

TDK-dolgozat

Tóth Zoltán Gábor

Szegedi Tudományegyetem
Informatikai Tanszékcsoport
Szoftverfejlesztés Tanszék

Folyamat metrikák hiba előrejelző képességének vizsgálata nyílt forráskódú környezetben

Tóth Zoltán

Programtervező Informatikus (MSc) szakos hallgató

Témavezetők:

Dr. Ferenc Rudolf egyetemi adjunktus

Dr. Siket István tudományos munkatárs

2012

Tartalomjegyzék

1. Bevezetés	2
2. Kapcsolódó munkák	4
3. A változtatáskövető és hiba előrejelző rendszer belső működése	6
3.1. A forráskód-pozíciók karbantartása	7
3.1.1. Forráskód elemek pozícióinak karbantartása egyértelmű esetekben	9
3.1.2. Forráskód elemek pozíciójának karbantartása nem egyértelmű esetekben.....	10
3.2. Metrikák és hiba-hozzárendelések forráskód elemekhez	12
3.2.1. Verziókövető rendszerekből származtatott metrikák kiszámítása forráskód elemekhez	13
3.2.2. A hibák hozzárendelése forráskód elemekhez.....	14
3.3. Az algoritmus	15
4. Eredmények	21
4.1. A változtatások hozzárendelése forráskód elemekhez	21
4.2. A tanulmányban használt metrikák közötti kapcsolatok	23
4.3. Gépi tanulási módszerek alkalmazása hiba előrejelzéshez	24
5. Összegzés és jövőbeli tervek.....	28
6. Függelék	31
7. Irodalomjegyzék.....	33

1. Bevezetés

Olyan világban élünk, ahol a környezetünk gyorsan változik és egyre újabb és újabb elektronikai eszközök jelennek meg. Mivel manapság már a legegyszerűbb készülékeket is speciálisan azokra készített szoftverek vezérlik, ezért a szoftverfejlesztés egyik nagy kihívása az elkészült szoftverek karbantartása és javítása, valamint azok adoptálása az új igényekhez. Azonban a folyamatos változások következtében a szoftver egyre inkább elhasználódik, egyre nehezebb lesz karbantartani, így egyre több hiba lesz megtalálható benne. Ennek következtében a tesztelés egyre költségesebb lesz. A hibák felderítésének egyik hatékony, ám annál költségesebb módszere a tesztelés. A tesztelési erőforrások elosztását segíthetjük azzal, ha hiba előrejelzéssel megadjuk a tesztelendő alkalmazás azon részeit, melyek nagyobb valószínűséggel fognak hibákat tartalmazni. Ilyen módon a tesztelési erőforrásokat hatékonyabban oszthatjuk el.

Mégis mi szerint tudjuk meghatározni a több hibával rendelkező komponenseket egy fejlesztett alkalmazásban? Milyen részek igényelnek több odafigyelést? A választ többféleképpen próbálták már megközelíteni. Korábban igazolták, hogy a statikus forráskód elemzésekkel előállított szoftver termék metrikák alkalmasak a hiba-előrejelzésre. Azonban az előrejelzés pontosításához a fejlesztés során használt verziókövető rendszerekből kinyerhető folyamat metrikákat is felhasználjuk, melyek jelentős eltérést mutatnak a forráskód metrikákhoz képest. A verziókövető rendszerek által nyilvántartott információkból kinyerhető metrikákat már vizsgálták korábban is, azonban a korábbi munkák elsősorban fájl szintű változásokat vettek figyelembe, míg jelen munkában a változásokat osztály és metódus szinten tudjuk követni.

A változtatás feldolgozás mellett nagy rendszerekben elengedhetetlen a hibák számontartása, nyomon követése. Erre a célra különböző hibakövető rendszereket használnak, melyekben a megtalált hibákról több információt is tárolnak. A kijavított hibák tulajdonságait felhasználva megkaphatjuk, hogy mely hibák milyen időintervallumban voltak jelen a rendszerben. A hibák javítása és a változtatások közötti kapcsolatot is felhasználjuk a hiba előrejelzésekhez.

Tanulmányunk célja a korábban kidolgozott folyamat metrikák számítási algoritmusának pontosítása, ennek implementálása, illetve a hibák forráskód elemekhez rendelésének továbbfejlesztése. Ennek segítségével lehetőségünk lesz a metrikák és a hiba

előfordulások közötti kapcsolatok vizsgálatára gépi tanulási módszerek segítségével. A vizsgálatokat a WebKit nyílt forráskódú rendszeren végeztük el

A vizsgálatok azt mutatták, hogy a verziókövető rendszerekből kinyerhető metrikák jó prediktoroknak bizonyultak a hibák előrejelzéséhez.

2. Kapcsolódó munkák

Ebben a fejezetben bemutatjuk azokat a tanulmányokat, melyek kapcsolódnak a kutatásunk témájához. Korábban már többen vizsgálták a verziókövető rendszerekből kinyerhető és felhasználható prediktorokat, és most ezen munkákat tekintjük át röviden

Általánosságban elmondható, hogy a változtatásokon alapuló folyamat metrikák hasznosabbak lehetnek a hibaszámok eloszlásának becslésére, mint a kódon alapuló termékmetrikák [1]. Például jobb indikátor lehet az a metrika, mely megmondja, hogy az adott kódrész hányszor módosult, mint az, hogy milyen hosszú az a kódrészlet. Az egyik hasznosnak bizonyuló mérőszám a kódrészek korára vonatkozik. Például vessünk össze két kódrészletet, amik hasonló metrikákkal rendelkeznek, esetleg hasonló funkciókat látnak el, viszont az egyik kódrészlet 5 éve a rendszerünkben van és már régóta nem változott, míg a másik csak pár hónapja került bele, és folyamatosan változik. A kettő közül melyikben lesz több hiba? Magától értetődik, hogy a „fiatal”, illetve a gyakran módosított kódrészekben még több hiba fordul elő. A változtatások méretét, gyakoriságát valamint a változások idejét felhasználva, hatékony becslési módszert készíthetünk. Ezeket felhasználhatjuk egy súlyozás kialakításához, mely segít meghatározni azokat a részeket, melyek hajlamosabbak a hibára. Kiszámoltak egy arányt is, miszerint egy év után ugyanabban a kódrészletben már csak egyharmad annyi hiba van [1].

Több tanulmányban találkozhatunk olyan metrikákkal, mint például a változtatást végző emberek száma [1] [2] [3]. Sokan azt hihetnék, ha többen írunk egy modult, akkor egy bizonyos emberszám felett nem csökkenne a hibaszám, hanem pont ellenkezőleg, nőne. A metrika alkalmazhatósága nem egyértelmű. Érték el vele sikereket is, de vannak olyanok is, akik nem fedeztek fel kézzel fogható összefüggést a fejlesztők száma és a hibák előfordulása között. Mivel kétes a metrika alkalmazhatósága, így a tanulmányunkban ezen prediktor használhatóságát is vizsgáljuk.

Fontos és használható metrikának ígérkezik az egy modulra levetített változtatások száma [1] [2]. A változtatásokat kisebb összetevőkre is bonthatjuk, ilyen módon mérhetjük a modulban a változtatott sorok számát is, melyet tovább bonthatunk hozzáadott és törölt sorokra is [3] [4] [5]. Ezen metrikák használhatósága jónak ígérkezik, bár köztük fellelhető korreláció is.

A témában számos megközelítés, tanulmány született már, melyekben egyéb metrikákat használnak fel a hibák előrejelzéséhez. Ilyen metrikák még a múltbeli hibák száma, a fejlesztők tapasztalata, akik a kódon változtattak, a változás által érintett fájlok száma. Továbbá az első és utolsó változtatás között eltelt idő, mely érintette az adott kódrészletet. Ezekről és még egyéb a témában használt metrikákról olvashatunk még a további tanulmányokban [6] [7] [8] [9] [10].

A fent említett tanulmányokban a metrikákat fájl szinten definiálták, aminek előnye, hogy a verziókövető rendszerekben a változások követése fájl szinten történik, és ennek megfelelően a korábbi változásokat is ilyen formában tudjuk kinyerni. Egy diff-nek nevezzük a forráskód két revíziója közötti különbséget leíró szöveges állományt. A változtatások során az előállított diff fájlok alapján egyértelműen és könnyen meg lehet határozni a fájl szintű metrikákat. Alacsonyabb szinten, például a metódusok szintjén, már nehezebb meghatározni az egyes elemekhez tartozó változásokat, így a metrikákat is. A következő fejezetben bemutatom, hogy milyen problémák merülnek fel, ha a fájlon belül található elemekre akarunk változásokat vagy metrikákat számolni, majd ismertetjük, hogy milyen megoldást dolgoztunk ki a problémára.

3. A változtatókövető és hiba előrejelző rendszer belső működése

A verziókövető rendszerekből kinyerhető információk forráskód elemekhez történő hozzárendelése nagyon összetett és egyes nyelveknél (például C vagy C++ esetén) szinte lehetetlen feladat még az emberek számára is. A bonyolultságát nem a komplexitása adja, hanem az a tény, miszerint vannak olyan változtatások, melyekről csak nehezen, alapos vizsgálatok után tudjuk egyértelműen eldönteni, hogy mely forráskód elemekhez rendelhetjük hozzá. Szerencsére az esetek legnagyobb részében a változások egyértelműen és könnyen beazonosíthatóak, és mi is ezzel foglalkozunk.

Ahhoz, hogy a forráskód elemek változásait pontosan követhessük szükséges, hogy minden egyes változás után statikusan elemezzük a rendszereket. A statikus elemzés alapján pontosan meg tudjuk mondani, hogy mely forráskód elem hol helyezkedik el a kódban, de az elemzés rengeteg erőforrással és idővel jár (Java rendszerek esetében ez elfogadhatónak mondható, de például C++ alkalmazások esetében nem). Meg lehet közelíteni úgy is az elemzés kérdését, hogy inkrementalitást viszünk bele a folyamatba, azaz nem elemezzük újra minden egyes revíziónál a rendszert, mindig csak a változtatásokat vesszük hozzá az új statikus forráskód elemzéshez. Az inkrementális elemzés C++ rendszerek esetében még így is nagy erőforrást igényel. Az egyetlen módszer a pontos kezdő- és befejező sor azonosítására csak a statikus elemzés lenne (minden revízióhoz), azonban nagy rendszereknél ez szinte lehetetlen. Ezek után a célunk az, hogy minél több változást minél pontosabban kövessünk, így a lehető legtöbb forráskód elemnek (osztály, metódus, függvény) ismerjük a pozícióját. A változtatókövetés alapú forráskód pozíció beállítása során információt veszíthetünk, de sok eset van, amikor ezek jól véghezvihetőek és pontos eredményt adnak. Például Java nyelvben nagy segítséget jelent az, hogy egy fájlban csak egy publikus osztály szerepelhet, aminek a következtében a fájl szintű változások egyszerűen hozzárendelhetően az adott osztályhoz. Ennek ellenére Java-ban is problémát jelenthet a metódusok forráskód pozícióinak beazonosítása, továbbá a nem publikus osztályok létezése és az a tény, hogy egy java kiterjesztésű fájlban általában az első néhány sor nem része a publikus osztálynak (import, package, stb.).

A verziókövető rendszerek különböznek egymástól, így nem meglepő, hogy a diff és log kimeneti formátumok sem egyeznek meg. A tanulmányomban a diff-ek mindig két

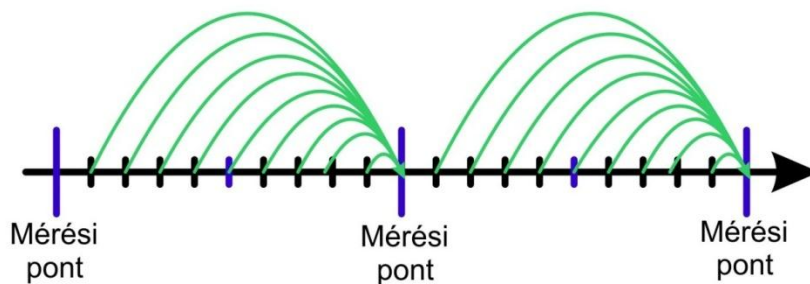
egymást követő revízió különbségét jelentik. A diff kimenetét minden esetben egy szöveges állományban tároljuk el. A log kimeneti fájlok alatt mindig az adott revízióhoz tartozó, a verziókövető rendszerből kinyerhető naplózási állományokat értjük. Ahhoz, hogy egy általánosan működő alkalmazást fejlesszünk, figyelembe kell vennünk a verziókövető rendszerek eltéréseit. Jelen tanulmányban megkülönböztetjük a Subversion [11] és a Git [12] rendszerek specifikusságát. A Subversion rendszer esetében is kiterjesztett diff formátumot alkalmazok, mely a Git-tel készített formátumhoz jobban hasonlít.

A következő kihívás, hogy a hibajavításokat hozzárendelhessük az egyes forráskód elemekhez. A hibajavítások hozzárendelése forráskód elemekhez nem egy új megközelítés. Ezt a technikát már többen alkalmazták korábban. Például az Impala rendszer Java nyelven implementált objektum-orientált rendszerek esetében képes beazonosítani azokat a forráskód elemeket, melyeket egy változtatás halmaz érintett [13]. Továbbá korábban a Szegedi Tudományegyetemen Szoftverfejlesztés Tanszékén is kifejlesztettek egy heurisztikát a hibák forráskód elemekhez rendelésére, és annak ellenére, hogy a megoldásukat sikeresen alkalmazták Mozilla rendszeren, ez a megoldás sem volt képes követni az elemek forráskód pozíciójának változását a két mérési pont között [14]. A tanulmányom egyik célja, hogy ezt az összerendelést, illetve eljárást tovább pontosítsa. A 3.2 (Metrikák és hiba-hozzárendelések) fejezetben részletesen ismertetem a hibajavítások hozzárendelésének mechanizmusát.

3.1. A forráskód-pozíciók karbantartása

Mivel csak bizonyos kiválasztott verziókat tudunk leelemezni, ezért két mérési pont közötti diff-ek feldolgozása során csak a forráskód információkat felhasználva, nehéz a forráskód elemekhez pontosan hozzárendelni a változásokat, mert nem ismert ez elemek pontos pozíciója. A 3.1. ábra azt a megoldást mutatja be, melyben az alap koncepció az, hogy nem állítunk semmilyen pozíciót sem a változtatásokkal együtt. A mérési pontoknál elvégzett elemzések alapján ismerjük az egyes forráskód elemek pozícióit, és ezt használjuk a hozzárendeléseknél. A mérési pontok között található egységek egy-egy változtatást (revíziót) jelölnek. A zöld nyilak jelképezik a hozzárendeléseket. Például, ha az első mérési pontban lévő forráskódot összehasonlítjuk a közvetlen utána található revízióbeli forráskóddal, akkor megkapjuk a két revízió közötti diff-et. A diff-ből megállapítható, hogy a forráskód mely sorai változtak. A változtatott sorok halmazát pedig megpróbáljuk

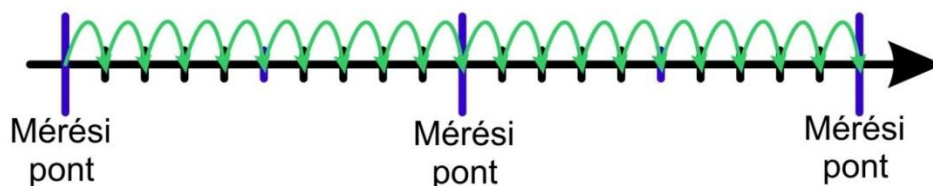
hozzárendelni az egyes forráskód elemekhez. A hozzárendelés a forráskód elemek pozíciója szerint történik, amit a következő mérési eredmények alapján kaptunk meg. A pozíciókat nem változtatjuk, mindig csak a következő mérési eredmények alapján végezzük a hozzárendelést. Tehát ebben az esetben elmarad az egyes forráskód elemek kezdő és végsorának karbantartása.



3.1. ábra – A forráskód pozíciók karbantartásának elvetése

A fenti megközelítés egyszerű, azonban ha két mérési pont között sok változás történik, akkor ez sok pontatlanságot okozhat.

A másik megközelítés az, hogy minden egyes új revízió feldolgozásával megpróbáljuk az elemek forráskód pozícióját is karbantartani. Ezt a megoldást mutatja be a 3.2. ábra. Itt minden egyes revízió hatására változtatjuk az elemek pozícióját az adott diff-ben történt változásoknak megfelelően. Tehát abban az esetben, amikor egyértelműen megállapítható, hogy egy változtatás, mely forráskód elemek pozícióit hogyan módosítja, akkor elvégezzük a pozíciók frissítését. Az eljárás így garantálja, hogy a lehető legfrissebb adatok alapján történjen meg a következő hozzárendelés.



3.2. ábra – Forráskód pozíciók karbantartása

Ebben az esetben nagyobb biztonsággal mondhatjuk egy későbbi hozzárendelésre is, hogy az konzisztens volt, viszont felvetődnek bizonyos problémák.

A verziókövető rendszerek fájl szinten dolgoznak, vagyis két fájl tartalmának különbségét képesek előállítani. Ezzel szemben mi a forráskód elemek változtatásait

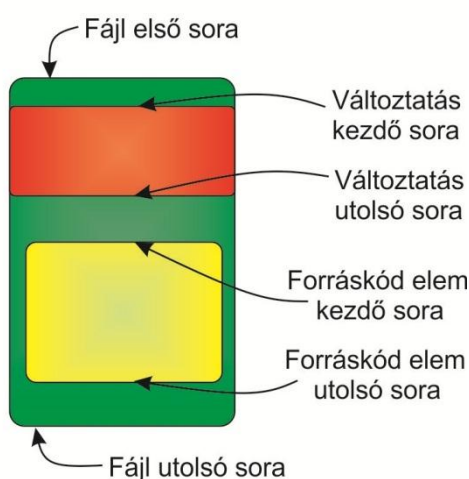
(forráskód pozíciót is) szeretnénk nyomon követni. Java nyelven egy fájl egy publikus osztályt jelöl, de például C++ programozási környezetben nincs ilyen megkötésünk. A metódusok szintjén ez a probléma hatványozottan érvényes. A következő alfejezetekben megvizsgáljuk azon eseteket, melyek a forráskód elemek pozícióinak frissítésekor felléphetnek.

3.1.1. Forráskód elemek pozícióinak karbantartása egyértelmű esetekben

A forráskód elemek változásait sok esetben egyértelműen frissíthetjük. A 3.3. ábra mutatja be a fájl, a forráskód elem és a változtatás viszonyát az 1. egyértelmű esetben. A zöld terület a fájl szimbolizálja, a sárga a vizsgált forráskód elemet (ebből lehet több is, azonban az ábrán a könnyebb érthetőség végett szerepel csak egy elem), a piros a változtatást jelképezi. Látható, hogy a változtatás és a forráskód elem törzse teljesen diszjunkt, azaz nem érintik egymást. Ekkor, ha a forráskód elem - az ábrán látható módon – a változtatás után található, akkor a forráskód elem első- és utolsó sorának pozíciója módosulni fog, mely módosítást egyértelműen el is végezhetjük. A változtatott sorok feldolgozása során számon tartjuk a hozzáadott és törölt sorok számát. A diff-ben a hozzáadott és a törölt sorokat különböző szimbólumokkal jelölik, így azok egyértelműen megkülönböztethetőek, és így az ε eltolás értéke is kiszámítható a következő módon:

$$\varepsilon = \text{hozzáadott sorok száma} - \text{törölt sorok száma}$$

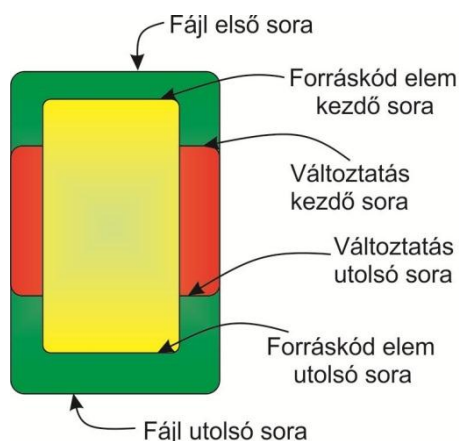
Ekkor a forráskód elem kezdő- és befejező sorának pozíciója ε -nal tolódik el.



3.3. ábra – Változtatás és forráskód elem viszonya – 1. egyértelmű eset

A másik egyértelmű esetet a 3.4. ábra mutatja be. Az előző példához hasonlóan az ábrán a zöld terület a fájl jelképezi, a sárga terület a forráskód elemet szimbolizálja, míg a

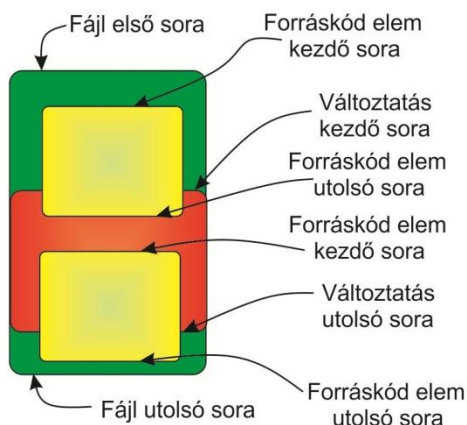
piros a változtatott területet jelenti. Látható, hogy ebben az esetben a változtatás egy adott forráskód elemen belül történt, azaz nem diszjunkt az elem és a változtatás, pontosabban az elem teljesen tartalmazza a változtatott sorok halmazát. Az ϵ eltolás megegyezik a korábban definiált eltolással, viszont azt ebben az esetben csak a forráskód elem utolsó sora pozíciójának módosításához használjuk fel. Az elem kezdő sora nem fog változni.



3.4. ábra Változtatás és forráskód elem viszonya – 2. egyértelmű eset

3.1.2. Forráskód elemek pozíciójának karbantartása nem egyértelmű esetekben

A forráskód elemek pozíciójának karbantartásakor felmerülhetnek bizonyos problémák, melyek akkor jelentkezhetnek, ha nem tudjuk egyértelműen meghatározni azt, hogy egy változtatás, mely elemeket hogyan érintette.

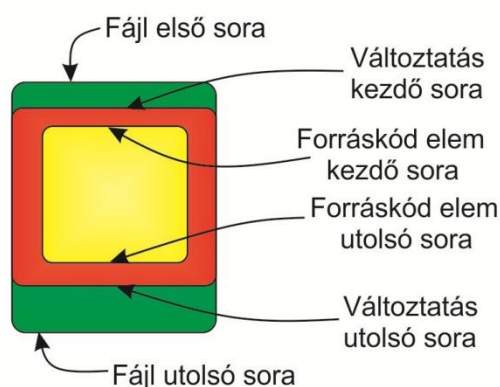


3.5. ábra - Változtatás és forráskód elem viszonya – 1. nem egyértelmű eset

Egy nem egyértelmű esetet mutat be a 3.5. ábra. Az ábrán látható zöld terület a fájlt testesíti meg. Most több forráskód elem is látható, melyeket a sárga területek szimbolizálnak. Ezek egymással diszjunktak, azaz egyik sem tartalmazza a másikat (például

egy osztályban lévő két metódus viszonya). A piros régió a változtatást jelképezi, mely mindkét forráskód elemet érinti. Ebben a helyzetben nem tudjuk egyértelműen megmondani, hogy az egyes forráskód elemek kezdő- és befejező sorainak pozíciói hogyan módosultak. A diff (változtatásokat tartalmazó fájl) feldolgozása során szét kellene választani a változtatások halmazát. A fenti esetben ez három halmazt jelentene: azok a változtatások, melyek az első forráskód elemhez tartoznak, melyek a második elemhez tartoznak és azok, melyek egyikhez sem. Az utóbbi halmaz lehet üres halmaz is. A probléma, hogy nem tudjuk meghatározni egy diff fájlban az egyes elemekhez tartozó változtatás halmazokat, mivel a verziókövető rendszerek fájl szinten tárolják és hajtják végre a változtatásokat.

A fent részletezett 1. nem egyértelmű esetben a változtatás nem feltétlen két forráskód elemet érint. Abban az esetben, ha több mint kettő forráskód elemet érint, akkor előáll egy másik eset, amit a 3.6. ábra - Változtatás és forráskód elem viszonya – 2. nem egyértelmű eset mutat be. A piros terület a változtatást, a sárga terület a forráskód elemet jelenti. Ebben az esetben a változtatás teljesen magában foglalja az adott forráskód elemet is (például egy metódust úgy módosítottak, hogy a metódus előtt és után is módosították még a sorokat). A probléma ugyanaz, mint az 1. nem egyértelmű esetben. A változtatáshalmazt 3 részre kell osztanunk, azaz meg kell találnunk a forráskód elemet érintő első és utolsó változtatott sort a halmazban, amiről már láttuk, hogy nagy kihívás.



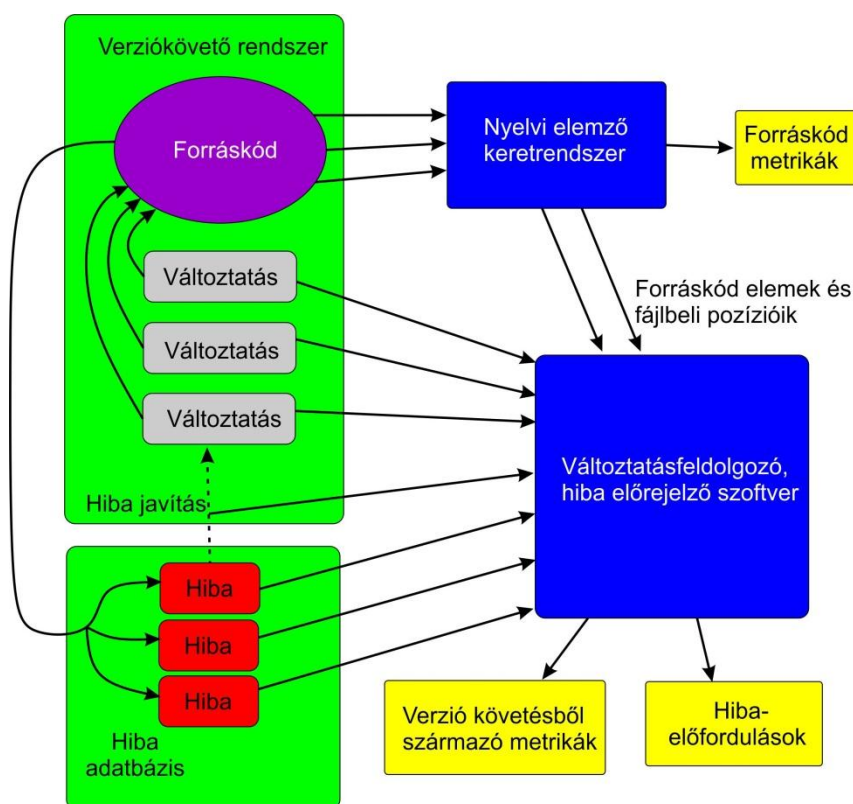
3.6. ábra - Változtatás és forráskód elem viszonya – 2. nem egyértelmű eset

A bemutatott két esetben kívül lehet még olyan szituáció, hogy az elem változását nem tudjuk beazonosítani, azonban a további esetekkel most nem foglalkozunk.

Jelen tanulmányban a nem egyértelmű esetekben nem frissítjük az elemek pozícióit, mivel ez hibás frissítéshez vezethet. A nem egyértelmű esetekben való frissítést heurisztikus módszerek tehetik lehetővé. Ehhez a nem egyértelmű változtatások további, mélyebb elemzése szükséges (például az elem nyitó és záró szimbólumának keresése). Az esetek nagy részében azonban sikerül egyértelműen hozzárendelni a változtatást a megfelelő forráskód elemhez. Például 1 havi intervallumra kiterjedő vizsgálat során a WebKit rendszerben közel 80%-ban sikerül a változtatások egyértelmű hozzárendelése a forráskód elemekhez. Az egy hónapos intervallum kiterjeszthető nagyobb időegységre is úgy, hogy a közel 80%-os arány ne sérüljön. Az arány megtartására született megoldás kulcsát a 3.3. fejezetben mutatjuk be.

3.2. Metrikák és hiba-hozzárendelések forráskód elemekhez

A tanulmányom fontos részét képezik azok a verziókövető rendszerekből származtatható metrikák, melyek segítségével pontosíthatjuk a hiba előrejelzéseket. Megvizsgálom a metrikák forráskód elemekhez való hozzárendelését is. Központi szerepet töltenek be az alkalmazásban a hibák, illetve a hibajavítások is.



3.7. ábra – A folyamat metrikákat kiszámoló és hibákat forráskód elemekhez rendelő rendszer felépítése

A 3.7. ábra mutatja be a hozzárendelési folyamatokat a rendszerben. Az ábra bal oldalán található a forráskód, melyet verziókövető rendszer használatával fejlesztenek, így minden változtatási halmaz később visszanyerhető. Az ábra másik zöld szekciója a hiba adatbázist testesíti meg, melyben a forráskódban fellelt hibákról tárolják az információt. A nyelvi elemző keretrendszer segítségével előállíthatóak különböző forráskód metrikák, továbbá ennek a kimenetét használva megtudhatjuk a forráskód elemek pozícióját is (azaz mely fájlban hol helyezkednek el). A tanulmányban ismertetésre kerülő alkalmazás felhasználja ezen információkat, továbbá a változtatásokat, a hibákat és az ezek között lévő kapcsolatokat, hogy új metrikákat állíthassunk elő és az elemekben talált és kijavított hibák számát is meghatározza.

3.2.1. Verziókövető rendszerekből származtatott metrikák kiszámítása forráskód elemekhez

A forráskód elemek pozícióinak karbantartása egy jelentős javítást, pontosítást jelent a feldolgozás során. Most megvizsgáljuk azokat a metrikákat, melyek felhasználhatóak a hiba előfordulások jóslásához. A változtatások feldolgozásakor különféle adatokat gyűjthetünk, melyek segíthetnek a hibajóslásban.

Az egyik metrika, melyet felhasználhatunk az, hogy változott-e az adott forráskód elem, vagyis minden elemhez egy bináris értéket rendelünk (például igaz és hamis értékek). Az igaz érték jelenti azt, hogy az adott forráskód elem módosult, hamis érték esetében nem érintette a forráskód elemet változtatás. A bináris osztályozásnál több információt tudhatunk meg, ha nem pusztán a változtatás tényét ragadjuk meg, hanem azt is megmondjuk, hogy egy elemet hányszor változtattak meg. Ez egy pontosabb értéket adhat. Például jelentős különbséget jelent, ha a bináris osztályozásnál két elemet sorolunk a „változott” osztályba, viszont az egyiket 100-szor a másikat csak egyszer módosították.

Többet információt nyújthat a módosítók száma. Változások halmazát egyszerre természetesen egy ember tudja végrehajtani, így két megadott revízió között azt számoljuk, hogy hány különböző ember hajtott végre változtatást. Abban az esetben, ha nem kötöttük volna ki a változtatást végző személyek különbözőségét, akkor a változtatások számát kapnánk.

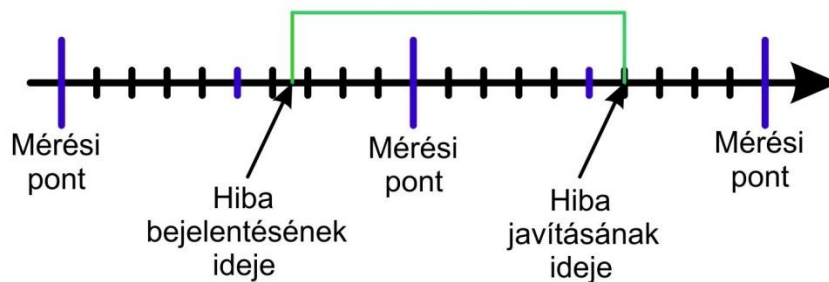
További metrikák, melyeket felhasználok a tanulmányban a sormódosítások száma. Tehát a törölt és hozzáadott sorok összege, melyeket külön-külön is hozzárendelünk az adott elemekhez.

A tanulmányomban használt metrikák definíciói: [1], [2],

- **Number of Committers (NCOM):** az adott forráskód elemet változtató különböző fejlesztők száma.
- **Number of Lines Changed (NLC):** Az adott forráskód elemhez tartozó változtatott sorok száma.
- **Number of Lines Added (NLA):** Az adott forráskód elemhez tartozó hozzáadott sorok száma.
- **Number of Lines Deleted (NLD):** Az adott forráskód elemhez tartozó törölt sorok száma.
- **Number of Bugs (NB):** Az adott forráskód elemet érintő hibák száma.
- **Number of Changes (NC):** Egy forráskód elem változtatásainak száma.

3.2.2. A hibák hozzárendelése forráskód elemekhez

Az egyes forráskód elemekhez hozzárendelhetjük a már megtalált és kijavított hibákat is, azaz megmondhatjuk, hogy egy adott elemet hány hiba érintett, vagy éppen érintette-e hiba egyáltalán. A hibákat egy arra alkalmas adatbázisban tárolják, ahonnan nyílt forráskódú projektek esetén ezeket le is kérhetjük. A hiba adatbázisok általában sok információt tartalmaznak egy-egy hibáról, azonban a tanulmányban ezek közül az adatok közül minket csak a hiba bejelentési és kijavítási ideje érdekel. Az általunk tanulmányozott rendszerben (WebKit) a Bugzilla adatbázist használják a hibák kezeléséhez [15]. A rendszerben található hibákat lekérhetjük egy szabványos szöveges állomány formájában is. A 3.8. ábra egy hiba életét mutatja be a rendszerben. Az ábra egy idővonalat mutat be, melyen három mérési pont látható (ezeken a pontokon készülnek elemzések). Az egységbeosztások a rendszer egy-egy revízióját jelentik. Láthatjuk továbbá egy hiba "életét" a rendszerben. Egy korábbi időpillanatban bejelentették a hibát, később pedig kijavították. A zölddel jelzett intervallumban volt megtalálható az adott hiba a rendszerben.



3.8. ábra – Egy hiba élete a rendszerben

A Webkit rendszerben a hibajavításoknak egyedi sémája van. A forráskód mellett karbantartanak úgynevezett ChangeLog fájlokat is, melyekben a változtatások szerepét írják le. Minden egyes változtatás végrehajtásnál megtalálható legalább egy ilyen fájl. A ChangeLog fájlokban jelzik, ha az adott változtatás halmaz egy hibajavításhoz köthető. Tehát a változtatások (diff) feldolgozása során megállapítható, hogy a változtatás hibajavítás volt-e. Abban az esetben, ha javítás volt, akkor a ChangeLog fájlból egyértelműen megkaphatjuk annak vagy azoknak a hibáknak az azonosítóját, melyeket javítani hivatott a változtatás. Ezek az azonosítók pontosan megegyeznek a Bugzilla rendszerben használt azonosítókkal. Miután már ismerjük az azonosítókat, lekérhetjük a hibákat leíró XML fájlokat. Tudjuk, hogy egy hibának mikor történt meg a javítása, mivel ez egybeesik a változtatás végrehajtásának időpontjával, amit egyszerűen kinyerhetünk a verziókövető rendszer által készített naplózó fájlból. A bejelentési dátumokat a Bugzilla rendszerből nyerhetjük ki. Az adott javításhoz tartozó patch alapján ismeretes, hogy mely forráskód elemeket érintette a hiba. Tehát kijelenthetjük azt, hogy a bejelentési és kijavítási időpontok közötti időintervallumban az érintett forráskód elemek hibát tartalmaztak.

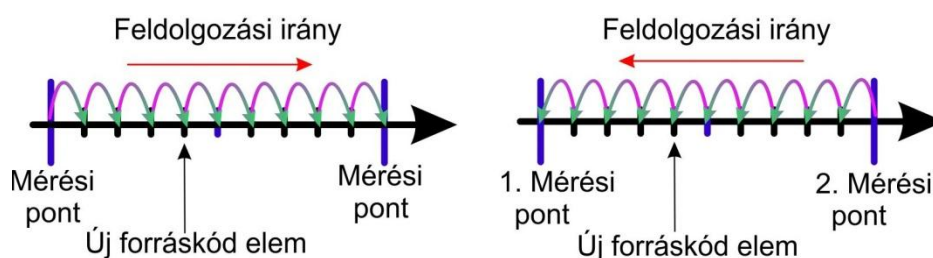
Előfordulhat az is, hogy egy hibajavításhoz több patch is tartozik. Ez például olyan esetekben fordulhat elő, amikor egy javítás nem volt teljes, azaz nem oldotta meg a problémát teljes egészében, vagy esetleg maga a javítás is hibás volt. Ekkor el kell döntenünk, hogy mit veszünk javításnak és mit nem. Ezt a 3.3 fejezetben bővebben fogom tárgyalni.

3.3. Az algoritmus

Az előző fejezetekben ismertetett elvek meghatároznak egy algoritmust, melyet ebben a fejezetben részletesen is ismertetek. A megoldás során több (néhol kissé technikai) elemeket is részleteiben megvizsgálók.

Első lépésként elő kell állítanunk az elemzendő rendszerről a szükséges statikus forráskód elemzéseket. A világban számos eszköz létezik ilyen célokra, azonban ebben a munkában az elemzést a Szegedi Tudományegyetem Szoftverfejlesztési Tanszéke által fejlesztett nyelvi elemzőcsomag segítségével valósítottuk meg. Ennek segítségével tetszőleges C++, Java vagy C# rendszerről előállíthatjuk a szükséges adatokat, azaz azt, hogy mely elemek hol találhatóak a forráskódban.

Miután rendelkezésre állnak az elemzési eredmények, elkezdhetjük a verziókövető rendszerben tárolt információk kinyerését és feldolgozását. Az első fontos cél az, hogy a jövőbeli hibák előfordulását, eloszlását szeretnénk becsülni. Az idő haladásával megegyező irányban végzett változtatás feldolgozás során képződhetnek új elemek, melyekről a kiindulási (memóriába töltött) halmazban nem rendelkezünk információval. Az új elemekről futás közben nem tudunk automatikusan információkat előállítani. Az új forráskód elemek azonban szerepelhetnek a szoftverben még sok revízió keresztül, és fontos szerepet is játszhatnak a hiba előrejelzésben. Ezzel ellentétben olyan forráskód elemekre, amik a változtatások feldolgozása közben megszűnnek, nem vagyunk kíváncsiak, mivel a jövőbeni hibák előfordulásakor nem is lesznek már jelen a rendszerben.



3.9. ábra – A változtatások feldolgozásának iránya

A 3.9. ábra mutatja be a megoldás előnyét. Az ábra bal és jobb oldali idővonala is két-két statikus forráskód elemzési pontot tartalmaz, továbbá mindkét esetben látható, hogy a rendszer fejlesztésekor az egyik változtatási pontnál új forráskód elem került be a rendszerbe. Az első (bal oldali) esetben az új elemről nem rendelkezünk információkkal, mivel az 1. mérési pont eredményeit használjuk fel kiindulási halmazként. A jobb oldali esetben (amikor a változtatások feldolgozásának iránya ellentétes az idő múlásának irányával) viszont rendelkezünk információkkal az új forráskód elemről, mivel a 2. mérési pont statikus forráskód elemzési eredményeit használtuk fel a kiindulási halmaz

meghatározásához, melyben már szerepelt az új elem is (ez csak akkor igaz, ha az új forráskód elemet nem törölték, még a 2. mérési pont előtt).

Ennek megfelelően a változtatások feldolgozását „visszafelé” mutató irányban végezzük el. A változtatások feldolgozásához szükség van magukra a változtatásokra is, melyeket az elkészített szoftver állít elő a verziókövető rendszerből. A verziókövető rendszerek általában (főleg nyílt forráskódú projektek esetén) csak lassabb válaszdőt képesek biztosítani. Azonban a hiba előrejelzések készítésekor többször is feldolgozhatunk egy-egy változtatási halmazt, mivel különböző méretű intervallumokra vizsgálhatjuk a rendszert (havi, negyedéves, féléves, stb.). Mivel a verziókövető rendszerbe feltett változások már megtörténtek, azaz nem fognak változni, ezért a program a feldolgozás mellett eltárolja változtatásokat lokálisan, így a későbbi használat során gyorsabban férhetünk hozzá a változtatásokat leíró információkhoz azáltal, hogy előbb megnézzük, hogy megvan-e lokálisan, és csak a hiányzókat kérjük le a verziókövető rendszerből.

A változtatások feldolgozásának megkezdésével egyszerre több műveletet is végre kell hajtani párhuzamosan. Az első a változtatások és a forráskód elemek egymáshoz rendelése. A változtatásokat leíró fájlok tartalma a verziókövető rendszerek által használt formátumtól függ, azonban mindezek ellenére több ponton megegyeznek a kimeneti formátumok.

```
1 diff --git a/A/B/C/XYZ.cpp b/A/B/C/XYZ.cpp
2 index flcffa0..114c789 100644
3 --- a/A/B/C/XYZ.cpp
4 +++ b/A/B/C/XYZ.cpp
```

3.10. ábra – Egy változtatásokat leíró állomány fejlécének szerkezete Subversion verziókövető rendszert használva

A 3.10. ábra egy lehetséges változatát mutatja be a változtatásokat tartalmazó állományok fejlécének. A fejlécből egyértelműen beazonosíthatjuk a régi és új állomány nevét. Az ábrán a régi és az új állomány neve megegyezik, mely jelen esetben az A/B/C/XYZ.cpp, mely azt jelenti, hogy az állományt nem nevezték át. A fenti fejléc a Subversion verziókövető rendszerrel készült, mely a kiterjesztett Git formátumot használja, így biztosítva azt, hogy a

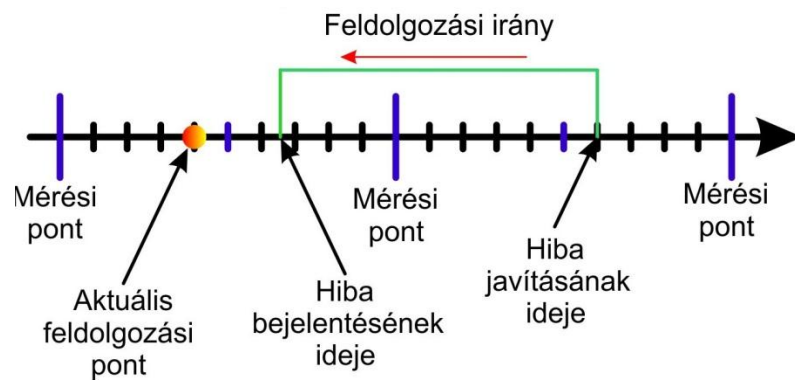
formátumok között minimális legyen az eltérés¹. Egy állományon belül több ilyen fejléc is megtalálható abban az esetben, ha több forrásfájlban történt módosítás. Minden, a fent bemutatott fejléchez tartozhat több, általunk szekciónak nevezett egység. A szekciók kezdetét egyedi alakú sorok jelzik, melyek például a Subversion rendszer esetében @@ -a,b +c,d @@ formában jelennek meg, ahol $a,b,c,d \in \mathbb{Z}^+$. Ezen számok a régi és az új állományra vonatkozólag azt adják meg, hogy mely sortól meddig terjedően jeleníti meg a változtatásokat a rendszer. Az egyes szekciókban egyértelműen jelzik, hogy mely sorok azok, melyek törlésre kerültek és melyek azok, amelyeket újonnan adtunk az állományhoz, így meghatározható az egyes szekciók első és utolsó változtatott sora. A változtatott intervallum ismeretében megkezdődhet a változtatások és a forráskód elemek párosítása, összerendelése. Az eljárás az adott állományhoz tartozó forráskód elemek között olyat keres, melyhez egyértelműen társítható a változtatás. Egyértelmű társításkor a forráskód elem pozícióját frissíthetjük a 3.1.1 fejezetben ismertetett módon.

A változtatások feldolgozási folyamatával párhuzamos, a 3.2.1 fejezetben ismertetett metrikák számolása is megtörténik. A változtatások és forráskód elemek párosításakor (mind egyértelmű és nem egyértelmű esetekben) a módosításokból nyerhető információkat is hozzárendelhetjük az adott forráskód elemhez. Ezzel frissíthetjük a forráskód elemekben módosított sorok számát, a változtatást végző személyek számát, a változtatások számát, stb.

Már korábban láthattuk, hogy a WebKit nyílt forráskódú projekt esetében egy változtatási halmazt leíró állományban mindig hivatkoznak egy vagy több ChangeLog állományra. A ChangeLog állományokat érintő szekciók feldolgozásakor egyértelműen kiderül, hogy az adott változtatási halmaz hibajavításhoz tartozott-e. Amennyiben hibajavításhoz tartozott, lekérjük az adott hiba tulajdonságait és hozzárendeljük őket a forráskód elemekhez a 3.2.2 fejezetben ismertetett módon. A WebKit rendszer változtatási halmazait leíró állományok között sokszor szerepelnek nagyméretű fájlok, melyekben sok forráskód elemet azonosíthatunk be. Abban az esetben, ha ezek a változtatások hibajavításokhoz köthetőek rendkívül sok összerendelés születik, és így a memóriában tárolt adathalmaz mérete is gyorsan képes növekedni. Ezen okból kifolyólag a rendszerbe egy extra

¹ Megjegyezzük, hogy bár a megoldást a SVN/GIT verziókövető rendszer formátumra mutatjuk be, a megoldás elég általános ahhoz, hogy más verziókövető rendszerek formátumaira könnyen lehessen alkalmazni.

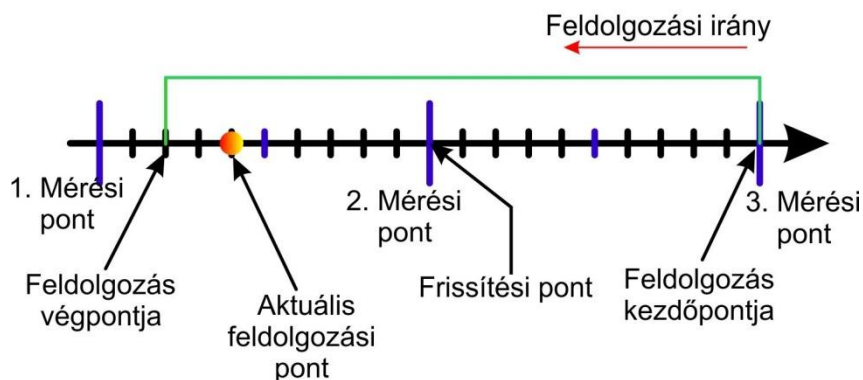
attribútumot vezettünk be, mely megadja, hogy hány változtatási halmaz (revízió) feldolgozása után kerüljön frissítésre a memóriában lévő hiba és forráskód elemeket összerendelő adathalmaz. A frissítési folyamat során azokat az összerendeléseket szüntetjük meg, melyek már nem játszanak szerepet a forráskódhoz tartozó hibák számának szempontjából. Ezt az eljárást a 3.11. ábra mutatja be. A hibák rendelkeznek a bejelentési és a kijavítási időpontokkal, melyek alapján meghatározható az az intervallum, melyben a hiba jelen volt a rendszerben. Az ábrán a hiba jelenlétének intervallumát a zöld vonal által kijelölt szakasz szemlélteti. Egy kitöltött kör jelöli az aktuális feldolgozási pontot, mely által meghatározott változtatások időpontja szintén ismert. Mivel a feldolgozandó változtatások korábbi időpontra esnek, mint a bemutatott hiba bejelentési ideje, így biztosak lehetünk abban, hogy több hibajavításnál már nem fognak hivatkozni az adott hibára.



3.11. ábra – A hibák és forráskód elemek összerendelési halmazának frissítése

A hiba és forráskód elemek összerendelését leíró halmaz frissítésekor a törölt párosítások által meghatározott forráskód elem attribútumait is frissítjük. Tehát egy törölt párosítás esetén a forráskód elemhez tartozó hibaszámot növeljük.

A különböző metrikák és hiba hozzárendelések számontartása mellett fontos, hogy amikor csak lehet, tartsuk karban magukat a forráskód elemeket is. Például tekintsük azt az esetet, amikor hetente készítünk forráskód elemzést egy adott rendszerről, és nagyobb, mint egy hetes intervallumra szeretnénk vizsgálatokat folytatni.



3.12. ábra – Forráskód elemek frissítése

A fent ismertetett állapotot a 3.12. ábra szemlélteti. Az ábrán a zöld vonallal kijelölt intervallum jelenti a vizsgált intervallumot. A 3. mérési pont eredményeit használjuk fel kiindulási halmazként. Korábban ismertettük, hogy akkor kapnánk minden forráskód elem pozíciójához a lehető legpontosabb eredményeket, ha minden revízióhoz tartozna mérés. Ismertettük ennek az elképzelésnek a hátrányait is. Abban az esetben, ha a vizsgálandó intervallumban mégis több méréssel rendelkezünk, érdemes az elemeket forráskód pozícióit frissíteni, így biztosítani, hogy az eredmény még pontosabb legyen. A 3.12. ábra esetében a 3. és a 2. mérési pont között némi elcsúszás jelentkezhet a forráskód elemek pozícióiban (a nem egyértelműen beazonosítható változtatások miatt). Viszont a 2. mérési pont eredményeit is felhasználva ismét biztosak lehetünk abban, hogy minden forráskód elem pozíciója a megfelelően beállított. Ezt kihasználva a 2. mérési pont és a feldolgozás végpontja közötti változtatásokat feldolgozva szintén a lehető legnagyobb pontossággal végezhetjük el a változtatások és a forráskód elemek párosítását. Az ismertetett elv egyszerűen kiterjesztett több mérési pont eredményeire is, azaz akárhány frissítés történhet a vizsgálat alatt. Azonban minél több frissítést szeretnénk beiktatni a mérések közé, annál több mérésre van szükségünk, így annál költségesebb a folyamat.

Ebben a fejezetben ismertettem a hiba előrejelzéshez használt algoritmus főbb elemeit, bemutatásra kerültek az információgyűjtési eljárások. A vizsgálat során rengeteg adat keletkezik, melyeket a következő fejezetben tovább vizsgálunk, hogy összefüggéseket találjunk.

4. Eredmények

Ebben a fejezetben az elért eredményeket ismertetjük. Kitérünk a forráskód elemek és a változtatások összerendelésére, a folyamat metrikák és a forráskód metrikák közötti kapcsolatokra is. Továbbá górcső alá vesszünk gépi tanulási módszereket is, melyekkel a hiba előfordulását és az előforduló hibák számát szeretnénk tanulni.

A Webkit rendszert két hosszabb intervallumon vizsgáltuk. Kettő darab körülbelül féléves időtartamra vetítettük le az eredményeket. A két intervallum együttese 2011. júniusa és 2012. májusa közti időtartamot jeleníti meg. Az intervallum lefedéséhez a 87778. revíziótól a 115000. revízióig végeztük a vizsgálatainkat. Ezt az időszakot két kisebbre osztottuk, így az egyik vizsgált időszak a 87778-98935 revíziók közé esik, míg a második időtartamot a 98935-115000 revízióig terjedő szakasz teszi ki. A fejezet további szakaszaiban a 87778-98935 revíziók közti intervallumra 1. vizsgált szakaszként, míg a 98935-115000 revíziók közti időtartamra 2. vizsgált szakaszként fogunk hivatkozni.

4.1. A változtatások hozzárendelése forráskód elemekhez

A 3.3. fejezetben ismertetett algoritmus első lépésben megpróbálja a változtatáshalmazban foglalt változtatásokat forráskód elemekhez rendelni. Ebben a fejezetben összehasonlítom a korábbi [14] tanulmányban ismertetett módszer eredményeit a jelenlegi tanulmányomban ismertetett eljárás eredményeivel. A korábbi algoritmus nem foglalkozott a forráskód elemek pozícióinak karbantartásával, míg az újabb módszer megpróbálja kiküszöbölni ezt a hiányosságot.

	1. vizsgált szakasz	2. vizsgált szakasz
Szekciók száma	184 312	175 066
Nem egyértelmű szekciók száma	3 955	4 828
Egyértelmű szekciók száma	21 116	22 283
Eldobott szekciók száma	159 241	147955
Nem eldobott szekciók száma	25 071	27 111
Nem egyértelmű szekciók száma/nem eldobott szekciók száma	0,157751	0,178082
Egyértelmű szekciók száma / nem eldobott szekciók száma	0,842249	0,821918

4.1. Táblázat - A két féléves intervallumon vizsgált változtatások forráskód elemekhez rendelésének a tulajdonságai

A változtatások forráskód elemekhez történő hozzárendelésekor a változtatási halmaz elemeit dolgozzuk fel. A változtatások fájlokhoz köthetőek, melyekben forráskód elemeket találhatunk. Azonban a változtatási halmazban vannak hivatkozások olyan forrásfájlokra, melyek jelen esetben nem befolyásolják az alkalmazás működését (például HTML fájlok, képek, stb.). Ezeket az elemek egyszerűen eldobjuk, és nem vesszük figyelembe.

A 4.1. Táblázatban a változtatások és a forráskód elemek összerendelésének néhány tulajdonságát adjuk meg két féléves intervallumon az új módszer használatával. A feldolgozás során nyilvántartjuk a szekciók számát, mely fogalmat Az algoritmus 3.3. fejezetben már megismerhettünk. A táblázatból kiolvasható azon szekciók száma, melyeket el kellett dobnunk, továbbá azon szekciók száma, melyek által tartalmazott változtatásokhoz egyértelműen hozzárendelhattük a megfelelő forráskód elemet, és azoknak számát is, melyekhez nem tudtunk egyértelműen forráskód elemet rendelni.

A következő, 4.2. Táblázatban a korábbi és az új módszer által elért változtatás – forráskód elem párosítás eredményeit mutatom be. Az algoritmus hatékonyságát egy adott intervallumon jellemezhetjük a

$$\frac{\text{egyértelmű szekciók száma}}{\text{nem eldobott szekciók száma}}$$

hányadossal, így a táblázatban már csak ezeket az értékeket tüntettük fel. A táblázatból kiolvasható, hogy a korábbi módszer alacsonyabb értékeket hoz, míg az új módszerrel, azaz a forráskód elemek pozíciójának karbantartásával elérhetőek magasabb párosítási értékek is.

	1. vizsgált szakasz	2. vizsgált szakasz
Korábbi módszer	0,761108	0,745186
Jelenlegi módszer	0,842249	0,821918

4.2. Táblázat – A változtatások és forráskód elemek összerendelésének eredménye két különböző módszerrel.

A magasabb érték azt jelenti, hogy sikerül több változtatást egyértelműen hozzárendelni forráskód elemekhez, mely által később pontosabb forráskód elem pozíciókat és pontosabb metrikákat kaphatunk, melyek fontos szerepet játszanak a hiba előrejelzésben.

4.2. A tanulmányban használt metrikák közötti kapcsolatok

Első lépésként megvizsgáljuk, hogy a metrikák mennyire függetlenek. A 4.3. Táblázatban és a 4.4. Táblázatban láthatjuk a Pearson-féle korrelációs értékeket a vizsgált metrikák között, ahol a 0,5-nél nagyobb értékek kiemeltük. Mivel túl sok forráskód metrikát használtunk a vizsgálat során, és nem célunk azok részletes elemzése, ezért csak hármat hasonlítottunk össze a folyamat metrikákkal, azokat, amelyek korábban a legjobbnak bizonyultak az elmúlt évek vizsgálatai során, illetve 3 különböző szempontját jellemzik a rendszernek (az LLOC a méretet, a CBO a csatolást és a WMC a komplexitást). A táblázatban szereplő korrelációk szignifikánsan 0,05 szignifikancia mellett, és amint látható, csak a CBO és 3 folyamat metrika között nem találtunk kapcsolatot. A táblázatban 3 érdekesebb kapcsolat-csoport figyelhető meg. Az első, hogy a hozzáadott, a törölt és a változott sorok száma között igen szoros kapcsolat van, ami nem meglepő. A második, hogy a forráskód metrikák között is szoros a kapcsolat annak ellenére, hogy a rendszer különböző karakterisztikáját mérik. A harmadik, hogy a változások száma és a különböző kommiterek száma között is igen magas a korreláció. Ezek mellett még említésre méltó például a forráskód metrikák és a hibák száma közötti kapcsolat, de a hibaszám és a változások/kommiterek száma közti kapcsolat sem elhanyagolható. További érdekesség, hogy a forráskód metrikák és a folyamat metrikák között nincs szoros kapcsolat, azaz ezen két metrika csoport valóban a rendszer különböző, egymástól független tulajdonságát jellemzi.

1. vizsgált szakasz	NCOM	NLA	NLD	NLC	NB	LLOC	CBO	WMC
NC	0,977	0,192	0,211	0,207	0,266	0,201	0,176	0,188
NCOM		0,187	0,203	0,201	0,263	0,213	0,174	0,202
NLA			0,876	0,973	0,072	0,081		0,1
NLD				0,963	0,078	0,082		0,089
NLC					0,077	0,084		0,098
NB						0,243	0,263	0,216
LLOC							0,675	0,929
CBO								0,619

4.3. Táblázat – Metrikák függetlensége az 1. vizsgált szakaszban

2. vizsgált szakasz	NCOM	NLA	NLD	NLC	NB	LLOC	CBO	WMC
NC	0,967	0,15	0,153	0,153	0,365	0,224	0,251	0,21
NCOM		0,152	0,146	0,15	0,389	0,239	0,247	0,228
NLA			0,966	0,992	0,062	0,037		0,039
NLD				0,991	0,063	0,034		0,03
NLC					0,063	0,036		0,035
NB						0,451	0,361	0,424
LLOC							0,673	0,935
CBO								0,627

4.4. Táblázat – Metrikák függetlensége a 2. vizsgált szakaszban

4.3. Gépi tanulási módszerek alkalmazása hiba előrejelzéshez

A hiba előfordulás és az előforduló hibák számának előrejelzéséhez elengedhetetlen, hogy gépi tanulási módszereket alkalmazzunk. Ilyen módon a múltban előállított adatok alapján (mint például a folyamat és forráskód metrikák) előrejelzést adathatunk a jövőbeni várható hibák számáról és eloszlásáról. A tanulási eredményeket a korábban meghatározott két intervallumra fogjuk ismertetni.

A WebKit rendszer vizsgálatokor kétféle gépi tanulási módszert alkalmaztunk. Az egyik módszerrel a hiba előfordulását szeretnénk vizsgálni (nominális érték), azaz ebben az esetben egy osztályozásról van szó. A másik esetben a hibák számára vagyunk kíváncsiak (numerikus érték), azaz egy regressziós gépi tanulási módszert kell alkalmaznunk. A gépi tanuláshoz a Weka, [16] Java alapú szoftvert használtuk, melyet a Waikato Egyetem Gépi Tanulási Csoportja fejlesztett.

A gépi tanulás során elért eredmények közül elsősorban az osztály szintű forráskód elemekre tértek ismertetjük. A tanulási folyamatokhoz a Weka RepTree döntési fáját használtuk, mely képes osztályozást is és regressziót is végezni. A tanulási folyamat során 10-szeres keresztvalidációt alkalmaztunk.

1. vizsgált szakasz	helyesen osztályozott	nem helyesen osztályozott
folyamat metrikák alapján	92.0606 %	7.9394 %
forráskód metrikák alapján	78.4712 %	21.5288 %
mindkét csoport metrikái alapján	92.356 %	7.644 %

4.5. Táblázat – 1. vizsgált szakaszhoz tartozó osztályozó gépi tanulás eredményei osztály szintű forráskód elemekhez

Először azt vizsgáltuk meg, hogy a metrikák mennyire alkalmasak annak előrejelzésére, hogy egy adott osztály hibás-e vagy sem, ahol hibásnak tekintettünk egy osztályt, ha legalább egy hibajavítást volt benne a vizsgált időszakban. A 4.5. Táblázat mutatja be az 1. vizsgált szakaszban elért eredményeket. 3 esetet különböztettünk meg. Az első esetben csak a folyamat metrikák alapján történt a gépi tanulás, a második esetben kizárólag a forráskód metrikák szolgálták a tanulás attribútumait, míg a harmadik esetben az összes folyamat és forráskód metrikát felhasználtuk a tanulás során. A táblázatban a „helyesen osztályozott” oszlopban szerepel az, hogy az elemek hány százalékát osztályozta helyesen az algoritmus (azaz a hibásat hibásnak és a nem hibásat nem hibásnak), a „nem helyesen osztályozott” oszlopban a rosszul osztályozott elemek aránya található (azaz hibásat jónak osztályozott vagy jót hibásnak). A nagy értékek elsőre nagyon ígéretesnek tűnnek a folyamat metrikák és az összes metrikák soraiban, azonban az eredményeket pozitívan befolyásolta az elvégzett kísérlet „egyszerűsége”, ugyanis ugyanazon az intervallumon számoltuk ki a folyamat metrikákat, amelyeken a hibajavításokat is beazonosítottuk, és így azok a forráskód elemek, melyek nem változtak, nem is lehetnek hibásak (ChangeLog feldolgozásából származnak a hibajavítások hivatkozásai). A vizsgált 1. szakasz esetében az osztályok közel fele nem változott, azaz ezekben hiba sem lehetett, amit könnyű volt megtanulni.

1. vizsgált szakasz	Hiba-osztály	Precision	Recall
Folyamat metrikák alapján	hibás	0.823	0.89
	nem hibás	0.959	0.932
Forráskód metrikák alapján	hibás	0.646	0.407
	nem hibás	0.812	0.92
Mindkét csoport metrikái alapján	hibás	0.832	0.89
	nem hibás	0.96	0.936

4.6. Táblázat – Az egyes metrikacsoportok alapján történő osztályozáshoz tartozó precision és recall értékek osztály szintű forráskód elemek esetén

Azonban az eredmények még így is biztatóak, mert a 92% körüli helyes osztályozás arra utal, hogy a változott osztályok között is figyelemre méltóan eredményeket ér el az osztályozás. A vizsgálat eme gyengeségének kiküszöbölésének lehetőségével az 5. fejezetben foglalkozunk.

A 4.6. Táblázat egy bővebb képet ad az osztályozás eredményéről. A precision oszlopban található érték megmutatja, hogy azok közül az elemek közül, amelyeket az adott csoportba sorolt a tanuló algoritmus, mennyi tartozott ténylegesen az adott csoportba. A recall érték pedig megadja, hogy az adott csoportba tartozó elemek mekkora részét találtuk meg valójában. Az ismertetett ok miatt a „nem hibás” osztályozásokhoz tartozó precision és recall értékek szintén nagyon magasak lesznek, így a következő táblázatokban a „nem hibás” osztályhoz tartozó értékeket nem tüntetem fel. A „nem hibás” osztály magas értékei mellett azonban a „hibás” osztályozáshoz tartozó eredmények is jó értékeket adnak. A táblázatból kiolvasható, hogy a folyamat metrikák felhasználásával jobb eredményeket érhetünk el.

A 4.7. Táblázat a 2. vizsgált intervallumra mutatja be az osztályozás eredményességét. A gépi tanulás során az 1. vizsgált szakaszhoz hasonló eredményeket kapunk. A táblázat alapján megerősítést nyer, hogy a folyamat metrikák alkalmazása előnyösebb lehet a hiba előrejelzésben.

2. vizsgált szakasz	helyesen osztályozott	nem helyesen osztályozott	Precision	Recall
folyamat metrikák alapján	93.1001 %	6.8999 %	0.867	0.957
forráskód metrikák alapján	72.2721 %	27.7279 %	0.619	0.627
mindkét csoport metrikái alapján	93.1322 %	6.8678 %	0.868	0.958

4.7. Táblázat – A 2. vizsgált szakaszhoz tartozó osztályozás értékei és a true osztályokhoz tartozó precision és recall értékek osztály szintű forráskód elemek esetén

Az egyszerű osztályozáson felül érdemes regressziós gépi tanulási módszereket alkalmazni, hogy a várható hibaszámot tanulhassuk. A regressziós gépi tanulás esetében is a Weka RepTree implementációt használjuk fel.

1. vizsgált szakasz	Korrelációs együttható	Átlagos abszolút hiba
folyamat metrikák alapján	0.6131	1.4684
forráskód metrikák alapján	0.3919	2.1451
mindkét csoport metrikái alapján	0.6817	1.5611

4.8. Táblázat - Az első vizsgált szakasz regressziós tanulási algoritmus eredményei osztály szintű forráskód elemekre

2. vizsgált szakasz	Korrelációs együttható	Átlagos abszolút hiba
folyamat metrikák alapján	0.6536	0.9904
forráskód metrikák alapján	0.4607	1.6512
mindkét csoport metrikái alapján	0.5901	1.1981

4.9. Táblázat – A második vizsgált szakasz regressziós tanulási algoritmus eredményei osztály szintű forráskód elemekre

A 4.8. Táblázat az 1. a 4.9. Táblázat a 2. vizsgált szakaszon futtatott regressziós gépi tanulási módszer eredményeit ismerteti az osztályszintű forráskód elemekhez. A táblázatok tartalmazzák a tanuláshoz használt attribútumok közötti Pearson-féle korrelációs együtthatókat és az átlagos abszolút hibák értékét. Megállapítható, hogy a folyamat metrikák regresszió esetében is jobb eredményeket érnek el. A második szakaszban mind a folyamat, mind a forráskód alapú metrikák szerinti tanulással jobb értékeket érünk el, mint az első szakaszban, azonban a két táblázat arányaiban hasonló értékeket mutat.

5. Összegzés és jövőbeli tervek

A tanulmányban ismertetett probléma nem új, már léteztek korábbi megoldások is hiba előrejelzéshez. Több tanulmányban próbáltak változásokövető rendszerekből kinyerhető adatok (metrikák, mérőszámok) alapján hiba előrejelzéseket adni a jövőre nézve. Ezen módszerek működnek is, azonban ezek az algoritmusok általában fájl szinten dolgoznak. Viszont az objektum-orientált nyelvek esetében ezt nehéz a nyelvi elemekhez, például osztályokhoz rendelni, melynek következtében nehéz a megszerzett információkat megfelelően hasznosítani (például az egység tesztek is osztályokra vonatkoznak és nem fájlokra). A tanulmányunkban egy lehetséges pontosítási eljárást adtunk a hiba előrejelzéshez azzal, hogy a változtatások információit a forráskód elemek szintjén végeztük el. Bemutattuk azokat a problémákat, melyek ezen az alacsony szinten felbukkantak, ugyanakkor lehetséges megoldásokat is demonstráltunk, melyekkel a korábban kidolgozott módszer továbbfejlesztéseként folyamatosan nyomon követhetjük a forráskód elemek pozícióinak változását. A változtatások és a forráskód elemek összerendelésekor nincs minden esetben egyértelmű megoldás, azonban a forráskód elemek pozíciójának karbantartásával 4-6%-kal növelhető az egyértelmű hozzárendelések aránya. Továbbá a megoldásunknak köszönhetően rendelkezünk azzal a fontos információval is, hogy mennyire volt pontos algoritmus, és ha „túl alacsonynak” találjuk a pontosságát, akkor további mérési pontok beiktatásával tovább javíthatjuk azt.

A változtatások követésének során 6 különböző metrikát számoltam ki, ugyanakkor több forráskód alapú metrikát is vizsgáltunk a tanulmányban. A metrikák vizsgálatakor kiderült, hogy a folyamat és a forráskód metrikák elkülönülnek egymástól, szerepük nem egyezik meg a hiba előrejelzésben.

A tanulmányban használt metrikákat felhasználtuk gépi tanulási modellek tanuló halmazaként is. Tüzetesebben megvizsgáltunk egy osztályozás és egy regressziós algoritmus által generált eredményeket is. Végeredményként azt tapasztaltuk, hogy a változtatások feldolgozásával előállított metrikák lényegesen jobb eredményt szolgáltatnak a hiba előrejelzésben, mint a forráskód alapú metrikák.

A tanulmányban több pontosítási, javítási lehetőséget vettünk számba, azonban maradtak még be nem tömött lyukak. A változtatások feldolgozása során lehetőségünk van

további metrikák előállítására is, amik tovább javíthatják majd az előrejelzéseket. Ilyen metrika lehet például a forráskód elem kora vagy a fejlesztők tapasztalata. Továbbá érdekes lehet megvizsgálni, hogy bizonyos elemek mennyire változnak együtt, és ha sok együtt változás után egyszer csak az egyiket változtatják meg, akkor az okoz-e hibát.

A forráskód elemek pozícióinak karbantartásakor felléphetnek olyan esetek, amikor nem tudjuk egyértelműen meghatározni, hogy hogyan tolódott el az adott elem első és utolsó sorának pozíciója. Ezekben az esetekben további vizsgálatokat végezhetünk, például a változott sorok szintaktikus elemzésével (megkeresem az adott elem első sorát a változott részben, és az alapján meghatározhatom az új kezdő pozíciót), esetleg valamilyen heurisztika alapján becslést adhatunk az elmozdulásokra.

A WebKit nyílt forráskódú rendszer az idő folyamán már sok nyomot hagyott maga után. Ezeket a nyomokat a tanulmányban most felhasználtuk, hogy a jövőben a WebKit rendszerben fellelhető hibák számáról és eloszlásáról mondhassunk információkat. Azonban sokszor kevés egy adott rendszer vizsgálata. Fontos, hogy több rendszert is alaposabban megvizsgáljunk, mivel így bizonyosodhatunk meg a módszerünk általánosságáról.

A jelen tanulmányban két, közel féléves időintervallumot ragadtunk ki a WebKit rendszer életéből. Ahhoz, hogy a lehető legoptimálisabb megoldást kapjuk, minimalizálnunk kell a rendszeren végzendő nyelvi elemzések számát, viszont minél pontosabb előrejelzéseket szeretnénk előállítani. Tehát fontos, hogy különböző méretű intervallumokra is futtassuk az alkalmazást, mivel így egy sokkal jobb képet kaphatunk arról, hogy mekkora az az ideális intervallum, melynek eredményeit megfelelően felhasználhatjuk a hiba előrejelzésekhez, de nem kell túl sokszor nyelvi elemzést végeznünk.

A gépi tanulási módszerek utat engednek a tényleges hiba előrejelzéshez, ami a tanulmány fő célja egyben. A hiba előrejelzés célja, hogy egy adott rendszer múltbeli időintervallumain végzett tanulási folyamatot követően tényleges előrejelzést adhatunk majd a rendszerben keletkező hibákra, megjelölve azokat a forráskód elemeket, melyek várhatóan a hibák legnagyobb hányadát fogják magukban foglalni. Jelen kísérletben azonban csak azt vizsgáltuk meg, hogy egy időintervallumra mért metrikák mennyire képesek az adott intervallumba eső hibákat megtalálni, azonban ez még nem előrejelzés. Célunk, hogy a metrikák hiba előrejelző képességeit is megvizsgáljuk, azaz több kisebb egymást követő

intervallumon esetében kiszámoljuk a metrikákat és a hibaszámokat, és a korábbi intervallumokra számolt metrika értékek alapján próbáljuk meg megtalálni a következő intervallumokba eső hibás elemeket.

6. Függelék

A vizsgálatok során az alábbi termék metrikákat használtuk fel, azonban ezek definíciói és kiszámítása nem a szerző saját eredménye (Siket István a [17] doktori dolgozatának része).

- **Lines of Code (LOC):** Az osztály (és annak metódusainak) sorainak a száma.
- **Logical Lines of Code (LLOC):** Az osztály (és annak metódusainak) nem üres és nem komment sorainak a száma.
- **Sum of Lines Of Code (SLOC):** Az osztályban található metódusok sorainak számának az összege.
- **Number of Outgoing Invocations (NOI):** Az osztály metódusai által közvetlenül hívott metódusok és függvények halmazának a számossága.
- **Number of Incoming Invocations (NII):** Azon függvények és metódusok halmazának számossága, amelyek hívják az osztály valamely metódusát.
- **Number of Methods (NM):** Az osztály 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 Attributes (NA):** Az osztály lokális és örökölt attribútumainak a száma.
- **Number of Attributes Local (NAL):** Az osztály lokális attribútumainak a száma.
- **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 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 (NFMA_{ni}):** Az osztály metódusai által közvetlenül hívott nem saját lokális metódusok halmazának a számossága.

- **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 $RFC(C) = |M \cup \bigcup_{i=1}^n R_i|$ 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 végrehajthatnak válaszul egy, az adott osztályból példányosított objektum felé irányuló hívásnál.
- **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 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.
- **Weighted Methods per Class (WMC):** Az osztály metódusainak súlyozott összege, ahol a metódus súlyának a McCabe-féle ciklomatikus komplexitást használjuk.
- **Nesting Level (NL):** Az osztályban található metódusok „beágyazásai szintjeinek” a maximuma, ahol egy metódusok beágyazási szintjét a metóduson belül található egymásba ágyazott elágazások mélységének a maximuma.
- **Nesting Level elseif (NLE):** Megegyezik a Nesting Level metrikával, a különbség annyi, hogy az „else if” szerkezet nem növeli a beágyazás mélységét.

Irodalomjegyzék

- [1] Todd L. Graves, Alan F. Karr, J.S. Marron, and Har, *Predicting Fault Incidence Using Software Change History*, 2000.
- [2] Audris Mockus and David M. Weiss, *Predicting Risk of Software Changes*, 2000.
- [3] Erik Arisholm, Lionel C. Briand, Magnus J. Fugleru, *Data mining Techniques for Building Fault-proneness Models in Telecom Java Software*, 2007.
- [4] Ranjith Purushothaman and Dewayne E. Perry, *Toward Understanding the Rhetoric of Small Source Code Changes*, 2005.
- [5] Emad Shihab and Ahmed E. Hassan, Bram Adams, Zhen , *An industrial study on the Risk of Software Changes*, 2012.
- [6] Elaine J. Weyuker, Thomas J. Ostrand, Robert M. Be, *Using Developer Information as a Factor for Fault Prediction*, 2007.
- [7] Elaine J. Weyuker, Thomas J. Ostrand, Robert M. Be, *Using the number of developers to enhance defect prediction models*, 2008.
- [8] Erik Arisholm and Lionel C. Briand, *Predicting fault-prone Components in a Java Legacy System*, 2006.
- [9] Sunghun Kim, E. James Whitehead, Jr., Yi Zhang, *Classifying Software Changes: Clean or Buggy?*, 2008.
- [10] Robert M. Bell, Thomas J. Ostrand, Elaine J. Weyuk, *The limited impact of individual developer data on software defect prediction*, 2011.
- [11] Subversion, <http://subversion.tigris.org/>.
- [12] Git, <http://git-scm.com/>.
- [13] Lile Hattori, Gilson dos Santos Jr., Fernando Cardoso és Marcus Sampaio, *Mining*

Software Repositories for Software Change Impact Analysis: A Case Study, 2008.

[14] Tibor Gyimóthy, Rudolf Ferenc, and István Siket, *Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction*.

[15] Bugzilla, <http://www.bugzilla.org/>.

[16] Weka 3: Data Mining Software in Java, <http://www.cs.waikato.ac.nz/ml/weka/>.

[17] Siket István, *Szoftver termék metrikák alkalmazása a szoftverkarbantartás területén*, 2010.

[18] The Webkit Open Source Project, <http://www.webkit.org/>.