is. A <0,1-34> osztályozáshoz tartozó eredményeket a 3.11. táblázat tartalmazza, míg a 3.12. táblázat mutatja a <0,1,2-12,13-34> osztályozás eredményeit.

Metrikák	Döntési fa	Neuron háló
NML	62,91%(4,50%)	62,22%(4,59%)
NOA	58,45%(6,13%)	60,15%(4,39%)
NOC	57,95%(6,00%)	57,96%(6,00%)
CBO	63,79%(4,68%)	63,25%(5,08%)
RFC	61,94%(4,24%)	62,16%(4,25%)
LCOM	62,75%(4,22%)	62,16%(4,80%)
LCOMN	62,75%(4,22%)	62,16%(4,58%)
LLOC	62,78%(5,94%)	62,44%(5,58%)

3.12. táblázat. A tanuláshoz tartozó helyesség értékek (<0,1,2-12,13-34>)

A táblázatokból leolvasható, hogy mindkét modell esetében a CBO metrikának van a legnagyobb helyesség értéke. Az első osztályozás (<0,1-34>) esetében az átlagos helyesség érték a döntési fára 69,77%, míg a neuron háló esetében 69,46%, azaz mindkettő egy kicsit jobb, mint az a modell, ahol az összes metrikát felhasználtuk (ahol ez az érték 69,58% volt a döntési fára, és 68,77% a neuron háló esetében).

A 3.2.1. alfejezetben bemutatott pontosság és teljesség értékeket is meghatároztuk a gépi tanulási módszerek eredményeire. Az eredmények a 3.13. táblázatban láthatóak, mivel azonban a pontosság és a teljesség definíciói csak a <0,1-34> osztályozás esetében értelmezhetőek, ezért csak ezekre számoltuk ki. A táblázatban a tíz különböző esethez

Pontosság	Döntési fa	Neuron háló
NML	62,34%(10,21%)	65,75%(12,51%)
NOA	63,13%(12,70%)	61,36%(17,35%)
NOC	-	-
CBO	69,13%(11,88%)	70,63%(10,84%)
RFC	65,15%(13,46%)	63,99%(10,07%)
LCOM	63,96%(12,07%)	63,92%(12,50%)
LCOMN	63,96%(12,07%)	66,70%(11,09%)
LLOC	66,81%(08,37%)	65,29%(08,05%)
Többv.	68,38%(11,47%)	68,94%(13,46%)
Teljesség	Döntési fa	Neuron háló
NML	65,33%(10,80%)	60,19%(08,59%)
NOA	41,09%(18,24%)	40,52%(18,08%)
NOC	-	-
CBO	67,02%(11,88%)	65,13%(11,81%)
RFC	56,91%(15,64%)	61,66%(16,13%)
LCOM	60,59%(12,66%)	60,36%(13,34%)
LCOMN	60,59%(12,66%)	60,62%(09,24%)
LLOC	64,41%(10,95%)	65,85%(12,28%)
Többv.	67,84%(11,96%)	64,76%(11,43%)

3.13. táblázat. A pontosság és teljesség értékek az egyes modellekre (<0,1-34>)

tartozó pontosság és teljesség értékek átlagai és szórásai találhatók. A táblázatban külön-külön szerepel az egyes metrikákhoz tartozó modellek eredménye, valamint az utolsó sorban feltüntettük a többváltozós modellek eredményeit is.

Az eredményeket megvizsgálva azt tapasztaltuk, hogy ismét a CBO metrika bizonyult a legjobbnak. A döntési fa esetében a CBO helyesség értéke (69,13%) meglepő módon nagyobb, mint a többváltozós modell helyesség értéke (68,38%), míg a teljesség értéke (67,02%) nem sokkal marad el a többváltozós modell teljesség értékétől (67,84%). A neuron háló esetében szintén a CBO metrika rendelkezik a legnagyobb pontosság értékkel, továbbá a teljesség értéke is nagyobb, mint a többváltozós modellé, bár a legnagyobb értékkel az LLOC metrika rendelkezik. A NOC esetében nem tudtuk kiszámolni a pontosság és teljesség értékeket, mert amikor a NOC metrikát használtuk a tanulás során, akkor mindkét modell az összes osztályt hibamentesre osztályozta. A NOA metrika pontosság értéke (63,13% és 61,36%) csak egy kicsit gyengébb, mint a többi modell pontosság értéke, viszont a teljesség értékek nagyon gyengék (41,09% és 40,52%). Összességében azt mondhatjuk, hogy a gépi tanulások esetében CBO a legjobb metrika, míg a NOC metrika nem használható hiba-előrejelzésre, és a NOA eredményei is eléggé gyengék, legalábbis a többi vizsgált metrikával összehasonlítva.

Osztályozás	Döntési fa	Neuron háló
<0,1-34>	58,01%(0,41%)	57,81%(0,64%)
<0,1,2-12,13-34>	50,88%(0,44%)	51,28%(0,27%)

3.14. táblázat. A többváltozós modellek helyesség értékei a Mozilla 1.0-án

Modell pontossága	Döntési fa	Neuron háló
Pontosság	80,95%(0,64%)	81,03%(2,28%)
Teljesség	58,15%(1,10%)	57,05%(2,06%)

3.15. táblázat. A pontosság és teljesség értékek a Mozilla 1.0-án (<0,1-34>)

Végezetül itt is megvalósítottuk a Mozilla 1.6-os verzióján kialakított többváltozós modellek kiértékelését a Mozilla 1.0-ás verzióján. A 3.14. és 3.15. táblázatok tartalmazzák a többváltozós modellek helyesség, pontosság és teljesség értékeit. A konklúziónk hasonló a 3.2.1. alfejezetben elmondottakhoz, azaz a modellek eredményei romlottak egy keveset a pontosság értékek kivételével, de még így is alkalmasak a hibák előrejelzésére.

3.3. A hipotézisek tárgyalása

A modellek és eredményeik részletes ismertetése után a 3.2. alfejezetben kimondott hipotéziseket fogjuk egyesével tárgyalni. Ehhez nem csak a regressziós analízisek eredményeit használjuk fel, hanem a helyesség, pontosság és teljesség értékeket is figyelembe vesszük. Ezek az értékek ki lettek számolva a logisztikus regresszió, a döntési fa és a neuron háló esetében is (míg a lineáris regressziónál nem lehetett értelmezni azokat). A 3.16. táblázatban összefoglaljuk a három modell ezen értékeit.

- **NML hipotézis**: Az NML szignifikáns volt a regressziós analízisek során, és a hozzá tartozó R^2 érték sem sokkal maradt el a legjobbaktól. A gépi tanulási eredmények is hasonlóak a regresszió eredményeihez, a helyesség értékek közel azonosak (66% körüliek), míg a pontosság és teljesség értékek ingadoznak egy kicsit, de egyik sem mondható sokkal jobbnak vagy rosszabbnak a másiknál.

  Az NML metrika esetében mind a négy analízis közel azonos eredményt hozott, így elvetjük a null-hipotézist, és helyette az alternatív hipotézist fogadjuk el.

- **NOA hipotézis**: A NOA metrika is szignifikáns volt mindkét regresszió esetén, bár a hozzá tartozó R^2 értékek lényegesen kisebbek voltak a legjobb értékeknél, továbbá a helyesség érték is csak 64% volt. A gépi tanulás esetében 64% alatt volt a helyesség érték, ami jelentősen kisebb a többi metrika értékénél (kivéve a NOC metrikát). A pontosság értékek szintén elmaradnak egy picit a többi metrika pontosság értékétől (bár itt jóval nagyobb a szórása az értékeknek), míg a teljesség értékek messze a legrosszabbak. Figyelembe véve mindhárom értéket azt mondhatjuk, hogy a logisztikus regresszió által előállított modell jobb eredményeket ért el, mint a gépi tanulási módszerek.

  Az eredményeket összefoglalva azt mondhatjuk, hogy elvetjük a NOA metrikához tartozó null-hipotézist, és az alternatív hipotézist fogadjuk el helyette. Azonban meg kell jegyeznünk, hogy a NOA nem olyan jó prediktor, mint a többi metrika, és a hasznosságának eldöntéséhez további vizsgálatok szükségesek.

- **NOC hipotézis**: A vizsgálatok során azt tapasztaltuk, hogy a NOC metrika sem a logisztikus, sem a lineáris regresszió esetében nem volt szignifikáns. A gépi tanulási módszerek esetében olyan modelleket kaptunk, amelyek minden osztályt

Metrika	Modell	Helyesség	Pontosság	Teljesség
NML	Logisztikus r.	65,38%	68,84%	55,24%
	Döntési fa	66,51%	62,34%	65,33%
	Neuron háló	66,20%	65,75%	60,19%
NOA	Logisztikus r.	64,04%	65,06%	45,17%
	Döntési fa	63,66%	63,13%	41,09%
	Neuron háló	63,47%	61,36%	40,52%
NOC	Logisztikus r.	57,96%	–	–
	Döntési fa	57,95%	–	–
	Neuron háló	57,96%	–	–
CBO	Logisztikus r.	69,77%	70,38%	69,12%
	Döntési fa	69,77%	69,13%	67,02%
	Neuron háló	69,46%	70,63%	65,13%
RFC	Logisztikus r.	66,01%	71,89%	53,60%
	Döntési fa	66,45%	65,15%	56,91%
	Neuron háló	66,76%	63,99%	61,66%
LCOM	Logisztikus r.	64,69%	81,34%	43,68%
	Döntési fa	66,67%	63,96%	60,59%
	Neuron háló	66,17%	63,92%	60,36%
LCOMN	Logisztikus r.	63,82%	85,02%	39,01%
	Döntési fa	66,67%	63,96%	60,59%
	Neuron háló	67,17%	66,70%	60,62%
LLOC	Logisztikus r.	66,85%	72,98%	54,58%
	Döntési fa	67,98%	66,81%	64,41%
	Neuron háló	67,58%	65,29%	65,85%
Többv.	Logisztikus r.	69,61%	72,57%	65,24%
	Döntési fa	69,58%	68,38%	67,84%
	Neuron háló	68,77%	68,94%	64,76%

3.16. táblázat. A helyesség, pontosság és teljesség értékek összefoglalása

hibamentesnek osztályoztak, ami alátámasztja a statisztikai módszereknél kapott eredményt. A szélsőséges osztályozások eredménye az 57,96%-os helyesség érték (az osztályok ekkora része hibamentes), míg a pontosság és teljesség értékeket nincs értelme kiszámolni.

Ezek alapján a NOC metrika esetében elfogadhatjuk a null-hipotézist, azaz ezen kísérlet alapján a NOC metrika nem alkalmas a hibák előrejelzésére.

- **CBO hipotézis**: Amellett, hogy a CBO metrika mind a logisztikus, mind a lineáris regresszió esetében szignifikáns volt, ez a metrika bizonyult a legjobbnak a vizsgált metrikák közül. A gépi tanulási módszerek is megerősítették ezeket az eredményeket, mivel majdnem minden esetben a CBO metrikának volt a legjobb a helyesség, pontosság és teljesség értéke (az egyetlen kivétel a neuron hálónál a teljesség érték volt, mert ott az LLOC értéke egy picit jobb volt). Mi több, a CBO metrika helyesség értékei még a többváltozós modell értékeinél is jobbak voltak, míg a pontosság és teljesség értékek is jobbnak bizonyultak több esetben.

  Így a CBO esetében nem csak azt mondhatjuk, hogy elvetjük a null-hipotézist

és elfogadjuk az alternatív-hipotézist, hanem azt is kijelenthetjük, hogy a nyolc vizsgált metrika közül a CBO a legjobb minden tekintetben.

- **RFC hipotézis**: Az RFC szintén szignifikáns volt mindkét regresszió esetében. A gépi tanulásokhoz tartozó helyesség értékek közel azonosak a logisztikus regresszionál kapott értékekkel (66%), továbbá a többi metrikával összehasonlítva egyike a legjobbaknak. A három modell pontosság és teljesség értékei szélesebb skálán mozognak, és az átlagnál jobbnak mondhatóak.

  Ezeket figyelembe véve elvetjük a null-hipotézist az RFC metrika esetében, és az alternatív hipotézist fogadjuk el.

- **LCOM hipotézis**: A statisztikai vizsgálatok eredményei alapján azt mondhatjuk, hogy az LCOM metrika szignifikáns. Amíg a helyesség értékek közel azonosak (64-66%) voltak, addig a logisztikus regresszióhoz tartozó pontosság érték jelentősen nagyobb, mint a gépi tanulási modellek értékei, míg a teljesség értékeknél a gépi tanulási módszerek teljesítenek jobban. Ezekre a nagy eltérésekre nem tudunk magyarázatot adni, így ennek kiderítéséhez további vizsgálatok szükségesek.

  Összességében azt mondhatjuk, hogy az LCOM esetében elvetjük a null-hipotézist, és elfogadjuk az alternatív hipotézist.

- **LCOMN hipotézis**: Az LCOM metrikához hasonlóan az LCOMN metrika is szignifikáns volt mindkét regresszió esetében, csak az R^2 együtthatói voltak rosszabbak, mint a LCOM metrikáé. A regressziós modell helyesség értéke közel azonos az LCOM értékével, míg a pontosság értéke nagyobb, és a teljesség értéke kisebb az LCOM hasonló értékeinél. A gépi tanulás esetében az LCOM és LCOMN metrikákhoz tartozó értékek közel azonosak. Ezekből is látszik, hogy az LCOM és LCOMN metrika nagyon hasonlít egymásra, és ezek alapján nem lehet eldönteni, hogy a kettő közül melyik metrika a jobb.

  Az LCOM-hoz hasonlóan az LCOMN esetében is elvetjük a null-hipotézist, és az alternatív hipotézist fogadjuk el.

- **LLOC hipotézis**: Az LLOC metrika mindkét regresszió esetén szignifikáns volt, és csak a CBO metrika volt jobb. Az LLOC metrika helyesség értékei jók voltak (csak a CBO értékei voltak jobbak), és a pontosság és teljesség értékek is nagyon jók voltak, amiből következik, hogy az LLOC metrika az egyik legjobb.

  Az összes elemzés ugyanazt az eredményt hozta, amely alapján elvetjük a null-hipotézist, és megtartjuk az alternatív-hipotézist.

3.4. A Mozilla fejlődésének tanulmányozása

Az eredményeket felhasználva megvizsgáltuk, hogy a Mozilla hogyan változott az 1.0-ás verziótól, amit 2002. júniusában adtak ki, az 1.6-os verzióig, ami 2004. januárjában jelent meg. A 3.17. táblázatban láthatók a Mozilla különböző verzióban található osztályok hibaszámai közti Pearson-féle korrelációs együtthatók, ahol az összes korreláció szignifikáns volt 0,01 szignifikancia szint mellett. Amint azt láthatjuk, ezek a korrelációs értékek igen magasak még az időben távol lévő verziók között is. Továbbá az 1.6-os verzió adatai segítségével kialakított modelljeinket teszteltük az 1.0-ás verzión

(lásd 3.2.1. és 3.3. alfejezet), ami alapján azt mondhatjuk, hogy a helyesség és teljesség értékek csökkenése ellenére a modellek még mindig használhatóak többek között a nem romló pontosság értékek miatt.

ver.	1.0	1.1	1.2	1.3	1.4	1.5	1.6
1.0	1.00	0.89	0.84	0.76	0.73	0.69	0.70
1.1		1.00	0.90	0.81	0.77	0.74	0.73
1.2			1.00	0.85	0.81	0.77	0.73
1.3				1.00	0.83	0.75	0.74
1.4					1.00	0.84	0.76
1.5						1.00	0.80
1.6							1.00

3.17. táblázat. A Mozilla osztályainak hibaszámai közti korrelációs értékek

A különböző verziók esetében megvizsgáltuk a metrikák eloszlását is, azonban a nagy mennyiségű adatból nehéz lenne komolyabb következtetéseket levonni. Ezért úgy döntöttünk, hogy „összevont" adatokon fogunk dolgozni annak ellenére, hogy tudtuk, ezekből nem lehet olyan megbízható következtetéseket levonni, mint az eredeti adatokból, viszont a tendenciák így is kivehetőek. A 3.18. táblázat tartalmazza a metrikák várható értékét (V. é.) és szórását (Sz.) a Mozilla összes vizsgált verziójára, míg a 3.1. ábra ugyanazt mutatja grafikus formában[4]. A szórások egyes metrikáknál igen nagyok, de ez nem meglepő ilyen nagy rendszer esetében, mint a Mozilla, amely több mint 3 000 osztályt tartalmaz.

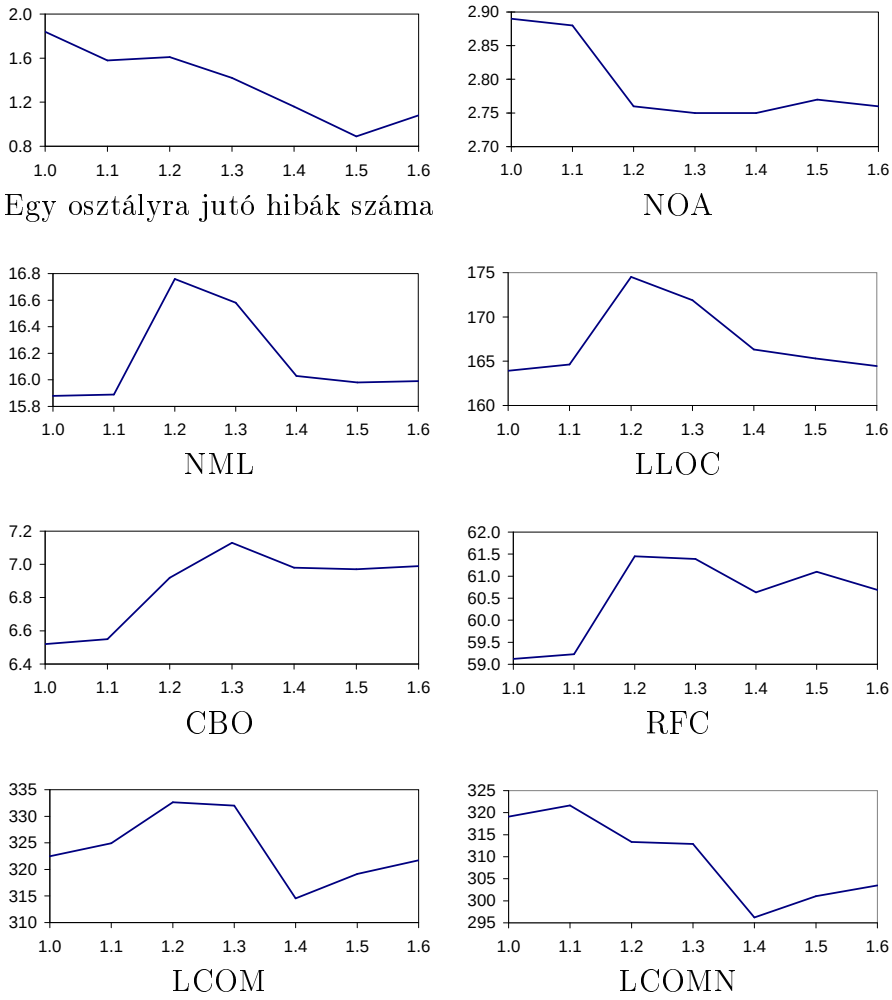
Ver.	Oszt. száma	Hibák száma	Hiba per osztály	NML		NOA		RFC	
				V. é.	Sz.	V. é.	Sz.	V. é.	Sz.
1.0	3 585	6612	1,84	**15,88**	23,68	**2,89**	2,97	**59,12**	86,01
1.1	3 624	5720	1,58	**15,89**	23,77	**2,88**	2,98	**59,23**	86,19
1.2	3 451	5549	1,61	**16,76**	24,31	**2,76**	3,03	**61,45**	88,53
1.3	3 491	4960	1,42	**16,58**	24,28	**2,75**	3,06	**61,39**	89,04
1.4	3 666	4243	1,16	**16,03**	23,79	**2,75**	3,16	**60,63**	90,59
1.5	3 689	3300	0,89	**15,98**	23,96	**2,77**	3,25	**61,10**	93,67
1.6	3 677	3961	1,08	**15,99**	24,06	**2,76**	3,25	**60,69**	92,44

Ver.	CBO		LCOM		LCOMN		LLOC	
	V. é.	Sz.	V. é.	Sz.	V. é.	Sz.	V. é.	Sz.
1.0	**6,52**	7,95	**322,47**	1754,78	**319,09**	1755,62	**163,93**	394,42
1.1	**6,55**	7,99	**324,93**	1776,92	**321,64**	1767,72	**164,63**	398,54
1.2	**6,92**	8,05	**332,63**	1603,85	**313,34**	1857,45	**174,52**	411,29
1.3	**7,13**	8,14	**332,00**	1626,51	**312,92**	1879,91	**171,88**	409,50
1.4	**6,98**	7,98	**314,54**	1609,34	**296,21**	1853,55	**166,32**	406,01
1.5	**6,97**	7,99	**319,15**	1726,65	**301,05**	1949,89	**165,32**	402,89
1.6	**6,99**	8,10	**321,70**	1752,48	**303,51**	1973,51	**164,44**	400,29

3.18. táblázat. A metrikák átlagainak változásai a Mozilla hét verziójában

A 3.1. ábra vizsgálatakor egy érdekes jelenséget figyeltünk meg. Az NML, CBO, RFC, LCOM és LLOC metrikák jelentősen megnőttek az 1.2-es verzióra, míg a hiba-előrejelzés szempontjából kevésbé jelentős NOA és LCOMN metrikák az ellenkező irányba változtak, azaz csökkentek. Az egy osztályra jutó hibák száma szintén nőtt, míg az osztályok száma csökkent.

[4]A NOC metrikát nem tüntettük fel, mert alkalmatlannak bizonyult a hiba-előrejelzésre.



3.1. ábra. A metrikák átlagainak változásai a Mozilla hét verziójában

Ezekből a megfigyelésekből arra következtettünk, hogy a Mozilla 1.2-es verziójában egy nagyobb átszervezés történt, aminek a következménye ez a nagy változás a metrika értékekben, illetve a hibaszámban. Természetesen ez csak találgatás, és sok egyéb tényező is szerepet játszhatott ezen változásokban, így a pontos válasz megtalálásához további vizsgálatok szükségesek.

3.5. Az eredmények összefoglalása

Ebben a fejezetben a Mozilla 1.6-os verzióját felhasználva vizsgáltuk a metrikák és az osztályokban található hibák száma, illetve az osztályok hibára való hajlamosságának kapcsolatát. Ehhez a Chidamber és Kemerer által definiált metrikákat választottuk, amit kiegészítettük az LLOC metrikával. A vizsgálatokhoz különböző statisztikai és gépi tanulási módszereket is használtunk, melyeknek az eredménye közel azonos lett. Azt tapasztaltuk, hogy a 8 vizsgált metrikából 7 esetben találtunk kapcsolatot, és egyedül a NOC metrika esetében nem volt kimutatható semmilyen összefüggés sem.

A metrikák hiba-előrejelző képességeinek elemzése statisztikai és gépi tanulási módszerekkel, illetve az eredmények alapján a modellek felállítása a szerző saját eredménye. A Mozilla változásának elemzése nem a szerző eredménye.

III. rész

Hiba-előrejelzésre alkalmas metrika-kategóriák meghatározása

4. fejezet

Az objektum-orientált metrika-kategóriák vizsgálata

Az előző fejezetben azt vizsgáltuk, hogy van-e kapcsolat a Chidamber és Kemerer által definiált objektum-orientált metrikák és a tradicionális LLOC metrika, valamint az osztályok hibára való hajlamossága között. Különböző szempontokat figyelembe véve kiértékeltük az eredményeket, és összehasonlítottuk, hogy melyik metrika mennyire alkalmas a hibák előrejelzésére. Ezzel a munkával bebizonyítottuk, hogy néhány metrika alkalmas hiba-előrejelzésre, azonban kevés metrikát vizsgáltunk meg ahhoz, hogy általános következtetéseket vonhassunk le. Ezért most azt fogjuk megvizsgálni, hogy különböző metrika-kategóriák milyen mértékben alkalmasak a hiba-előrejelzésre.

4.1. A metrika-kategóriák hiba-előrejelző képességeinek megismerése

Ebben a részben az előző fejezetben bemutatott kísérletet ismételjük meg, azonban a célunk más. Míg az előző esetben nyolc metrikát vizsgáltunk, addig itt öt metrika-kategóriát fogunk elemezni, és arra vagyunk kíváncsiak, hogy az adott metrika-kategória, pontosabban az abba tartozó metrikák mennyire alkalmasak a hibák előrejelzésére. Ehhez összesen 58 metrikát vizsgáltunk meg külön-külön, és ezek alapján következtettünk a kategóriák használhatóságára. A következőkben csak a metrika-kategóriákat fogjuk ismertetni, a metrikák definíciója az A függelékben található.

- 30 különböző *méret-alapú* metrikát tudunk kiszámolni, amelyek a rendszer alapvető tulajdonságait írják le számunkra azáltal, hogy megszámolnak „valamit” (például a program sorokat, osztályokat vagy a metódusokat) a rendszerben. Ezeknek a metrikáknak az előnyei, hogy könnyű kiszámolni azokat, pontosabban könnyebb, mint a többi metrikát, továbbá könnyű megérteni is őket. Egy példa a méret alapú metrikákra a tradicionális és jól ismert LLOC metrika, amelyet már az előző fejezetben is vizsgáltunk.

- A *komplexitás* metrikák a program vagy egy adott kódrészlet komplexitását mérik. Annak ellenére, hogy sok különböző komplexitást lehet definiálni és mérni, mi csak egy komplexitás metrikát, a Chidamber és Kemerer által definiált WMC (metódusok súlyozott összege) metrikát fogjuk vizsgálni. A metódusok súlyának a McCabe-féle ciklomatikus komplexitást választottuk.

- Az *öröklődési* metrikák – amelyekből 8-at számoltunk ki – a rendszer öröklődési hierachiájáról adnak információt. Annak ellenére, hogy ezen metrikákból nem lehet visszaállítani a teljes öröklődési fát, mégis leírják annak karakterisztikáját, ami fontos a rendszer öröklődésének vizsgálata szempontjából.

- Az objektum-orientáltság egyik alappillére az egységbezárás, ami azt jelenti, hogy az összetartozó adatokat és a rajtuk műveleteket végző operátorokat egy egységbe kell zárni, ahol az egység az osztály. A *kohézió* metrikák – melyek közül 11-et vizsgáltunk – azt mérik, hogy az adott osztály csak egy funkcionalitást valósít-e meg. Ha ez nem teljesül, akkor az osztály tagjai (metódusai és attribútumai) között gyenge a kohézió. A gyenge kohézió akkor szokott előfordulni, ha az osztály olyan adatokat tartalmaz, vagy olyan funkcionalitásokat valósít meg, amelyek nem tartoznak oda. Extrém esetekben az is előfordulhat, hogy az osztály tagjait szét tudjuk osztani kettő vagy több diszjunkt csoportra úgy, hogy a két csoport között nincs semmilyen kapcsolat. Ebben az esetben az osztályt célszerű lenne újratervezni.

- A rossz tervezés egy másik jele, ha egy osztály túl sok másik osztályt használ, vagy túl gyakran használja azokat. A használat tetszőleges dolgot jelenthet, mint például az egyik osztály metódusa hívja a másik osztály metódusát, vagy használja annak attribútumát. A *csatolás* metrikák ezeket a különböző típusú kapcsolatokat mérik. Mi 8-at vizsgáltunk meg ezek közül.

4.1.1. A Mozilla kiterjesztett elemzése

A 2.3. alfejezetben leírtuk, hogy hogyan elemeztük le a Mozilla hét különböző verzióját, hogyan gyűjtöttük össze a hibákat a Bugzillából, valamint hogy hogyan rendeltük hozzá a hibákat az osztályokhoz. Amint azt már jeleztük, a bemutatott módszeren változtattunk, illetve a különböző programokon javítottunk az eltelt időszakban, de a módszer alapötlete továbbra is a 2.3. alfejezetben ismertetett eljárás. Mielőtt azonban továbbmennénk, röviden ismertetjük a változásokat, és bemutatjuk, hogy azok milyen hatással voltak a kísérletben használt adatokra.

A második kísérlet esetében kilenc Mozilla verziót elemeztünk az 1.0-tól az 1.8 alfáig[1], és ahogyan korábban, úgy itt is csak az 1.6-os verziót használtuk. A sok hasonlóság ellenére az osztályokhoz rendelt hibák száma jelentősen megváltozott, mert míg az első esetben 3 961 hibát sikerül megtalálnunk, addig a másodikban 7 662 hibát sikerült az osztályokhoz rendelni. A nagy eltérés fő oka, hogy míg az első vizsgálatok idején az 1.6-os verzió még igen friss volt, és kevés hibát javítottak ki benne, addig az eltelt évek alatt sok hibát jelentettek be és javítottak ki. Így lehetséges az, hogy a második esetben majdnem kétszer annyi hibát sikerült beazonosítani.

Egy másik említésre méltó változás, hogy míg az első elemzésnél 3 192 osztályt használtunk a Mozilla 1.6-os verziójának vizsgálatakor, addig a második elemzésnél 17-tel többet, azaz 3 209 osztályt kaptunk. A változás azzal magyarázható, hogy nem szűrtünk ki olyan sok osztályt, mint az első vizsgálat alkalmával, azonban nem hisszük, hogy ez a néhány osztály lényegesen befolyásolná a kísérletet.

Ezen változásokat figyelembe véve a 4.1. táblázat tartalmazza a hibák eloszlását a Mozilla 1.6-os verziójára. A továbbiakban ezekkel az adatokkal fogunk dolgozni.

[1] A Mozillánál az 1.8-as verzió esetében az alfa változatot elemeztük, mert nem volt release verzió.

Az osztályok száma	%	A hozzárendelt hibák száma
1 284	40,01	0
760	23,68	1
415	12,93	2
197	6,14	3
115	3,58	4
92	2,87	5
71	2,21	6
35	1,09	7
35	1,09	8
28	0,87	9
25	0,78	10
23	0,72	11
16	0,50	12
12	0,37	13
14	0,44	14
11	0,34	15
31	0,97	16-20
12	0,37	21-25
10	0,31	25-30
10	0,31	31-40
5	0,16	41-50
6	0,19	51-60
2	0,06	60-72
3 209	100,00	7 662

4.1. táblázat. A hibák eloszlása a 1.6-os verzióban (megismételt mérés)

4.1.2. Logisztikus regresszió alkalmazása

A 3.2. fejezetben lineáris regresszió segítségével vizsgáltuk a metrikák és a hibák közti összefüggéseket. Emellett másik három módszert is alkalmaztunk a metrikák és az osztályok hibára való hajlamosságának vizsgálatára. A vizsgálatok során azt tapasztaltuk, hogy a különböző módszerek közel azonos eredményt adtak, ezért a jelenlegi vizsgálatban csak a logisztikus regressziót[2] alkalmazzuk 0,05 szignifikancia szinttel.

Az eredmények kiértékeléséhez és összehasonlításához a korábbiakban definiált R^2, *helyesség*, *pontosság* és *teljesség* értékek mellett bevezettünk még egyet, a *fedést* (recall), ami azt mondja meg, hogy a hibás osztályok hány százalékát találta meg a modell. Ezt úgy számoljuk ki, hogy a hibásnak jelzett hibás osztályok számát elosztjuk az összes hibás osztályok számával. Például a 4.2. táblázat mutatja a CBO metrika osztályozási eredményét, ahol a modell 1 477 hibás osztályt jelzett hibásnak az 1 925 (478+1 447) valóban hibás osztályból, ami azt jelenti, hogy a fedés értéke 75,17%.

[2] A logisztikus regresszió leírása megtalálható a 3.2.1. alfejezetben.

	Előrejelzett	
Tapasztalt	Nem hibás	Hibás
Nem hibás	757	527
Hibás	478 (695)	1 447 (6 967)

4.2. táblázat. A logisztikus regressziós modell eredménye a CBO metrika esetében

4.1.3. Az eredmények ismertetése

A következőkben a logisztikus regresszió eredményeit fogjuk ismertetni. A 4.3. táblázat tartalmazza azt a 17 metrikát, amelyek esetében a logisztikus regresszió által adott modell mindkét kategóriába (hibás és nem hibás) legalább 100 osztályt sorolt. Mind a 17 metrika esetében a regresszióhoz tartozó p-érték kisebb volt, mint 0,001, azaz minden esetben szignifikáns volt a kapott eredmény. Továbbá a Coh metrika kivételével minden esetben pozitív volt a metrika regressziós együtthatója, ami azt jelenti, hogy minél nagyobb a metrika értéke, annál valószínűbb, hogy az osztály tartalmaz hibát. A Coh metrika esetében a reláció fordított a metrika és az osztályok hibára való hajlamossága között, ami azt jelenti, hogy minél nagyobb a metrika értéke, annál kisebb az esélye, hogy az osztály hibás lesz.

Metrika	Kat.	R^2	Helyesség	Pontosság	Fedés	Teljesség
CBO	csat.	0,186	68,68%	73,30%	75,17%	90,93%
NOI	csat.	0,184	67,90%	72,82%	74,18%	89,62%
RFC	csat.	0,176	67,68%	72,09%	75,27%	90,92%
NFMA_{ni}	csat.	0,174	66,87%	72,26%	72,68%	88,80%
WMC	kompl.	0,161	67,28%	72,63%	72,94%	89,39%
NFMA	csat.	0,158	67,22%	72,19%	73,77%	89,27%
LOC	méret	0,154	67,68%	74,64%	69,87%	89,05%
LLOC	méret	0,147	67,19%	73,12%	71,64%	89,51%
NML	méret	0,128	66,38%	69,93%	77,09%	91,16%
RFC3	csat.	0,125	67,93%	71,35%	77,77%	91,80%
NMLD	méret	0,122	65,78%	68,94%	78,18%	91,48%
NAML	méret	0,114	66,19%	69,37%	78,13%	91,56%
NML_{pub}	méret	0,110	66,19%	69,23%	78,55%	91,36%
NMLD_{pub}	méret	0,104	65,60%	68,52%	78,91%	91,24%
AvgLOC	méret	0,090	64,07%	66,92%	79,32%	88,79%
LCOM4	koh.	0,085	63,79%	64,87%	86,44%	94,39%
Coh	koh.	0,050	65,96%	67,36%	91,60%	97,07%

4.3. táblázat. Az egyváltozós logisztikus regresszió eredménye

A metrika kategóriák vizsgálata előtt a 4.3. táblázat eredményeit áttekintjük és röviden a metrikákat is megvizsgáljuk. Annak ellenére, hogy az R^2 értékek jelentősen csökkentek a CBO metrikától (0,186) a Coh metrikáig (0,050), a *helyessége* értékek kevesebb, mint 5%-ot csökkentek, és a legkisebb érték (63,79%) – amely a LCOM4 metrikához tartozik – is jobb, mint a „triviális osztályozáshoz" (amikor az összes osztályt hibásnak jelezzük) tartozó érték (59,99%).

A LOC metrikának van a legnagyobb pontossága, ami azt jelenti, hogy négy hibásnak jelzett osztályból három valóban hibás. Sajnos ennek a metrikának van a legkisebb fedés értéke, ami csak 69,87%, így a hibás osztályok közel 30%-át nem találja meg. Az LLOC értékei nagyon hasonlók, ami egyáltalán nem meglepő, mivel mindkét metrika az osztályok sorainak számát méri, csak egy kicsit eltérő módon. A CBO metrika pontosság értéke csak 1,5%-kal kisebb, mint a LOC értéke, de a fedés értéke 5%-kal jobb. A Coh metrika rendelkezik a legnagyobb fedés értékkel (több mint 90%), de figyelembe véve, hogy csak 295 osztályt osztályozott hibamentesnek, a nagy fedés érték már kevésbé hasznos. Láthatjuk, hogy az lenne a legjobb, ha a pontosság és fedés értékeket egyszerre tudnánk növelni. Annak ellenére, hogy a definíciójuk nem zárja ki ennek lehetőségét, a gyakorlat azt mutatja, hogy ha a pontosság érték nő, akkor a fedés csökken, és fordítva. Ez a tendencia figyelhető meg a 4.3. táblázatban is, bár van némi „kilengés”[3].

A teljesség érték azt mutatja meg, hogy mennyi hibát találunk meg az adott módszerrel. A CBO metrika esetében 1 974 (a 4.2. táblázat alapján megkapjuk, hogy 527+ 1 477) osztályt jeleztünk hibásnak, ami az osztályok mindössze 61,51%-a, de ez az 1 974 osztály tartalmazta a hibás osztályok 75,17%-át (fedés), és ami még fontosabb, a hibák 90,93%-át is (teljesség). A Coh metrika rendelkezik a legnagyobb helyesség értékkel (97,07%), de az osztályok 87,71%-a hibásnak volt jelezve. A második legnagyobb teljesség értéke az LCOM4 metrikának van (94,39%), ami kevesebb, mint 3%-kal rosszabb csak, mint a Coh metrika értéke, viszont majdnem 8%-kal kevesebb osztályt jelzett hibásnak, így összességében ez jobbnak mondható. A többi metrika teljesség értéke 88,79%-tól 91,80%-ig mozgott, azaz közel azonosnak tekinthető, így az számít, hogy mennyi osztály lett hibásnak jelezve. Minél kevesebb, annál értékesebb a nagy teljesség érték.

A következőkben a metrikákat nem egyesével vizsgáljuk, hanem a metrika-kategóriákról mondunk véleményt. Ehhez az adott kategóriába tartozó metrikákat vesszük alapul, és az alapján próbáljuk meg a kategória „jóságát” meghatározni. Ehhez a 4.3. táblázat adatait, illetve a táblázatból levont következtetéseket is felhasználjuk.

Csatolás metrikák

A 4.3. táblázatból leolvasható, hogy a csatolás metrikák rendelkeznek a legnagyobb R^2 értékekkel (négy csatolás metrikának van a legnagyobb R^2 értéke). Továbbá a négy legnagyobb helyesség érték is csatolás metrikákhoz tartozik, bár nem ugyanarról a négy metrikáról van szó, mint az R^2 esetében. Emellett a csatolás metrikák „kiegyensúlyozott” pontosság és fedés értékekkel rendelkeztek. Továbbá a 8 vizsgált csatolás metrikából 6 metrika (CBO, NOI, RFC, NFMA_{ni}, NFMA és RFC3) hasznosnak bizonyult.

Habár nehéz egyértelmű rangsort felállítani a metrika-kategóriák között, az eredmények alapján mégis azt mondhatjuk, hogy a csatolás metrikák használhatók a hibák előrejelzésére a legjobban. Ebből az eredményből az is látszik, hogy a különböző típusú csatolások fontosak, bár nem azonos mértékben.

[3]Megjegyezzük, hogy a 4.3. táblázatban a metrikák csökkenő R^2 értékek szerint vannak rendezve és nem pontosság vagy fedés értékek szerint.

Méret alapú metrikák

A 4.3. táblázatban található méret-alapú metrikákat két nagyobb csoportra oszthatjuk. Az egyik csoportba azok a metrikák tartoznak, amelyek az osztályok sorait mérik valamilyen formában, azaz a LOC, az LLOC és a AvgLOC metrikák. Ez az eredmény nem mondható újnak, hiszen korábban már számos esetben igazolták ezt az összefüggést. Meglepőnek sem nevezhető, mert várható volt, hogy a nagyobb, pontosabban több sorral rendelkező osztályokban várhatóan több hiba található.

A másik csoportba azok a metrikák (NML, NMLD, NAML, NML_{pub} és NMLD_{pub}) tartoznak, amelyek az osztály metódusait mérték. Korábban már igazolták [4, 28], hogy ha egy osztálynak több metódusa van, akkor több hibát tartalmaz. Az újdonságot az jelentette, hogy a publikus metódusok jó hiba-előrejelzőnek bizonyultak, szemben a nem publikus metódusok számával[4].

A méret-alapú metrikákról nehéz egységes véleményt kialakítani, mert a 30 vizsgált metrikából mindössze 8 metrika bizonyult hasznosnak, ami nem túl jó eredmény. Másfelől az a 8 metrika csak kicsivel rosszabb eredményt ért el, mint a csatolás metrikák. Ezek alapján azt mondhatjuk, hogy a méret-alapú metrikák bizonyos csoportjai alkalmasak a hibák előrejelzésére, azonban nem érik el a legjobb csatolás metrikák eredményeit.

Komplexitás metrika

A komplexitás esetében nem igazán beszélhetünk metrika kategóriáról, mert csak egy komplexitás metrikát, a WMC-t vizsgáltuk. A WMC metrika kicsit gyengébb eredményeket mutatott, mint a legjobb csatolás metrikák, és a legjobb méret alapú metrikákhoz közel azonosat. Ezen eredmények alapján azt mondhatjuk, hogy a vizsgált komplexitás metrika alkalmas a hibák előrejelzésére, azonban ezen metrika alapján nem mondhatjuk azt, hogy a komplexitás metrikák általában alkalmasak a hibák előrejelzésére.

Kohéziós metrikák

A két szignifikánsnak talált kohéziós metrika első ránézésre ellentmond egymásnak, mert a Coh metrikának fordított irányú a kapcsolata a hibák számával, míg az LCOM4 esetében pozitív irányú a kapcsolat. Ez azonban nem ellentmondás, mert a Coh metrika a kohéziót méri, míg az LCOM4 a kohézió hiányát. Ezen eredények alapján azt mondhatjuk, hogy minél kisebb a kohézió az osztályon belül, annál több hiba lesz benne.

A kapott eredények ellenére mégsem jelenthetjük ki, hogy a kohéziós metrikák alkalmasak lennének a hibák előrejelzésére, mert a 11 metrikából csak 2 esetében tudtunk kimutatni szignifikáns kapcsolatot. Továbbá ezen két metrika a leggyengébb metrikák közt található (lásd 4.3. táblázat). Így nehéz egyértelműen megítélni a kohéziós metrikákat, de az látszik, hogy van valamekkora kapcsolat a kohézió hiánya és az osztályok hibára való hajlamossága között. Azonban ezen kísérlet alapján nem mondhatjuk azt, hogy a kohéziós metrikák jelentős szerepet kaphatnak a hibák előrejelzésében.

[4] A 30 méret alapú metrika között szerepeltek olyanok is, amelyek a nem publikus metódusok számát mérték.

Öröklődési metrikák

A 8 vizsgált öröklődési metrika közül egy sem bizonyult alkalmasnak a hibák előrejelzésére a Mozilla esetében. Ebből azt a következtetést vonhatjuk le, hogy ezek a metrikák nem alkalmasak a hibák előrejelzésére. Azonban meg kell vizsgálnunk, hogy ennek mi az oka, illetve ezt a vizsgálatot más rendszerek esetében is meg kell ismételni, mielőtt általánosan kijelentjük, hogy az öröklődési metrikák nem alkalmasak az osztályokban található hibák előrejelzésére[5].

Metrika-kategóriák eredményeinek összegzése

A kapott eredményeket úgy foglalhatnánk össze, hogy a csatolás metrikák bizonyultak a legjobb hiba-előrejelzőknek szinte minden szempontból. Az egyetlen vizsgált komplexitás metrika, valamint a méret alapú metrikák közül a sorok, a metódusok, illetve a publikus metódusok számát mérő metrikák szintén igen jó eredményt értek el. A kohéziós metrikák megítélése igen nehéz, és a jelen vizsgálatok alapján igen korlátozott mértékben alkalmasak a hibák előrejelzésére. De ha ezen eredmények alapján azt mondjuk, hogy nem alkalmasak, akkor sem tévedünk nagyot. Az öröklődés metrikák esetében a kapott eredmény teljesen egyértelmű, és azt mondhatjuk, hogy a Mozilla esetében nincs kapcsolat az öröklődési metrikák és az osztályok hibára való hajlamossága között.

4.2. Kapcsolódó munkák

Számos esetben vizsgálták már az objektum-orientált metrikák és a szoftver minősége közti kapcsolatatot [8, 6, 7, 15, 51, 57, 58]. Ezeknek összefoglalásait többek között Subramanyan és Krishnan [49], illetve Briand és munkatársai [7] készítették el. Itt csak néhány olyan munkát ismertetünk, amelyek szorosan kapcsolódnak a vizsgálatainkhoz. Az eredmények ismertetését az adott cikkben használt metrika-jelölésekkel mutatjuk be, de a következő (a 4.2.1.) alfejezetben összehasonlítjuk a kapott eredményeket.

Basili és munkatársai [4] a Chidamber és Kemerer által definiált hat metrika és az osztályok hibára való hajlamossága közti kapcsolatot vizsgálták. A vizsgálatokhoz nyolc kis és közepes méretű C++ rendszert használtak, amit egyetemi diákok fejlesztettek. Mivel minden, a fejlesztéssel kapcsolatos információ (például a hiba riportok és azok javításai) a rendelkezésükre állt, ezért meg tudták határozni az osztályokban található hibák számát. A rendszerek 180 osztályt tartalmaztak, amelyekre a GEN++ [16] segítségével számolták ki a metrika értékeket. A metrikák és az osztályok hibára való hajlamossága közti kapcsolat vizsgálatára logisztikus regressziót használtak. A vizsgálataik során a DIT, a NOC és az RFC metrikák bizonyultak a legjobbaknak, a CBO is jó eredményt adott, míg a WMC eredménye gyengének mondható. A kísérletükben nem találtak kapcsolatot az LCOM metrika és az osztályok minősége között.

Fioravanti és Nesi [24] a Basili féle projektet [4] használták fel, de a kísérletükben azt vizsgálták, hogy hogyan használhatóak a metrikák a hibák felfedezésében. Összesen 226 metrikát számoltak ki, és a céljuk az volt, hogy egy olyan, minél kevesebb metrikából álló modellt készítsenek, amelynek a helyessége nagyobb, mint 90%. A

[5] Meg kell említenünk, hogy a korábbi vizsgálatok során (3. fejezet) azt tapasztaltuk, hogy a NOA metrika és az osztályok hibára való hajlamossága között volt kapcsolat. Így nem jelenthetjük ki egyértelműen, hogy az öröklődési metrikák nem alkalmasak a hibák előrejelzésére.

céljukat elérték, mert egy olyan modellt alakítottak ki, amely 42 metrikából állt, és a helyessége nagyobb volt, mint 97%. Ez a modell azonban túl sok metrikát használt ahhoz, hogy a gyakorlatban alkalmazható legyen, ezért megpróbálták jelentősen csökkenteni a metrikák számát úgy, hogy még egy használható modellt kapjanak. Végül kaptak egy 12 metrikából álló modellt, amelynek a helyessége még majdnem 85% volt. A modellben a CO, ICP_L, LCOM1, LCOM2, NAI, NAML, NDSTT, NFR, NMImp, RFC_∞, STMTS és TCC metrikák szerepeltek.

Yu és munkatársai [56] tíz metrikát[6] választottak, hogy a metrikát és a hibák közti kapcsolatot vizsgálják. A vizsgálatokhoz egy nagy hálózati rendszer kliens oldali komponensét használták, amit három tapasztalt szoftverfejlesztő fejlesztett. A rendszer Java nyelvű volt, amely 123 osztályból és közel 34 000 programsorból állt. Egyváltozós regressziós analízist alkalmaztak, és azt kapták, hogy a CBO_{in}, az RFC_{in} és a DIT metrikák alkalmatlanok a hibák előrejelzésére, míg a másik hét metrika (WMC, LOC, CBO_{out}, RFC_{out}, LCOM, NOC és Fan-in) esetében szignifikáns kapcsolatot találtak, de a metrikák hiba-előrejelző képessége különböző volt.

Subramanyam és Krishnan [49] egy nagyobb C++ és Java nyelven írt e-kereskedelmi alkalmazást választottak. A rendszer 405 C++ és 301 Java osztályt tartalmazott. Az osztályok mérete mellett a Chidamber és Kemerer metrikák közül a WMC, CBO és DIT metrikák és a hibák közti kapcsolatot vizsgálták többváltozós regressziós analízissel. Mindkét nyelvre külön-külön elvégezték a kísérleteket, majd összehasonlították az eredményeket. Az eredmények azt mutatták, hogy az osztályok mérete mindkét nyelv esetében jó prediktornak bizonyult, míg a WMC és a CBO metrikákat csak a C++ nyelv esetén találták használhatónak.

Olague és munkatársai [30] a Rhino projektet [45] választották a vizsgálataikhoz. A Rhino a Mozilla egy nyílt forráskódú Java nyelvű terméke, melynek fejlesztéséhez agilis szoftverfejlesztési folyamatot alkalmaztak. A Rhino hat különböző verzióját vizsgálták meg, melyekben az osztályok száma 100 és 200 között mozgott. A Rhino Bugzillájából összegyűjtött hibákat az osztályokhoz rendelték. A kísérletükben a Chidamber és Kemerer metrikák, a Brito e Abreu féle MOOD metrikák [17] és a Bansiya és Davis által definiált QMOOD metrikák [3] hibaelőrejelző képességeit vizsgálták egyváltozós logisztikus regressziós segítségével. Az RFC mind a hat esetben szignifikáns volt, a CBO metrika[7] a hat vizsgált esetből ötször bizonyult szignifikánsnak, az LCOM98 és a WMC metrika négyszer volt szignifikáns, míg a maradék két metrika (DIT és NOC) esetében nem, vagy csak nagyon gyenge kapcsolatot találtak. A QMOOD metrikák közül a CIS és a NOM volt a legjobb, míg a többi QMOOD metrika csak egy-két esetben bizonyult használhatónak. A MOOD metrikák esetében csak két esetben találtak szignifikáns kapcsolatot, azaz általánosságban elmondhatjuk, hogy a MOOD metrikák nem alkalmasak a hibák előrejelzésére. Emellett többváltozós lineáris regresszió felhasználásával több modellt kialakítottak. Azt tapasztalták, hogy a C&K metrikákon alapuló modellek voltak a legjobb hiba-előrejelzők, de a QMOOD metrikákon alapuló modellek csak egy kicsivel teljesítettek rosszabbul.

[6] Valójában 8 metrikát vizsgáltak, mert a CBO és RFC metrikáknak két különböző változatát használták.

[7] Az általuk használt CBO definíciója egy kicsit eltért az általunk használt CBO definíciójától.

4.2.1. Az eredmények összehasonlítása

A metrikák vizsgálatának lezárásaként nem csak a kapott eredményeket foglaljuk össze, hanem elmagyarázzuk azok jelentését is. Ennek első lépéseként a kapott eredményeket összehasonlítjuk néhány korábbi munkával, hogy lássuk, az általunk kapott eredmények mennyire vannak összhangban azokkal. Emellett a két kísérletünk eredményeit is összehasonlítjuk, mert találtunk eltéréseket a kapott eredményekben.

A különböző munkák összevetése a 4.4. táblázatban látható. A Saját$_1$ oszlopban az első kísérletünk [28] eredményei találhatók, míg a Saját$_2$ tartalmazza a kiterjesztett vizsgálatok [47] eredményét. A további oszlopokban Basili [4], Yu [56], Subramanyan [49] és Olague [30] munkáinak eredményei találhatók. Annak ellenére, hogy számos metrikát megvizsgáltunk, csak a hat Chidamber és Kemerer metrika és az LLOC metrika eredményét fogjuk összehasonlítani, mivel ezek szerepeltek a legtöbbször a vizsgálatokban. A 4.4. táblázatban + jelöli ha az adott vizsgálatban pozitív kapcsolatot találtak a metrika és a hibák száma között, míg a − jelöli, ha a kapcsolat fordított irányú. A ++ és −− jelölik, ha metrika kiemelkedően jónak bizonyult. A 0 azt jelenti, hogy az adott kísérletben vizsgálták a metrikát, de nem találtak összefüggést a metrika és a hibák száma között, míg az üresen hagyott helyek azt jelentik, hogy az adott metrikát nem vizsgálták.

Metrika	Saját$_1$	Saját$_2$	Basili	Yu	Subramanyan	Olague
NML	++	++	+	++	+	++
NOA	+	0	++	0	0	
NOC	0	0	−−	−−		0
CBO	++	++	+	+	+	++
RFC	++	++	++	+		++
LCOM	+	+	0			
LLOC	++	++			++	

4.4. táblázat. A különböző kutatások eredményeinek összehasonlítása

Az összehasonlítás alapján azt mondhatjuk, hogy a két méret alapú metrika (NML és LLOC), valamint a két csatolás metrika (CBO és RFC) esetében minden vizsgálat ugyanarra az eredményre jutott, mely szerint ezek a metrikák alkalmasak a hibák előrejelzésére. A másik három metrika (NOA, NOC és LCOM) esetében az eredmények változóak voltak, mert néhány esetben sikerült kapcsolatot találni a metrika a hibák száma között, míg más esetben nem. Ezért ezen három metrika esetében nem lehet egyértelműen állást foglalni, és további vizsgálatok szükségesek hiba-előrejelző képességeinek validálására.

4.3. Az eredmények összefoglalása

A fejezetben a korábbi vizsgálatunkat kiterjesztve öt metrika-kategória hiba-előrejelző képességét vizsgáltunk meg, melyhez 58 objektum-orientált metrikát használtunk fel. A kísérletben a csatolás metrikák bizonyultak a legjobbnak minden szempontból, de a komplexitás metrika, illetve a bizonyos méret méret-alapú metrikák is figyelemre méltó eredményt értek el. Ezzel szemben a kohéziós metrikák esetében csak igen gyenge

kapcsolatot sikerül igazolnunk, míg az öröklődés metrikák alkalmatlannak bizonyultak a hiba-előrejelzésre.

A fejezetben bemutatott metrika-kategóriák hiba-előrejelző képességeinek vizsgálatát szerző saját eredményének tekinti.

5. fejezet

Az eredmények gyakorlati alkalmazása

Az előző fejezetekben megmutattuk, hogy van kapcsolat a metrikák és a program minősége között. A 2. fejezetben ismertettük a Columbus keretrendszert, melynek segítségével kiszámolhatjuk a metrika értékeket parancssorból. Annak ellenére, hogy ez a megoldás alkalmas nagy rendszerek elemzésére is, használata nem kényelmes a mindennapi munka során.

Ebben a fejezetben olyan eszközöket mutatunk be, amelyek támogatják a metrikák hatékony és kényelmes kiszámítását, melynek következtében lehetőség nyílik a metrikák alkalmazására a mindennapi munkánk során. A programok ismertetése mellett bemutatjuk azok ipari környezetben történő alkalmazását is.

5.1. Az eszközök bemutatása

A 2. fejezetben már bemutattuk, hogy egy rendszer elemzése és a metrikák előállítása néhány parancs kiadásával végrehajtható, és az eredményt egy csv fájlban megkaphatjuk. Ez a megoldás alkalmas arra, hogy egy rendszert egyszer elemezzünk, és a kapott eredményeket felhasználjuk különböző célokra, azonban nem alkalmas arra, hogy a mindennapi munkánk során használjuk. Ezért továbbfejlesztettük a keretrendszert, hogy könnyen és egyszerűen használható eszközöket kapjunk. Az eszközök kifejlesztésénél azt is figyelembe vettük, hogy kik és hogyan fogják felhasználni az eredményeket.

A felhasználók egyik csoportjába a programozók tartoznak, akik munkájára jellemző, hogy egy nagyobb rendszer egy kisebb részét fejlesztik valamilyen fejlesztőkörnyezet (IDE – Integrated Development Environment) segítségével. Az ő munkafolyamatuk támogatására egy olyan eszköz szükséges, amelyik integrálható a fejlesztőkörnyezetükbe, és gyorsan ki tudja számolni a metrikákat azokra a részekre, amelyeket éppen fejlesztenek. Ez az eszköz a *SourceAudit*, amelyet az 5.1.1. alfejezetben mutatunk be.

A menedzserek és vezetők nem használnak fejlesztőkörnyezetet annak kiszámítására, hogy egy adott osztály metrikái mekkorák, vagy hogyan változtak. A számukra legkedvezőbb folyamat az lenne, ha a teljes rendszer folyamatosan elemezve lenne, és vizsgálhatnák, hogy a teljes rendszer metrikái mennyire jók vagy rosszak, vagy hogy az elmúlt időszakban hogyan változtak a metrika értékek. Ennek a folyamatnak a támogatására a *SourceInventory* eszközt ajánljuk, amit az 5.1.2. alfejezetben ismertetünk részletesen. Mint látni fogjuk, a menedzserek mellett a fejlesztők munkájában is nagy segítség lehet a SourceInventory.

Az értekezés a metrikák vizsgálatáról szól, azonban az itt bemutatásra kerülő eszközök nem csak a metrikák kiszámítására alkalmasak. Ezért röviden összefoglaljuk a

programok által előállított további eredményeket, mert az eszközök ismertetésénél nem tudjuk elkerülni ezek megemlítését.

Mind a három általunk elemzett nyelv esetében vannak olyan általános szabályok, amelyek be nem tartása a program hibás működéséhez vezet. Egy példa erre, hogy a C++ nyelv esetében ha `new[]` operátorral foglaltunk memóriát, akkor `delete[]` operátorral kell azt felszabadítani. Ha `delete[]` operátor helyett a `delete` operátort használjuk, akkor a program futásának az eredménye nem definiált (ISO/IEC 14882:2003(E) [35]). A *kódolási szabályellenőrző* exporterek olyan programozási hibákat és veszélyes helyeket mutatnak meg a forráskódban, amelyek statikus elemzéssel felismerhetők. A kimenet tartalmazza a megtalált problémák pontos helyét (melyik fájl melyik sorában található, és a probléma leírását), valamint statisztikát is készít, hogy mely forráskódelemekben (metódus, osztály és névtér) mennyi szabálysértés volt.

A *klón detektor* a forráskód különböző helyein található hasonló[1] kódrészeket találja meg és párosítja össze. Az eredmény a programkód másolatok listája, valamint különféle klón-alapú metrikákat is megtudunk a rendszerről, mint például a teljes rendszer klón lefedettsége.

A harmadik nagy csoportba a *bad smellek* [25] tartoznak. A bad smell a programozók körében elterjedt zsargon, ami olyan „tünetet” jelent a forráskódban, ami potenciális hibaforrás. A bad smellek valójában nem hibák, csak utalások lehetséges hibaforrásokra, így ajánlott az ilyen helyek alaposabb vizsgálata, és a szükséges javítások elvégzése.

5.1.1. SourceAudit

A *SourceAudit* eszközt a szoftverfejlesztők számára fejlesztettük ki, hogy a mindennapi munkájukat támogassuk. A SourceAudit nem egy önálló program, hanem egy addin, ami a telepítése után integrálódik a különböző fejlesztőkörnyezetekbe. A Microsoft Visual Studio alá telepített változattal C/C++ és C# nyelvű rendszereket tudunk elemezni, míg az Eclipsehez készült változat Java nyelvű rendszerek elemzésére alkalmas. Annak ellenére, hogy a különböző SourceAudit változatok különböző fejlesztőkörnyezetekben működnek és különböző nyelveket elemeznek, a működésük nagyon hasonlít, így csak a Visual Studiohoz készült változatot fogjuk bemutatni.

A SourceAudit telepítése[2] után a Visual Studioban engedélyezni kell a SourceAudit plugint[3]. Ezután megjelenik a SourceAudit toolbar (5.1. ábra), és készen áll a használatra. Először a toolbaron található gombokat mutatjuk be.

5.1. ábra. A SourceAudit toolbar

[1] Koschke és munkatársai [37] definiálták, hogy mikor tekinthetünk két kódrészt azonosnak, illetve Bakota és munkatársai [2] vizsgálták a klónok hatását a Mozilla esetében.

[2] A SourceAudit telepítését nem részletezzük külön, mert nagyon egyszerű. A felhasználónak csak a telepítési könyvtárat kell megadni, valamint azt, hogy a Start menübe kerüljön-e ikon, ezután a program automatikusan feltelepül és azonnal használható.

[3] A plugin aktiválásához a *Tools* menüpontból a *Customize...* pontot kell kiválasztani, majd a felugró ablakban engedélyezni kell a *SourceAudit Toolbar*t.

Extract the project: Ennek segítségével az aktív projektet elemezhetjük inkrementálisan. Ez azt jelenti, hogy a rendszernek csak azt a részét elemezzük újra, amelyik változott a legutolsó elemzés óta. Ezzel az elemzés ideje nem az elemzendő rendszer méretétől függ (pontosabban attól is, de csak kis mértékben), hanem az utolsó elemzés óta történt változásoktól. Ez nagy rendszerek gyakori elemzésénél nagy előnyt jelent, és jelentősen csökkenti az elemzés idejét.

Re-extract the whole project: Ha valamiért a rendszert teljesen újra kell elemeznünk, akkor erre is lehetőségünk van a *re-extract* funkcióval. Mivel egy rendszer teljes újraelemzése sokáig tarthat, ezért ezt csak különösen indokolt esetben ajánljuk.

Stop the extract: Bármikor megállíthatjuk az elemzést a *Stop* gombbal. Ebben az esetben félbeszakad az elemzési folyamat, de az *Extract* funkció segítségével bármikor folytathatjuk onnan, ahol megállítottuk.

Settings for the checkers: A *Settings* ablakot hozza fel, amiben az elemzéssel és metrikákkal kapcsolatos beállításokat adhatjuk meg. A beállítások részletes leírását később adjuk meg.

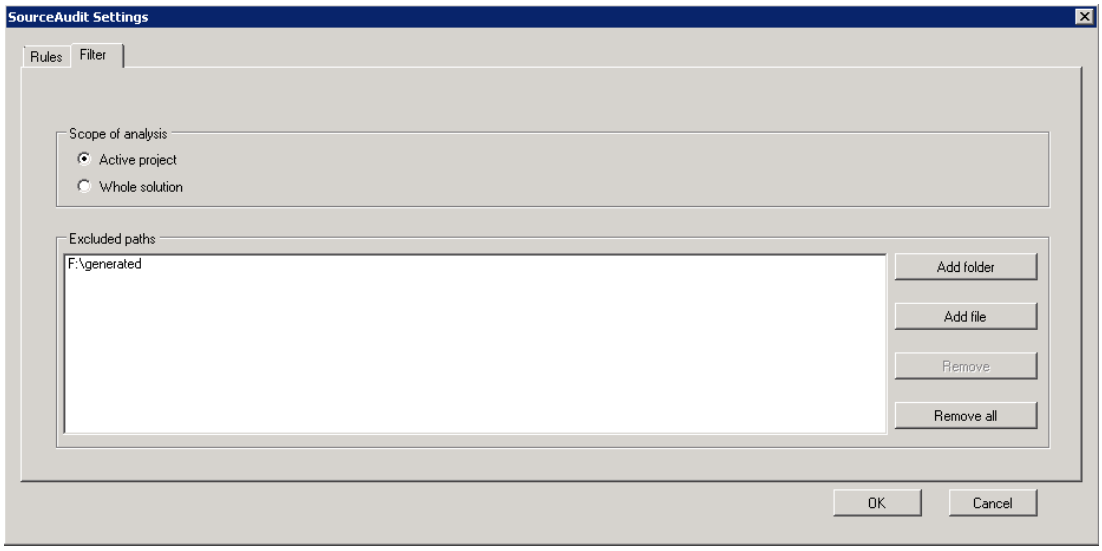
Show the SourceAudit metrics window: Ezzel aktiválhatjuk azt az ablakot, amelyben az elemzés után a rendszer metrikái megjelennek (5.4. ábra).

About: Megnyitja a CopyRight ablakot.

Beállítások

A *Settings* ablakon módosíthatjuk a működéssel kapcsolatos beállításokat, amelyeket két nagyobb csoportra oszthatunk. Az egyikbe a projekt elemzésével kapcsolatos beállítások tartoznak (5.2. ábra), míg a másikban azt választhatjuk ki (5.3. ábra), hogy milyen típusú (például metrikák vagy klónok) eredményekre vagyunk kíváncsiak, illetve azon belül is milyen értékekre (például látni szeretnénk a CBO metrika értékeit, de a NOA metrikára nem vagyunk kíváncsiak).

Az 5.2. ábra mutatja az általános beállításokhoz tartozó ablakot. Itt megadhatjuk, hogy a teljes rendszert elemezzük (*Whole solution*) vagy csak az aktív projektet (*Active project*). Ennek csak akkor van jelentősége, ha a megnyitott solutionben több projektünk is van, különben mindegy, hogy melyiket választjuk, ugyanazt az eredményt kapjuk. Abban az esetben, ha több projekt is van, és azt választjuk, hogy csak az aktív projektet elemezzük, akkor csak az éppen aktív projekt lesz elemezve (ilyen mindig létezik abban az esetben, ha meg van nyitva egy solution a Visual Studioban), és minden eredmény ez alapján lesz kiszámolva. Ennek az az előnye, hogy ha egy nagy rendszerünk van, akkor annak az elemzése sokáig tarthat, míg egy projekt elemzése gyorsan lefut. A hátránya, hogy bizonyos metrikák és más fontos eredmények nem lesznek pontosak (a felhasználó erről egy figyelmeztetést is kap), mert például nem tudjuk megmondani egy adott osztály gyerekeinek a számát, vagy nem tudjuk megtalálni a klónokat addig, amíg a teljes rendszert nem elemeztük. Ennek ellenére ez a funkció igen hasznos, mert például a „lokális” metrikákat – mint amilyenek a komplexitások, bizonyos csatolások vagy az osztályok és metódusok sorainak a száma – így is megkaphatjuk. Ha a solutionben található összes projektet elemezzük (Whole



5.2. ábra. A SourceAudit elemzéssel kapcsolatos beállításai

solution), akkor minden metrika érték pontos lesz. A hátrány, hogy az első elemzés nagyon lassú lehet, viszont a következő elemzések már gyorsan megvannak.

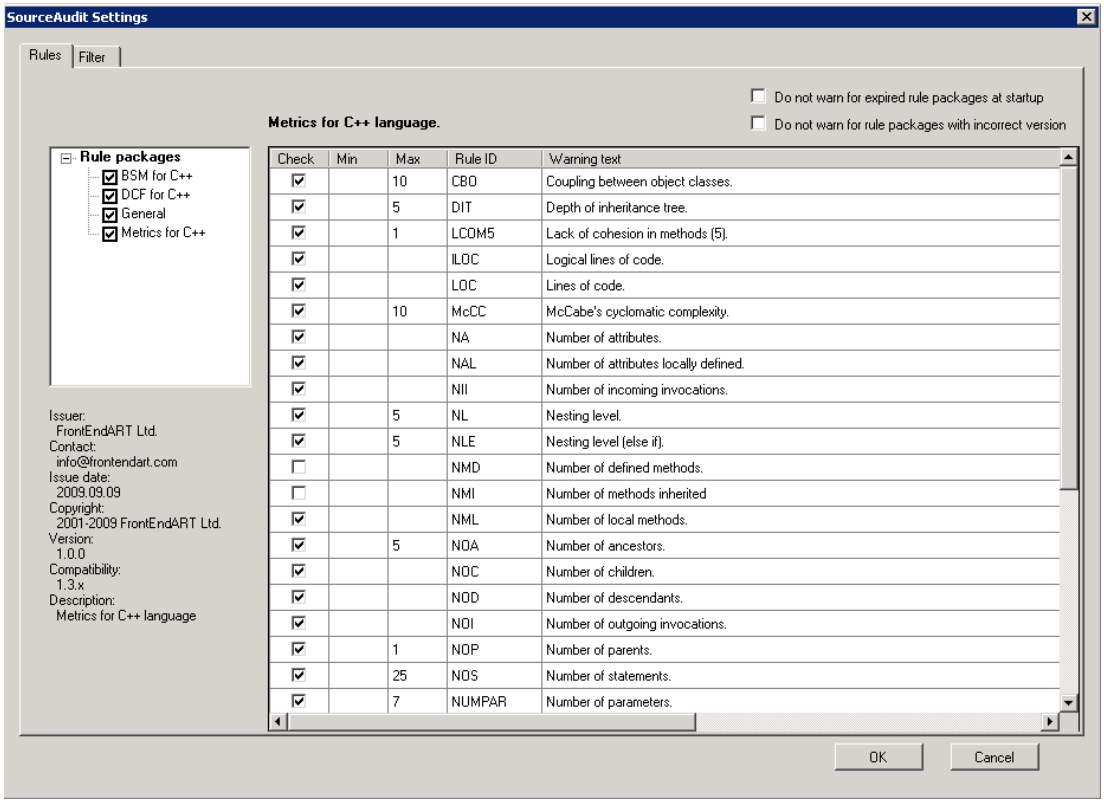
Ugyanezen az ablakon adhatjuk meg azokat a könyvtárakat és fájlokat is, amelyeknek az „eredményeire" nem vagyunk kíváncsiak. Az elemzést nem befolyásolják ezek a beállítások, azonban a kiszűrt elemeknek az eredményei nem fognak megjelenni. Az ábrán látható példában az *F:\generated* könyvtár alatt található fájlokat szűrtük ki.

A *Settings* ablak másik fülén (5.3. ábra) az eredményekkel kapcsolatos beállítások találhatók. Itt megadhatjuk, hogy milyen típusú eredményekre vagyunk kíváncsiak. Lehetőségünk van teljes csoportok (metrikák, kódolási szabályok, kód duplikációk és bad smellek) kikapcsolására azáltal, hogy a *Rule packages* alatt lévő csomagok közül nem választjuk ki azt, amelyikre nem vagyunk kíváncsiak. Emellett lehetőségünk van a csoportokon belül egyesével is kiválasztani a checkboxok használatával, hogy melyiket akarjuk látni, és melyiket nem. Továbbá számos metrika esetében meg lehet adni korlátot is. A metrikák esetében ezeknek a korlátoknak a megjelenítésben van szerepük, amit a metrikák megjelenítésénél ismertetünk.

A SourceAudit működését befolyásolják a *Settings* ablak jobb felső sarkában található további checkboxok is (az 5.3. ábra). Ha valamelyik csomagnak lejárt a licensze, akkor a SourceAudit induláskor figyelmeztet erre. Ha szeretnénk a figyelmeztetéseket eltűntetni, akkor a *Do not warn for expired packages at startup* checkboxot ki kell választani. A *Do not warn for rule packages with incorrect version* checkboxnak hasonló a szerepe, csak ez a nem megfelelő verzióval rendelkező exporterekre hívja fel a figyelmet.

A metrikák megjelenítése

Az elemzés befejezése után az eredményeket különböző ablakokban kapjuk meg. Ezek közül az egyik a *SourceAudit Metrics* ablak (5.4. ábra), amelyben a forráskód elemek metrikáit jelenítjük meg. Ebben az ablakban a forráskód elemek logikai tartalmazás alapján egy fába vannak rendezve, amelyet szétnyithatunk vagy összecsukhatunk. A



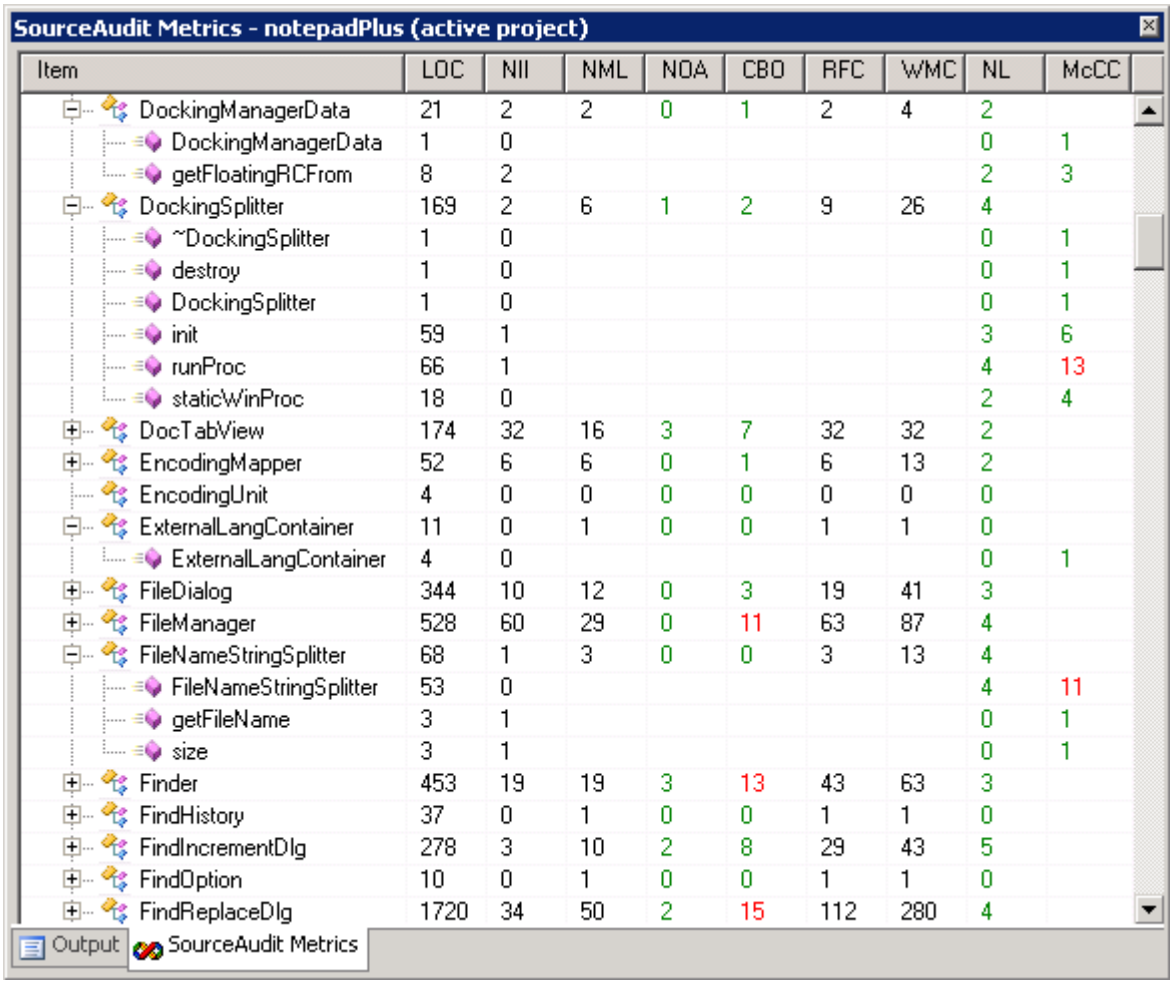
5.3. ábra. Metrikákkal kapcsolatos beállítások a SourceAuditban

forráskód elem sorában szerepelnek a metrikái, de csak azoknak a metrikáknak az oszlopában vannak értékek, amely értelmezve vannak az adott elemre. Ha egy metrikára nem lehet beállítani semmilyen korlátot sem (az 5.3. ábrán látszik, hogy például a LOC metrikára nem lehet beállítani korlátot), akkor a hozzá tartozó metrika értékek feketével jelennek meg. Ha egy metrikára beállítottunk valamilyen korlátot (például a CBO metrika esetében 10 a korlát), akkor ezt a metrika értéket zöld színnel jelenítjük meg, ha kisebb, mint az adott korlát (például a `FileDialog` osztályhoz tartozó 3-as érték zöld), különben pirossal (például a `Finder` osztály 13-as metrikája). Ha egy metrika érték túl nagy, akkor az nagyobb valószínűséggel tartalmaz hibát, ezért az ezekhez tartozó metrika értékek piros jelennek meg, így felhívják az esetleges problémára a fejlesztők figyelmét.

Lehetőségünk van arra is, hogy az elemeket egy adott metrika szerint növekvő vagy csökkenő sorrendbe rendezve vizsgáljuk az adott szkópon belül[4]. Ezt az adott metrika oszlop fejlécére kattintva kaphatjuk meg.

További segítség, hogy a SourceAudit bármely eredmény ablakában (így természetesen a metrika ablakban is), ha kettőt kattintunk egy elemen (vagy a jobb egérgombbal aktiválható menüből kiválasztjuk a *Go to source* menüpontot), akkor az elemet tartalmazó fájl megnyílik a Visual Studioban, és a kurzor odaugrik az adott fájl megfelelő elemére.

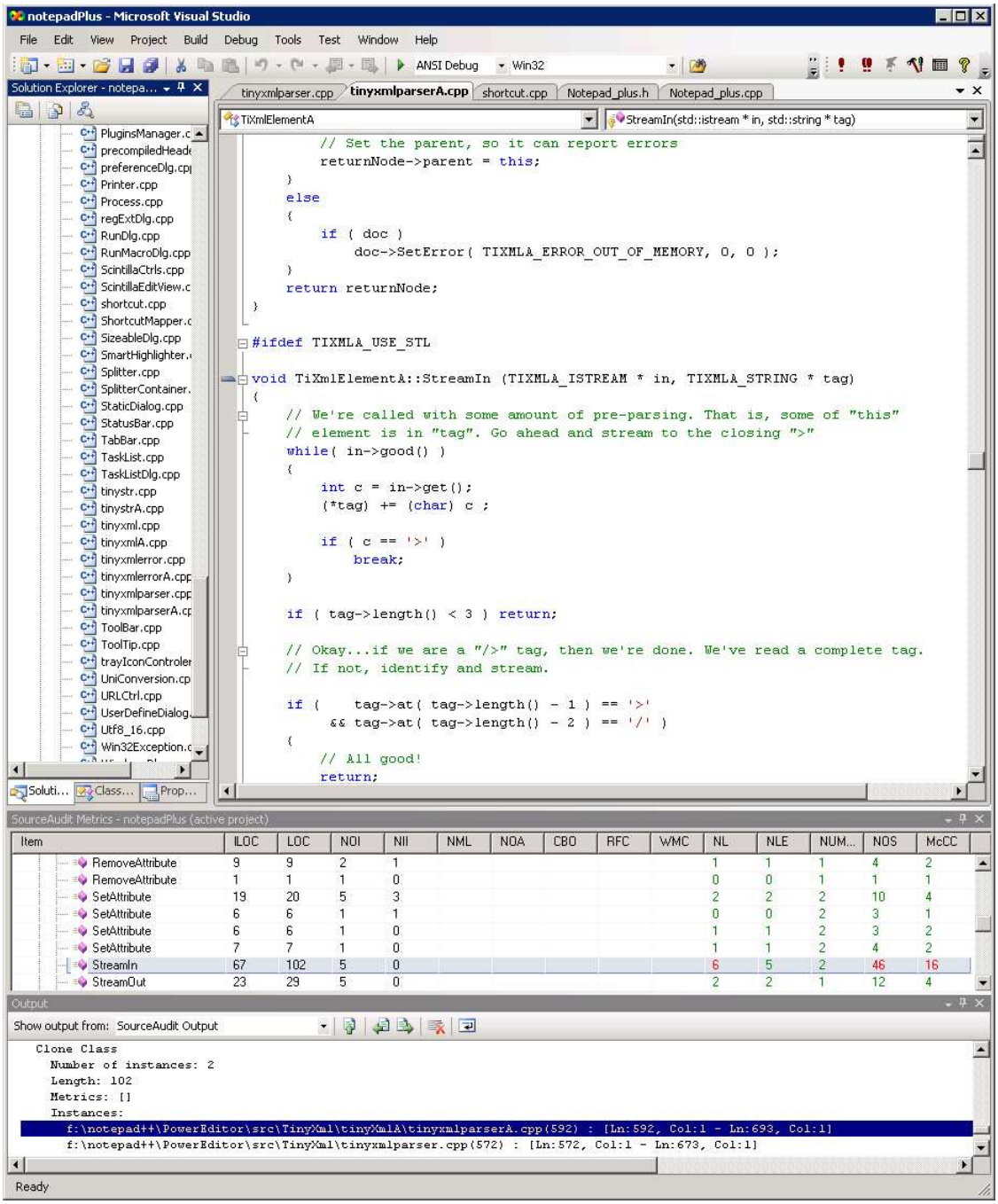
[4]Azért nem lehet az összes elemet egy adott metrika szerint rendezni, mert a fa struktúrát meg kell tartani.



5.4. ábra. A *SourceAudit metrics* ablak

A SourceAudit integrációja a Visual Studioba

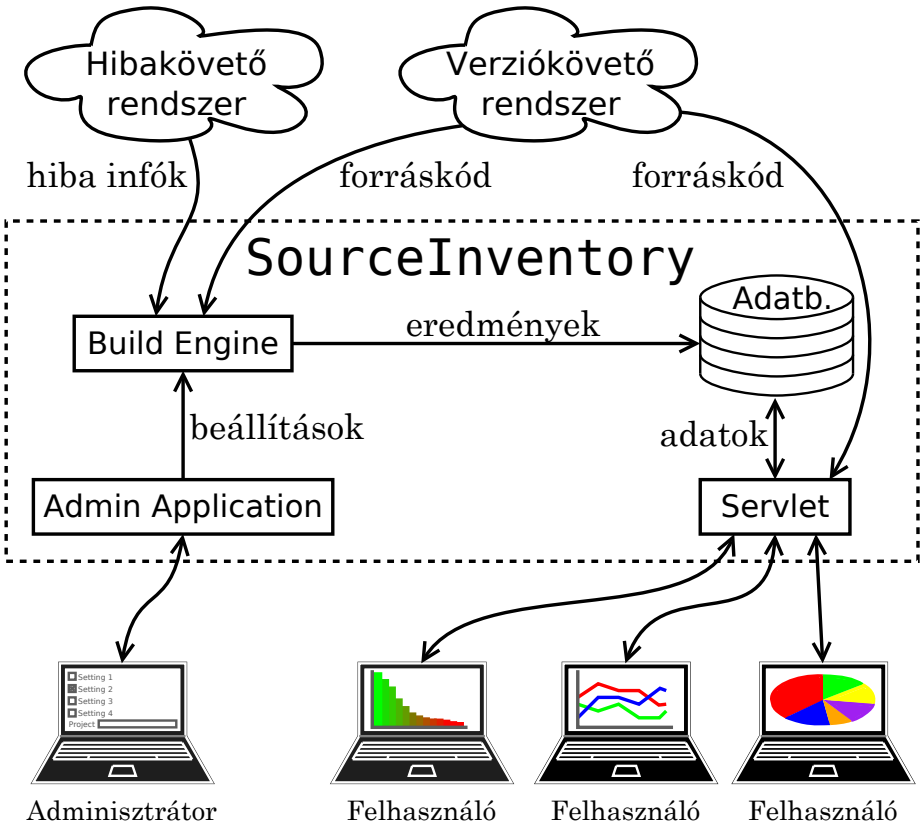
A SourceAudit Visual Studioba történő integrációját az 5.5. ábra mutatja, amelyen látható a SourceAudit toolbar, a metrika ablak, és az az output ablak, melyben a klón példányok találhatók éppen.



5.5. ábra. A SourceAudit integrációja a Visual Studioban

5.1.2. SourceInventory

Annak ellenére, hogy a SourceAudit és a SourceInventory ugyanarra a technológiára épül, és ugyanazokat az eredményeket tudják előállítani, mégis nagyon eltér a működésük és a felhasználásuk. Anélkül, hogy mélyebben belemennénk a részletekbe, először röviden ismertetjük a SourceInventory működését. A SourceInventory architektúráját az 5.6. ábra mutatja. Mielőtt a rendszert használnánk, az adminisztrátornak be kell állítania a működéshez szükséges információkat[5]. Ezt az *Admin Application* által biztosított adminisztrációs felületen lehet megtenni. Az Admin Application továbbítja ezeket az információkat a *Build Engine* felé, ami a megadott időközönként végrehajtja a feladatot. A Build Engine a megadott időpontban letölti a forráskódot a *verziókövető rendszerből*, elvégzi a megadott rendszer elemzését a 2. fejezetben bemutatott módon, majd a *hibakövető rendszerből* összegyűjti a hibákat, és a forráskódelemekhez rendeli. Ezeket az *eredményeket* továbbítja az *adatbázis felé*, ahol a SourceInventory az elemzett rendszerek eredményeit tárolja. Ezután a *felhasználók* egy böngésző segítségével kapcsolódhatnak a SourceInventoryhoz, azon belül a *servlet*hez, majd a megfelelő projekt kiválasztása után használhatják azt (a kezdő képernyőt az 5.7. ábra mutatja). A kliens a szervlettel „kommunikál”, ahol a szervlet a kérések kiszolgálására az adatokat az adatbázisból nyeri ki, míg a vizsgált rendszer forráskódját a verziókövető rendszerből tölti le, és továbbítja a kliens felé.

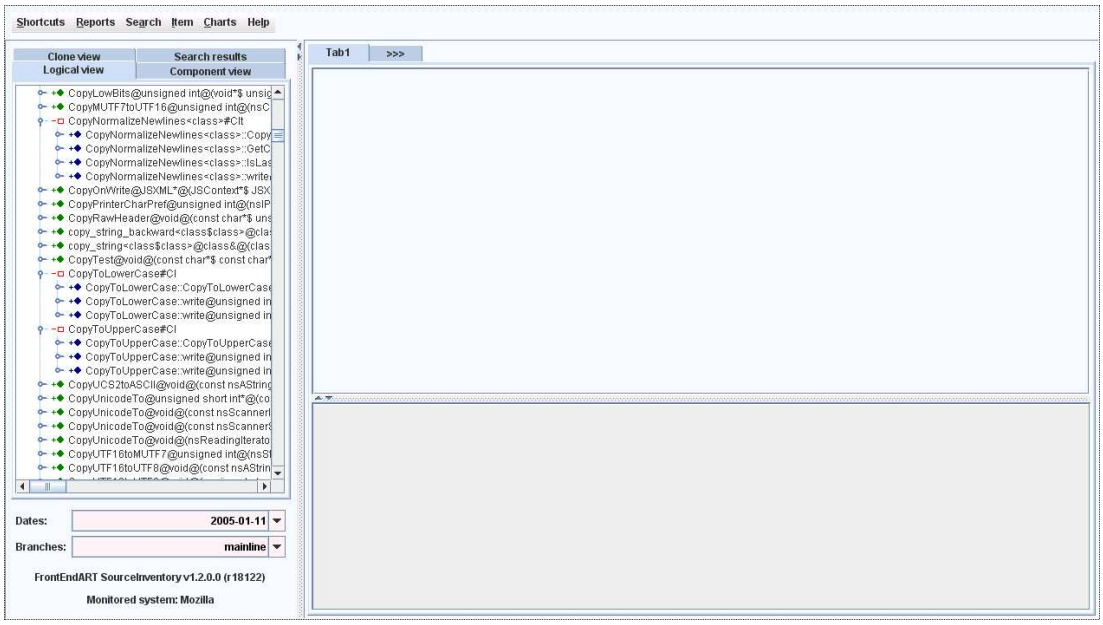


5.6. ábra. A SourceInventory architektúrája

A bejelentkezés után az 5.7. ábrán látható kezdő képernyő jelenik meg. A képernyő

[5]Mivel a SourceInventory adminisztrációs beállításai technikai jellegűek, ezért azokkal nem fogunk külön foglalkozni.

fel van osztva három nagy részre, ahol a felosztás arányát tetszőlegesen módosíthatjuk az elválasztó szegélyek mozgatásával.



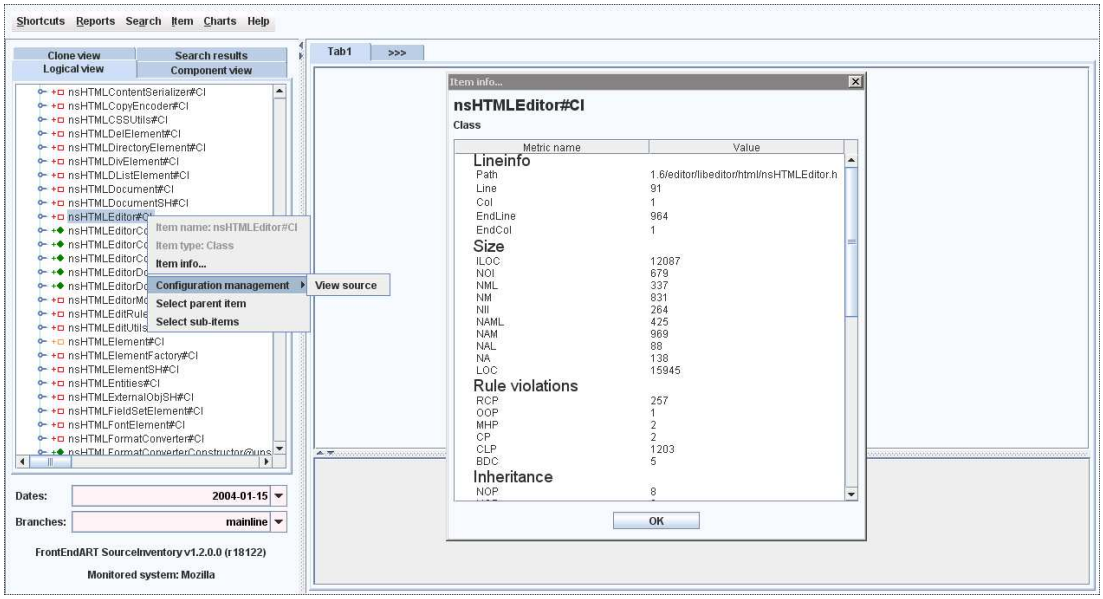
5.7. ábra. A SourceInventory kezdő oldala

A bal oldali részen választhatjuk ki, hogy a rendszernek mely nézetére vagyunk kíváncsiak. Három nézet esetében az elemek egy fa-struktúrában jelennek meg, amit szét lehet nyitni vagy össze lehet csukni azokat. A logikai nézetben (*Logical view*) a fa az elemek logikai kapcsolatai alapján van felépítve, ami megegyezik a SourceAudit metrika ablakánál bemutatott hierarchiával (5.4. ábra). A komponens nézetben (*Component view*) a hierarchia a rendszer komponenseitől indul, amely alatt az elemek már a logikai kapcsolatok alapján szerveződnek, de csak azok az elemek jelennek meg egy komponens alatt, amelyek az adott komponensben vannak. A klón nézetben (*Clone view*) a rendszerben található klón osztályok és klón példányok szerint szerint épül fel a fa. Mind a három nézet esetében a különböző típusú elemek különböző alakzatokkal és színekkel vannak jelölve. Lehetőségünk van név vagy metrika érték alapján keresni az elemek között, melynek az eredménye a *Search results* nézetben jelenik meg.

A jobb oldalon lévő felső ablakban jelennek meg az éppen aktuális „művelet” eredményei, míg az alatta lévő ablakban ezzel kapcsolatos információkat láthatunk. A későbbiekben fogunk látni konkrét példákat is erre.

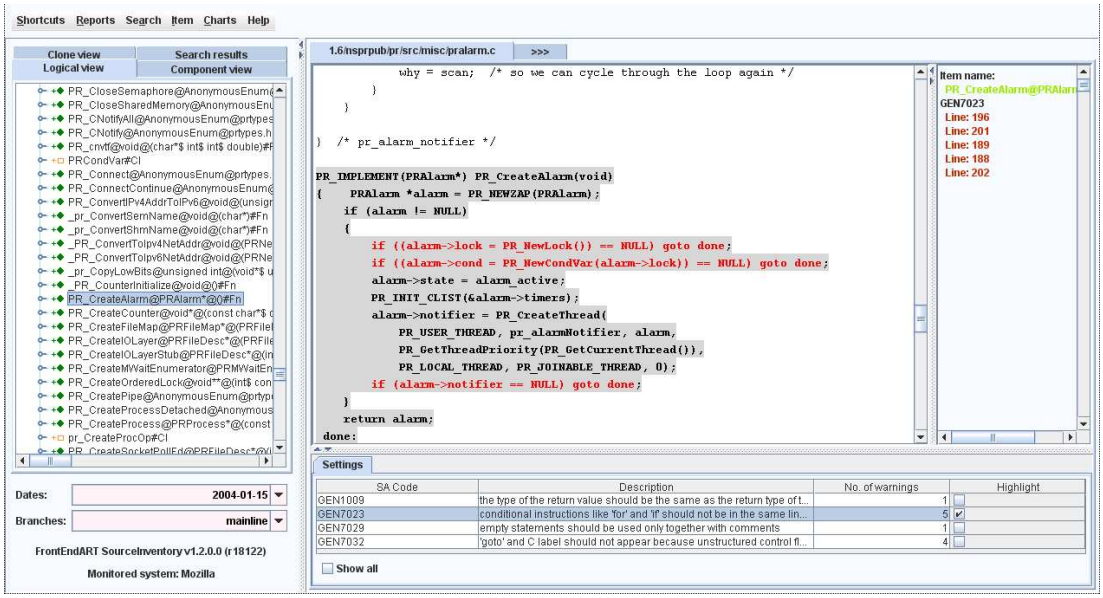
Ha egy rendszernek több fejlesztési ága is létezik, akkor a bal alsó sarokban lévő *Branches* melletti legördülő menü segítségével válthatunk a fejlesztési ágak között. A különböző időpontokhoz tartozó mérések közötti váltásokat a *Dates* mellett található legördülő menüvel tehetjük meg.

Az értekezésnek nem célja a SourceInventory részletes bemutatása, ezért a folytatásban csak néhány egyszerű használati esetet emelünk ki. A továbbiakban metrika alatt nem csak az objektum-orientált metrikákat értjük, mert metrikákat számolunk a bad smellek és a klónok alapján is, illetve az elemben található kódolási szabálysértések száma is metrikának tekinthető. Sőt, az adatok megjelenítése szempontjából ezek az értékek teljesen hasonlóan kezelhetők, mint az objektum-orientált metrikák, ezért továbbiakban nem különböztetjük meg ezeket.



5.8. ábra. Az `nsHTMLEditor` osztály metrikái

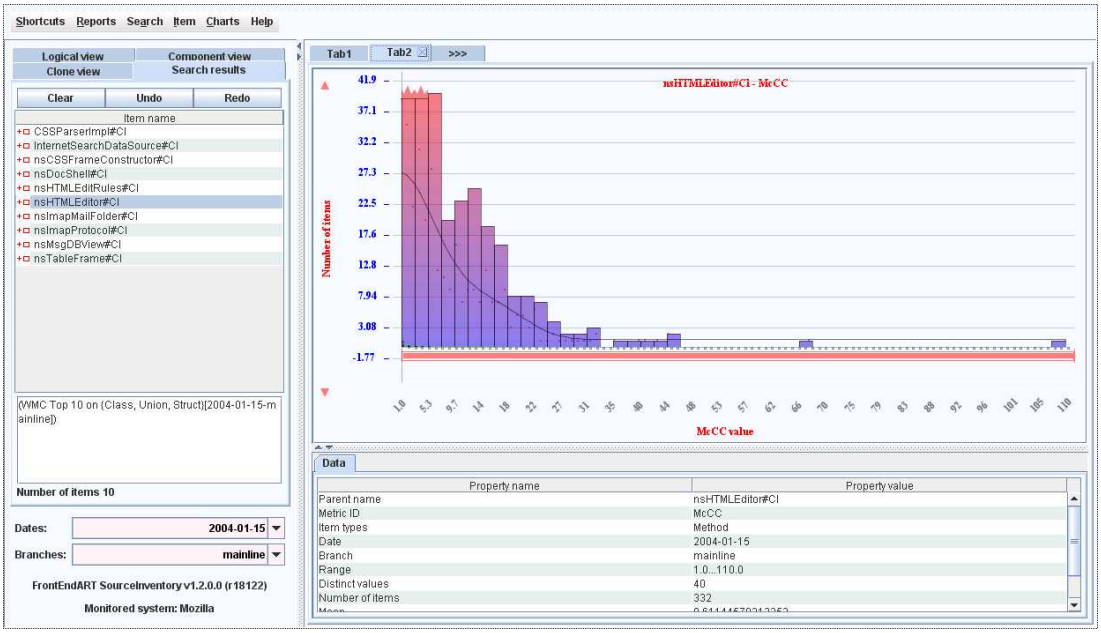
Az 5.8. ábrán látható, hogy egy elemet kiválasztva egy gyorsmenüt aktiválhatunk. Az *Item info ...* menüpont segítségével egy külön ablakban megkaphatjuk az adott elem összes adatát. Például az 5.8. ábrán látható ablakról leolvashatjuk, hogy az `nsHTMLEditor` az *editor/libeditor/nsHTMLEditor.h* fájlban található, a 91. sor 1. oszlopában kezdődik és a 961. sor 1. oszlopáig tart. Ez alatt megtalálható az `nsHTMLEditor` osztály összes metrikája csoportosítva.



5.9. ábra. Az `PR_CreateAlarm` függvény forráskódjának megjelenítése

A gyorsmenü *Configuration management* menüpontja egy újabb menüt hoz fel (5.8. ábra), amelyen a *View source* segítségével megjeleníthetjük a kiválasztott elem forráskódját. Ilyenkor a teljes fájl megjelenik, amin belül a kiválasztott elem forráskódja

szürke háttérrel van kiemelve. Az 5.9. ábrán a `PR_CreateAlarm` függvény forráskódja látható. Az alatta található *Settings* ablakban megjelennek azok a bad smellek és szabálysértések, amelyek megtalálhatók az adott elemben. Ha ezek közül valamelyiket kiválasztjuk a mellette található checkbox segítségével, akkor a forráskód mellett megjelenik egy ablak, amiben azt láthajuk, hogy a kiválasztott bad smell vagy szabálysértés melyik sorban található. Ezekre kattintva a forráskód az adott sorra lesz pozícionálva. Hogy még könnyebb legyen beazonosítani a problémás részeket, ezek pirossal ki vannak emelve a forráskódban.



5.10. ábra. Hisztogram az `nsHTMLEditor` osztály metódusainak McCC metrikáiról

Lehetőségünk van az elemek keresésére név vagy metrika érték alapján. Az 5.10. ábrán a *Search results* ablakban látható annak a keresésnek az eredménye, ahol a rendszer 10 legnagyobb CBO értékkel rendelkező osztályára voltunk kíváncsiak. Miután ez megvan, több lehetőség is rendelkezésünkre áll az elemek metrikáinak vizsgálatára. Az egyik lehetőség, hogy készíthetünk egy hisztogramot egy elem gyerekeire valamely metrika alapján. Az 5.10. ábrán az `nsHTMLEditor` osztály metódusainak McCC metrika értékeinek a hisztogramja látható. Az alatta található *Data* ablakban a hisztogramról láthatunk olyan információkat, amelyek megmondják, hogy hány elem alapján készült, mi a legnagyobb és legkisebb metrika érték, illetve további statisztikai adat is megtalálható.

Az elemek metrikáinak időbeli változásának vizsgálatára az egyik lehetőség a *timeline*. Az 5.11. ábrán látható az `nsHTMLEditor` CBO, WMC és LOC metrikáinak változás a Mozilla 1.0-ás verziótól az 1.8-as verzióig. Ebből leolvasható, hogy mind a három metrika érték jelentősen nőtt a vizsgált verziókban. A *Data* ablakon a timelineról találhatunk különböző statisztikai adadtokat.

5.11. ábra. Az `nsHTMLEditor` osztály CBO, WMC és LOC metrikáinak változásai

5.2. A metrikák ipari alkalmazásai

Zárásként néhány ipari példát mutatunk, melyek alátámasztják, hogy az eddig bemutatott eszközök és eredmények nem csak elméleti síkon léteznek, hanem a gyakorlatban, ipari körülmények között is alkalmazható technológiáról van szó. Valószínű az olvasó első kérdése az lenne, hogy vajon a saját kódjuk minőségének mérésére használják-e az általuk kidolgozott technológiát. Természetesen igen a válasz a kérdésre. A SourceInventory segítségével minden éjszaka elemezzük a tanszékünkön történő összes olyan fejlesztést, ami valamely általunk elemezhető nyelven történik, majd a változásokról mindenki (fejlesztők és vezetők egyaránt) kap egy összefoglaló levelet. Így reggelre láthatjuk, hogy az elmúlt egy nap alatt mennyit változott a kód, illetve annak minősége. Emellett a fejlesztőknek rendelkezésükre áll a SourceAudit is, amellyel a forráskód minőségét a verziókövető rendszerbe történő kommitálás előtt tudják ellenőrizni. Ezzel elkerülhetik, hogy a másnap reggeli riportokból szerezzenek tudomást az általuk okozott minőségi romlásról, és utólag kelljen javítani azt.

A saját kód elemzése mellett számos nyílt forráskódú rendszert is folyamatosan, hetente elemzünk. Ezek közé tartozik a Mozilla, az openOffice.org, a WebKit vagy éppen az Eclipse, hogy csak a legnagyobbakat említsük.

Több olyan ipari partnerünk is van, akik évek óta használják az általunk kidolgozott technológiát és termékeket a saját kódjuk minőségének biztosítására. A folyamatos mérés és monitorozás mellett dolgozunk azon is, hogy ahol lehetséges (azaz rendelkezésre áll megfelelő hiba-adatbázis) kialakítsunk egy, a metrikákon alapuló hibaelőrejelző modellt is. A módszer teljesen megegyezik a Mozilla esetében ismertetett eljárással, csak specializálni kell az adott rendszerre.

5.3. Kapcsolódó munkák

Széles a választék az olyan termékekből [36, 39], amelyek többek között kiszámolják egy adott rendszer metrikáit, megjelenítik azt valamilyen formában, illetve felhasználják azokat a szoftver minőségének mérésére. Ezek bemutatása helyett inkább a metrikák egy érdekes és ötletes felhasználását ismertetjük.

A CodeCity [54, 55] nem képes a metrikák kiszámítására, hanem a más eszközök által kiszámolt és FAMIX [14] formátumba kimentett adatokat használja. Az adatokat egy város formájában jeleníti meg, ahol az osztályokból házak épülnek, míg a névterek határozzák meg a lakótömböket vagy városrészeket. Az városban az elemek egymásra épülése megegyezik azzal, ahogy a forráskódban névterek és osztályok szkóp szerinti tartalmazzák egymást. A metrika értékek többek között az épületek méreteit (szélesség, hosszúság és magasság), illetve azok színét határozzák meg. Az így kapott városképpel egyszerre jelenítettük meg a rendszer elemeit és metrikáit, ezáltal a rendszer vizsgálatát átalakítottuk egy város kialakításának megítélésére. Ha a városban találunk nem megfelelő házakat, akkor az azt jelenti, hogy a rendszerben is találtunk nem megfelelő osztályokat, azaz a város vizsgálatával valójában magát a rendszert vizsgáljuk.

5.4. Az eredmények összegzése

Bemutattunk két olyat programot, amelyek a Columbus technológiára épülnek, és annak a használhatóságát növelik, illetve az eredmények hatékonyabb és gyorsabb felhasználását segítik elő. Ezáltal mindkét program könnyen beilleszthető a mindennapi fejlesztési folyamatokba, és alkalmasak a folyamatos minőségbiztosítás segítésére. Az eredmények ismertetése igazolja, hogy nem csak elméleti eredményekről beszélhetünk, hanem egy olyan módszert mutattunk be a korábbi fejezetekben, amely az iparban is használható.

A SourceAudit és a SourceInventory bemutatásának a célja a korábbi fejezetek eredményeinek ipari alkalmazhatóságának szemléltetése. Mivel a szerző csak tanácsadóként vett rész ezen eszközök kifejlesztésében, így nem tekinti saját eredményének ezeket.

IV. rész

A szoftverfejlesztők tapasztalatainak felhasználása a minőségi modellek javítására

6. fejezet

A szoftverfejlesztők metrikákról kialakult véleményének megismerése

A dolgozat eddigi részében igazoltuk, hogy a metrikák és a program minősége között van kapcsolat, így a metrikák használata a szoftverfejlesztés különböző fázisaiban javíthatja a szoftver minőségét. Az 5. fejezetben bemutattunk néhány programot, amelyek segítségével ki tudjuk számolni a metrika értékeket, és folyamatosan nyomon tudjuk követni azok változásait, mellyel a szoftver minőségének változását is megfigyelhetjük.

A legjobb eszközök sem érnek azonban sokat, ha olyan emberek kezébe kerülnek, akik nem értenek hozzá. Ezért ebben a fejezetben a fejlesztők metrikákról kialakult véleményeit fogjuk megvizsgálni, hogy lássuk, megvan-e a szükséges tudásuk a metrikán alapuló technológiák megfelelő alkalmazásához. Az eszközök nem megfelelő használata – és itt elsősorban nem a programok nem megfelelő használatára utalunk, hanem a módszertan nem megfelelő használatára – nemcsak hogy nem segíti a fejlesztés menetét, hanem hátráltatja azt, illetve nem várt mellékhatásai lehetnek.

Általánosan azt mondhatjuk, hogy a metrikát jól ismerő szakemberek szerint a metrikák hatékonyan használhatóak a szoftverfejlesztés különböző fázisaiban. Másfelől a fejlesztők alig használják a metrikákat a mindennapi munkájuk során, mert vagy nem ismerik eléggé azokat, vagy ismerik, de nem tudják hogyan kell alkalmazni azokat. Ezért egy *kérdőívet* állítottunk össze [48], aminek a segítségével vizsgáltuk a fejlesztők ismereteit és nézeteit az objektum-orientált metrikákról, továbbá vizsgáltuk azt is, hogy a tapasztalat hogyan befolyásolja a metrikák gyakorlati alkalmazását. Az vizsgálat eredményének segítségével a hiba-előrejelző modelljeinket javíthatjuk, hogy például a fejlesztők által elfogadott, bár rossz metrika értékekkel rendelkező kódok minőségét pontosabban becsüljük meg.

6.1. A kérdőív bemutatása

A kérdőív több mint 50 kérdést tartalmazott, de helyhiány miatt nem tudjuk az összes kérdést és eredmény bemutatni. Ezért csak nagyvonalakban mutatjuk be a kérdőívet, majd a legérdekesebb kérdéseket és a legfontosabb eredményeket fogjuk csak részletesen tárgyalni.

A kérdőív három nagyobb részre osztható. Az első rész (6.1.1. alfejezet) néhány általános kérdést tartalmazott a résztvevők tapasztalataival és szakértelmével kapcsolatban. Ezen kérdések segítségével egy általános képet kapunk magukról a résztvevőkről.

A kérdőív hátralévő részében a részvevők objektum-orientált metrikákról kialakult

véleményét vizsgáltuk, valamint azt, hogy szerintük milyen összefüggés van a metrikák és a program megértés, illetve tesztelés között. Mivel az objektum-orientált programozás alapegysége az osztály, ezért csak osztály-szintű metrikákat vizsgáltunk. A kérdőívben sok metrika-csoportot és azon belül több metrikát vizsgálhattuk volna, de ebben az esetben túl hosszú lett volna. Így csak négy általános kategóriát (*méret*, *komplexitás*, *csatolás* és *kód duplikáció*) választottunk, illetve minden kategóriából egy metrikát vizsgáltunk.

- A *méret*-alapú metrikák közül a *Logical Lines of Code (LLOC)*, azaz a nem üres és nem komment sorok száma metrikát választottuk.

- A *Weighted Method for Class (WMC)* – a választott *komplexitás* metrika – ami az osztály metódusainak komplexitásainak az összege, ahol a metódus komplexitásának mérésére a McCabe-féle ciclomatikus komplexitást használtuk.

- A *csatolás* metrikák – amelyek az osztályok közötti kapcsolatokat mérik – közül a *Coupling Between Object classes (CBO)* metrikát választottuk, amely azt méri, hogy egy osztály hány másik osztályt „használ".

- A negyedik választott csoport a *kód duplikációk* vagy *klónok*, amelyek a forráskódban található kódmásolatok különböző „jellemzőit" mérik. Ezek közül a *Clone Instances (CI)* metrikát használtuk a kérdőívben, ami az osztályban található klón példányok számát méri.

A kérdőív középső részében (6.1.2. alfejezet) a metrikákat egyesével vizsgáltuk, azaz ugyanazt a kérdést tettük fel mind a négy metrikára, hogy lássuk, hogyan alkalmazzák azokat ugyanannak a problémának a megoldására. A kérdőív végén (6.1.3. alfejezet) található kérdések a metrikákat együttesen vizsgálják, és a résztvevőknek a metrikákat kellett rangsorolniuk azok fontossága szerint.

Mind az 50 résztvevő, akik kitöltötték a kérdőívet, a Szegedi Tudományegyetem Szoftverfejlesztés Tanszékének dolgozói, és különböző ipari és K+F projekteken dolgoztak. A részvevők összetétele széles skálán mozgott a kezdő diákoktól a tapasztalt programozókig, így a tapasztalatuk és a szakértelmük nagyon különbözött. Ezért megvizsgáltuk azt is, hogy a különböző mértékű tapasztalat hogyan befolyásolja a metrikák gyakorlati alkalmazásainak megítélését. Ez azt jelenti, hogy a válaszok és azok eloszlásainak ismertetése mellett statisztikai módszereket is alkalmaztunk, hogy megtudjuk, a tapasztalat befolyásolta-e az válaszokat vagy sem.

A következőkben a kérdőív fontosabb részeit fogjuk bemutatni, valamint azokat a következtetéseket ismertetjük, amelyeket a kérdésekre adott válaszokból levontunk. Ehhez minden kérdés vagy kérdéscsoport (ha a kérdések összetartoznak) után megadjuk a lehetséges válaszokat, valamint hogy hány résztvevő jelölte meg azt (százalékos formában). Ezen felül minden kérdés után bemutatjuk a kérdésekből levont következtetéseinket is.

6.1.1. A résztvevők tapasztalatának felmérése

Az első kérdések a résztvevők szakmai tapasztalatait és szakértelmét mérik. Mivel nem volt értelme az egyes kérdésekből külön-külön következtetéseket levonni, ezért

ezt a kérdés-csoportot egyben vizsgáltuk, és csak a kérdések bemutatása után ismertetjük a következtetéseinket. Ennek megfelelően a kérdések és a lehetséges válaszok a következők:

Kérdés: *Mekkora programozói tapasztalattal rendelkezik?*

- Kevesebb, mint 2 hónap tapasztalatom van. (8%)
- 2-12 hónap tapasztalatom van. (14%)
- 1-3 év tapasztalatom van. (24%)
- Több mint 3 év tapasztalatom van. (54%)

Kérdés: *Milyen a jártassága a C programozásban? (Ugyanezt a kérdést tettük fel C++, Java, C#, and SQL nyelvekre.)*

Válaszok	C	C++	JAVA	C#	SQL
• Nem ismerem a C nyelvet.	2%	0%	0%	32%	6%
• Alapvető ismereteim vannak a nyelvről.	20%	20%	8%	32%	34%
• Saját célra készítettem néhány programot C nyelven.	32%	14%	34%	22%	20%
• Részt vettem C nyelven megvalósított projektben, de rendszeresen nem programozom benne.	30%	32%	24%	8%	30%
• Rendszeresen programozom C nyelven.	16%	34%	34%	6%	10%

Kérdés: *Milyen a jártassága nyílt-forrású szoftverfejlesztés területén?*

- Nincsen semmi tapasztalatom ezen a területen. (22%)
- Vizsgáltam már nyílt-forrású kódot, de magam nem írtam még patch-et. (52%)
- Írtam már elfogadott patch-et. (16%)
- Rendszeresen fejlesztek nyílt-forrású projektben. (10%)

Kérdés: *Milyen ismeretei vannak szoftver termék metrikákról?*

- Nincsen semmi ismeretem. (10%)
- Hallottam, olvastam vagy tanultam a metrikákról. (58%)
- Ismerem a metrikákat és a fejlesztés során figyelembe veszek néhányat. (16%)
- Alapos ismereteim vannak erről a területről (pl. kutatási terület). (16%)

Kérdés: *Milyen a jártassága szoftver tesztelés területén?*

- Nincsen semmi tapasztalatom ezen a területen. (10%)
- Vannak tapasztalataim a tesztelésről. (34%)
- Az általam megírt kódot szoktam tesztelni. (40%)
- A napi feladataim közé tartozik a tesztelés. (16%)

	Prog. tap.	Metr. ism.	C tap.	C++ tap.	Java tap.	C# tap.	SQL tap.	Nyílt-f. tap.
Prog. tap.	1,000							
Metr. ism.	0,425	1,000						
C tap.	0,494	0,242	1,000					
C++ tap.	0,252	0,324	0,393	1,000				
Java tap.					1,000			
C# tap.						1,000		
SQL tap.	0,256				0,351		1,000	
Nyílt-f. tap.	0,344		0,534	0,281				1,000
Teszt. tap.								

6.1. táblázat. Korrelációs együtthatók a tapasztalatok és szakértelmek között

Először azt vizsgáljuk meg, hogy volt-e valamilyen kapcsolat a fent említett tapasztalatok és szakértelmek között. Az eredményeket röviden elemeztük annak ellenére, hogy a kérdésekre adott válaszokból levont következtetéseket nem lehetett általánosítani, mert a válaszok erősen függtek a tanszékünk összetételétől és az itt végzett munkától. A vizsgálathoz a Kendal τ rank korrelációt [10] alkalmaztuk 0,05 szignifikancia szinttel. A 6.1. táblázat tartalmazza a szignifikáns korrelációs együtthatókat. Ezek az eredmények rávilágítanak a tanszékünk egy-két speciális vonására. Például, a legtapasztaltabb programozóink főleg C/C++ nyelven programoznak (a hozzá tartozó korrelációs együtthatók 0,494 és 0,252) és sokan közülük nyílt-forrású fejlesztésekben is részt vesznek (0,344), ahol minden projekt C/C++ nyelvű (0,534 és 0,281). A Java és C# nyelvek kevésbé használtak a tanszéken (hiányoznak a szignifikáns korrelációk), de azok az alkalmazások, amelyek Java nyelven íródtak, adatbázisokat is használnak, aminek a következménye a Java és SQL közti korreláció (0,351).

Kérdés	Junior résztvevők száma	Senior résztvevők száma
Programozói tapasztalat	23 (46%)	27 (54%)
Metrika ismeret	34 (68%)	16 (32%)
C++ tapasztalat	17 (34%)	33 (66%)
Nyílt-forrású tapasztalat	37 (74%)	13 (26%)
Tesztelési tapasztalat	22 (44%)	28 (56%)

6.2. táblázat. A junior és senior résztvevők aránya

A következőkben azt is elemezni fogjuk, hogy a különböző területeken szerzett tapasztalat és szakértelem hogyan befolyásolja a válaszokat. A fenti 9 kérdés közül csupán 5-nek az eredményét fogjuk felhasználni (a C, Java, C# és SQL nyelveken szerzett tapasztalatokat nem vesszük figyelembe). A további vizsgálatokhoz a résztvevőket két csoportra osztottuk: *junior* (tapasztalatlanok az adott területen) illetve *senior* (tapasztaltak az adott területen). Ennek megfelelően az egy kérdéshez tartozó 4 vagy 5 lehetséges választ két diszjunkt csoportra kellett osztanunk. Mivel nem volt pontos definíció arra, hogy mikor számít valaki egy adott területen tapasztaltnak, ezért mi magunk határoztuk meg a határt a két kategória között. A 6.2. táblázat tartalmazza az 5 kérdéshez tartozó kategorizálás eredményeit.

6.1.2. Kérdések a metrikák gyakorlati alkalmazásairól

A következő kérdések a metrikákat vizsgálják egyesével, ami azt jelenti, hogy minden kérdésben csak egy metrika szerepel. A metrikák hasznosságának vizsgálata mellett azt is megnéztük, hogy van-e szignifikáns különbség a junior és senior fejlesztők válaszai között. Pearson-féle χ^2 tesztet alkalmaztunk 0,1 szignifikancia szinttel annak vizsgálatára, hogy a tapasztalat szignifikánsan befolyásolja-e metrikák megítélését. A vizsgálathoz felállított null-hipotézis és alternatív-hipotézis a következő:

- A **null-hipotézis** az, hogy metrikák megítélése nem függ a résztvevők tapasztalatától.
- Az **alternatív-hipotézis** az, hogy metrikák megítélése függ a résztvevők tapasztalatától.

Minden egyes kérdés esetében elvégeztük a tesztet, és vagy elfogadtuk a null-hipotézist, vagy elvetettük, és helyette az alternatív hipotézist fogadtuk el.

Tudtuk, hogy a minta mérete kicsi, így a teszt kevésbé lesz megbízható, továbbá a mintát a tanszékünkről gyűjtöttük, így nem lesz általánosítható másik szoftverfejlesztő csapatokra. Ezen tények ellenére az eredmények rávilágítottak arra, hogy ezt a témát alaposabban is meg kell vizsgálni, mert az eredmények potenciális problémákat fedeztek fel a metrikák gyakorlati használatával kapcsolatban.

A következőkben minden kérdést feltettünk mind a négy metrikával. Minden egyes kérdés esetében megvizsgáltuk a kapcsolatot a válaszok és az előző alfejezetben bevezetett tapasztalat és szakértelem kategóriák között. Az eredményeket a kérdések után ismertetjük.

A metrikák használata a program megértésben és tesztelésben

Az első két kérdéscsoporttal azt vizsgáltuk, hogy a metrikákat hogyan lehet használni a program megértésben és tesztelésben. A kérdést a *méret-alapú* metrikával és *program megértéssel* ismertetjük, de ugyanez a kérdés volt feltéve *komplexitás*, *csatolás* és *klón* metrikákkal is, valamint mind a négy kérdést megismételtük *program tesztelésre* is.

I. kérdés: *Tegyük fel, hogy egy olyan rendszert kell* megismernie *(*tesztelnie*), aminek a fejlesztésében nem vett részt. Befolyásolja-e a rendszer* megértését *(*tesztelését*) a rendszerben található található osztályok* mérete *(*komplexitása, csatolása *vagy a bennük található* klónok száma*)?*

- V_1: Igen, a több kisebb osztályból álló rendszert könnyebb megérteni.
- V_2: Igen, a kevesebb, de nagyobb osztályból álló rendszert könnyebb megérteni.
- V_3: Az osztályok méretének nincs jelentősége a megértés szempontjából, azaz mindegy, hogy több kisebb vagy kevesebb nagyobb osztályt kell megérteni.
- V_4: Nem tudom eldönteni.
- V_5: Szerintem a méret önmagában nem elegendő annak eldöntésére, hogy az adott rendszert mennyire könnyű vagy nehéz megérteni. Ezért további szempontokat (metrikákat) javasolnék (szöveges magyarázat megadható).

Metrika-kategória	Megértés					Tesztelés				
	V_1	V_2	V_3	V_4	V_5	V_1	V_2	V_3	V_4	V_5
Méret	28%	6%	10%	6%	**50%**	**36%**	10%	16%	6%	32%
Komplexitás	**68%**	6%	6%	0%	20%	**80%**	2%	2%	2%	14%
Csatolás	**56%**	12%	6%	4%	22%	**80%**	2%	2%	0%	16%
Klónok	**64%**	8%	14%	4%	10%	**64%**	8%	14%	4%	10%

6.3. táblázat. A válaszok eloszlása az I. kérdés esetében

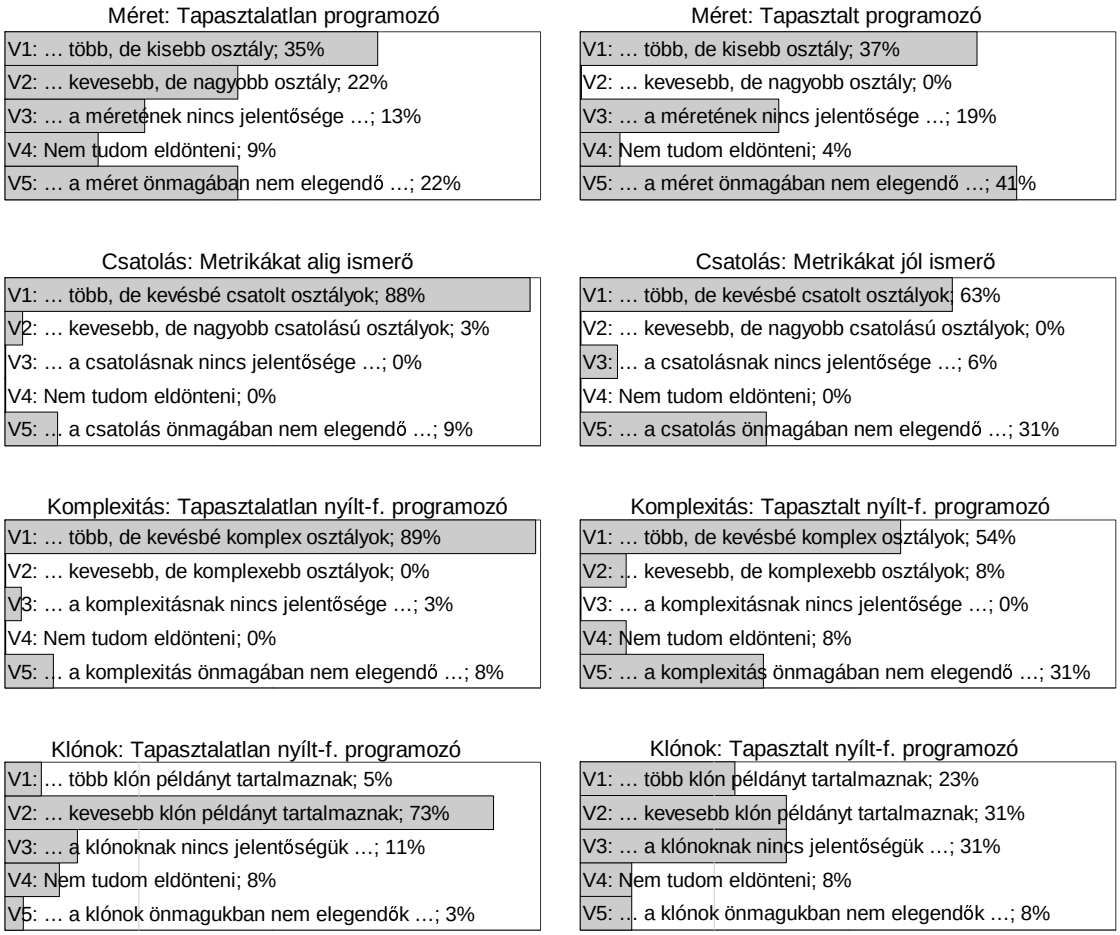
A 6.3. táblázat tartalmazza az I. kérdésre adott válaszok eloszlásait. A vastaggal kiemelt számok jelölik azokat a válaszokat, amelyeket legtöbben jelöltek meg az adott kérdésre. A program megértés szempontjából a résztvevők fele mondta azt, hogy a méret alapú metrika helyett mást választana (V_5), míg 28%-a mondta, hogy könnyebb megérteni a több, de kisebb osztályból álló rendszert (V_1). A tesztelés esetében ezek az arányok jelentősen eltérnek, mert a résztvevők 36%-a jelölte meg, hogy könnyebb tesztelni a több kisebb osztályból álló rendszert (V_1), de csak egy kicsit kevesebben jelölték meg (32%), hogy a méret helyett más metrikát választanának (V_5). A komplexitás, csatolás és klónok esetében a résztvevők több mint fele mondta, hogy könnyebb a rendszert megérteni, ha több, de kevésbé komplex osztályból áll, vagy több osztály van a rendszerben, de azok közt kisebb a csatolás, vagy több osztály van, de azok kevesebb klónt tartalmaznak (V_1). A tesztelés esetében lényegesen többen jelölték meg ugyanezeket a válaszokat (V_1).

A vizsgálatok során azt tapasztaltuk, hogy a megértés szempontjából nem volt szignifikáns különbség a junior és a senior résztvevők válaszai között. Másfelől a tesztelés szempontjából a 20 vizsgált esetből 4-nél (azaz az esetek 20%-ánál) azt tapasztaltuk, hogy a tapasztalat és a szakértelem szignifikánsan befolyásolta a metrikák megítélését. A 6.1. ábra mutatja a junior és senior résztvevők válaszainak az eloszlását azon kérdések esetében, ahol a két csoport válaszai között szignifikáns különbséget találtunk. Ezen négy eset magyarázata és a belőlük levonható következtetések a következők:

- *Programozói tapasztalat és a méret-alapú metrika kapcsolata*: A programozásban tapasztalatlan résztvevők 22%-a gondolta úgy, hogy könnyebb a kevesebb nagyobb osztályt tesztelni, míg a programozásban tapasztaltak teljesen elutasították ezt a válasz-lehetőséget. Másfelől sokkal több (41% a 22%-kal szemben) programozásban tapasztalt résztvevő mondta azt, hogy a méret önmagában nem elegendő.

- *Metrika ismeret és a csatolás metrika kapcsolata*: A metrikát jól ismerő résztvevők közül 25%-kal kevesebben gondolták azt, hogy az alacsonyabb csatolású osztályokat könnyebb tesztelni, míg a tapasztaltak közül szignifikánsan többen (8% a 31%-kal szemben) választották azt, hogy a csatolás önmagában nem elegendő a tesztelés megítélésére.

- *Tapasztalat nyílt-forráskódú fejlesztésben és a komplexitás kapcsolata*: A nyílt-forráskódú fejlesztésben tapasztalatlan résztvevők jelentős része (89%) szerint a több, de kevésbé komplex osztályokat könnyebb tesztelni, míg a tapasztalt nyílt-forráskódú fejlesztőknek csupán 54%-a – azaz 34%-kal kevesebben – osztja ezt a véleményt. Viszont a nyílt-forráskódú fejlesztők közel egyharmada (31%) mondta, hogy a komplexitás önmagában nem elegendő, míg a másik csoport mindössze 8%-a válaszolta ugyanezt.



6.1. ábra. A junior és senior fejlesztők válaszainak eloszlása az I. kérdés esetében

- *Tapasztalat nyílt-forráskódú fejlesztésben és a klónok kapcsolata*: A legtöbb tapasztalatlan nyílt-forráskódú fejlesztő (73%) azt mondta, hogy a kevesebb klónt, de több osztályt tartalmazó rendszert könnyebb tesztelni. Ezzel szemben a tapasztaltak megosztottak ebben a kérdésben, mert három válasz is közel azonos jelölést kapott (23%, 31% és 31%).

Elfogadható kivételek a rossz metrika értékekre

Amikor egy adott programrész rossz metrika értékekkel rendelkezik (például nagy a csatolás metrikája), akkor ajánlott annak a résznek az átszervezése, hogy növeljük a kód minőségét. Azonban vannak olyan esetek, amikor mégis elfogadjuk a rossz metrika értéket. Például nem fogunk újratervezni egy jól ismert tervezési mintát pusztán a nem megfelelő metrikái miatt. A következő kérdéssel azt vizsgáljuk, hogy a fejlesztők milyen okokat tudnak elfogadni a rossz metrika értékekre.

II. kérdés: *Milyen indokokat tudna elfogadni arra, hogy egy osztály* mérete *túl nagy legyen? (Többet választ is meg lehet jelölni.)*

- V_1: Semmi sem indokolhatja.
- V_2: Egy jól ismert tervezési minta miatt túl nagy a méret.
- V_3: A megvalósított funkcionalitás követeli meg a nagy méretet.

- V_4: A nagy méretű osztály kódja generált.
- V_5: Valamilyen API-nak kell eleget tennie.
- V_6: Amennyiben nem nehezíti a megértést, tesztelést, stb.
- V_7: Letesztelt, jól működik, kár lenne hozzányúlni.
- V_8: Nem tudom eldönteni.
- V_9: Egyéb indok (szöveges magyarázat).

Metrika	V_1	V_2	V_3	V_4	V_5	V_6	V_7	V_8	V_9
Méret	2%	34%	52%	56%	56%	24%	28%	2%	4%
Komplexitás	2%	36%	80%	46%	30%	36%	28%	2%	0%
Csatolás	4%	36%	56%	38%	52%	26%	26%	8%	0%
Klónok	18%	42%	22%	68%	24%	12%	14%	4%	4%
Átlag	6.5%	37%	52.6%	52%	40.5%	24.5%	24%	4%	2%

6.4. táblázat. A különböző válaszok elfogadási arányai a II. kérdés esetében

A 6.4. táblázat tartalmazza a II. kérdésre adott válaszok eloszlását[1]. Az első érdekes észrevétel, hogy annak ellenére, hogy sok résztvevő azt mondta, hogy a rossz metrika-értékek megnehezítik a programok megértését és tesztelését, a *méret*, *komplexitás* és *csatolás* esetében csak nagyon kevesen jelölték meg azt, hogy semmilyen indokot nem fogadnának el (V_1) vagy hogy nem tudják eldönteni (V_8). Igaz, hogy többen utasították el a klónokat (18%), de még mindig nem olyan sokan, így általánosságban azt mondhatjuk, hogy bizonyos körülmények között a rossz metrika értékek is elfogadhatók. A V_2 és V_7 közti válaszok mindegyike jelentős számú szavazatot kapott. A leginkább elfogadott indok a *megvalósított funkcionalitás* (V_3) és a *generált kód* (V_4) volt, de a *tervezési minta* (V_2) és az *előre megadott API* (V_5) is sok szavazatot kapott. A maradék két válaszlehetőség (V_6 és V_7) szintén nem elhanyagolható, bár azokat kevésbé támogatták.

Megvizsgáltuk a junior és senior részvevők válaszai közötti különbséget is. Mivel az adott kérdésekre több választ is meg lehetett jelölni, ezért ezeket a válaszokat külön-külön kezeltük, és azt vizsgáltuk, hogy a junior és senior résztvevők válaszai eltérnek-e az adott kivételek (lehetséges válaszok) esetében. Mivel a V_1, V_8 és V_9 válaszokat nagyon kevesen jelölték meg, ezért azokat nem elemeztük. Valójában ez nem nagy veszteség, mivel ezek a válaszok az „én nem tudom” válasz szinonimái. A 6.5. táblázat tartalmazza a vizsgálat eredményét, ahol a sorokban a lehetséges kifogások szerepelnek, míg az oszlopokban a metrika-csoportok vannak. Ahol szignifikáns különbséget találtunk a junior és senior résztvevők válaszai között, ott az adott tapasztalat vagy szakértelem rövidítését beírtuk a táblázat megfelelő sorába. Például a *teszt* a második sorban (V_3) és a második oszlopban (komplexitás) azt jelenti, hogy szignifikáns különbség van a tapasztalt és tapasztalatlan tesztelők válaszai között.

Mivel túl sok szignifikáns különbséget találtunk, ezért általánosan fogjuk vizsgálni azokat, és csak egy konkrét példát fogunk részletesen ismertetni. Először a tapasztalat és szakértelem szempontjából ismertetjük az eredményeket. A metrikák ismerete

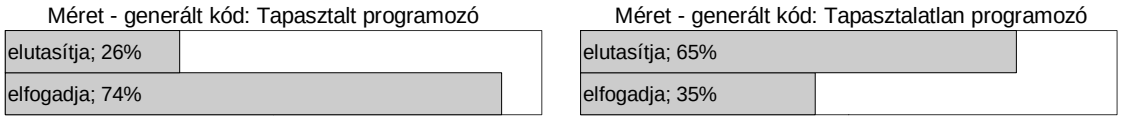
[1]Mivel a kérdésre tetszőleges számú választ meg lehetett jelölni, ezért itt a válaszok arányainak összege nem 100%.

	Méret	Komplexitás	Csatolás	Klónok
V_2				
V_3		teszt	ny.f., teszt	
V_4	tap, met, C++, teszt	tap, met, C++, teszt	tap, met	tap, met, C++, teszt
V_5	tap, met			
V_6				
V_7	tap, met, C++, ny.f.	met	ny.f.	met, test

6.5. táblázat. A szignifikáns eltérések a II. kérdés esetében

(met) a 24-ből 8-szor, azaz az esetek 33%-ban befolyásolta szignifikánsan a metrikák megítélését. A programozói tapasztalat (tap) és a tesztelői tapasztalat (teszt) 6-szor (25%) befolyásolta a metrikák megítélését, míg a C++ nyelven szerzett tapasztalat (C++) 4-szer (16,7%). A nyílt-forráskódú tapasztalatnak volt a legkisebb befolyása a válaszokra, mert csak 3 esetben (12,5%) különbözött szignifikánsan a junior és senior résztvevők véleménye. Ebből a kérdésből azt a következtetést vonhatjuk le, hogy minden vizsgált tapasztalat és szakértelem területnek komoly befolyása van a metrikák megítélésére.

A következőkben a kapott eredményeket a lehetséges válaszok szempontjából fogjuk megvizsgálni, és azt nézzük meg, hogy melyik hány esetben osztotta meg a résztvevőket. A *generált osztályok* (V_4) kérdése osztotta meg leggyakrabban a junior és senior résztvevőket: a programozói tapasztalat és a metrikák ismerete minden esetben megosztotta a résztvevők véleményét, míg a C++ tapasztalat és a tesztelési tapasztalat háromszor befolyásolta a válaszokat szignifikánsan. Ami ezek után meglepő, hogy a nyílt-forráskódú tapasztalatnak ezen válasz esetén semmilyen befolyása nem volt. A *tesztelt kód* (V_7) volt a másik válasz, amelyik nagyon megosztotta a résztvevők válaszait. Ebben az esetben a *méret-alapú* metrikák megítélése négyszer különbözött szignifikánsan, a klónok megítélése kétszer különbözött, míg a másik két metrika csoport megítélése az ötből négy esetben lényegében azonos volt. Összegezve azt mondhatjuk, hogy ebben az esetben a 20 esetből 8-ban különbözött szignifikánsan a junior és senior résztvevők véleménye. Ezen eredmények után meglepő lehet, hogy a *tervezési minták* (V_2) és a *jól érthető kód* (V_6) megítélése nem különbözött szignifikánsan. Továbbá, hogy a *megvalósított funkcionalitás* (V_3) és az *adott API-nak történő megfelelés* (V_5) válaszok csak két-három esetben osztották meg a résztvevők véleményét.



6.2. ábra. A junior és senior résztvevők válaszai a II. kérdésre

A hely hiánya miatt nem tudjuk mind a 27 szignifikáns különbséget egyesével elemezni, ezért csak egy példát fogunk bemutatni. A 6.2. ábra mutatja, hogy a tapasztalt programozók 74%-a fogadja el a nagy generált osztályokat, és csupán 26%-uk utasítja el azokat. Ezzel szemben a tapasztalatlan programozók csupán 35%-a fogadja el a nagy osztályokat, míg 65%-a elutasítja. Ez a példa is mutatja, hogy a junior és senior résztvevők véleménye mennyire különbözhet.

A tesztelési erőforrások hatékonyabb szétosztása a metrikák segítségével

A tesztelés a szoftverfejlesztés egy igen fontos fázisa. A célja az, hogy a programban található összes hibát felfedezzük, azonban ez nagyobb programok esetében lehetetlen, mivel a tesztelési erőforrások (tesztelők száma, tesztelésre fordítható idő, ...) korlátozottak. Ezért a rendelkezésre álló tesztelési erőforrásokat szét kell osztanunk a program különböző részei között, de nem mindegy, hogy ezt hogyan tesszük meg. Minél jobban szét tudjuk osztani a tesztelési erőforrásokat, annál hatékonyabb lesz maga a tesztelés, ami többek közt azt jelenti, hogy több hibát találunk meg ugyanakkora ráfordítással. A következő kérdések segítségével azt vizsgáltuk meg, hogy a résztvevők hogyan osztanák el a tesztelési erőforrásokat akkor, ha rendelkezésükre állnának metrika információk. Egy nagyon egyszerű példa (csak két osztály használunk) segítségével vizsgáljuk ezt a kérdést.

III. kérdés:*Tegyük fel, hogy egy ismeretlen rendszer két osztályát kell tesztelni, ahol az A osztály 1000 sorból áll (LLOC), míg a B osztály 5000 sort tartalmaz. Továbbá tegyük fel, hogy a két osztály minősége közel azonos. A fejlesztés során az A osztály mérete 10%-kal, míg a B osztályé 2%-kal növekedett. Hogyan osztaná szét a rendelkezésre álló tesztelési erőforrásokat?*

- V_1: Csak az A osztályt tesztelném.
- V_2: A rendelkezésre álló tesztelési erőforrás 90%-át az A osztály tesztelésére, míg 10%-át a B osztály tesztelésére fordítanám.
- V_3: A rendelkezésre álló tesztelési erőforrás 75%-át az A osztály tesztelésére, míg 25%-át a B osztály tesztelésére fordítanám.
- V_4: A rendelkezésre álló tesztelési erőforrást fele-fele arányban osztanám szét a két osztály között.
- V_5: A rendelkezésre álló tesztelési erőforrás 25%-át az A osztály tesztelésére, míg 75%-át a B osztály tesztelésére fordítanám.
- V_6: A rendelkezésre álló tesztelési erőforrás 10%-át az A osztály tesztelésére, míg 90%-át a B osztály tesztelésére fordítanám.
- V_7: Csak a B osztályt tesztelném.
- V_8: Nem a méret alapján ítélném meg.
- V_9: Nem tudom eldönteni.

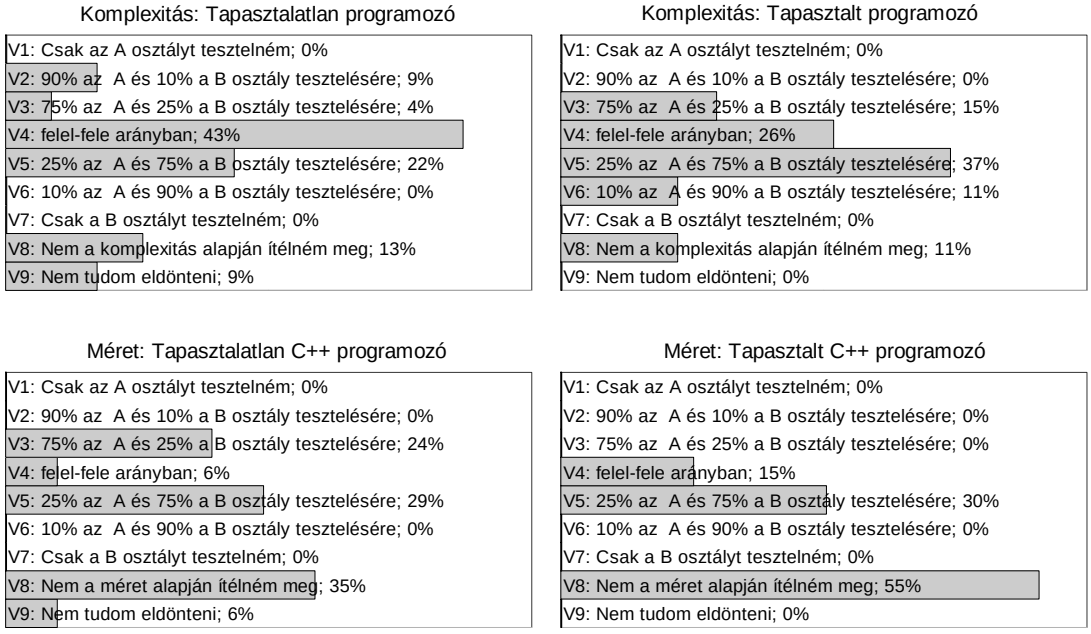
Ugyanezt a kérdést tettük fel a komplexitásra (WMC) is, ahol az A osztály komplexitása 100, míg a B osztályé 500 volt, továbbá a csatolásra (CBO) is, ahol az A osztály csatolása 20, míg a B osztályé 100 volt. Ezt a kérdést nem tettük fel a klónokra, mert nem lett volna értelme.

A 6.6. táblázat tartalmazza a válaszokat a III. kérdésre. Majdnem a résztvevők fele (48%) válaszolta azt, hogy a tesztelési erőforrásokat nem a méret alapján ítélnék meg (V_8), míg 30%-uk válaszolta, hogy a tesztelési erőforrás 75%-át az A osztályra költené (V_5). A komplexitás esetében a legtöbben (36%) azt válaszolták, hogy a tesztelési erőforrásokat egyformán kell szétosztani a két osztály között (V_4), de csak egy kicsivel kevesebben (30%) mondták azt, hogy a tesztelési erőforrások 75%-át az A osztályra

Metrika	V_1	V_2	V_3	V_4	V_5	V_6	V_7	V_8	V_9
Méret	0%	0%	8%	12%	30%	0%	0%	**48%**	2%
Komplexitás	0%	4%	10%	**34%**	30%	6%	0%	12%	4%
Csatolás	0%	2%	10%	**28%**	**28%**	6%	4%	20%	2%

6.6. táblázat. A III. kérdésre adott válaszok eloszlása

költenék (V_5). A csatolás esetében a V_4 és V_5 válasz ugyanannyi szavazatot kapott (mind a kettőt a résztvevők 28%-a választotta), így ez az eredmény nagyon hasonlít a komplexitásnál kapott eredményre.



6.3. ábra. A junior és senior fejlesztők válaszainak eloszlása a III. kérdésre

Azt tapasztaltuk, hogy a III. kérdés esetében a junior és senior fejlesztők által adott válaszok két esetben tértek el szignifikánsan. A 6.3. ábra szemlélteti ezt a két esetet, valamint a magyarázatuk a következő:

- *Programozói tapasztalat és a komplexitás metrika kapcsolata*: A tapasztalt programozók 48%-a (37% plusz 11%) gondolta úgy, hogy a tesztelés szempontjából az abszolút komplexitás a mérvadó és nem a fejlesztés során bekövetkezett változás (V_5 és V_6). Ezzel szemben a tapasztalatlan programozók közül legtöbben (43%) egyenlő mértékben osztanák szét az erőforrásokat (V_4).
- *C++ nyelvű tapasztalat és a méret-alapú metrika kapcsolata*: A senior C++ programozóknak több mint a fele (55%) mondta azt, hogy nem a méret alapján terveznék meg a tesztelést (V_8), míg 30%-uk mondta azt, hogy az abszolút méret sokkal fontosabb, mint a fejlesztés során tapasztalt növekedés (V_5). A junior C++ programozók sokkal megosztottabbak, mivel két egymásnak ellentmondó válasz (V_3 és V_5) kapott sok szavazatot (24% és 29%).

Az eddig bemutatott négy kérdés-kategória mellett voltak további kérdések is, amelyek szintén a metrikákat vizsgálták egyesével. Ezeket röviden ismertetjük, de nem elemezzük a válaszokat.

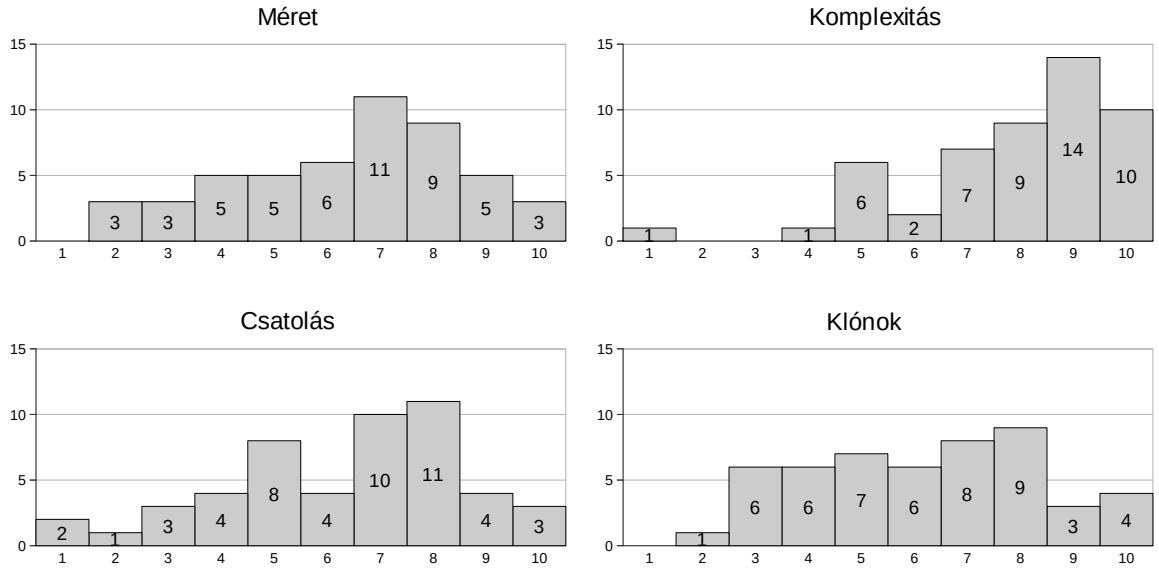
A kérdőív elkészítéséhez 10 C/C++ rendszert elemeztünk, és kiszámoltuk a metrikáikat. A 10 elemzett rendszer közül 6 ipari és 4 nyílt-forráskódú rendszer volt. Az ipari projektek közt volt többek között telekommunikációs rendszer és grafikai alkalmazás is, míg a négy nyílt-forráskódú rendszer a Tamarin [50], a WebKit [53], a Mozilla [41] és az OpenOffice.org [42]. Minden rendszerre kiszámoltuk a vizsgált metrikák átlagát, valamint azt, hogy az osztályok hány százalékának a metrikája haladja meg az átlag harmadát. Ezután minden rendszerre mindkét értéket név nélkül feltüntettük egy diagrammon. A résztvevőknek az volt a feladatuk, hogy a diagramm segítségével a rendszerek minőségét becsüljék meg egy 7 fokozatú skálán. A skála a nagyon rossz minőségűtől a nagyon jó minőségűig terjedt. Ezt a kísérletet mind a négy metrikával elvégeztük.

Azt is vizsgáltuk, hogy a résztvevők szerint mennyi lenne az osztályok optimális mérete, komplexitása vagy csatolása, ahol válaszként egy minimum és egy maximum értéket lehetett megadni. A kód duplikációk esetében azt kérdeztük, hogy mi az a kódméret, amelynél hosszabb klónokat már nem engedhetünk meg a kódban[2].

6.1.3. Kérdések a metrikák relatív fontosságáról

A kérdőív harmadik felében a metrikák egymáshoz viszonyított fontosságát illetve használhatóságát mértük különböző szempontok alapján. Ez azt jelenti, hogy egy kérdésben több metrikát is felhasználtunk, és a résztvevőknek ki kellett választaniuk, hogy mely metrika a fontosabb, vagy rangsorolniuk kellett azokat.

Ebből a részből csak egy kérdést (IV. kérdés) fogunk részletesen tárgyalni, amely a metrikák fontosságát mérte a tesztelés szempontjából. A résztvevőknek az volt a feladata, hogy értékeljék a négy metrikát (méret, komplexitás, csatolás és kód duplikációk) aszerint, hogy melyik mennyire fontos a tesztelés szempontjából. Mind a négy metrikát 1-től 10-ig kellett osztályozni, ahol az 1 jelentette a legkevésbé fontosat, míg a 10 jelentette a legfontosabbat.



6.4. ábra. A résztvevők válaszainak eloszlása a IV. kérdésre

[2]Megjegyezzük, hogy ez a kérdés nem áll kapcsolatban az eddig vizsgált kód duplikáció metrikával, mivel az a klón-példányok számát mérte, míg ez a kérdés a klón-példányok hosszára kérdez rá.

A 6.4. ábra mutatja a IV. kérdésre adott válaszok hisztogramját. Amint azt láthatjuk, a résztvevők szerint a *komplexitás* metrika a legfontosabb, mivel a két legnagyobb súly kapta a legtöbb szavazatot, továbbá a súlyok átlaga (7,88) ebben az esetben a legnagyobb. A másik három metrika esetében a két legtöbb szavazatot kapott súly a 7 és a 8, továbbá a súlyok átlaga (a méret esetében 6,40, a csatolásra 6,34 és a klónokra 6,20) is közel azonos. Annak ellenére, hogy a három metrika esetében a válaszok eloszlása egy kicsit eltér (lásd 6.4. ábra), az mondhatjuk, hogy a fontosságuk közel azonos, de a részvevők szerint nem annyira fontosak, mint a komplexitás. Ez az eredmény kicsit meglepő a 3. és a 4. fejezetben kapott eredmények fényében, ahol azt kaptuk, hogy a csatolás (CBO) és a méret (LLOC) jobb eredményt adott, mint a komplexitás (WMC).

6.2. Az eredményeinek lehetséges felhasználásai

A kérdőívben csak néhány metrikát használtunk, illetve a szoftverkarbantartásnak csak egy egészen kis területét érintettük, azonban eredmények mégis hasznosak számunkra. Sok esetben kiderült, junior és senior fejlesztők véleménye szignifikánsan eltért, ami azt jelenti, hogy a különböző területen szerzett tapasztalat jelentősen befolyásolta a metrikák gyakorlati alkalmazását. Továbbá számos alkalommal tapasztaltuk azt, hogy még a részvevők homogén csoportjának (például a tapasztalt fejlesztőknek) sem volt egységes a véleménye a metrikák használhatóságáról (például a 6.1. ábrán a méret megítélése). Ezek a nem várt eredmények rámutatnak arra, hogy a metrikák gyakorlati alkalmazása sokkal nehezebb, mint az gondolnánk. Ahhoz, hogy kiaknázzuk a metrikákban rejlő lehetőségeket, meg kell ismertetnünk a fejlesztőkkel a metrikákat, és meg tanítanunk nekik a metrikák megfelelő használatát. Továbbá meg kell vizsgálnunk azokat az eseteket, ahol a fejlesztők véleménye eltért az „elméleti" eredményektől. Például a tesztelés szempontjából a fejlesztőknek fontosabb a komplexitás, mint a csatolás vagy a méret, ami ellentmond a korábbi fejezetek eredményeinek. Egy másik példa, hogy a résztvevők több mint a fele azt válaszolta, hogy a generált kód esetében a rossz metrika érték nem jelentik azt, hogy a kód maga is rossz lenne. Ezeket az eredményeket felhasználva finomíthatjuk a hiba-előrejelző modelljeinket, hogy figyelembe vegyék a fejlesztők véleményét, és a „speciális körülmények között" annak megfelelően működjenek. Ezzel jobb eredményeket érhetünk el, és ami még fontosabb, a felhasználók számára is elfogadhatóbb előrejelzéseket tudnánk előállítani.

6.3. Kapcsolódó munkák

Kevés olyan kutatás folyt eddig, amelyben a szoftverfejlesztők tudását vagy képességeit vizsgálták volna. Azonban a mi kísérletünk is rávilágít arra, hogy a fejlesztők véleményének megismerése elengedhetetlen a különböző módszerek hatékony alkalmazásához. Ezt támasztja alá José és munkatársai [13] által elvégzett vizsgálat is.

Az ISO/IEC 9126 nemzetközi szabvány [34] definiálja a szoftver minősége és a ISO/IEC 9126 által definiált alacsonyabb szintű minőségi jellemzők közti kapcsolatot. A Software Improvement Group (SIG) bevezetett még egy szintet az alacsony szintű minőségi jellemzők alá, amely a rendszer tulajdonságaiból állt, majd meghatározta a kapcsolatot a két szint között [31]. José és munkatársai egy kérdőív segítségével a különböző szinten definiált tulajdonságok közti kapcsolatot vizsgálták a karbantarthatóság szempontjából. A SIG 22 szakembere vett részt a kísérletükben. Az volt a feladatuk,

hogy mind a 4 alacsony szintű karbantarthatósági jellemzőt összehasonlítsák egymással (6 összehasonlítás), továbbá ugyanezt kellett tenniük mind a 4 alacsony szintű jellemzőhöz tartozó 9 rendszer tulajdonságai esetében is (4-szer 36 összehasonlítás). Így a résztvevőknek összesen 150 összehasonlítást kellett elvégezniük. Az összehasonlításhoz egy 1-től (egyformán fontosak) 5-ig (kiemelkedően fontosabb) terjedő skálán kellett osztályozniuk a tulajdonságok egymáshoz viszonyított fontosságát. A vizsgálattal a következő három kérdést szerették volna megválaszolni:

- *A szakemberek egyetértenek-e a súlyok nagyságában?*
  A kísérlet azt mutatta, hogy az alacsony szintű jellemzők esetében a 4 esetből 2-szer nem volt egyetértés a szakemberek közt, míg a rendszer tulajdonságok esetében a 36-ból 7-szer különbözött a véleményük.

- *A vizsgálat eredményeként kapott súlyozott fontosság mennyire esik egybe a korábban a SIG által definiált kapcsolatokkal [31]?*
  Az eredmények 21 esetben megerősítették a SIG által definiált kapcsolatokat, azonban 2 esetben új kapcsolatot találtak, míg 6 esetben a definícióban szereplő kapcsolat a szakemberek véleménye alapján nem létezik. A vizsgálatnál nem vették figyelembe azt a 7 esetet, amelynél nem volt volt egyetértés a szakemberek között.

- *Felhasználhatjuk-e az eredményeket a létező modell finomítására?*
  Az alacsony szintű minőségi jellemzők és a karbantarthatóság közti kapcsolat esetében ez eredmények alapján a tesztelhetőség súlyát növelni, míg a stabilitás súlyát csökkenteni kellene. Azonban a szakemberek véleménye nem volt annyira egységes, hogy ezt a változtatást javasolhassák.
  A rendszer tulajdonságai és az alacsony szintű jellemzők esetében már sokkal nagyobb volt a szakemberek közti egyetértés, melynek következtében javaslatot tettek néhány változtatásra.

6.4. Az eredmények összegzése

Azt szerettük volna megtudni, hogy a fejlesztők milyen mértékben képesek a metrikák nyújtotta előnyöket a gyakorlatban is hasznosítani. Ezért készítettünk egy kérdőívet, aminek a segítségével felmértük, hogy a fejlesztők hogyan használnák a metrikákat a különböző szoftver karbantartási feladatok esetében, illetve megtudtuk, hogy számukra melyek a fontos metrikák. Továbbá igazoltuk azt is, hogy a különböző területeken szerzett tapasztalat jelentősen befolyásolja a metrikák használhatóságának megítélését. Ezek az eredmények segítenek pontosítani a hiba-előrejelző modelljeinket.

A bemutatott vizsgálat ötlete és témája nem a szerző eredménye. Annak részletes kidolgozását, végrehajtását, valamint az eredmények kiértékelését a szerző saját eredményének tekinti.

7. fejezet

Az eredmények összegzése

Az értekezésben az objektum-orientált metrikák ipari alkalmazhatóságát vizsgáltuk. Ennek első lépéseként az objektum-orientált metrikák kiszámítását tárgyaltuk. Ismertettünk néhány problémát azok közül, amelyek felmerülhetnek egy nagy C++ rendszer elemzésénél. Ennek megoldására kidolgoztuk a Columbus technológiát, melynek segítségével képesek vagyunk tetszőleges C++ rendszert elemezni. A Columbus technológia részeként kidolgoztunk egy nyelvfüggetlen modellt, amely képes egy objektum-orientált rendszer magas szintű ábrázolására, illetve megvalósítottuk a C++ nyelv konverzióját erre a nyelvfüggetlen modellre. A metrikák kiszámítása is a nyelvfüggetlen modellen történik, amely modell segítségével hasonlóan tudunk Java és C# programokat is ábrázolni, illetve azok metrikáit kiszámítani. Kidolgoztunk továbbá egy heurisztikát, melynek segítségével automatikusan összegyűjthetjük egy adott szoftver hibakövető rendszeréből a bejelentett és kijavított hibákat, és a forráskódelemekhez rendelhetjük azokat. A Mozilla 9 különböző verzióját elemeztük, kiszámoltuk az objektum-orientált metrikákat, és a Mozilla hibakövető rendszeréből összegyűjtött hibákat hozzárendeltük az osztályokhoz.

Először a Mozilla 1.6-os verzióján vizsgáltuk a Chidamber és Kemerer [12] által definiált 6 objektum-orientált metrika, illetve az LLOC metrika és az osztályokban található hibák száma közti kapcsolatot. A vizsgálatokhoz lineáris és logisztikus regressziót, illetve döntési fát és neuron hálót használtunk, de az eredmények azt mutatták, hogy nincs lényeges különbség a módszerek eredményei között. A vizsgált metrikák közül a CBO metrika bizonyult a legjobb hiba-előrejelzőnek, de az LLOC metrika csak kevéssel volt rosszabb. A NOC metrika kivételével a további metrikák is alkalmasnak bizonyultak a hibák előrejelzésére, csak különböző mértékben. Az objektum-orientált metrikák és a hibák száma közti kapcsolatot már korábban számos esetben igazolták, így a kísérletünk igazi jelentősége az, hogy az elsők közt igazoltuk ezt az összefüggést ipari méretű rendszer esetében.

Az első kísérletben csak néhány metrikát vizsgáltunk meg, ezért a kísérletet kiterjesztettük, melyben már a metrika-kategóriákat vizsgáltunk. A vizsgálatokhoz felhasznált 58 objektum-orientált metrikát az öt metrika-kategória (méret, komplexitás, öröklődés, kohézió és csatolás) valamelyikébe soroltuk, és a kategóriák hasznosságát a kategóriába tartozó metrikák határozták meg. Ez alapján azt mondhatjuk, hogy a csatolás metrikák a legjobb hiba-előrejelzők, de a vizsgált komplexitás metrika és bizonyos méret-alapú metrikák is alkalmasak a hibák előrejelzésére. A kísérlet alapján a kohéziós metrikák alig, míg ez öröklődés metrikák egyáltalán nem voltak alkalmasak a hibák előrejelzésére.

Végezetül azt vizsgáltuk meg, hogy a fejlesztők hogyan alkalmaznák a metrikákat a szoftverfejlesztésben. Ehhez készítettünk egy kérdőívet, melyben három objektum-orientált metrika mellet egy klón metrikát is felhasználtunk, és arra voltunk kíváncsiak, hogy a szoftverfejlesztők hogyan alkalmaznák a metrikákat a szoftver megértésben és tesztelésben. Ezzel egy átfogó képet kaptunk arról, hogy mely területeken lehet hatékonyan alkalmazni a metrikákat. Emellett statisztikai módszerek alkalmazásával vizsgáltuk, hogy a különböző területeken szerzett tapasztalatnak milyen hatásai vannak a metrikák gyakorlati alkalmazására. Igazoltuk, hogy a tapasztalat számos esetben szignifikánsan befolyásolja a szoftverfejlesztők metrikákról kialakult véleményét.

A. Függelék

Az objektum-orientált metrikák definíciói

Méret alapú metrikák

Average of Lines of Code (AvgLOC) Az osztályban található metódusok LOC metrikáinak az átlaga.

Logical Lines of Code (LLOC) Az osztály (és annak metódusainak) nem üres és nem komment sorainak a száma.

Lines of Code (LOC) Az osztály (és annak metódusainak) sorainak a száma.

Number of Attributes (NA) Az osztály lokális és örökölt attribútumainak a száma.

Number of public Attributes ($\mathbf{NA}_{pub}$) Az osztály lokális és örökölt publikus attribútumainak a száma.

Number of protected Attributes ($\mathbf{NA}_{prot}$) Az osztály lokális és örökölt protected attribútumainak a száma.

Number of private Attributes ($\mathbf{NA}_{priv}$) Az osztály lokális és örökölt privát attribútumainak a száma.

Number of Attributes Local (NAL) Az osztály lokális attribútumainak a száma.

Number of public Attributes Local ($\mathbf{NAL}_{pub}$) Az osztály lokális publikus attribútumainak a száma.

Number of protected Attributes Local ($\mathbf{NAL}_{prot}$) Az osztály lokális protected attribútumainak a száma.

Number of private Attributes Local ($\mathbf{NAL}_{priv}$) Az osztály lokális privát attribútumainak a száma.

Number of Attributes and Methods (NAM) Az osztály összes attribútuma és metódusa (NA + NM).

Number of Attributes and Methods Local (NAML) Az osztály összes lokális attribútuma és metódusa (NA + NM).

Number of Methods (NM) Az osztály metódusainak a száma, melybe beletartoznak az örökölt metódusok és a deklarációk is.

Number of public Methods ($\mathbf{NM}_{pub}$) Az osztály publikus metódusainak a száma, melybe beletartoznak az örökölt metódusok és a deklarációk is.

Number of protected Methods ($\mathbf{NM}_{prot}$) Az osztály protected metódusainak a száma, melybe beletartoznak az örökölt metódusok és a deklarációk is.

Number of private Methods ($\mathbf{NM}_{priv}$) Az osztály privát metódusainak a száma, melybe beletartoznak az örökölt metódusok és a deklarációk is.

Number of Methods Local (NML) Az osztály lokális metódusainak a száma (amibe a deklarációk és a definíciók is beletartoznak).

Number of public Methods Local ($\mathbf{NML}_{pub}$) Az osztály lokális publikus metódusainak a száma (amibe a deklarációk és a definíciók is beletartoznak).

Number of protected Methods Local ($\mathbf{NML}_{prot}$) Az osztály lokális protected metódusainak a száma (amibe a deklarációk és a definíciók is beletartoznak).

Number of private Methods Local ($\mathbf{NML}_{priv}$) Az osztály lokális privát metódusainak a száma (amibe a deklarációk és a definíciók is beletartoznak).

Number of Methods Defined (NMD) Az osztály definiált metódusainak a száma (amibe az örökölt metódusok is beletartoznak).

Number of public Methods Defined ($\mathbf{NMD}_{pub}$) Az osztály definiált publikus metódusainak a száma (amibe az örökölt metódusok is beletartoznak).

Number of protected Methods Defined ($\mathbf{NMD}_{prot}$) Az osztály definiált protected metódusainak a száma (amibe az örökölt metódusok is beletartoznak).

Number of private Methods Defined ($\mathbf{NMD}_{priv}$) Az osztály definiált privát metódusainak a száma (amibe az örökölt metódusok is beletartoznak).

Number of Methods Locally Defined (NMLD) Az osztály lokális, definiált metódusainak a száma.

Number of public Methods Locally Defined ($\mathbf{NMLD}_{pub}$) Az osztály lokális, definiált publikus metódusainak a száma.

Number of protected Methods Locally Defined ($\mathbf{NMLD}_{prot}$) Az osztály lokális, definiált protected metódusainak a száma.

Number of private Methods Locally Defined ($\mathbf{NMLD}_{priv}$) Az osztály lokális, definiált privát metódusainak a száma.

Number of Incoming Invocations (NII) Azon függvények és metódusok halmazának számossága, amelyek hívják az osztály valamely metódusát.

Komplexitás metrikák

Weighted Methods per Class (WMC) Az osztály metódusainak súlyozott összege, ahol a metódus súlyának a McCabe-féle ciklomatikus komplexitást használjuk.

Öröklődés alapú metrikák

Average Inheritance Depth (AID) Az AID értéke nulla azokra az osztályokra, amelyeknek nincs ősosztálya. A többi osztály esetében az AID értéke az osztály közvetlen ősosztályaihoz tartozó AID értékek átlaga plusz egy.

Class-to-Leaf Depth (CLD) Az öröklődési hierarchiában az osztály alatt található szintek maximuma.

Depth of Inheritance Tree (DIT) A leghosszabb út az osztálytól valamely gyökér ősosztályáig az öröklődési hierarchiában.

Number of Ancestors (NOA) Az osztály őseinek a száma.

Number of Children (NOC) Az osztály közvetlen leszármazottainak a száma.

Number of Descendants (NOD) Az osztály összes leszármazottjának a száma.

Number of Parents (NOP) Az osztály közvetlen őseinek a száma.

Number of Methods Inherited (NMI) Az osztály ősosztályaiban található összes függvénydeklaráció száma.

Kohéziós metrikák

Cohesion (Coh) Tegyük fel, hogy az osztálynak m darab metódusa van, amelyek az $\{A_j, j = 1, \ldots, a\}$ attribútumokat használják. Jelölje $v(A_j)$ azt, hogy az A_j attribútumot hány metódus használja. Ekkor a Coh értékét a $\frac{\sum_{j=1}^{a} v(A_j)}{m*a}$ képlettel adhatjuk meg.

Connectivity 1 (Co1) Connectivity (1). Tekintsünk egy irányítatlan G gráfot, amelynek a csúcsai az osztály metódusai, és akkor van él két csúcs között, ha használnak közös attribútumot, vagy az egyik metódus hívja a másikat. Jelölje V a csúcsok, míg E az élek számát. Ekkor Co1 $= 2 * \frac{(E-(V-1))}{(V-1)(V-2)}$.

Connectivity 2 (Co2) Tekintsünk egy irányítatlan G gráfot, amelynek a csúcsai az osztály metódusai, és akkor van él két csúcs között, ha használnak közös attribútumot, vagy az egyik metódus hívja a másikat. Jelölje V a csúcsok, míg E az élek számát. Ekkor Co2 $= \frac{2*E}{V(V-1)}$.

Lack of COhesion in Methods (LCOM) Az osztály azon metóduspárjainak a száma, amelyek nem használnak közös attribútumot, mínusz azoknak a száma, amelyek használnak. Ha ez a különbség negatív lenne, akkor az LCOM értéke 0.

Lack of COhesion in Methods allowing Negative values (LCOMN) Az osztály azon metóduspárjainak a száma, amelyek nem használnak közös attribútumot, mínusz azoknak a száma, amelyek használnak.

Lack of COhesion in Methods 3 (LCOM3) Az osztály azon metóduspárjainak a száma, amelyek nem használnak közös attribútumot

Lack of COhesion in Methods 4 (LCOM4) Tekintsünk egy irányítatlan G gráfot, amelynek a csúcsai az osztály metódusai, és akkor van él két csúcs között, ha használnak közös attribútumot. A metrika értéke a G-ben található összefüggő komponensek száma.

Lack of COhesion in Methods 5 (LCOM5) Tekintsünk egy irányítatlan G gráfot, amelynek a csúcsai az osztály metódusai, és akkor van él két csúcs között, ha használnak közös attribútumot vagy az egyik metódus hívja a másikat. A metrika értéke a G-ben található összefüggő komponensek száma.

Lack of COhesion in Methods 6 (LCOM6) Jelölje $\{M_i, i = 1, \ldots, m\}$ az osztály metódusait, $\{A_j, j = 1, \ldots, a\}$ az attribútumait, és $v(A_j)$ azt, hogy az A_j attribútumot hány metódus használja. Ekkor az LCOM6 metrika értékét a $\frac{\frac{1}{a} * \sum_{j=1}^{a} v(A_j) - m}{1-m}$ képlettel számolhatjuk ki.

Number of Local Method Access (NLMA) Az NLMA metrika értéke az osztály azon saját (lokális vagy örökölt) metódusainak a száma, amelyeket az osztály saját metódusai hívnak.

Number of Local Method Access without Inheritance ($\mathbf{NLMA}_{ni}$) Az NLMA_{ni} metrika értéke az osztály azon saját lokális metódusainak a száma, amelyeket az osztály saját metódusai hívnak.

Csatolás metrikák

Coupling Between Object classes (CBO) Egy osztály közvetlenül kapcsolódik egy másikhoz, ha használja annak metódusait és/vagy attribútumait. A CBO azon osztályok száma, amelyekhez a vizsgált osztály közvetlenül kapcsolódik.

Number of Foreign Methods Accessed (NFMA) Az osztály metódusai által közvetlenül hívott nem saját és nem örökölt metódusok halmazának a számossága.

Number of Foreign Methods Accessed without inheritance ($\mathbf{NFMA}_{ni}$) Az osztály metódusai által közvetlenül hívott nem saját lokális metódusok halmazának a számossága.

Number of Outgoing Invocations (NOI) Az osztály metódusai által közvetlenül hívott metódusok és függvények halmazának a számossága.

Response set For a Class (RFC) Az RFC metrika egy C osztályra az $RFC(C) = \left|M \cup \bigcup_{i=1}^{n} R_i\right|$ képlettel számolható ki, ahol M az n lokális metódussal rendelkező C osztály metódusainak a halmaza, míg R_i az osztály i-edik metódusa által közvetlenül hívott metódusok halmaza. Azaz az RFC metrika azon metódusok számát jelöli, amelyek végrehajtódhatnak válaszul egy, az adott osztályból példányosított objektum felé irányuló hívásnál.

Response set For a Class 1 (RFC1) Az RFC1 metrikát ugyanúgy számoljuk ki, mint at RFC-t, csak (az M halmaz meghatározásakor) figyelembe vesszük az örökölt metódusokat is.

Response set For a Class 2 (RFC2) Az RFC2 metrikát ugyanúgy számoljuk ki, mint at RFC-t, csak a metrika értékébe beleszámoljuk a tranzitívan hívott metódusokat is.

Response set For a Class 3 (RFC3) Az RFC3 az RFC1 és RFC2 metrika kombinációja, azaz figyelembe vesszük az örökölt metódusokat és az tranzitívan hívottakat egyaránt.

B. Függelék

Magyar nyelvű összefoglaló

Bevezetés

Egy szoftver életciklusa nem ér véget a program megírásával. Miután átadták az ügyfélnek az alkalmazást, következik a szoftver karbantartása és továbbfejlesztése. Ezért annak ellenére, hogy az ügyfelek számára elsősorban a megbízhatóság vagy a használhatóság a legfontosabb szempont, addig a fejlesztés és a költségek szempontjából a rendszer karbantarthatóságára is figyelnünk kell. A szoftver karbantartás egyik fontos része a szoftverben található hibák felfedezése és azok kijavítása. Mivel a tesztelés során lehetetlen az összes hibát megtalálni, ezért a tesztelés célja az, hogy minél több hibát találjunk meg, minél kevesebb költséggel. Bármilyen segítség, ami növeli a tesztelés hatékonyságát, egyben növeli a szoftver minőségét is, és csökkenti a költségeket.

Korábban számos kísérletben igazolták, hogy az objektum-orientált metrikák és az osztályok hibára való hajlamossága, azaz a szoftver minősége között létezik kapcsolat. Ez azt jelenti, hogy az objektum-orientált metrikák folyamatos mérésével és vizsgálatával javíthatjuk a szoftver minőségét és karbantarthatóságát, valamint segítségükkel növelhetjük a tesztelés hatékonyságát.

Az értekezés témája az objektum-orientált metrikák vizsgálata, amely három nagyobb részre osztható. Először bemutattuk az általunk kidolgozott technológiát, amely lehetővé teszi nagy rendszerek esetében az objektum-orientált metrikák kiszámítását. Ezután a Mozilla rendszer esetében igazoltuk, hogy van kapcsolat a metrikák és az osztályok hibára való hajlamossága között. Az értekezés zárásaként pedig a metrikák gyakorlati alkalmazását vizsgáltuk, illetve bebizonyítottuk, hogy szignifikáns kapcsolat van a programozói tapasztalat és a metrikák megítélése között.

A Columbus technológia

(A kapcsolódó eredmények a [21, 28] publikációkban találhatóak.)

Ahhoz, hogy nagy C++ rendszerek esetében is vizsgálhassuk a metrikák és a szoftver minősége közti kapcsolatot, először a metrika értékeket kell kiszámolnunk. Azonban az ipari méretű rendszerek nagyon bonyolultak lehetnek, melyeknek már a fordításuk sem egyszerű. Több nagy rendszer megvizsgálásával számos problémát találtunk, amelyek nagyon megnehezítik a rendszerek elemzését és ezzel együtt a metrikák kiszámítását. A problémák megoldására kidolgoztuk egy technológiát, amelynek segítségével egyszerűen, az eredeti kód megváltoztatása nélkül képesek vagyunk tetszőleges C++ rendszert elemezni. A kidolgozott technológia platformfüggetlen, azaz Windows

és Linux operációs rendszereken is működik, illetve a különböző fordítóprogramokat (például Microsoft Visual Studio vagy GCC) is egyformán támogatja. A technológia segítségével elemeztük a Mozillát és az OpenOffice.org-ot, illetve sikeresen alkalmaztuk több ipari rendszer estében is, amelyek közül a legnagyobb 30 millió programsorból állt.

A metrikák kiszámítására kidolgoztunk egy nyelvfüggetlen modellt, amely alkalmas objektum-orientált rendszerek magas szintű ábrázolására. Továbbá megvalósítottuk a C++, Java és C# nyelvek konverzióját erre a nyelvfüggetlen reprezentációra, így ezen nyelvek esetében ezt használjuk az objektum-orientált metrikák kiszámítására.

Egy szoftverben talált hibákat a szoftverhez tartozó hibakövető rendszerben tartják nyílván, amely eltárol minden információt a hibákról azok bejelentésétől a kijavításáig. Kidolgoztunk egy módszert a bejelentett és kijavított hibák automatikus kigyűjtésére, majd egy heurisztika segítségével az osztályokhoz rendeltük azokat.

A bemutatott technológiák segítségével a Mozilla kilenc verziójának elemzése után kiszámoltuk a metrikákat, illetve a hibakövető rendszerét felhasználva meghatároztuk a Mozilla osztályaiban található hibák számát.

Metrika alapú hiba-előrejelző modellek

(A kapcsolódó eredmények a [28] publikációban találhatók.)

A Columbus technológiát felhasználva a Mozilla különböző verzióira kiszámoltuk a Chidamber és Kemerer által definiált metrikákat [12] és a hagyományos nem-üres és nem-komment sorok száma (LLOC) metrikát, valamint meghatároztuk az osztályokban megtalált és kijavított hibák számát. Egy Mozilla verziót kiválasztottunk, hogy lineáris és logisztikus regresszió, valamint döntési fa és neuron háló segítségével vizsgáljuk az osztályok metrikái és az osztályokban található hibák száma, illetve az osztályok hibára való hajlamossága közti kapcsolatot. A metrikákat elemeztük különkülön is, illetve megvizsgáltuk, hogy több metrika együttes használatával javíthatjuk-e a modellek hatékonyságát. A kapott eredmények azt mutatták, hogy a különböző módszerek közel azonos eredményt adtak, mely szerint szinte minden tekintetben a CBO metrika volt a legjobb hiba-előrejelző, amitől az LLOC csak kevéssel maradt el. A további öt vizsgált metrika gyengébben teljesített, míg NOC alkalmatlannak bizonyult a hibák előrejelzésére. A több metrikát használó modell sem tudott egyértelműen jobb eredményt elérni, mint a CBO metrika önmagában.

Korábban már számos kísérlettel igazolták, hogy van kapcsolat az objektum-orientált metrikák és a hibák száma között, így az eredményeink nem voltak meglepőek. Azonban a korábbi vizsgálatok mind kis, vagy közepes méretű rendszeren történtek, így a vizsgálatunk igazi jelentősége az, hogy az elsők között igazoltuk a metrikák és a hibák közti kapcsolatot egy nagy rendszer esetében.

A metrika-kategóriák hiba-előrejelző képességeinek vizsgálata

(A kapcsolódó eredmények a [47] publikációban találhatók.)

Az első kísérletben csak nyolc metrikát vizsgáltunk meg, amelyekből nehéz általános következtetéseket levonni. Ezért kiterjesztettük a kísérletet metrika-kategóriák vizsgálatára. Ehhez 58 metrikát használtunk fel, amelyeket az öt kategória (méret,

komplexitás, csatolás, öröklődés és kohézió) egyikébe soroltunk, és a kategória hibaelőrejelző képességét a benne található metrikák alapján határoztuk meg. Mivel korábban a négy alkalmazott módszer közel azonos eredményt adott, ezért itt csak a logisztikus regressziót alkalmaztuk. 17 metrika esetében találtunk szignifikáns kapcsolatot a metrikák és a hibák között, amely alapján egyértelműen kijelenthettük, hogy a csatolás metrikák a legjobbak a hibák előrejelzésében. Emellett a komplexitás metrikák, illetve a méret-alapú metrika-kategórián belül a sorok számát különböző módon mérő metrikák, valamint a metódusok és a publikus metódusok számát mérő metrikák is hasznosnak bizonyultak. A kohéziós metrikák nagyon gyenge eredményt értek el, míg egyetlen öröklődés metrika sem volt alkalmas a hibák előrejelzésére.

A metrikák használhatóságának vizsgálata a fejlesztők szemszögéből

(A kapcsolódó eredmények a [48] publikációban találhatók.)

A Columbus technológia alkalmas arra, hogy tetszőleges rendszer esetében kiszámoljuk a metrika értékeket. Igazoltuk, hogy van összefüggés a metrikák és a szoftver minősége között, így azt gondolhatnánk, hogy minden rendelkezésünkre áll ahhoz, hogy a metrikák gyakorlati alkalmazásával javítsuk a szoftver minőségét. Azonban a metrikák eredményes gyakorlati alkalmazásához elengedhetetlen, hogy maguk a fejlesztők is tisztában legyenek a metrikák használatával. Ellenkező esetben csak hátráltatja a munkájukat a szoftver minőségének javítása helyett.

Készítettünk egy felmérést, amelyben a szoftverfejlesztők véleményét vizsgáltuk három objektum-orientált metrika és egy kód duplikációval kapcsolatos metrika felhasználásáról. Arra voltunk kíváncsiak, hogy szerintük milyen kapcsolat van a metrikák és a program megértése és tesztelése között, valamint megvizsgáltuk, hogy hogyan alkalmaznák a metrikákat különböző gyakorlati problémák megoldásában. A válaszok alapján egy általános képet kaptunk a szoftverfejlesztők metrikákról kialakult véleményéről, illetve megtudtuk, hogy szerintük mely területeken lehet hatékonyan alkalmazni a metrikákat, és melyeken nem.

Mivel a kísérletben résztvevő 50 szoftverfejlesztő különböző tapasztalatokkal rendelkezett, így megvizsgáltuk, hogy a tapasztalat hogyan befolyásolja a metrikák megítélését. Több esetben is igazoltuk, hogy szignifikáns különbség volt a tapasztalt és tapasztalatlan fejlesztők véleménye között, azaz a tapasztalat jelentősen befolyásolja a metrikák megítélését.

A kísérlet rávilágított arra, hogy a legjobb eszköz sem ér sokat, ha nem tudják helyesen használni, azaz nem elegendő a fejlesztők kezébe adni a „metrikákat”, meg is kell tanítani őket a megfelelő használatára. Továbbá a fejlesztők véleményének figyelembe vételével a hiba-előrejelző modelljeinket is javíthatjuk. Például a generált kódot és annak metrikáit másképp kell kezelni, és ezt a rendszer minőségének vizsgálatakor is figyelembe kell vennünk.

Konklúzió

Az értekezésben bemutattunk egy technológiát, amelynek segítségével nagy méretű C++, Java vagy C# nyelvű rendszerek esetében is képesek vagyunk kiszámolni az objektum-orientált metrikákat anélkül. Kidolgoztunk egy módszert, melynek segítségével egy adott szoftver hibakövető rendszeréből automatikusan össze tudjuk gyűjteni

a bejelentett és kijavított hibákat, majd egy heurisztika segítségével a forráskód elemekhez rendelhetjük azokat. Különböző statisztika és gépi tanulási módszerek alkalmazásával igazoltuk, hogy van kapcsolat a metrikák és a szoftver minősége között, majd meghatároztuk azokat a metrika-kategóriákat is, amelyek a legalkalmasabbak ennek mérésére.

Ezeket felhasználva egy olyan „eszközt" dolgoztunk ki, amely alkalmas a szoftver minőségének mérésére és a minőség változásának nyomon követésére. Azonban a fejlesztők körében elvégzett felmérés felhívja a figyelmet arra, hogy a fejlesztőknek meg kell tanítani használni és értelmezni a metrikákat ahhoz, hogy azok valóban alkalmasak legyenek a szoftver minőségének javítására.

Appendix C

Summary in English

Introduction

The lifecycle of software does not end with its programming. After delivering the software to the customer, the following step is the software maintenance and improvement. In addition to the improvement and cost aspect, we also have to take care about the software maintainability, although that the reliability and the usability are the most important qualities from the customers' aspects. One of the most important parts of software maintenance is finding the bugs in the software. Since it is impossible to find all bugs during testing phase, therefore the aim of testing is to find as many bugs as possible with the lowest cost. Any kind of help, that improves the efficiency of testing, improves the quality of the software and at the same time decreases the cost.

Previously many researches proved that there is connection between the object-oriented metrics and the fault-proneness property of the classes. This means that with the continuous measurement and analyzes of the object-oriented metrics, the quality and the maintainability of the software can be improved and the testing of the software can be more efficient.

The topic of the dissertation is the examination of the object-oriented metrics, which can be divided into three main parts. First, we introduced our technology for calculating object-oriented metrics of large software systems. Next, we proved that there was connection between the object-oriented metrics and the fault-proneness property of the class in the case of Mozilla. In the end, we examined the practical use of the metrics, and proved that there was significant relationship between programming experience and the judgement of metrics.

The Columbus technology

(Results related to this thesis can be found in [21, 28] publications.)

Before analyzing the connection between the metrics and software quality for large C++ systems, the metric values have to be calculated. Although industrial systems can be very complicated, their compilation is even more difficult. After examining several large systems many problems have been found, which made the analysis of the systems and the metrics calculation very difficult. Therefore we have worked out a technology, which helped us to be able to analyze an arbitrary C++ system easily, without modifying its source code. This technology is platform-independent, which means that it works on Windows and Linux, and supports the different compilers (for

example, Microsoft Visual Studio or GCC) as well. By applying this technology, we analyzed Mozilla and openOffice.org and applied it successfully with more industrial systems as well, among which the largest one contained 30 million lines of code.

We developed a language independent model that is a suitable high level representation of the object-oriented systems. In addition, we implemented the conversion from C++, Java and C# languages to that language independent model, so it can be used for calculating object-oriented metrics for these languages.

Bugs found in software are usually stored in its bug tacking system, which keeps all information about the bug from its report to its correction. We worked out a method for the automatic collection of the reported and corrected bugs and we associate these bugs with the classes by applying a heuristic.

With the introduced technology, we analyzed nine Mozilla versions, calculated the metric values, and determined the number of bugs found in the classes by using the bug tracking system of Mozilla.

Metric-based fault-predictor models

(Results related to this thesis can be found in [28] publication.)

By using the Columbus technology, we calculated the Chidamber and Kemerer metric suite [12] and the traditional non-empty and non-comment lines of code (LLOC) metric for different Mozilla versions, and we determined the number of the found and corrected bugs occurred in the classes. One Mozilla version was chosen to examine the relationship between the metric values and the fault-prone property of the classes. For this analysis logistic and linear regression and decision tree and neural network were used. We analyzed the metrics individually, but we examined whether the use of more metrics in one model could improve its quality. The results indicated that the different methods have had very similar results. From almost all aspects the CBO metric was the best fault-prone, but LLOC was only a slightly worse. The other five metrics have given poorer results while the NOC metric was useless for fault-prediction. The models using more metrics could not achieve obviously better results than the CBO metric itself.

As we mentioned many research validated that there is connection between the object-oriented metrics and the number of bugs so our results were not surprising. On the other hand, the earlier studies used only small or medium sized systems, so the main value of our results is that we are one of the firsts, who have proved the connection between the metrics and the bugs on a large system.

Examining the fault-prediction capabilities of metrics-categories

(Results related to this thesis can be found in [47] publication.)

In our first experiment only eight metrics were used so we cannot make general conclusions. Therefore we extended our experiment to metrics-categories as well. 58 metrics were chosen and they were classified according to the five metric-categories (size, complexity, coupling, inheritance and cohesion) and the fault-proneness capability of a category was determined by its metrics capability. Since the four different previously applied methods gave almost the same results, we decided to apply here

only the logistic regression. We have found significant correlation between the metrics and bugs in 17 cases, so we could say that the coupling metrics are the best fault-predictors. Besides, the complexity metrics, and the metrics from size category, which measured the lines of code of a class in different way or counted the methods and public methods were useful as well. Cohesion metrics gave very poor results and none of the inheritance metrics was suitable for bug prediction.

Investigating the usefulness of metrics from the point of view of developers

(Results related to this thesis can be found in [48] publication.)

By using the Columbus technology the metric values for an arbitrary system can be calculated. We proved that there is correlation between the metric values and the quality of the software so we may think that we have gained everything for improving the quality of the software by the practical use of metrics. But for the effective practical use of the metrics it is indispensable that the developers themselves should know how to use the metrics. Otherwise it only slows their work instead of improving the quality of the software.

Therefore we created a Survey to examine the developers' opinion about the use of three object-oriented and one code duplication metrics. We wanted to see that according to the participants what kind of relation was between the metrics and program comprehension & testing. Besides we also examined how they would apply the metrics in connection with practical problems. This way we got a general picture about the developers' opinion in connection with the metrics, and we got to know that according to their experiments in which part of software engineering could be the metrics applied efficiently.

Since the 50 participants, who took part in the experiment had different experience, we analyzed how the experiment influences the judgement of metrics. We proved in many cases, that there was a significant difference between the answers of the experienced and inexperienced participants, which means, that the experience significantly influences the judgement of the metrics.

This experiment reflects that the best tool does not worth much if we do not know how to use it, so it is not enough to give the "metrics" to the developers; we have to teach them to its use.

Besides we can improve our fault-predictor models by taking into account the developers' opinions. For example, the generated code and its metrics have to be treated differently, and we have to beware of this during the investigation of the software' quality.

Conclusions

We presented a technology that is suitable for calculating the object-oriented metrics for large C++, Java and C# systems. We have worked out a method for collecting automatically the reported and corrected bugs from a bug tracking system and with the help of our heuristic these are associated with the source code elements. By applying different statistical and machine learning methods we proved that there is connection between the metrics and the quality of the software, and then we determined those metric-categories, which are the most suitable for fault-prediction.

Using these results, we worked out such a "tool" that is able to measure the quality of the software and it is also able to follow the change of its quality. But the result of our survey attracts our attention to the fact that we have to teach the developers how to use and read the metrics, to make them suitable for improving the software's quality.

Irodalomjegyzék

[1] K. Ayari, P. Meshkinfam, G. Antoniol, and M. Di Penta. Threats on Building Models from CVS and Bugzilla Repositories: the Mozilla Case. In *Proceedings of the 2007 conference of the center for advanced studies on Collaborative research*, pages 215–228, October 2007.

[2] Tibor Bakota, Rudolf Ferenc, and Tibor Gyimóthy. Clone Smells in Software Evolution. In *Proceedings of the 23nd International Conference on Software Maintenance*, pages 24–33, October 2007.

[3] J. Bansiya and C.G. Davis. A Hierarchical Model for Object-Oriented Design Quality Assessment. *IEEE Transactions on Software Engineering*, 28:4–17, 2002.

[4] Victor R. Basili, Lionel C. Briand, and Walcélio L. Melo. A Validation of Object-Oriented Design Metrics as Quality Indicators. In *IEEE Transactions on Software Engineering*, volume 22, pages 751–761, October 1996.

[5] Christopher M. Bishop. *Neural Networks for Pattern Recognition.* Clarendon Press, Oxford., 1995.

[6] Lionel C. Briand, Walcelio L. Melo, and Jürgen Wüst. Assessing the Applicability of Fault-Proneness Models Across Object-Oriented Software Projects. In *IEEE Transactions on Software Engineering*, volume 28, pages 706–720, July 2002.

[7] Lionel C. Briand and Jürgen Wüst. Empirical Studies of Quality Models in Object-Oriented Systems. In *Advances in Computers*, volume 56, September 2002.

[8] Lionel C. Briand, Jürgen Wüst, John W. Daly, and D. Victor Porter. Exploring the Relationships between Design Measures and Software Quality in Object-Oriented Systems. In *The Journal of Systems and Software*, volume 51, pages 245–273, 2000.

[9] Bugzilla for Mozilla. `http://bugzilla.mozilla.org`.

[10] Bruce J. Chalmer. *Understanding Statistics.* CRC Press, 1986.

[11] Yih-Farn Chen, Emden R. Gansner, and Eleftherios Koutsofios. A C++ Data Model Supporting Reachability Analysis and Dead Code Detection. *IEEE Transanctions on Software Engineering*, 24(9):682–694, 1998.

[12] S.R. Chidamber and C.F. Kemerer. A Metrics Suite for Object-Oriented Design. In *IEEE Transactions on Software Engineering 20,6(1994)*, pages 476–493, 1994.

[13] José Pedro Correia, Yiannis Kanellopoulos, and Joost Visser. A Survey-based Study of the Mapping of System Properties to ISO/IEC 9126 Maintainability Characteristics. In *The International Conference on Software Maintenance (ICSM'09)*, pages 61–70. IEEE Computer Society, September 2009.

[14] S. Demeyer, S. Ducasse, and M. Lanza. A Hybrid Reverse Engineering Platform Combining Metrics and Program Visualization. In *Proceedings of WCRE'99*, 1999.

[15] Giovanni Denaro and Mauro Pezzè. An Empirical Evaluation of Fault-Proneness Models. In *ICSE 2002: Proceedings of the 24th International Conference on Software Engineering*, pages 241–251, March 2002.

[16] Premkumar T. Devanbu. GENOA: A Customizable, Language- and Front-End independent Code Analyzer. In *Proceedings of the 14th international conference on Software engineering*, pages 307–317, June 1992.

[17] Fernando Brito e Abreu and Walcelio Melo. Evaluating the Impact of Object-Oriented Design on Software Quality. *IEEE Third International Software Metrics Symposium*, pages 90–99, 1996.

[18] Eclipse Homepage.
`http://www.eclipse.org`.

[19] Rudolf Ferenc and Árpád Beszédes. Data Exchange with the Columbus Schema for C++. In *Proceedings of the 6th European Conference on Software Maintenance and Reengineering (CSMR 2002)*, pages 59–66. IEEE Computer Society, March 2002.

[20] Rudolf Ferenc, Árpád Beszédes, Mikko Tarkiainen, and Tibor Gyimóthy. Columbus – Reverse Engineering Tool and Schema for C++. In *Proceedings of the 18th International Conference on Software Maintenance (ICSM'02)*, pages 172–181. IEEE Computer Society, October 2002.

[21] Rudolf Ferenc, István Siket, and Tibor Gyimóthy. Extracting Facts from Open Source Software. In *Proceedings of the 20th International Conference on Software Maintenance (ICSM 2004)*, pages 60–69. IEEE Computer Society, September 2004.

[22] Rudolf Ferenc, Susan Elliott Sim, Richard C Holt, Rainer Koschke, and Tibor Gyimóthy. Towards a Standard Schema for C/C++. In *Proceedings of the 8th Working Conference on Reverse Engineering (WCRE 2001)*, pages 49–58. IEEE Computer Society, October 2001.

[23] P. Finnigan, R. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. Mueller, J. Mylopoulos, S. Perelgut, M. Stanley, and K. Wong. The Software Bookshelf. In *IBM Systems Journal*, volume 36, pages 564–593, November 1997.

[24] F. Fioravanti and P. Nesi. A Study on Fault-Proneness Detection of Object-Oriented Systems. In *Fifth European Conference on Software Maintenance and Reengineering (CSMR 2001)*, pages 121–130, March 2001.

[25] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Pub Co, 1999.

[26] FrontEndART Szoftver Kft.
`http://www.frontendart.com`.

[27] Michael W. Godfrey and Eric H. S. Lee. Secrets from the Monster: Extracting Mozilla's Software Architecture. In *Proceedings of the 2nd International Symposium on Constructing Software Engineering Tools (CoSET 2000)*, pages 15–23, June 2000.

[28] Tibor Gyimóthy, Rudolf Ferenc, and István Siket. Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction. In *IEEE Transactions on Software Engineering*, volume 31, pages 897–910. IEEE Computer Society, October 2005.

[29] Maurice H. Halstead. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., New York, NY, 1977.

[30] Sampson Gholston Hector M. Olague, Letha H. Etzkorn and Stephen Quattlebaum. Empirical Validation of Three Software Metrics Suites to Predict Fault-Proneness of Object-Oriented Classes Developed Using Highly Iterative or Agile Software Development Processes. In *IEEE Transactions on Software Engineering*, volume 33, pages 402–419, June 2007.

[31] Ilja Heitlager, Tobias Kuipers, and Joost Visser A. A Practical Model for Measuring Maintainability. *In 6th International Conference on the Quality of Information and Communications Technology, (QUATIC 2007)*, pages 30–39, September 2007.

[32] Ric Holt, Andreas Winter, and Andy Schürr. GXL: Towards a Standard Exchange Format. In *Proceedings of WCRE'00*, pages 162–171, November 2000.

[33] D. Hosmer and S. Lemeshow. *Applied Logistic Regression*. Wiley-Interscience, 1989.

[34] International Standards Organization. *Software engineering - product quality - part 1: Quality model*, ISO/IEC 9126-1 edition, 2001.

[35] International Standards Organization. *Programming languages – C++*, ISO/IEC 14882:2003(E) edition, 2003.

[36] Klocwork Inc. *Klocwork Issues and Metrics Reference*, 2008.

[37] Rainer Koschke, Raimar Falke, and Pierre Frenzel. Clone Detection Using Abstract Syntax Suffix Trees. In *13th Working Conference on Reverse Engineering*, pages 253–262, October 2006.

[38] Wei Li and Sallie Henry. Object-Oriented Metrics that Predict Maintainability. In *Journal of Systems and Software*, volume 23, pages 111–122, November 1993.

[39] C. Marinescu, R. Marinescu, P. F. Mihancea, and R. Wettel. iPlasma: An Integrated Platform for Quality Assessment of Object-Oriented Design. In *In ICSM (Industrial and Tool Volume*, pages 77–80. Society Press, 2005.

[40] Thomas J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, 1976.

[41] The Mozilla Homepage.
`http://www.mozilla.org`.

[42] OpenOffice.org Home Page.
`http://www.openoffice.org/`.

[43] J.R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[44] Christian Robottom Reis and Renata Pontin de Mattos Fortes. An Overview of the Software Engineering Process and Tools in the Mozilla Project. In *Proceedings of the Workshop on Open Source Software Development*, pages 155–175, February 2002.

[45] Rhino Home Page.
`http://www.mozilla.org/rhino`.

[46] Ferenc Rudolf. *Modelling and Reverse Engineering C++ Source Code*. PhD thesis, University of Szeged, 2004.

[47] István Siket. Evaluating the Effectiveness of Object-Oriented Metrics for Bug Prediction. *Periodica Polytechnica*, Budapest, 2009 Accepted for publication.

[48] István Siket and Tibor Gyimóthy. The Software Developers' View on Product Metrics; *A Survey-based Experiment. Annales Mathematicae et Informaticae*, Eger, 2010 Accepted for publication.

[49] Ramanath Subramanyan and M. S. Krishnan. Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects. In *IEEE Transactions on Software Engineering*, volume 29, pages 297–310, April 2003.

[50] Tamarin Home Page.
`http://www.mozilla.org/projects/tamarin/`.

[51] Mei-Huei Tang, Ming-Hung Kao, and Mei-Hwa Chen. An Empirical Study on Object-Oriented Metrics. In *Proceedings of the 6th International Symposium on Software Metrics*, pages 242–249, November 1999.

[52] László Vidács, Árpád Beszédes, and Rudolf Ferenc. Columbus Schema for C/C++ Preprocessing. In *Proceedings of the 8th European Conference on Software Maintenance and Reengineering (CSMR 2004)*, pages 75–84. IEEE Computer Society, March 2004.

[53] WebKit Home Page.
`http://webkit.org/`.

[54] Richard Wettel and Michele Lanza. Visualizing Software Systems as Cities. In *In Proc. of the 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 92–99, 2007.

[55] Richard Wettel and Michele Lanza. CodeCity: 3D Visualization of Large-Scale Software. In *In companion Proceedings of the 30th International Conference on Software Engineering, Research Demonstration Track*, pages 921–922, 2008.

[56] Ping Yu, Tarja Systä, and Hausi Müller. Predicting Fault-Proneness using OO Metrics: An Industrial Case Study. In *Sixth European Conference on Software Maintenance and Reengineering (CSMR 2002)*, pages 99–107, March 2002.

[57] Yuming Zhou and Hareton Leung. Empirical Analysis of Object-Oriented Design Metrics for Predicting High and Low Severity Faults. *IEEE Transactions on Software Engineering*, 32(10):771–789, 2006.

[58] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. Predicting Defects for Eclipse. In *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, page 9, May 2007.