

TDK Dolgozat

Gyimesi Gábor, Gyimesi Péter

Forráskód hibák jellemzése GitHub-on végzett adatbányászat segítségével

Készítette :

*Gyimesi Gábor, Gyimesi Péter
programtervező informatikus
II. évf. MSc hallgatók*

Témavezetők:

*Tóth Zoltán Gábor
PhD hallgató
Dr. Ferenc Rudolf
adjunktus*

**Szegedi Tudományegyetem Természettudományi és Informatikai Kar
Szoftverfejlesztés Tanszék**

Szeged

2014

Tartalomjegyzék

Bevezetés	5
1. Kapcsolódó munkák	7
2. GitHub	10
2.1. GitHub statisztika	11
2.2. Git verziókezelő rendszer	11
2.3. Feladatkezelő és hibakövető rendszer	12
2.4. GitHub REST API	13
2.4.1. Tárolóhoz tartozó feladatok lekérdezése	13
2.4.2. Feladathoz tartozó események lekérdezése	14
2.4.3. Commit adatainak lekérdezése	14
2.5. GitHub Java API	15
2.5.1. Felhasználó hitelesítés	15
2.5.2. Szolgáltatások használata	16
3. A metrikák	17
3.1. Forráskód metrikák	17
3.2. Verziókövető rendszerből kinyert metrikák	18
4. Adatbányászat GitHub-ról	19
4.1. A projekt választási szempontok	19
4.2. A választott projektek	21
4.3. Adatok összegyűjtése	24
4.4. Nyers adatok feldolgozása	25
4.5. Forráskód verziók elemzése	25
4.6. Hibajellemzők kinyerése	26
4.7. Kinyert adatok összefűzése	29
4.8. Adatok szűrése	30
5. Összegzés	32

6. További tervek	33
7. Függelék	34
7.1. A hibajavításokban érintett commit-ok diagramjai	34
Irodalomjegyzék	40

Bevezetés

A szoftverek forráskód hibáinak jellemzése manapság egy gyakran kutatott terület. A programozók a fejlesztőkörnyezetek által nyújtott segítség ellenére hajlamosak elkövetni hibákat, illetve a gyakori módosítások és nem megfelelő specifikáció miatt is jelentkezhetnek hibák, ezért fontos, hogy minél több eszköz segítségével könnyítsük a hibák automatikus detektálását. Az automatikus detektáláshoz szükséges a hibák jellemzése, mely alapján felismerhetők a további hibák. Ez sok kérdést vonz maga után. Hogyan definiáljuk a hibát? Hogyan adjuk meg a már megtalált hibákat? Milyen tulajdonságai lehetnek a hibáknak? Ez a dolgozat ezekre a kérdésekre keres válaszokat.

Gyakori megoldás valamilyen ellenőrző program írása, amely egy megadott szabályrendszer szerint felkutatja a hibákat a forráskódban. Ehhez szükséges a hibákat előre ismerni és definiálni, valamint megírni egy olyan szoftvert, ami képes ezeket a hibákat megtalálni. Látszik, hogy már a hibák ismerete is nagy feladat lehet, hiszen minden programozási nyelvnek megvannak a sajátosságai, amik hibához vezethetnek. De ha csak egy nyelvnél maradunk, akkor sem egyszerű a gyakori hibákon kívülieket összegyűjteni. Egy másik lehetőség a hibák jellemzésére, ha nem a kódra mondunk szabályokat, hanem a kódból próbáljuk kinyerni ezeket az információkat. Ehhez az kell, hogy egy forráskódról tudjuk, hogy hiba van benne. A hibás kódrészeket pedig jellemezhetjük különböző szempontokból.

A szoftverek fejlesztéséhez számos eszközt használnak: feladatkezelő, konfiguráció menedzselő, hibakövető, verziókövető, dokumentáló eszközök, stb. Rendelkezésre áll számos fizetős és nyílt forráskódú szoftver is ezekre a célokra. Több webes szolgáltatás is épül ezeknek az igényeknek a kielégítésére¹. A legnépszerűbbek a SourceForge, Bitbucket, Google Code és a GitHub. Ezek általában több szolgáltatást is nyújtanak, mint például forráskód hosting és felhasználó kezelés. Mindegyiknek megvannak a sajátosságai, előnyei és hátrányai. Némelyik saját API-val is rendelkezik, melyek jól definiáltak és felhasználhatóak az adatok olvasására vagy módosítására is. Ez lehetőséget biztosít az adatok programokkal végzett lekérésére, így számos érdekes felhasználási módja lehet. Az egyik ilyen felhasználási lehetőség az adatbányászat. Lehetőség van a felhasználók viselkedésének, együttműködésének vizsgálatára vagy akár a forráskód elemzésére is. Mivel a legtöbb ilyen szolgáltatás funkciói között szerepel a hibakövetés, felmerül a gondolat, hogy ezt az információt hogyan lehetne felhasználni a hibák

¹http://en.wikipedia.org/wiki/Comparison_of_open-source_software_hosting_facilities

jellemzésére? A hibakövető rendszerek általában biztosítanak egy felületet, melyen a felmerülő esetleges hibákat lehet bejelenteni. Ezeket a hibabejelentéseket valahogy párosítani kell a forráskódhoz. Több megoldás is létezik. Egy bevett gyakorlat a verziókövető rendszerekben, hogy a módosításokhoz kommentben leírják mit módosítottak, és gyakran egy hiba azonosítót is megadnak. Ezt felhasználva meghatározhatjuk azt a forráskód verziót, amelyben a hiba megtalálható. Egyes szolgáltatások ezt automatikusan elvégzik. Azonban ez nem mindig teljesül, mivel vagy nem elérhető egy ilyen hiba adatbázis, vagy a kommentezési konvenciót nem mindig követik. Amennyiben mégis sikerül meghatározni a hibás forráskód verziót, még mindig kérdéses, hogy ebből milyen információt nyerhetünk ki. Ezért valamilyen módon tovább kell szűkíteni a hibás kódrészt. Itt jönnek képbe újra a verziókövető rendszerek. A legtöbb lehetőséget biztosít a változások lekérésére valamilyen diff² formátumban. Ez alapján már sor szinten azonosítani lehet a hibás kódrészeket. A jellemzésük többféleképpen kivitelezhető. Használhatunk szöveges hasonlóságot valamely más hibás kóddal, amennyiben rendelkezésünkre áll egy ilyen adatbázis. Használhatjuk továbbá a statikus forráskód metrikákat, melyekhez csak egy eszköz kell, ami képes előállítani azokat. Lehetséges még a jellemzéshez felhasználni a fentebb említett szolgáltatásokból kinyerhető egyéb információkat, mint például a felhasználói statisztikákat.

A munkánkhöz az adatokat a GitHub-ról gyűjtöttük. Kiválasztottunk 13 projektet, melyek alkalmasak az ilyen módú vizsgálatra, majd letöltöttük a projekthez tartozó adatokat. Hibáknak a hibakövető rendszerben található bejelentéseket tekintettük. Ezeknek a jelöléséhez meghatároztuk a diff fájlok alapján, hogy mely forráskód elemeket érintették. A jellemzéshez a statikus forráskód metrikákat választottuk, illetve definiáltunk további jellemzőket a kinyert adatokból. A metrikákat egy olyan eszköz segítségével határoztuk meg, mely metódus szinten is képes számolni azokat. Az így kapott adatokból felépítettünk egy olyan adatbázist, amely metódus szinten is jellemzi a hibákat, szemben az eddigi adatbázisokkal, melyek osztály, de leginkább fájl szinten tartalmazznak ilyen információkat.

A dolgozat elején az ezen a területen elért eredményekről lesz szó. Ezt követően részletezzük a GitHub tulajdonságait, szolgáltatásait és az elérhető adatokat. A harmadik fejezet a metrikákról tartalmaz rövid összefoglalót. A munkánk lényegi részét a negyedik fejezet írja le. Az utolsó fejezetben a további terveinket részletezzük.

²http://en.wikipedia.org/wiki/Diff_utility

1. fejezet

Kapcsolódó munkák

A hibák jellemzésére és felismerésére többféle módszert is kidolgoztak. Ezek egyike a Zhou és társai által kidolgozott BugLocator [13] nevű program, mely szöveges hasonlóság alapján keres további hibákat a forráskódban. Egy hiba adatbázisból indul ki, melyből kinyeri, hogy melyik fájlokban voltak hibák és ezekhez keres hasonló hibákat. Az új hibabejelentésekhez is képes ajánlásokat tenni, hogy mely fájlokban lehet a hiba. Ezt úgy éri el, hogy az új hibabejelentéshez a leírás alapján természetes nyelvfeldolgozás segítségével meghatározza a hasonló bejelentéseket, és a hozzájuk tartozó hibás kódrészekhez hasonló fájlokat rangsorolja. Így megkönnyíti a fejlesztők dolgát, hogy ne kelljen annyi fájlt átnézniük. A hiba adatbázis felépítéséhez nyújt segítséget a ReLink¹ nevű program. Ez képes visszaállítani az elveszett kapcsolatot a hibabejelentések és a forráskód között, amit szintén nyelvfeldolgozási módszerekkel ér el. Elemzi a hibához tartozó, hibakövető rendszerből kinyerhető információkat mint például bejelentő, hiba-leírás, kommentek, dátum, stb. A párosításhoz összehasonlítja ezeket a forráskód módosításokhoz tartozó információkkal: komment, módosított fájlok, fejlesztő, stb., majd pedig hasonlóság alapján összeköti ezeket.

A Rebug-Detector [11] nevű program, melyet Wang és kollégái készítettek, szintén a nyelvfeldolgozás segítségével keres kapcsolódó, vagy hasonló forráskód hibákat. Abból indul ki, hogy ha egy metódusban hiba van, akkor a hozzá hasonló, felüldefiniált vagy kiterjesztett metódusokban is valószínűleg hiba van. A metódusokban lecseréni a változóneveket a változó típusára, és így szöveges hasonlóság alapján keres hasonló metódusokat. A hibabejelentésből kinyert információk segítségével pedig képes ajánlatokat tenni egy bejelentett, de még ki nem javított hibához.

Amennyiben egy hiba javításához többször kell módosítani a forráskódot, gondot okozhat az érintett forráskód elemek meghatározása. Minden módosítást meg kell vizsgálni, ki kell gyűjteni a módosított forráskód elemeket, és végül ezeket meg kell keresni a még hibás verzióban. A verziókezelő rendszerek módosításainak követéséről megjelent cikk, melyet Tóth Z. és munkatársai publikáltak [12], jól leírja a módszer menetét, valamint nehézségeit. Ennek során a

¹<https://code.google.com/p/bugcenter/>

forráskód pozíciók frissítésével érték el, hogy nem fontos minden verziót elemezni a forráskód elemek pozíciójának meghatározásához. Nagy számú elemzendő verzió esetén ezzel jelentős mértékben lehet időt és erőforrást spórolni.

Kalliamvakou és munkatársai készítettek egy tanulmányt [9], melyben a GitHub-ról kinyerhető adatok minőségével és tulajdonságával foglalkoztak. Részletezi többek között, hogy a GitHub-on található projektek milyen természetűek, például inaktívak, csak tárhelyként használják, nem önálló projekt, stb. Az eredményekből kiderül, hogy a kinyerhető adatok összességében jól felhasználhatóak különböző elemzésekhez, de fenntartással kell kezelnünk azokat. Érdekes az aktív projekteket vizsgálni, sok módosítással és sok fejlesztővel, valamint a célra minél alkalmasabbakat kiválasztani.

Bird és kollégái az elosztott forráskód kezelő rendszerekről, pontosabban a Git-ről készítettek egy tanulmányt [3]. Munkájuk során a verziókezelő rendszer használatát, és a használatával rendelkezésre álló információkat vizsgálták. A központosított és az elosztott rendszerek közötti különbségekből adódó eltéréseket fejtették ki, például commit-ok törölhetőségéből, módosíthatóságából vagy áthelyezhetőségéből adódó problémákat. Fő céljuk az elosztott forráskód kezelő rendszerek adatai iránt érdeklődő kutatók munkájának segítése volt, hogy elkerülhessék a buktatókat a Git adatok feldolgozása és elemzése során.

Bangcharoensap és munkatársai egy, az Eclipse-en végzett tanulmány [2] során egy módszert dolgoztak ki a hibás fájlok gyors meghatározására hibabejelentésekből kiindulva. A módszer három megközelítést használt a fájlok rangsorolására: nyelvfeldolgozás, forráskód feldolgozás, és kód változás vizsgálata. A nyelvfeldolgozás során a rangsort a hibás kód és a többi forráskód közötti szöveges hasonlóság alapján állították fel. Forráskód feldolgozása során a forráskód metrikák alapján határozták meg a rangsort. A kód változásának vizsgálata során pedig változás metrikákat használtak hibás fájlok jóslására.

A hibákat nem csak a forráskód egy verziójának metrikái alapján próbálhatjuk meg előrejelezni, hanem a forráskód időbeli változása során a metrikáinak változása is utalhat hiba megjelenésre. A forráskód idővel változó metrikái és az előforduló hibák közötti kapcsolatról Couto és kollégái írtak [5]. Munkájuk során bizonyítékot kerestek arra, hogy a forráskód metrikáinak változása jelezheti hibák megjelenését. Ehhez egy forráskódelem metrikáinak időbeli változását vették alapul, és statisztikai hipotézis segítségével kerestek hasonló metrika változásokat.

A PROMISE [10] egy olyan adatbázis, mely számos nyílt forráskódú és ipari projektekből gyűjtött hibákat tartalmaz. Célja olyan publikus adathalmazok és eszközök összegyűjtése, amelyek alkalmasak olyan modellek építésére, melyek felhasználhatóak különböző előrejelzésre a szoftverfejlesztés területén.

Egy hasonló adatbázis a Bug prediction dataset [8]. Abból a célból jött létre, hogy a különböző hiba előrejelző rendszerek teljesítményét mérni lehessen, illetve összehasonlítani egymással. A metrikákat és a hibákat osztály szinten tartalmazza. A mi módszerünkben metódus szintű metrikák meghatározására törekszünk.

A hiba adatbázisok sorát az iBUGS [7] adathalmazzal zárjuk. Az iBUGS az automatikus

defektus lokalizáló eszközök tesztelésére létrehozott hiba adatbázis, mely több élesben használt projektből vett információkat tartalmaz. Az iBUGS valódi, bejelentett és ellenőrzött hibákat gyűjt össze. Az összegyűjtött információk forrásául a verziókövető rendszerek és a hibakövető rendszerek szolgáltak. Az összegyűjtött hibák a következő projektekből származtak (zárójelben a projektből kinyert hibák száma látható):

- AspectJ - A Java nyelv kiegészítése aspektusorientált programozás támogatásával (223)
- Rhino - Java nyelven írt JavaScript értelmező (32)
- JodaTime (Beta) - A beépített Java dátum- és időkezelő osztályok kiterjesztése (8)

A minta adatbázis előállítását megpróbálták automatikus módon is elvégezni [6], melyet aztán összevetettek az eredeti, manuálisan validált adatbázissal. Az iBUGS ígéretes lehet, mivel viszonylag sok (263) hibát sikerült validálniuk. A tanulmányunkban egy széleskörű kutatást végeztünk annak érdekében, hogy egy még több hibát magába foglaló adatbázis jöhessen létre a megfelelőnek ítélt Java nyelvű projektek felhasználásával.

2. fejezet

GitHub

A forráskód hosting szolgáltatók közül több is szóba került. A SourceForge több százezer nyílt forráskódú projektnek ad otthont, és több mint 3 millió regisztrált felhasználója van¹. Főbb tulajdonságai közé tartozik, hogy a legtöbb verziókövető rendszert támogatja, továbbá rendelkezik feladatkövető rendszerrel, wikivel, fórummal és webes fájl menedzserrel. Több API is rendelkezésre áll, melyekkel írni és olvasni lehet a SourceForge-on található projektekhez tartozó adatokat². A projektek között is vannak népszerűek, melyeket napi szinten fejlesztenek, töltenek le. Azonban nagyon gyakori a projekteknél, hogy egy külső hibakövető rendszert használnak és gyakran előfordul az is, hogy egy GitHub projekt van az egész mögött, ott tárolják a forráskódot és a hibabejelentéseket. A Google Code szintén rendelkezik feladatkövető rendszerrel, wikivel, valamint támogatja a Subversion, Git és a Mercurial verziókövető klienseket, azonban nem rendelkezik a szükséges API-val az adatok lekéréséhez. A Bitbucket hasonlít a GitHub-hoz, de támogatja a Mercurial verziókövetőt is, illetve a feladatkövetéshez a JIRA nevű rendszert használja. A legfontosabb különbsége a GitHub-hoz képest, hogy ingyen létrehozhatunk privát projekteket is (felhasználók számának korlátozásával), ezért nélimég kevesebb nyílt rendszer található rajta.

A GitHub³ napjaink egyik legnépszerűbb forráskód tároló szolgáltatása. A szolgáltatás a Git verziókezelő rendszert használja, és alapvetően Git tárolókat biztosít a felhasználóinak, de azt saját funkciókkal egészíti ki. Mivel a Git egy parancssori eszköz, ezért a legfőbb tulajdonsága a GitHub-nak, hogy grafikus webes felületet biztosít a Git tárolók kezeléséhez. A webes felületen lehetőség van többek közt a fájlok tartalmának megtekintésére (a forráskód nyelvének megfelelő szintaxis kiemeléssel), a Git fejlesztési ágak listázására, és az egyes commit-okkal okozott változások megtekintésére. Ezen kívül számos együttműködést segítő funkcióval rendelkezik, mint például tárolónkénti feladatkezelő rendszer, wiki oldalak, felhasználók hozzáférés vezérlése, de részletes statisztikákat is biztosít diagramok formájában. Továbbá megemlítenéd, hogy a Git tárolókat Subversion tárolóként is használhatjuk, azaz bármely SVN klienssel elérhetjük

¹<http://sourceforge.net/about>

²<http://sourceforge.net/p/forgedocumentation/API/>

³<https://github.com/>

azokat⁴. Alapvetően az ingyenes használat során publikus tárolókat biztosít a GitHub, amelyet bárki elérhet, de az előfizetők privát tárolót is létre tudnak hozni. Számos nagy, nyílt forráskódú rendszerhez használják a projekt menedzselésére, többek között: node.js, ruby on rails, spring keretrendszer, symphony, zend framework, elasticsearch, netty, jenkins, etc. Több mint 3,5 millió felhasználó használja a GitHub-ot, és több mint 16 millió tárolót hoztak létre⁵. A GitHub biztosít egy API-t⁶, amely lehetővé teszi a projektek menedzselését és az adatok lekérését.

2.1. GitHub statisztika

Mint ahogy fentebb olvasható, a GitHub-ot rengetegen használják, a létrehozott tárolók száma milliós nagyságrendű. Ez a nagy mennyiségű információ adatbányászat szempontjából hasznos lehet. Ahogy fentebb szerepel, GitHub lehetőséget biztosít bárki számára az adatok eléréséhez egy API-n keresztül. Ezt a lehetőséget kihasználva létrehozták a GitHub Archive⁷ projektet, amely 2011 eleje óta a publikus eseményeket folyamatosan rögzíti, és az adatokat elérhetővé teszi 2011 elejéig visszamenőleg. Az adatok nyers JSON formátumban, időszakokra bontva letölthetők HTTP-n keresztül, de elérhetővé tették Google BigQuery-vel⁸ is, amely egy webes felületen keresztül lehetőséget biztosít lekérdezések gyors futtatására a teljes adatbázison.

Ezt kihasználva könnyen tudunk statisztikát készíteni, például 2014 január és október között létrehozott tárolók által tartalmazott projektek programozási nyelvére vonatkozóan⁹. Ehhez mindössze egy lekérdezést kell írunk, amelynek az eredményét egy táblázat formájában kapjuk meg, ami a 2.1 táblázatban látható.

2.2. Git verziókezelő rendszer

A Git¹⁰ egy ingyenes, nyílt forráskódú, elosztott verziókezelő rendszer, amely kis projektektől kezdve egészen nagy, ipari méretű projektekhez is gyorsan és hatékonyan használható. A rendszert Linus Torvalds fejlesztette ki a Linux kernel fejlesztéséhez, és eredetileg egy olyan alacsony szintű verziókezelő rendszernek tervezte, amelynek a kezeléséhez más rendszerek nyújtanak felhasználóbarát felületeket. Mára már egy teljes értékű, önmagában használható verziókezelő rendszerré nőtte ki magát.

A Subversion verziókezelő rendszertől nagyban eltér. A Git jóval gyorsabb, és a tárolók sokkal kevesebb helyet foglalnak, mint SVN esetén. Főbb tulajdonsága, hogy széles körű támogatást ad a nem-lineáris fejlesztéshez. Sokkal jobb lehetőséget biztosít branch-ek vagy magya-

⁴<https://help.github.com/articles/support-for-subversion-clients/>

⁵<https://github.com/features>

⁶<https://developer.github.com/v3/>

⁷<http://www.githubarchive.org/>

⁸<https://cloud.google.com/bigquery/what-is-bigquery>

⁹<http://adamgard.com/blog/top-github-languages-for-2013-so-far/>

¹⁰<http://git-scm.com/>

Nyelv	Tárolók száma
JavaScript	631 788
Java	448 370
Ruby	398 804
CSS	284 284
PHP	281 054
C	256 885
Python	255 287
C++	130 122
Objective-C	101 272
C#	97 262
Shell	80 150
Perl	67 620
CoffeeScript	35 539
R	35 397
Go	29 424
Scala	18 400
Lua	14 350
VimL	14 086
TeX	13 535
Swift	13 319

2.1. táblázat. 2014-ben létrehozott tárolók száma nyelvenként

rul fejlesztési ágak létrehozására és összevonására mint az SVN. Az egyes fejlesztési ágak saját történettel rendelkeznek, amely azt jelenti, hogy egy ágon a többi fejlesztési ág zavarása nélkül dolgozhatunk. SVN-nel ellentétben, a Git egy elosztott rendszer, hasonló mint a BitKeeper vagy a Mercurial. A kliens a teljes tároló történetét lemásolja, így kevesebb adatforgalomra van szükség a kliens és a szerver között a történet böngészése során. Egy másik előnye az elosztott rendszernek, hogy szerver hiba esetén a kliensekről könnyen visszaállítható a teljes tároló. A Git lehetőséget biztosít lokális commit-ok és fejlesztési ágak használatára, így a fejlesztő saját magának tud különböző verziókat létrehozni és azok között váltani anélkül, hogy szerverhez kellene fordulnia. A Git tároló története kriptográfiailag hitelesített oly módon, hogy a központi tárolóba publikált verziók elnevezése függ a történetben oda vezető úttól. Ezzel szemben az SVN csak számozza a verziókat. Az SVN-nek a Git-tel szemben egy előnye, hogy a kliensek sokkal kiforrottabb felhasználói felülettel rendelkeznek. A Git önmagában csak egy parancssori eszköz, ezért nehézkes a használata, de léteznek hozzá grafikus felülettel rendelkező szoftverek. Egy ilyen grafikus felületű rendszer a GitHub.

2.3. Feladatkezelő és hibakövető rendszer

A GitHub egy nagyon egyszerű feladatkezelő rendszerrel rendelkezik, amely hibakövetésre is alkalmas. Egy feladat létrehozásakor mindössze egy címet és egy leírást lehet megadni, fel-

használóhoz és mérföldkőhöz lehet rendelni, illetve fel lehet címkézni a feladatot. A rendszer alapból biztosít néhány címkét, például „bug”, „duplicate” és „enhancement”, de igény szerint testreszabhatóak. A címkék a feladatok kategorizálására használhatóak, ezáltal megkönnyítve a feladatok közötti keresést. Egy projekthez nem csak a hozzáadott felhasználók tudnak feladatot létrehozni, hanem gyakorlatilag bármely GitHub felhasználó. Ekkor mindössze egy címet és egy leírást lehet megadni. Ez utóbbi funkció lehetőséget ad a hibák bejelentésére is. Optimális esetben például az így felvett bejelentésekről a projekten dolgozó felhasználók eldöntik, hogy tényleg hiba-e, majd felcímkézik a megfelelő (például „bug”) címkével. De ezek az alapfunkciók elegendőek ahhoz, hogy egy jól használható feladatkezelő és hibakövető rendszert kapjunk, mivel a GitHub egy további funkciója, hogy a feladatokra az azonosítójukkal hivatkozhatunk hozzászólásokban, illetve a commit-ok során írt megjegyzésekben. Ennek a hasznossága abban nyilvánul meg, hogy a feladat (vagy hibabejelentés) webes felületén minden hozzá kapcsolódó hozzászólás és esemény (címké hozzáadása, hivatkozások, lezárás) megjelenik, például ha egy commit megjegyzésben hivatkoznak a feladatra, akkor magára a commit-ra mutató link is megjelenik az oldalon, amire rákattintva egy webes felületen egyből láthatóak a commit által tartalmazott módosítások. Commit megjegyzésből lehetőség van továbbá a feladat lezárására is egy megfelelő kulcsszó használatával (pl: „Fixes #324”).

2.4. GitHub REST API

A GitHub API egy RESTful API, amely HTTPS kéréseken keresztül lehetőséget biztosít többek között tárolók, felhasználók és feladatok menedzselésére, illetve adatok lekérdezésére és módosítására. A küldött és fogadott adatok típusa JSON.

Az API publikusan használható, szinte minden projekt adatai szabadon elérhetőek, viszont hiteles felhasználói adatok nélkül óránként maximum 60 kérést engedélyez a rendszer. Hiteles felhasználói adatokkal óránként akár 5000 kérés is engedélyezett. A listázó lekérdezések esetén figyelembe kell venni, hogy egyszerre maximum 100 elemet tartalmazhat a válaszlista, ennek kezelésére egy lapozó mechanizmust biztosít az API, viszont figyelni kell arra, hogy minden egyes oldal lekérése külön lekérdezésnek számít, azaz növeli a kérések számát.

A továbbiakban néhány felhasznált API hívást ismertetünk.

2.4.1. Tárolóhoz tartozó feladatok lekérdezése

Lehetőségünk van lekérdezni egy tárolóhoz tartozó feladatokat. Ehhez szükségünk van a tároló illetve a tároló tulajdonosának azonosítójára.

```
GET /repos/:owner/:repo/issues
```

Az `owner` paraméter a tároló tulajdonosának azonosítója, a `repo` pedig a tároló azonosítója. Az url további paraméterekkel bővíthető a megszokott módon, amelyek segítségével szűr-

hető, hogy milyen feladatokat szeretnénk listázni. Egy ilyen paraméter például a `state`, amely lehet `closed`, `open` vagy `all`, és ez értelemszerűen a lezárt illetve nyitott feladatok szűrésére használatos. Egy másik lényeges paraméter a `labels`, amellyel megadható, hogy mely címkékkel rendelkező feladatokat kérdezzük le. A fentebb említett lapozó mechanizmus egyik paramétere a `per_page`, amely azt mondja meg, hogy oldalanként hány elem szerepeljen, illetve a `page`, amely az oldalszámot adja meg. Egy példa a paraméterek használatára:

```
GET /repos/:owner/:repo/issues?state=closed&labels=bug
```

A kérésre kapott válasz egy JSON tömb, amely a feltételeknek megfelelő feladatokat tartalmazza. Az egyes feladatokhoz tartalmazza többek között a feladat azonosítóját, állapotát, címét, leírását, a címkéket, valamint a létrehozási és frissítési dátumát.

2.4.2. Feladathoz tartozó események lekérdezése

Az egyes feladatokhoz lekérdezhetőek a hozzá tartozó események, amelyek többek közt tartalmazzák a feladatra való hivatkozásokat egy commit-ból vagy egy hozzászólásból. Ehhez a tároló és a tároló tulajdonosának azonosítóján kívül szükség van a feladat azonosítójára is.

```
GET /repos/:owner/:repo/issues/:issue_number/events
```

Az `owner` és `repo` paraméterek megegyeznek a fentiekkel, az `issue_number` a feladat azonosítója. A kérésre kapott válasz egy JSON tömb, amely az eseményeket tartalmazza. Az eseményekhez az azonosítót, url-t az esemény részletesebb oldalára, eseményt létrehozó felhasználót, esemény típusát, commit azonosítót és esemény időpontját tartalmazza.

Az események típusai közül néhány fontosabb¹¹:

- **closed**: A feladat le lett zárva az adott felhasználó által. Ha a commit azonosító nem `null`, akkor arra a commit-ra mutat, amelyik megjegyzéséből hivatkoztak a feladatra a „Fixes/Closed #NN” szintaxissal.
- **referenced**: A feladatra hivatkoztak egy commit megjegyzésből. Ha a commit azonosító nem `null`, akkor arra a commit-ra mutat, amelyikből hivatkoztak rá.

2.4.3. Commit adatainak lekérdezése

Lehetőségünk van egy commit-hoz tartozó adatok lekérdezésére is. Ehhez a tároló és a tároló tulajdonosának azonosítóján kívül a commit azonosítójára van szükség.

```
GET /repos/:owner/:repo/commits/:sha
```

¹¹<https://developer.github.com/v3/issues/events/#events-1>

Az `owner` és `repo` paraméterek megegyeznek a fentiekkel, az `sha` a commit azonosítója. A kérésre kapott válasz egy JSON objektum, amely a commit adatait tartalmazza. Az adatok többek között a felhasználó adatai, commit ősei, összes sor hozzáadások és törlések száma, valamint az érintett fájlok, amelyekről több információt is tartalmaz, például fájlnev és fájl diff.

2.5. GitHub Java API

A GitHub API használatához számos nyelven készült segédkönyvtár¹², amely az adott nyelven könnyen elérhetővé teszi az API által nyújtott szolgáltatásokat. Jelenleg a hivatalos és folyamatosan naprakészen tartott könyvtárak az Octokit¹³ könyvtárai, amelyek három környezetet támogatnak: Ruby, Objective-C, .NET, továbbá fejlesztés alatt áll egy a GO nyelvhez. Ezeken kívül számos nemhivatalos könyvtár érhető el különböző környezetekhez. Egy ilyen segédkönyvtár a Java-hoz a Github Java API.

A készítőik célja a könyvtár létrehozásakor az volt, hogy teljes mértékben támogassa a GitHub v3 API-t. Jelenleg a könyvtárat a GitHub Mylyn csatolóhoz használják, hogy Eclipse-ből elérhetőek és kezelhetőek legyenek a GitHub feladatok, tárolók és egyéb funkciók. (A Mylyn projekt lényege olyan beépülő modulok készítése, amelyek támogatást adnak a feladatkezelő rendszerek és egyéb eszközök használatára Eclipse-ből.)

A GitHub Java API lényegében az adatokat a GitHub API JSON sémájának megfelelő Java osztályokra képi le. Három főbb csomagból áll:

- **Kliens**: ez felelős a GitHub API-val való kommunikációért, kérések kezeléséért, továbbá az adatok JSON-ból Java modellre való leképezéséért.
- **Szolgáltatások**: ez a csomag tartalmazza azokat az osztályokat, amelyek az API hívásokat indítják, és az adatok Java-s reprezentációjával térnek vissza. Ezek az osztályok az egyes adattípusokhoz vannak definiálva, például „IssueService”, vagy „RepositoryService”.
- **Modell**: ez tartalmazza a modell osztályokat, amelyek reprezentálják az összes adattípust ami elérhető az API-n keresztül.

2.5.1. Felhasználó hitelesítés

A GitHub API-t érdemes hitelesített felhasználóként használni a kérések számára vonatkozó korlát miatt. Ezt a `GitHubClient` osztállyal tudjuk megtenni, melynek használata a 2.1 forráskódon látható.

```
1 GitHubClient client = new GitHubClient();
```

¹²<https://developer.github.com/libraries/>

¹³<https://github.com/octokit>

```
2 client.setCredentials("username", "password");
```

2.1. forráskód. GitHub Java API kliens használata példa.

2.5.2. Szolgáltatások használata

A GitHub Java API a különböző adatokra vonatkozó funkciókat szolgáltatásokra bontja. A szolgáltatások metódusainak paramétereinek és visszatérési értékeinek típusai a Java-s reprezentációja az adatoknak. Abban az esetben, ha a lekérdezés egy listát ad vissza, akkor a kód automatikusan lekezel a GitHub API által oldalanként engedélyezett maximum elemek számát, azaz szükség esetén több kérést indít a szerver felé a lapozási mechanizmust használva. A szolgáltatások használatát a 2.2 forráskód szemlélteti.

```
1 RepositoryService repoService = new RepositoryService();
2 Repository repo = service.getRepository("repouser", "repo");
3
4 CommitService commitService = new CommitService();
5 List<RepositoryCommit> repoCommits =
    commitService.getCommits(repo);
```

2.2. forráskód. GitHub Java API szolgáltatás használata példa.

Egy szolgáltatást hitelesített felhasználóval nagyon egyszerű használni, csupán a szolgáltatás konstruktorának paraméterül kell adni a megfelelő klienst. Erre egy példa a 2.3 forráskódon látható.

```
1 GitHubClient client = new GitHubClient();
2 client.setCredentials("username", "password");
3
4 RepositoryService repoService = new RepositoryService(client);
5 Repository repo = service.getRepository("repouser", "repo");
```

2.3. forráskód. GitHub Java API kliens és szolgáltatás használata példa.

3. fejezet

A metrikák

A kifejlesztett szoftver rendszerek bizonyos szempontok szerinti jellemzése igen nehéz feladat, mivel rengeteg szubjektív faktor is szerepet játszhat egy-egy karakterisztika leírásában. A metrikák segítségével a projektek bizonyos tulajdonságait tudjuk objektív módon mérni. Ezek a tulajdonságok sokfélék lehetnek. Vonatkozhatnak magára a forráskódra, a projekt menedzsmentre vagy a szoftver futására is. Mérésüknek több különböző módja lehet. A szoftverre vonatkozó metrikák segítségével következtethetünk a szoftver magasabb szintű tulajdonságaira is, mint például a forráskód minőségére vagy a hibák eloszlására, de felhasználhatóak például költségbecslésre, teljesítmény-optimalizálásra, vagy különböző szoftverminőséget támogató tevékenységekre. Esetünkben a statikus forráskód metrikák és a GitHub-ról kinyerhető metrikák az érdekesek. Ezeket felhasználva jellemezhetjük a hibás forráskódot fájl szinten, vagy akár forráskód elem (osztály, metódus) szinten.

3.1. Forráskód metrikák

Az objektumorientált forráskód metrikák területe már régóta kutatott témakör [4], így nem véletlen, hogy a szoftver metrikák mérésére már számos eszközt fejlesztettek ki¹. Ezek az eszközök számos programnyelven megírt rendszer részletes vizsgálatára alkalmasak. Léteznek statikus forráskód elemző eszközök például az alábbi programozási nyelvekhez:

- Ada
- C/C++
- Java
- JavaScript
- Python

¹http://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis

A fenti programnyelvek közül a Java nyelvhez a Szegedi Tudományegyetem Szoftverfejlesztés Tanszék munkatársai által kifejlesztett SourceMeter for Java elemzőkészletet használjuk fel. A SourceMeter for Java eszköz képes statikus forráskód metrikák kiszámítására, a Java nyelvre megadott szabálysértések ellenőrzésére, továbbá a duplikált kódrészek lokalizálására. A tanulmány szempontjából ezek közül a forráskódra kiszámolt metrikák halmaza a releváns. A kiszámolt forráskód metrikákat több típusba sorolhatjuk. Eszerint lehetnek:

- méret,
- öröklődés,
- csatolás,
- kohézió és
- komplexitás

alapú metrikák. A metrikákat felhasználhatjuk különböző minőségbiztosítási módszerekhez is. Egy ilyen minőségbiztosítási felhasználás lehet például egy minősítő modell kifejlesztése[1]. Egy másik ezzel kapcsolatos terület lehet a hibák eloszlása és a kiszámított metrikák korrelációjának megállapítása, melyre az előállítani kívánt hiba adatbázisok remek lehetőséget biztosítanak. Az előállított statikus forráskód metrikák pontos definíciója megtalálható a <https://sonarqube.sourcemeter.com/plugins/home/sm-help> oldalon.

3.2. Verziókövető rendszerből kinyert metrikák

A statikus forráskód metrikák mellett meghatároztunk néhány más természetű metrikát is a rendelkezésünkre álló adatokból. A verziókövető rendszerből kinyerhető információból megállapíthatjuk, hogy az egyes fájlokon hány módosítás történt a vizsgált verzióig, illetve ezen módosítások közül mennyi volt hibajavítás. Egy általánosabb metrikát is számoltunk, mégpedig az adott verzió létrehozási dátuma alapján a nyitott hibabejelentések számát. A GitHub által rendelkezésünkre bocsátott felhasználói statisztika alapján további metrikaként felvettük az utoljára módosító felhasználó összes módosításainak számát a projekten. Ezeket fájl szinten számoltuk. A GitHub-ról kinyert adatok alapján előállított metrikák teljes listája a következő:

- módosítások száma
- hibajavítások száma
- nyitott hibabejelentések száma
- a módosítást elvégző személy módosításainak száma

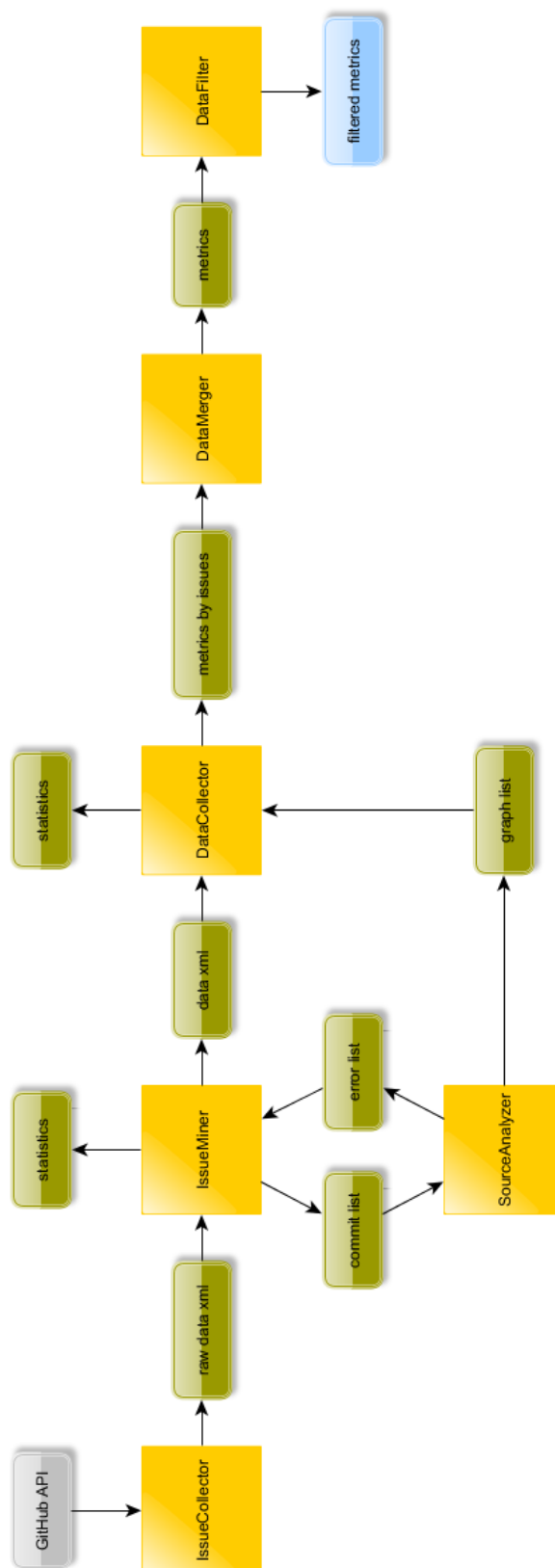
4. fejezet

Adatbányászat GitHub-ról

Az adatok feldolgozását több lépésben végeztük el. Első lépésként a GitHub-on található adatokat gyűjtöttük össze. Ezt az IssueCollector nevű program végezte el. Az így kapott nyers adatokat ezután feldolgoztuk, statisztikát készítettünk, amihez az IssueMiner nevű programot írtuk. Következő lépésként a SourceAnalyzer segítségével letöltöttük a szükséges forráskód verziókat és elemeztük azokat a SourceMeter for Java elemző készlettel. Ezeknek az elemzéseknek az eredményét ezután összekötöttük a GitHub-ról letöltött adatokkal, meghatároztuk a hibás kódrészeket, valamint jellemeztük azokat. Erre a célra a DataCollector nevű program szolgált. Kimenetként minden hibabejelentéshez előállt egy fájl, mely a metrikákat tartalmazza. Ahhoz, hogy egy közös adatbázist kapjunk a hibákról, összefűztük ezeket a fájlokat a DataMerger nevű program segítségével. Ebben az adatbázisban több olyan adat is keletkezett, mely megtalálható hibás, illetve hibamentes kód jellemzéseként is, ezért az ilyen bejegyzések szűrésére készítettük a DataFilter nevű programot. Így a végén előállt egy adatbázis, mely tartalmazta az összes kinyert hiba jellemzését, valamint megfelelően volt szűrve, hogy ne tartalmazzon félrevezető adatot. A folyamatot és a komponenseket a 4.1. ábra szemlélteti.

4.1. A projekt választási szempontok

A GitHub-on lévő projektek választásakor számos szempontot vettünk figyelembe. Először is Java nyelvű projekteket vizsgáltunk, és azok között is a minél nagyobb méretűekre koncentráltunk, hogy alkalmas legyen az ily módú elemzésre. Fontos volt továbbá, hogy megfelelő számú commit-tal rendelkezzen, jól elkülöníthetően címkézzék a hibabejelentéseket, valamint a commit megjegyzésekben hivatkozzanak a megfelelő hibabejelentésekre. Továbbá preferáltuk a jelenleg is aktív projekteket. A keresés során sok olyan projekttel találkoztunk, melyek a legtöbb szempontból megfeleltek volna, azonban külső hibakezelő rendszert használtak, így nehézkes lett volna a feldolgozásuk.



4.1. ábra. A folyamat komponensei

4.2. A választott projektek

A kikötött szempontok alapján 13 projektet választottunk ki. A projektek néhány jellemzőjét a 4.2 és 4.3 ábra, illetve a 4.1 táblázat mutatja.

Az alábbi projektek kerültek kiválasztásra:

- **JUnit**¹: Egy Java-s keretrendszer unit tesztek írására.
- **Mission Control Technologies**²: A NASA által eredetileg űrrepülési műveletekhez fejlesztett valós idejű monitorozó és vizualizációs platform, mely felhasználható bármilyen más adat monitorozáshoz is.
- **OrientDB**³: Egy népszerű dokumentum gráf alapú NoSQL adatbázis. Főként a skálázhatóságáról és a sebességéről híres.
- **Neo4j**⁴: A világ vezető gráf adatbázisa. Nagy teljesítményre képes.
- **MapDB**⁵: Egy sokoldalú, gyors és könnyen használható Java-s adatbázis motor.
- **mcMMO**⁶: A Minecraft alap játék mechanikájára épülő RPG játék.
- **Titan**⁷: Egy nagy teljesítményű, jól skálázható gráf adatbázis.
- **Oryx**⁸: Egy nyílt forráskódú gépi tanulás algoritmusokat biztosító szoftver, amely hatalmas méretű adatokon való alkalmazásukat teszi lehetővé.
- **jHipster**⁹: Egy sokoldalú Java-s webalkalmazás generáló szoftver.
- **Universal Image Loader**¹⁰: Képek betöltését segítő Androidos segédkönyvtár.
- **Netty**¹¹: Egy asszinkron eseményvezérelt hálózati keretrendszer.
- **ANTLR v4**¹²: Népszerű szoftver a nyelvfeldolgozás területén. Szintaktikus és szemantikus nyelvi elemzőket állít elő.
- **Elasticsearch**¹³: Népszerű RESTful keresőmotor.

¹<https://github.com/junit-team/junit>

²<https://github.com/nasa/mct>

³<https://github.com/orientechnologies/orientdb>

⁴<https://github.com/neo4j/neo4j>

⁵<https://github.com/jankotek/MapDB>

⁶<https://github.com/mcMMO-Dev/mcMMO>

⁷<https://github.com/thinkaurelius/titan>

⁸<https://github.com/cloudera/oryx>

⁹<https://github.com/jhipster/generator-jhipster>

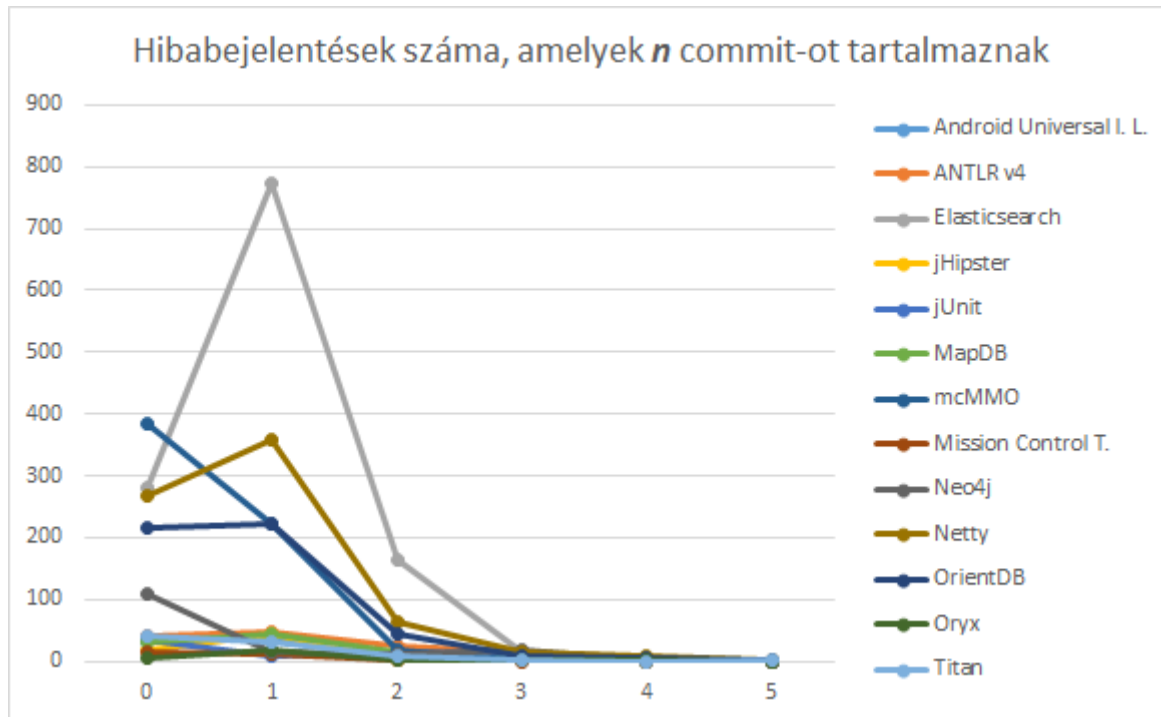
¹⁰<https://github.com/nostr13/Android-Universal-Image-Loader>

¹¹<https://github.com/netty/netty>

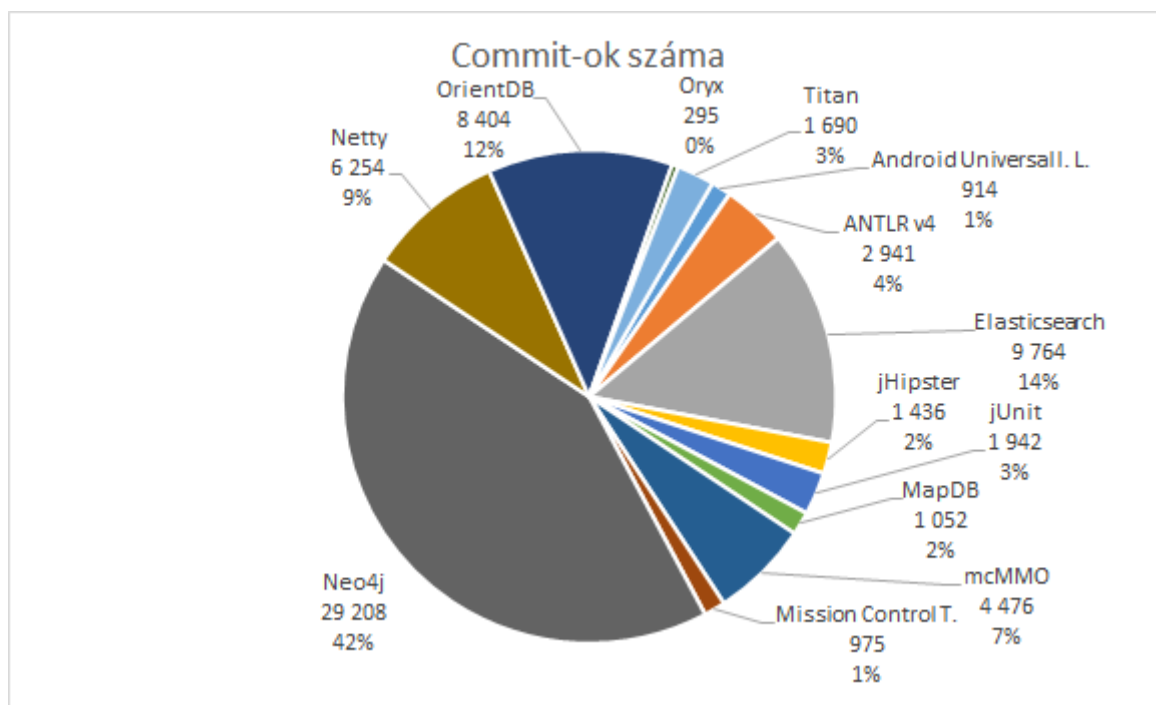
¹²<https://github.com/antlr/antlr4>

¹³<https://github.com/elasticsearch/elasticsearch>

A 4.2 ábrán az látható, hogy a lezárt hibabejelentésekhez mennyi commit tartozik. Ebből látható, hogy viszonylag sok olyan eset van, ahol nincs egy commit sem. Ennek számos oka lehet, például nem hivatkoztak a hibabejelentésre a commit során, olyan hibabejelentésről van szó amit már javítottak, vagy csak nem tervezik javítani a hibát.



4.2. ábra. Hibabejelentések száma a hozzájuk tartozó commit-ok alapján



4.3. ábra. A projektekhez tartozó commit-ok száma

	Commit-ok száma	Hibabejelentésekhez tartozó commit-ok száma	Hibabejelentések száma	Nyitott hibabejelentések száma	Lezárt hibabejelentések száma	Hibabejelentésenként a commitok átlagos száma
Android Universal I. L.	914	52	80	5	75	0,69
ANTLR v4	2 941	109	146	16	130	0,84
Elasticsearch	9 764	979	1 331	91	1 240	0,79
jHipster	1 436	52	68	0	68	0,76
jUnit	1 942	66	75	4	71	0,93
MapDB	1 052	80	109	18	91	0,88
mcMMO	4 476	251	635	11	624	0,40
Mission Control T.	975	15	37	9	28	0,54
Neo4j	29 208	76	268	112	156	0,49
Netty	6 254	567	747	28	719	0,79
OrientDB	8 404	362	710	212	498	0,73
Oryx	295	29	27	0	27	1,07
Titan	1 690	50	88	6	82	0,61

4.1. táblázat. Statisztikai adatok a választott projektekről

A 4.3 ábrán a projektekhez tartozó commit-ok számának aránya látható, amely a projektek aktivitását és méretét szemlélteti. Ebből jól látható, hogy különböző méretű projekteket választottunk az adatok sokszínűsége végett.

A 4.1 táblázat pontosabb képet ad a projektekről. A táblázatban a projektekhez tartozó hibabejelentések és a commit-ok száma látható. Feltűnő lehet a Neo4j projekt esetén, hogy a rengeteg commit ellenére kevés hibabejelentés és hozzájuk tartozó commit van. Ez azzal magyarázható, hogy nem használják olyan intenzíven a GitHub felület adta lehetőségeket.

4.3. Adatok összegyűjtése

A kiválasztott projektek adatait a GitHub Java API-t használva, a GitHub API-n keresztül mentettük le. Az adatok lementésére azért volt szükség, hogy legyen egy állandóan, bármikor elérhető adatforrásunk. Az API használata során a kérések számának korlátozása az adatok teljesen automatikus lementését nem tette lehetővé, ezért a problémára azt a megoldást alkalmaztuk, hogy ha a rendelkezésre álló kérések száma egy bizonyos érték alá csökken, akkor a program tevékenyen várakozik, amíg a GitHub rendszere nem enged egy bizonyos értéknél több kérést.

Az adatok lementéséhez a tároló és a tároló tulajdonosának azonosítójára, és címkék azon listájára van szükség, amelyek az adott projektben a hibabejelentéseket jelölik. Ezen kívül még meg kell adni egy érvényes GitHub felhasználónév és jelszó párost, amelyet az API hívásokhoz használ. A lementett adatok tartalmazzák a tárolóhoz rendelt felhasználókat (Contributors), nyitott illetve zárt hibabejelentéseket (Issue), továbbá az összes commit (Commits) fontosabb adatait. A felhasználókhoz (User) az azonosítójukat illetve a tárolón végzett commit-ok számát tároljuk. A nyitott hibabejelentésekről (OpenIssues) csak a létrehozásuk dátumát tároljuk el. A zárt bejelentésekről (ClosedIssues) a létrehozás dátumát, a lezárás dátumát, és a hozzájuk tartozó commit-ok azonosítóját és létrehozásuk dátumát tároljuk el. A commit-okhoz (Commit) eltárolt adatok között szerepel az azt létrehozó felhasználó azonosítója, fejlesztési ág azonosítója, commit ősei (Parents) és az érintett fájlok (Files) adatai a diff-el együtt (Patch).

Az így összegyűjtött adatokat projektenként egy XML fájlba mentjük le, amelynek a struktúrája a 4.1 forráskódon látható.

```
1 <Project user="repouser" repo="repo">
2   <Contributors>
3     <User id="456" contributions="15" />
4     ...
5   </Contributors>
6   <ClosedIssues count="5">
7     <Issue number="123" created_at="..." closed_at="..." >
8       <Commit sha="..." created_at="..." event="referenced" />
9     </Issue>
10    ...
11  </ClosedIssues>
```

```

12 <OpenIssues count="6">
13   <Issue number="234" created_at="..." />
14   ...
15 </OpenIssues>
16 <Commits>
17   <Commit sha="..." date="..." author="456" tree="...">
18     <Parents>
19       <Parent sha="..." />
20     </Parents>
21     <Files>
22       <File filename="..." status="modified" additions="3"
23         deletions="2" changes="5" >
24         <Patch><![CDATA[ ... ]]></Patch>
25       </File>
26       ...
27     </Files>
28   </Commit>
29   ...
30 </Commits>
31 </Project>

```

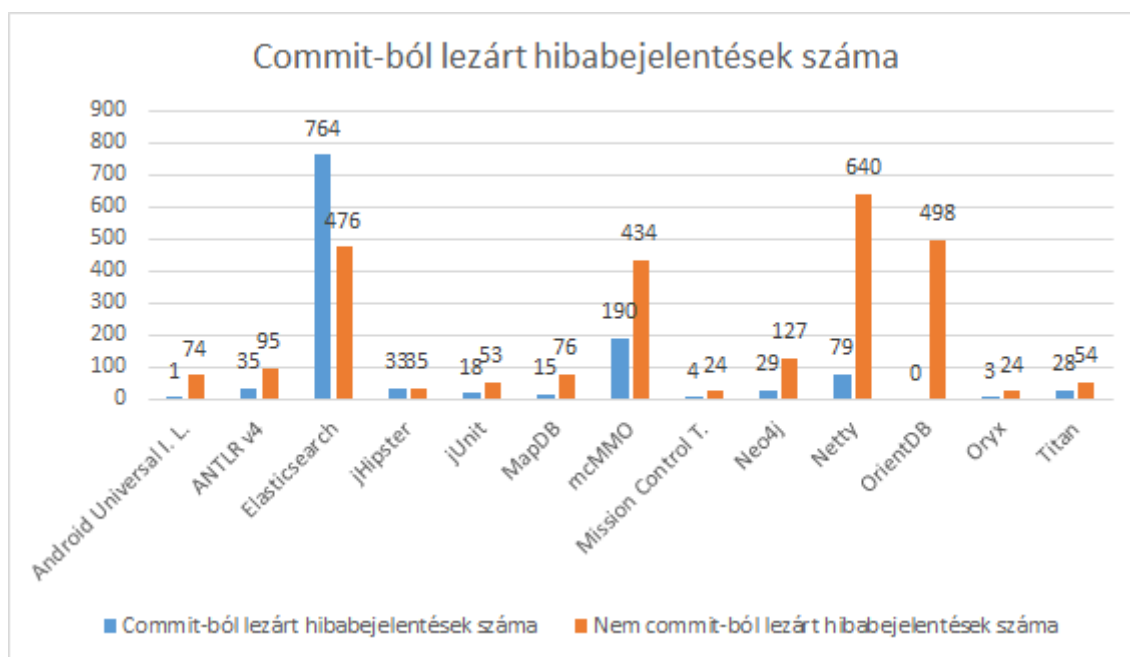
4.1. forráskód. Egy projekt GitHub-ról lementett adatainak az XML sémája.

4.4. Nyers adatok feldolgozása

A GitHub-ról lementett adatok csak a projektek adatainak egy nyers lokális reprezentációja. Ez az összes commit adatait tartalmazza, amelyek közül csak a hibabejelentésekhez tartozóakra és a bejelentésenkénti legelső commit előtti commit-ra van szükség. Ezen commit-ok között is előfordulhat olyan, amelyet el kell távolítani, mert már nem érhető el Git-en keresztül. A nyers adatok feldolgozása során ezeket a szűréseket végezzük el, továbbá statisztikákat is készítünk a projektek összehasonlítása végett. A statisztikai adatok között szerepel például commit-ok száma, hibabejelentések száma és hibabejelentésenkénti commit-ok átlagos száma (4.1 táblázat). Ezeken kívül olyan adatokat is kinyerhetünk, mint például a commit-okból lezárt hibabejelentések száma, amivel a GitHub ezen funkciójának használatának gyakoriságát tudjuk mérni (4.4 ábra). Ebből szintén az látható, hogy változatosak a projektek, van ahol használják, és van ahol egyáltalán nem.

4.5. Forráskód verziók elemzése

Az előző lépésben előálló commit azonosítók alapján letöltöttük a projektek megfelelő verzióit. Ez nem ment zökkenőmentesen, mivel bizonyos commit-ok akkor jöttek létre, amikor a különböző fejlesztési ágakat vonták össze és ezeket a commit-okat nem lehet letölteni. Az ilyen esetekben léteznek egy alternatív commit, melynek ugyanaz a tartalma, viszont ezeket le lehet



4.4. ábra. A commit-ból és nem commit-ból lezárt hibabejelentések száma

tölteni. Előfordult olyan is, hogy azért nem lehetett letölteni a commit-okat, mert törlésre kerültek. Ilyenkor egyszerűen töröltük ezeket a bejegyzéseket. A problémás commitok kiszűrését úgy oldottuk meg, hogy naplóztuk az ilyen hibákat, és töröltük a kiindulási halmazból, majd újra meghatároztuk a vizsgálandó commit-okat.

Ezután a forráskód elemek és a hozzájuk tartozó metrikák meghatározása céljából ezeket a programokat leelemeztük a SourceMeter for Java segítségével. A hibabejelentésekhez tartozó, a hibát még tartalmazó, valamint a hibát már nem tartalmazó verzióhoz szükségesek a metrikák, így azokat a teljes elemzőkészlettel elemeztük. A többi verziót csak az érintett forráskód elemek meghatározása céljából kellett vizsgálnunk, így azokhoz nem használtuk az összes eszközt, ezzel időt és tárhelyet spórolva. Az eredményeket gráfok formájában kaptuk meg. Ezek a gráfok tartalmazzák a forráskód elemeket csomag, osztály és metódus felbontásban, a statikus forráskód metrikákat, illetve a forráskód elemek forráskód pozícióját.

A fentebbi műveleteket a SourceAnalyzer nevű komponens végzi el. Az elemzett commit-ok és az összes commit viszonyát a Függelékben található diagramok szemléltetik. A felső sorban lévő jelek jelölik az elemzett, az alsó sorban lévőek pedig a nem elemzett verziókat. Látható belőlük, hogy mikortól kezdte el a hibakövetést használni, és hogy arányában mekkora részét érintették a hibajavítások a commit-oknak. Az elemzett commit-ok sűrűségéről egy összefoglalót tartalmaz a 4.2. táblázat.

4.6. Hibajellemzők kinyerése

A következő lépés a két adathalmaz - az elemzési gráfok, és a GitHub-ról kigyűjtött adatok - összekapcsolása, és a hibák jellemzőinek kinyerése. Ezzel a lépéssel a következőket határoztuk

meg:

- commit-ok által érintett forráskód elemek
- az érintett forráskód elemek statikus forráskód metrikái
- commit-onként a fájlok addigi módosításainak és fixszeinek száma
- commit-onként a fájlokat utoljára módosító felhasználó
- commit-onként a keletkezési idejében nyitott hibabejelentések száma

```

1 @@ -190,7 +190,7 @@ public void enterOuterAlt(ParserRuleContext
    localctx, int altNum) {
2     * If the parser is creating parse trees, the current
    symbol
3     * would also be added as a child to the current
    context (node).
4     */
5 -     protected Object consume() {
6 +     public Object consume() {
7         Object o = getCurrentInputSymbol();
8         getInputStream().consume();
9         if ( buildParseTrees ) {
10 @@ -227,6 +227,10 @@ public ParserRuleContext
    getInvokingContext(int ruleIndex) {
11         return null;
12     }
13
14 +     public ParserRuleContext getContext() {
15 +         return _ctx;
16 +     }
17 +
18     public boolean inContext(String context) {
19         // TODO: useful in parser?
20         return false;

```

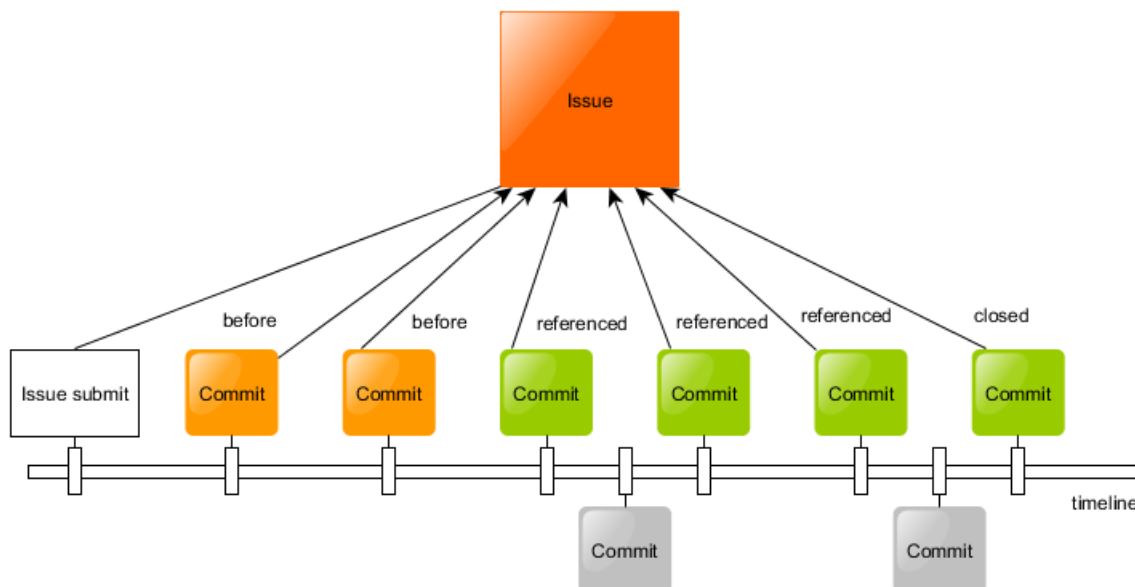
4.2. forráskód. Példa diff fájl.

Ennél a pontnál rendelkezésünkre álltak a gráfok, a GitHub projekt felhasználói adatai és a hibabejelentésekhez tartozó legelső, a legelső előtti, illetve az utolsó commit-ok azonosítója. A DataCollector feladatának része bizonyos metrikák számítása. Ezekhez szükséges még az egyes módosítások által érintett forráskód elemek meghatározása. A módosításokhoz tartozó unified diff alapján megkaphatjuk azokat a sorokat, melyeket érintett a változás egy fájlban. Egy ilyen diff fájl tartalmazhat egy vagy több eltérést (4.2. forráskód). Az eltérés tartalmazza a módosított rész előtti és utáni három-három sort is feltéve, hogy nem a fájl elején vagy a

végén van a módosítás. A konkrétan érintett sorokhoz pedig tartalmazza a korábbi, majd a későbbi verziókat, melyekből visszanyerhető az eredeti és a módosult szövegrészlet. Az ilyen módosításokhoz tartozik még egy-egy kezdő sorszám az érintetlen, és a módosult részhez. Ezekből, illetve az eltérés sorainak száma alapján megkaphatjuk azt a sor tartományt, melyben található a módosítás. Érdeemes kivonni ebből a tartományból azt a három-három sort, melyek a környezetet tartalmazzák, hogy pontosabb eredményt kapjunk. A GitHub-által generált diff fájlok tartalmazzák az eltérésekhez, hogy melyik metódusban találhatóak. Számunkra ez nem hordoz extra információt, mivel ebből nem tudjuk visszanyerni az összes forráskód elemet, melyet érint a módosítás. Ezután nincs más dolgunk, mint az adott fájlban lévő forráskód elemeket megvizsgálnunk, és amelyeknek a forráskód pozíciója metszi a módosított sorokat, azok a módosított elemek. A forráskód elemeket a teljes nevükkel azonosítjuk, például: `org.antlr.v4.test.TestTopologicalSort.testRepeatedEdges()`. A módosított fájlokat nem kell külön meghatározni, mivel a diff fájlok egy-egy fájlhoz tartoznak. Ezek alapján a program kigyűjti a módosított osztályokat és metódusokat.

Így már elég információnk van a metrikákhoz meghatározásához. Kiszámolja egy adott forráskód verzióra a létrehozási idejében nyitott hibabejelentések számát. Ezt úgy hajtja végre, hogy minden egyes commit-ra megvizsgálja, hogy hány hibabejelentés létrehozási ideje esik korábbra, lezárási ideje pedig későbbre, mint a commit módosítási ideje. Továbbá kiszámítja, hogy az egyes fájlokhoz hány módosítás tartozik. Ehhez először keletkezési idő szerinti sorrendbe rendezi a commit-okat. A legkorábbi commit-ban módosított fájlknál növeli a számlálót eggyel, majd az összes többi commit-ban is növeli a számlálót azoknál a fájlknál, amelyeknek az elérési útja - a fájlnevvvel együtt - megegyezik a módosítottakéval. Tehát a fájlokat elérési út alapján különböztetjük meg. Időrendben az összes commit-ra végrehajtva ezt, megkapjuk a kívánt metrikát. A folyamat közben egyből számolódik a javítások száma is oly módon, hogy ha az adott módosítás egy hibabejelentés utolsó commit-ja, akkor a hibajavítások számlálóját is frissül. Meghatározza a fájlokhoz továbbá, hogy hány módosítása van összesen a projektnek annak a felhasználónak, aki utoljára módosította azt. Ehhez rendelkezésünkre áll a GitHub felhasználói statisztikája, mely tartalmazza ezt az információt. A feladat az utoljára módosító felhasználó kiválasztása. Ezt az előbbi logika szerint lehet kivitelezni azzal a módosítással, hogy ebben az esetben a felhasználó azonosítóját frissítjük a fájlokon. Ezen felhasználói módosítások száma nem az adott commit keletkezési idejére érvényes, hanem az adatok GitHub-ról történő letöltésének idejére.

Ezt követően meghatározza a hibabejelentésekhez azokat a commit-okat, melyek a bejelentés és az első módosítás közé esnek. Azért lényegesek ezek a commit-ok, mert ezekben a verziókban találhatóak meg a hibák. A hibabejelentés és a commit-ok viszonyát a 4.5. ábra jelöli. Ezeknek a hibáknak a bejelöléséhez összegyűjti a hibabejelentéshez tartozó commit-okból a módosított forráskódelemeket, majd növeli ezeknek az elemeknek a hiba számlálóját a megfelelő commit-okban. A forráskód elemeket továbbra is a teljes nevükkel azonosítjuk. Ezzel eljutottunk arra a pontra, ahol meg van minden metrika és minden érintett forráskód elem. Ezt



4.5. ábra. A hibabejelentés és a commit-ok kapcsolata

követően a program ezeket kimentí csv formátumban. A kimentéshez még be kell tölteni a statisztikus forráskód metrikákat a gráfokból. A gráfok egyszer már feldolgozásra kerültek, de a nagy méretük memória problémákat vetne fel, ha betöltve tárolnánk az összeset. Ennek megoldása céljából az első betöltés során csak a módosított forráskód elemek kerültek meghatározásra. Az adatok exportálását hibabejelentésként végzi el a program, amihez betölti a hibát még tartalmazó, és a hibát már nem tartalmazó gráfot, melyekből kinyeri a forráskód metrikákat. Külön fájlba menti a fájlokhoz, a metódusokhoz és az osztályokhoz tartozó metrikákat. Két féle adatbázist állít elő. Az egyikben a metrikákhoz hozzárendeli, hogy hibás forráskódhoz tartoznak-e, a másikban pedig az adott forráskód elemében található hibák számát. Így hat féle kimenet jön létre. A csv fájlok első sorában található egy fejléc a metrikák neveivel. Legvégül pedig készít egy statisztikát a forráskód elemek és a diff fájlok párosításáról. Ez tartalmazza a commit-okhoz tartozó eltérések és a metszett forráskód elemek számát.

4.7. Kinyert adatok összefűzése

A DataCollector által előállított csv-k hibabejelentésekre vannak lebontva. Ahhoz, hogy egy nagy adatbázist kapjunk, össze kell fűznünk ezeket az adatokat. Minden fájlban eltérő lehet a fejléc, ugyanis egy szabálysértésnek a darabszáma csak akkor jelenik meg, ha legalább egyszer előfordul a hibabejelentéshez tartozó forráskód elemekben. A csv fájlok konzisztens módon történő összefűzését a DataMerger végzi el. Betölti a csv-eket és a soraikból a fejléccel kulcs-érték párokat alkot. Kimentésnél ha egy sorhoz nem tartozik bizonyos metrika, akkor azt nulla értékkel írja ki. Ez nem rontja el az adathalmazt, mivel csak szabálysértés darabszámok hiányoznak, ami egyenértékű azzal, hogy nincs olyan szabálysértés a forráskód elemében. Ezzel mind a hat

	Átlag (nap)	Maximum (nap)	Az elemzések között a nem elemzett verziók átlagos száma
Android Universal I. L.	7.65	105.07	7.44
ANTLR v4	5.09	131.94	11.8
Elasticsearch	0.77	29.88	4.68
jHipster	3.03	80.53	12.92
jUnit	14.74	303.47	11.04
MapDB	3.67	49.85	5.52
mcMMO	1.74	258.64	8.22
Mission Control T.	23.62	311.01	33.81
Neo4j	5.50	94.87	58.87
Netty	2.42	65.07	10.22
OrientDB	0.83	82.72	5.36
Oryx	2.77	23.90	3.46
Titan	9.36	197.61	17.04

4.2. táblázat. Statisztika az elemzett forráskód verziók között eltelt időről

féle adathalmazt előállítottuk. A halmazok méretét a 4.3. táblázat foglalja össze.

Adatbázis jellege	Adatsorok száma (db)	Adatsorok száma szűrés után (db)
Metódus szintű metrikák	27487	4079
Osztály szintű metrikák	15007	3877
Fájl szintű metrikák	10634	1714

4.3. táblázat. Az előállított adatbázisok mérete

4.8. Adatok szűrése

Az adatok között előfordulnak olyan sorok, melyekben megegyeznek a metrikák, viszont az egyik hibás forráskódhoz tartozik, a másik pedig már kijavítotthoz. Az ilyen részek félrevezetőek lehetnek, ezért az adatok szűrésére készítettük a DataFilter nevű programot. Ez többféleképpen képes szűrni az adatokat. Meg lehet adni egy arányt, melyet ha nem ér el az azonos metrikájú, különböző osztályba tartozó sorok száma, akkor eldobja a program. Ha például 30-30 ilyen metrika van hibás, és nem hibás forráskódhoz, akkor a kettő aránya 1, vagyis további feldolgozás - például gépi tanulás - szempontjából nem hordoz releváns információt. Amennyiben az előbbi arány paraméternek 2-t adunk meg, úgy a program eldobja ezeket a sorokat. Azokban az esetekben, ahol az arány megfelelő, még mindig kérdéses, hogy mi legyen azokkal a sorokkal. Meg is tarthatjuk, de előfordulhat, hogy félrevezető lehet. Egy lehetséges megoldás, ha a nagyobbik darabszámúakat tartjuk csak meg. Példának vegyük az 5-20 darabszámokat.

Az előbbi megoldás hatására 0-20 kerül a szűrt halmazba. Egy másik lehetséges megoldás, ha kivonjuk a kisebbik darabszámot. Ebben az esetben a példából 0-15 lesz. Az is megoldás, ha a nagyobb darabszámú osztályból csak egy példányt tartanánk meg, viszont ezzel elveszítenénk az egyes adatsorok súlyát, ami értékes információ lehet bizonyos felhasználás szempontjából. A programban paraméterek alapján megválasztható, hogy melyik módszer alapján szűrjön. Az így előállítható adathalmazok már alkalmasabbak különféle felhasználásra. Lefuttattuk az adatbázisra 1.5-ös aránnyal, valamint kivonásos szűréssel. Az így kapott halmaz méreteit a 4.3. táblázat tartalmazza.

5. fejezet

Összegzés

A tanulmány során kidolgoztunk egy módszert és annak implementációját, mely előállít egy adatbázist a hibákról a statikus forráskód metrikák segítségével. Az eddigi módszerek osztály, de leginkább fájl szinten számoltak metrikákat. Ebből a szempontból a mi módszerünk metódus szinten képes ezt megtenni, ezáltal egyedülálló, a többihez képest sokkal gazdagabb információ-tartalmú adatbázist hozva létre. Az sem elhanyagolható, hogy a metódus szintű részletességgel az egyes forráskód elemek kerülnek jellemzésre, szemben az eddigi fájl vagy osztály szinthez képest. Egy másik fontos szempont, hogy a GitHub-on található adatokat használja fel a rendszerünk, amellyel szintén kiemelkedik az eddigi munkák közül, melyek néhány nagy méretű, pontos menedzsment mellett létrejött adathalmazokat dolgoztak fel. Valamint a GitHub-on lévő projekteket vizsgálva kiválasztottuk a 13 leginkább alkalmas Java nyelvű projektet, melyekből fel is építettünk egy adatbázist. Ez az adatbázis rendelkezik a fentebb felsorolt tulajdonságokkal, így új lehetőségeket ad a hibák és a forráskód metrikák közötti összefüggések vizsgálatára.

6. fejezet

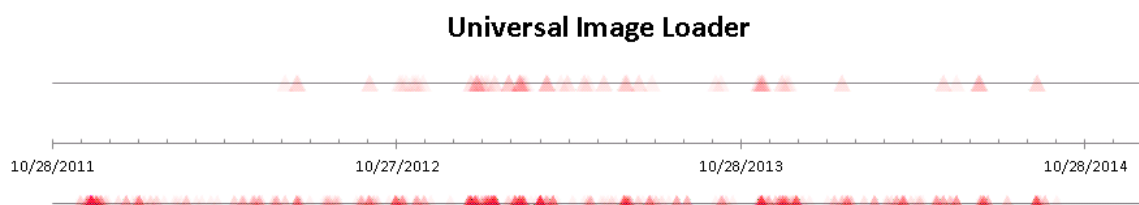
További tervek

Terveink között van az adatbázis bővítése további projektekkel és további adatforrások bevonása, mint például a SourceForge vagy a Bitbucket. Tervezzük továbbá a metrikák finomítását és újabb jellemzők meghatározását is. Az általunk számított fájl szintű metrikák további elemzésekkel levihetők alacsonyabb színre, az osztályokra és a metódusokra, valamint a GitHub-ról kinyerhető adatokból további metrikákat definiálhatunk. Célunk az adatok felhasználása a hibák és a forráskód metrikák közötti összefüggések vizsgálatára, illetve az esetleges eredmények alkalmazása a forráskód hibák automatikus felismerésének elősegítésére.

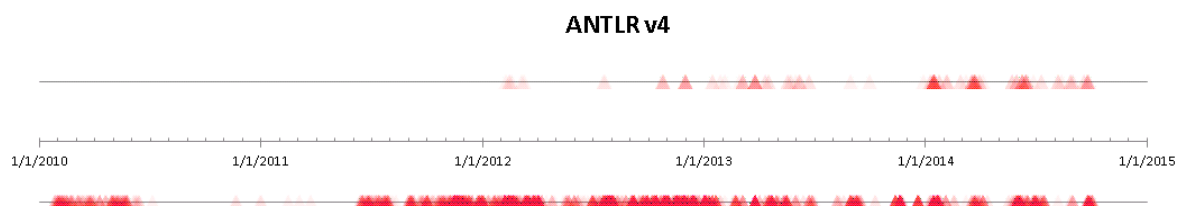
7. fejezet

Függelék

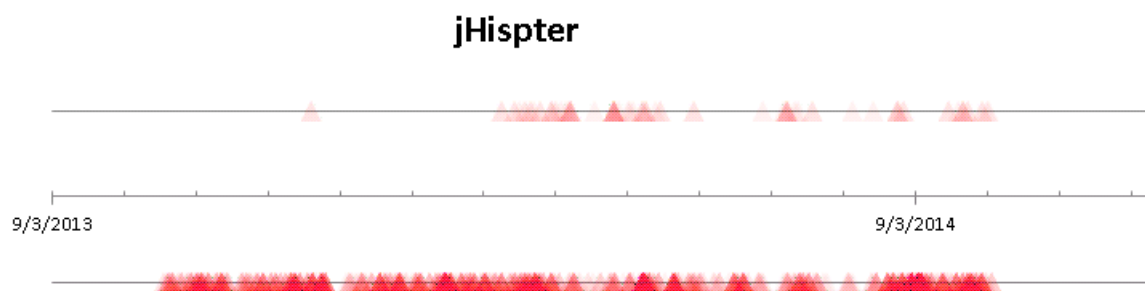
7.1. A hibajavításokban érintett commit-ok diagramjai



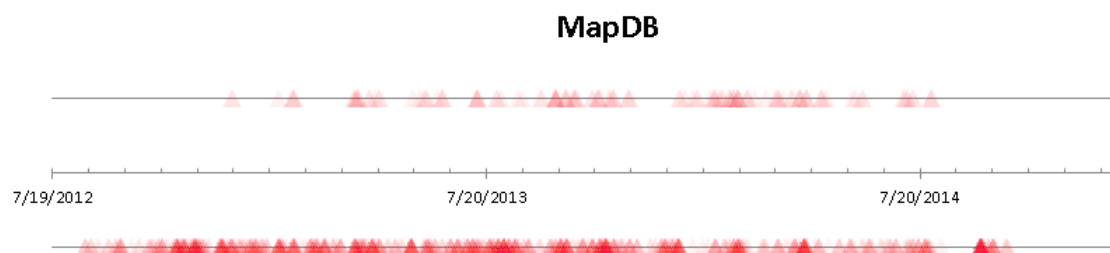
7.1. ábra. Universal Image Loader



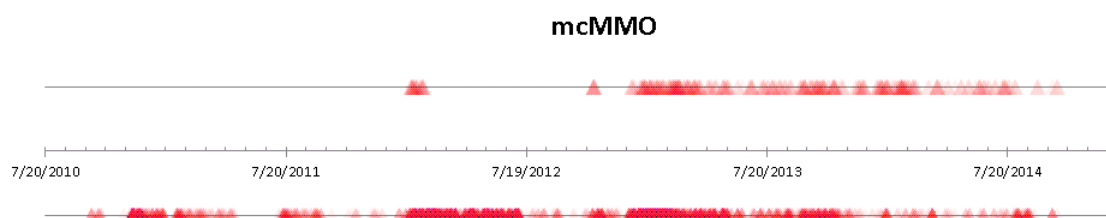
7.2. ábra. ANTLR v4



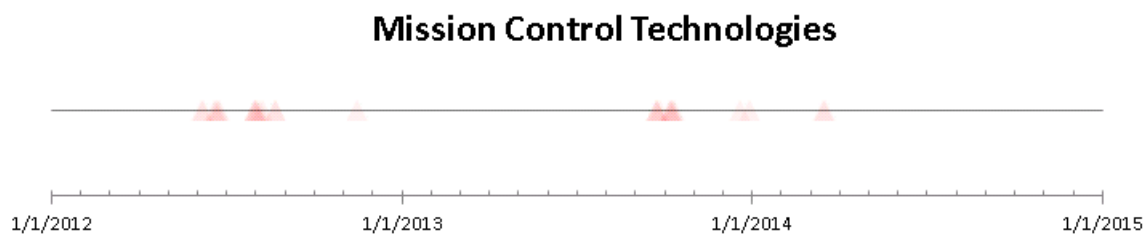
7.3. ábra. jHispter



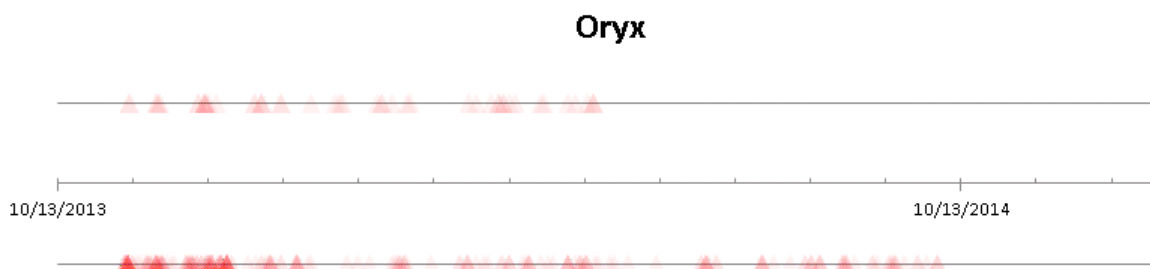
7.4. ábra. MapDB



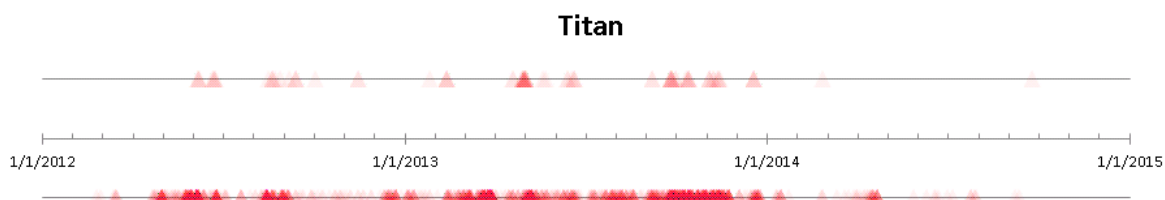
7.5. ábra. mcMMO



7.6. ábra. Mission Control Technologies

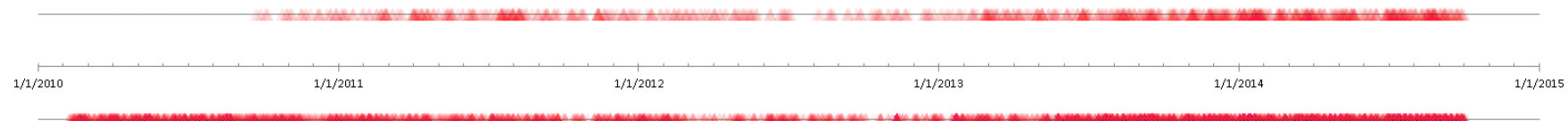


7.7. ábra. Oryx



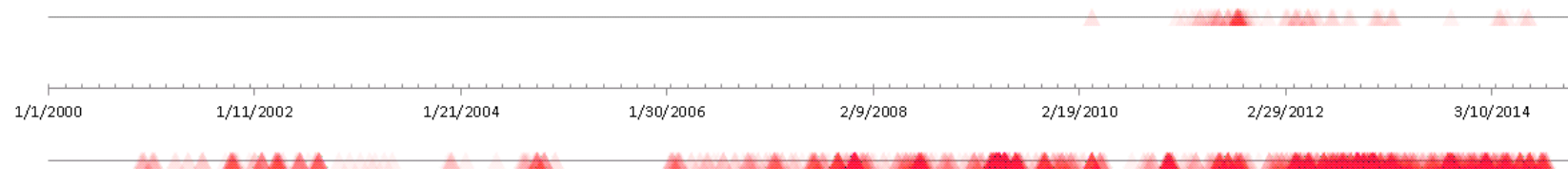
7.8. ábra. Titan

Elasticsearch



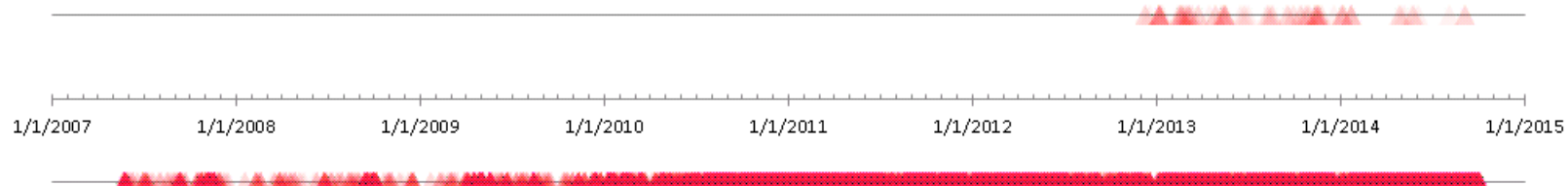
7.9. ábra. Elasticsearch

JUnit



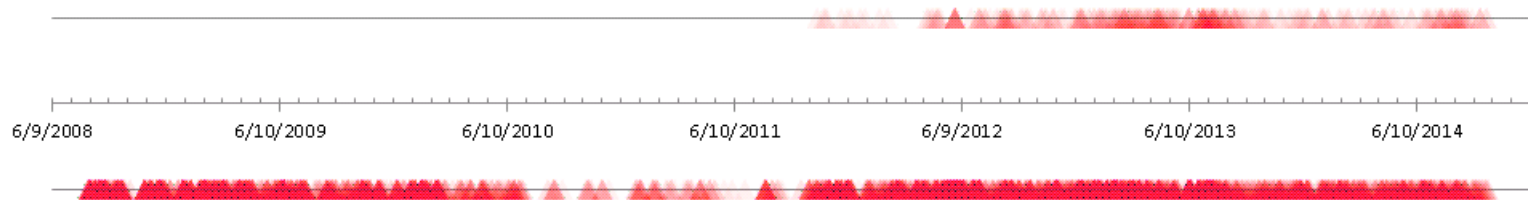
7.10. ábra. JUnit

Neo4j

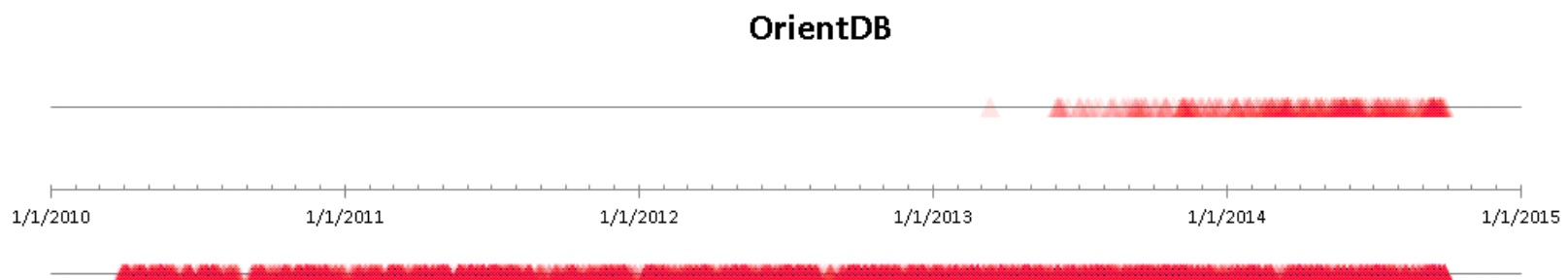


7.11. ábra. Neo4j

Netty



7.12. ábra. Netty



7.13. ábra. OrientDB

Irodalomjegyzék

- [1] T. Bakota, P. Hegedus, P. Kortvelyesi, R. Ferenc, and T. Gyimothy. A probabilistic software quality model. In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, pages 243–252, sept. 2011.
- [2] P. Bangcharoensap, A. Ihara, Y. Kamei, and K. Matsumoto. Locating source code to be fixed based on initial bug reports - a case study on the eclipse project. In *Empirical Software Engineering in Practice (IWESEP), 2012 Fourth International Workshop on*, pages 10–15, Oct 2012.
- [3] C. Bird, P.C. Rigby, E.T. Barr, D.J. Hamilton, D.M. German, and P. Devanbu. The promises and perils of mining git. In *Mining Software Repositories, 2009. MSR '09. 6th IEEE International Working Conference on*, pages 1–10, May 2009.
- [4] Shyam R Chidamber and Chris F Kemerer. A metrics suite for object oriented design. *Software Engineering, IEEE Transactions on*, 20(6):476–493, 1994.
- [5] C. Couto, C. Silva, M.T. Valente, R. Bigonha, and N. Anquetil. Uncovering causal relationships between software metrics and bugs. In *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*, pages 223–232, March 2012.
- [6] Valentin Dallmeier and Thomas Zimmermann. Automatic extraction of bug localization benchmarks from history. Technical report, Universitat des Saarlandes and Saarbrücken and Germany, 2007.
- [7] Valentin Dallmeier and Thomas Zimmermann. Extraction of bug localization benchmarks from history. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 433–436. ACM, 2007.
- [8] Marco D’Ambros, Michele Lanza, and Romain Robbes. An extensive comparison of bug prediction approaches. *Proceedings of MSR 2010 (7th IEEE Working Conference on Mining Software Repositories)*, pages 31 – 41, 2010.
- [9] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. The promises and perils of mining github. *MSR 2014 Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 92–101, 2014.

- [10] Tim Menzies, Bora Caglayan, Zhimin He, Ekrem Kocaguneli, Joe Krall, Fayola Peters, and Burak Turhan. The promise repository of empirical software engineering data, June 2012.
- [11] Deqing Wang, Mengxiang Lin, Hui Zhang, and Hongping Hu. Detect related bugs from source code using bug information. *Computer Software and Applications Conference (COMPSAC)*, 2010.
- [12] Toth Z., Novak G., Ferenc R., and Siket I. Using version control history to follow the changes of source code elements. *Software Maintenance and Reengineering (CSMR)*, 2013.
- [13] Jian Zhou, Hongyu Zhang, and David Lo. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. *Software Engineering (ICSE), 2012 34th International Conference on*, 2012.