Statička analiza koda na primjeru složene poslovne aplikacije

Solić, Matija

Master's thesis / Diplomski rad

2016

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: Josip Juraj Strossmayer University of Osijek, Faculty of Electrical Engineering, Computer Science and Information Technology Osijek / Sveučilište Josipa Jurja Strossmayera u Osijeku, Fakultet elektrotehnike, računarstva i informacijskih tehnologija Osijek

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:200:442379

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: 2024-01-19

Repository / Repozitorij:

Faculty of Electrical Engineering, Computer Science and Information Technology Osijek
STATIČKA ANALIZA KODA NA PRIMJERU SLOŽENE POSLOVNE APLIKACIJE

Diplomski rad

Matija Solić

Osijek, 2016.
SADRŽAJ

1. Uvod ........................................................................................................................................ 1

2. Uvod u statičku analizu kôda ................................................................................................ 2
   2.1. Potreba za statičkom analizom kôda ............................................................................... 2
   2.1.1. Važnost statičke analize ............................................................................................ 3
   2.2. Prednosti i nedostaci statičke analize ............................................................................... 5
   2.3. Podjela statičke analize ..................................................................................................... 6
   2.3.1. Podjela prema tipu kôda ............................................................................................ 6
   2.4. Vrste statičke analize ........................................................................................................ 8
   2.4.1. Provjera tipa podataka .............................................................................................. 9
   2.5. Mjerne jedinice statičke analize ..................................................................................... 13
   2.5.1. Mjerne jedinice složenosti kôda .............................................................................. 13

3. Teorija statičke analize ........................................................................................................ 16
   3.1. Programski modeli .......................................................................................................... 16
   3.1.1. Leksička analiza ...................................................................................................... 16
   3.1.2. Raščlanjivanje (parsing) .......................................................................................... 17
   3.1.3. Apstraktna sintaksa ................................................................................................. 18
   3.1.4. Semantička analiza .................................................................................................. 18
   3.1.5. Analiza toka programa ............................................................................................ 19
   3.2. Algoritmi za provođenje statičke analize ....................................................................... 22
   3.2.1. Apstraktna interpretacija ......................................................................................... 22
   3.2.2. Transformacija suda (predikatna transformacija) ................................................... 22
   3.2.3. Provjera modela sustava .......................................................................................... 23
   3.3. Pravila koja koriste alati za statičku analizu ................................................................... 23
   3.4. Prikaz rezultata analize ................................................................................................... 24

4. Opis alata za statičku analizu programa ........................................................................... 27
   4.1. Alat FindBugs ................................................................................................................. 27
   4.2. Alat PMD ........................................................................................................................ 29
   4.3. Alat Jenkins .................................................................................................................... 29
   4.4. Alat SonarQube .............................................................................................................. 31

5. Provоđenje testiranja na kodу ............................................................................................ 33
   5.1. Podešavanje alata za rad s Eclipse-om ........................................................................... 33
   5.2. Podešavanje Jenksinsa ................................................................................................... 38
5.3. Podešavanje SonarQube-a ........................................................................................................ 43

6. Analiza dobivenih rezultata ........................................................................................................ 45
   6.1. Analiza korištenja statičke analize u različitim dijelovima razvoja projekta .............. 47
       6.1.1. Perspektiva programera .................................................................................. 47
       6.1.2. Perspektiva voditelja projekta ........................................................................ 49
       6.1.3. Perspektiva vodstva ....................................................................................... 53
   6.2. Detaljna analiza projekta ............................................................................................... 56
       6.2.1. SonarQube analiza pomoću FindBugs pravila ............................................. 57
       6.2.2. PMD analiza unutar Eclipse razvojnog okruženja ....................................... 62
   6.3. Ispravljanje pogrešaka ................................................................................................. 62

7. Zaključak ............................................................................................................................... 64

Literatura ...................................................................................................................................... 65
Sažetak ......................................................................................................................................... 67
Abstract ...................................................................................................................................... 67
Životopis ..................................................................................................................................... 68
Prilozi .......................................................................................................................................... 69
   P.6.1. Popis pronadenih pogrešaka pomoću SonarQube pravila .................................... 69
   P.6.2. Popis pronadenih pogrešaka pomoću PMD pravila .............................................. 69
1. UVOD

Cilj diplomskog rada je proučiti i opisati kako se statička analiza programskog koda može iskoristiti za poboljšanje njegove kvalitete, koje su metrike pri tome najznačajnije, te kako ih pratiti. Diplomski rad izrađen je uz pomoć tvrke Siemens Convergence Creators koja je ustupila programski kod za analizu.

Statička analiza kôda predstavlja proces provjere kôda bez njegovog izvršavanja. U radu će se moći vidjeti da različiti alati koriste različita načela pri izradi statičke analize te da se dobrim odabirom alata i automatizacijom njegove upotrebe proces statičke analize može učiniti jednostavnim i iznimno korisnim. Proces statičke analize može se integrirati i automatizirati unutar samog razvojnog okruženja (Eclipse), unutar produkcijskog (integracijskog) okruženja (Jenkins) ili se za to može koristiti specijalizirani alat koji će rezultate analize prikazati na najbolji način (SonarQube). U svakom slučaju, najbitnije je da se rezultati statičke analize i ključne metrike kvalitete programa stalno prate te da se pravovremeno poduzimaju mjere poboljšanja kvalitete kôda (ispravljanje pogrešaka, uvođenje smjernica pisanja dobrog kôda, refaktoriranje i sl.). Da bi to bilo tako, u taj proces moraju biti uključeni svi: od programera i arhitekata, osoba zaduženih za testiranje pa sve do osoba zaduženih za upravljanje projektom ili životnim ciklusom programskog proizvoda.


Ovim putem želio bih zahvaliti ljudima iz tvrke Siemens Convergence Creators na uloženom trudu i vremenu.
2. UVOD U STATIČKU ANALIZU KÔDA


U diplomskom radu biti će objašnjeno što je statička analiza i koja je njena važnost u softverskoj industriji danas, koje vrste analize postoje, koji su alati najkorišteniji i kako statička analiza kôda izgleda u praksi.

2.1. Potreba za statičkom analizom kôda

Prema [1], u '70. godinama prošlog stoljeća programeri su došli do zaključka, da za povećanje kvalitete programa, moraju uvesti neku metodu provjere kôda dok je projekt još u razvoju. Rezultat toga su službeni pregledi i inspekcije kôda u kojima skupina ljudi pregledava kod, liniju po liniju, te analizira moguće propuste i probleme. U početku, izvorni kod je bio mali te su programeri, koji provode analizu, lakše i jednostavnije otkrivali pogreške. No, s godinama, veličina izvornog kôda se povećala te je programerima postalo sve teže otkriti moguće propuste i pogreške. Postoje razni razlozi zbog kojih sposoban programer može previdjeti pogrešku. Primjer razloga može biti umor. Programer satima prolazi kroz tisuće linija kôda i može se dogoditi da jednostavno previdi neki propust. Drugi razlog može biti fokusiranje na određeni dio kôda, pri čemu se zapostavlja neki drugi, manje bitni, dio kôda. Programeri najčešće ključnom dijelu kôda posvećuju više pozornosti i pažnje te posebno paze da taj kod radi ispravno pri čemu, nesvjesno, čine propuste u manje bitnim dijelovima programa.

Kako je problem otkrivanja pogrešaka postao sve veći i teži za riješiti, programeri su odlučili upotrijebiti računalo za pronalaženje eventualnih pogrešaka. Računalo može obraditi veliki broj

2.1.1. Važnost statističke analize

Za razumijevanje važnosti statističke analize potrebno je promotriti proces izrade programa. Proces izrade programa prikazan je dijagramom na Slika 2.1 koja je preuzeta iz [2].

Slika 2.1. Ciklus razvoja programa

njihovo otklanjanje. Za svaki otkriveni problem u koraku održavanja ciklus razvoja programa se ponavlja iz početka.

Važnost statičke analize lakše je shvatiti ukoliko se uzme u obzir cijena otklanjanja pogrešaka u različitim stadijima razvoja programa. Slika 2.2 preuzeta iz [3], prikazuje ovisnost troškova otklanjanja pogreške rastu o fazi u kojoj se projekt nalazi. Iz slike je vidljivo da troškovi u fazi planiranja (definiranja zahtjeva), analiziranja i dizajniranja te implementiranja minimalno rastu, dok troškovi nakon te točke eksponencijalno rastu.

Idealno, želja je pogreške ne raditi ili ih ispraviti već u fazi planiranja, no to nije moguće. Ono što je moguće je otkriti pogreške u fazi implementacije. Prema [1], statička analiza omogućuje otkrivanje pogrešaka već u ranoj fazi implementacije. Ona ne ovisi o ostatku kôda te se može izvoditi od samog početka pisanja kôda dok je za dinamičku analizu potrebno imati gotov modul kôda, što znači da je upotrebom statičke analize moguće ranije otkrivanje pogrešaka. Također, moguće je otkriti pogreške u dizajnu koje su skupe za otklanjanje u kasnijim fazama jer programski moduli često ovise jedni o drugima. Statička analiza ima svoje prednosti i nedostatke koje su detaljnije opisane u potpoglavlju 2.2. Statička analiza nije zamjena za dinamičku analizu niti je zamjena za službene preglede kôda od strane programera. Ona je najučinkovitija kada se koristi u kombinaciji s ostalim alatima. Izvor [3] tvrdi da statička i dinamička analiza zasebno
mogu otkriti do 85% grešaka u kodu. Korištenjem oba pristupa istovremeno korisnost se povećava do 95%. Povećanje od 10% možda ne djeluje značajno no ukoliko se u obzir uzme cijena otklanjanja pogreške u kasnijim fazama, zaključak je da se povećenje od 10% isplati jer osigurava veliku uštedu resursa.

2.2. Prednosti i nedostaci statičke analize


Prednosti statičke analize su:

- Uključuje sve moguće skupove ulaznih podataka. Statička analiza omogućava otkrivanje problema koji se ne pojavljuju pri dinamičkom testiranju jer provjerava sve moguće načine izvršavanja programa te otkriva moguće probleme.
- Prema [4], moguće je odrediti uzrok problema, a ne samo simptome. Primjer ovog je otkrivanje uzroka prelijevanja spremnika (engl. buffer overflow). Dinamička analiza može otkriti da se preljev spremnika dogodio ali ne može precizno utvrditi razlog. Ova činjenica je važna, jer omogućuje provjeru je li pogreška u potpunosti uklonjena ili je samo otklonjen jedan slučaj koji dovodi do pogreške.
- Prema [4], pogodnija je za otkrivanje sigurnosnih propusta. Prethodno spomenute činjenice omogućavaju alatu za statičku analizu da provjeri cijeli kod, prođe kroz sve moguće iteracije izvođenja te zaključi o mogućim sigurnosnim problemima. Dinamički alati provjeravaju specifične situacije te mogu predvidjeti problem, koji kasnije može prouzročiti velike probleme. Dodatan razlog zašto su alati za statičku analizu pogodniji za pronalaženje sigurnosnih propusta je njihova nadograđivost. Ukoliko istraživači
pronadu novi propust sve što moraju je napisati pravilo koje može otkriti problem. Novo nastalo pravilo moguće je primijeniti na bilo kojem kodu, dok bi se kod dinamičke analize morali pisati dodatni specifični testovi.


Nedostaci statičke analize:


2.3. **Podjela statičke analize**

Alate za statičku analizu kôda moguće je podijeliti prema tipu kôda kojeg analiziraju te prema vrsti problema kojeg mogu otkriti. Podjela prema tipu opisana je u nastavku poglavlja dok je podjela prema problemima opisana u potpoglavlju 2.4.

2.3.1. **Podjela prema tipu kôda**

kôda odstranjene su sve informacije koje nisu nužne. Osim odstranjivanja suvišnih informacija strojni kod može sadržavati drugačiju implementaciju napisanih funkcija. U nastavku su opisane specifičnosti navedenih analiza.

**Analiza izvornog kôda**


Primjer alata koji rade analizu nad izvornim kodom su PMD, CheckStyle, alati koji provjeravaju sintaksu, pravopis te stil.

**Analiza strojnog kôda**

Prema [4], analiza strojnog kôda podrazumijeva analiziranje kôda koji se izvršava pokretanjem programa. Provodenje analize na samom strojnom kodu je teško jer strojni kod sadrži samo naredbe koje procesor može izvršiti. Zbog tog razloga analiza se obavlja nad apstraktnim
podacima koji opisuju strojni kod. Proces apstrakcije podataka opisan je u potpoglavlju 3.1.3, za sada je nužno spomenuti da se pri apstrakciji stvara apstraktno sintaksno stablo (AST) te tablica simbola.

Prednost alata koji rade takvu analizu zasniva se na činjenici da analizu rade nad kodom koji je spreman za izvođenje na prevedenoj platformi. Ova činjenica omogućuje pronalaženje pogrešaka koje su se dogodile pri prevođenju kôda. Također, analizom strojnog kôda provjera se jedan slijed izvođenja koji je određen prevođenjem programa, dok se pri analizi izvornog kôda provjeravaju svi mogući sljedovi izvođenja. Zahvaljujući ovom smanjen je broj lažno-požitivnih izvještaja. Budući da strojni kod u sebi sadrži naredbe iz uključenih biblioteka i drugih programa, pomoću analize strojnog kôda moguće je provjeriti da li biblioteke odnosno uveženi programi imaju pogreške. Nadalje, za razliku od analize izvornog kôda, pri analizi strojnog kôda nije potrebno znati specifičnosti svih jezika koje program sadrži jer je kod već preveden i objedinen u strojnom kodu.

Prema [4], problemi analize strojnog kôda proizlaze iz činjenice da se analiza odvija nad apstraktnim podacima. Rezultat analize ovisi o kvaliteti analize, odnosno odnosi o točnosti AST-a i tablice simbola. Čest je slučaj da alat, pri izradi AST-a i tablice simbola, nema dovoljno podataka pa mora aproksimirati stanja. Zbog toga, analiza može predvidjeti pogreške ili prijaviti pogrešku koja nije pogreška. Poželjno je koristiti obje analize kako bi se uklonile lažne prijave.

Primjer alata koji koristi strojnu analizu kôda je FindBugs. Zbog navedenih prednosti, alati koji koriste strojnu analizu pogodni su za analiziranje sigurnosnih propusta.

2.4. Vrste statičke analize

Alate za statičku analizu može se podijeliti u slijedeće skupine:

- Provjera tipa podataka
- Provjera stila
- Razumijevanje programa
- Potvrdivanje ispravnosti programa (program verification)
- Provjera svojstava
- Pronalaženje pogrešaka
- Sigurnosna provjera
2.4.1. Provjera tipa podataka

Prema [4], provjera tipa podataka je jedan od najkorištenijih oblika statičke analize. Iako je jedan od najkorištenih alata programeri rijetko razmišljaju o njemu. Razlog tome je što prevoditelj radi provjeru tipa podataka prema standardima zadanog jezika. Otkrivanje ovih pogrešaka spriječava pojavljivanje pogrešaka prilikom izvođenja. Provjera tipa podataka eliminira cijeli niz problema koji se mogu pojaviti. Prema [4], primjer problema kojeg je moguće pronaći je pridruživanje podatku tipa „int“ podatak tipa „double“.

Provjera tipa podatka ima ograničenu mogućnost pronalaženja problema te može prijaviti lažno-pozitivne rezultate. Primjer 2.1, preuzet iz [4], prikazuje Java kod za koji je prijavljen lažno-pozitivan rezultat.

```
short s = 0;
int i = s; /* Provjera podatka ovo dozvoljava */
short r = i; /* Provjera podatka prijavljuje pogrešku */
```

**Primjer 2.1. Provjera tipa podatka s prijavljenom pogreškom**

Prevoditelj ne dozvoljava prevođenje Java kôda iz Primjer 2.1, jer kod krši pravilo o pridruživanju izraza tipa int short tipu, iako je programerova namjera nedvosmislena.

Osim lažno pozitivnih prijava mogu se dogoditi i lažno negativne prijave. Primjer 2.2, preuzet iz [4], prikazuje Java kod kojeg je prevoditelj preveo bez prijavljanja pogreške iako ona postoji.

```
Object[] objs = new String[1];
objs[0] = new Object();
```

**Primjer 2.2. Provjera tipa podatka bez prijavljene pogreške**

Kod prikazan Primjer 2.2 sadrži pogrešku pridruživanja tipa podatka. Prevoditelj ne otkriva pogrešku jer, iz njegove perspektive, programer pridružuje polju tipa Object objekt tipa Object. Ono što prevoditelj ne provjerava je da se pridruživanjem String objekta tip polja promijenio u String. Ovo je moguće, jer klasa String nasljeđuje klasu Object.

**Provjera stila**

Alati za provjeru stila uobičajeno koriste manji broj jednostavnijih pravila koja su površnija od pravila za provjeru tipa. Prema [4], alati koji omogućuju samo provjeru stila posjeduju pravila vezana uz razmake, konvenciju o imenovanju, popis nevažećih (engl. deprecated) funkcija, pravila o komentiranju, strukturi programa i sl. Kako većina programera ima vlastiti stil pisanja programa, dobar alat za provjeru stila mora biti podesiv kako bi ga programer prilagodio svojim
potrebama. Pogreške koje alati za provjeru stila prijavljaju većinom su vezani uz čitljivost i sposobnost održavanja kôda. Alati za provjeru stila ne prijavljaju pogreške koje su vezane za izvođenje programa.

Primjer 2.3, preuzet iz [4], prikazuje C kod koji ne uzima u obzir sve vrijednosti tipa `Color`. Prevoditelj će pri prevodjenju prijaviti pogrešku da vrijednosti “green” i “blue” nisu uzete u obzir.

```c
typedef enum { red, green, blue } Color;
char* getColorString(Color c) {
    char* ret = NULL;
    switch (c) {
        case red:
            printf("red");
            return ret;
    }
}
```

**Primjer 2.3. Provjera stila pisanja**

Budući da se stilovi pisanja kôda često razlikuju od osobe do osobe, provjeru stila teško je primijeniti na gotovom ili započetom projektu. Ispрављати pogrešке на већ započетом projektu je vremenski i financijski skup posao koji ne donosi značajne prednosti. Za optimalnu prednost provjeru stila potrebno je koristiti od početka izrade projekta. Primjer alata za provjeru stila su PMD te Checkstyle.

**Razumijevanje programa**


Primjer alata koji ima ovu funkcionalnost je Fujaba. Fujaba je alat otvorenog izvornog kôda koji omogućava programeru prebacivanje UML dijagrama u Java izvorni kod i obrnuto.
**Potvrđivanje ispravnosti programa i provjera svojstava**

Prema [4], potvrđivanje ispravnosti programa pokušava dokazati da je kod vjerodostojna implementacija specifikacije. Ukoliko specifikacija pruža potpun opis što program treba napraviti, alat koji vrši potvrđivanje ispravnosti programa može izvršiti provjeru jednakosti (engl. equivalence checking) kako bi utvrdio da se kod i specifikacija u potpunosti podrudaraju. Provjera jednakosti često se koristi pri provjeri sklopljenja dok se rijetko koristi pri provjeri programa. Razlog tome je što programerij rijetko posjeduju specifikacije koje su dovoljno detaljne da bi mogle biti korištene za provjeru jednakosti, a pisanje dodatnih specifikacija često zahtjeva više vremena nego što je potrebno za kodiranje. Dodatan problem predstavlja činjenica da alati za potvrđivanje jednakosti nisu u mogućnosti obraditi kod veće veličine.

Za savladavanje spomenutih problema razvijene su metode za potvrđivanje ispravnosti prema djelomičnoj specifikaciji koja opisuje samo dio ponašanja programa. Ova metoda potvrđivanja naziva se provjera svojstva. Većina alata za provjeru svojstva rade na načelu provjere modela ili primjenjujući logične zaključke (engl. applying logical inference). Većina alata usredotočena su na pronalaženje privreizbornih sigurnosnih svojstva. Privremena sigurnosna svojstva određuju sljed događaja koji program ne smije izvršiti. Primjer jednog takvog događaja je da se memorijska adresa ne bi trebala čitati nakon što se isprazni. Najčešće alati za provjeru svojstava imaju mogućnost pisanja vlastitih specifikacija kako bi programerij mogli provjeriti specifična svojstva za svoj program.


**Pronalaženje pogrešaka**

Prema [4], cilj alata za pronalaženje pogrešaka je otkrivanje dijelova kôda koji se ne ponašaju onako kako je programer zamislio. Većina alata za pronalaženje pogrešaka su jednostavna za korištenje jer u sebi sadržavaju skup pravila. Pravila upisuju obrasce pogrešaka koje se mogu pojaviti u kodu. Napredniji alati mogu proširiti svoju bazu pravila tako da izvedu zaključke o
zahtjevima iz samog kôda. Izvor [4], kao primjer navodi Java program u kojem postoji sinkronizacijska brava (engl. lock) koja onemogućava pristup korištenoj varijabli. Alat za pronalaženje pogrešaka otkriva da sinkronizacijska brava postoji na 99 od 100 mjesta gdje se varijabla koristi te zaključuje da se vjerojatno i na 100. mjestu treba koristiti.


Primjer alata za otklanjanje pogrešaka je FindBugs.

**Sigurnosna provjera**

Prema [4], sigurnosna provjera zasniva se na istim algoritmima i tehnikama koje koriste ostali alati. Zbog određenosti i specifičnosti zadatka, algoritmi i tehnike primjenjuju se drugačije nego kod ostalih alata.

Prvi alati za sigurnosnu provjeru radili su kao tražilice, prolazili su kroz kod i provjeravali da li postoje pozivi funkcije za koje je poznato da se mogu zloupotrebljavati. Primjer takve funkcije je strcpy(). Prema ponašanju i funkcionalnosti, prve alate za sigurnosnu provjeru, moguće je usporediti sa alatima za provjeru stila. Problemi koje su izvještavali programeru u nekim slučajevima nisu bili stvarni sigurnosni propusti, ali su predstavljali razlog za povećanu brigu o sigurnosti programa.

negativan izvještaj. S ovim na umu, alati za sigurnosnu provjeru služe kao pokazivač na kod kojeg je potrebno detaljnije proučiti pri pregledavanju kôda od strane programera.

2.5. **Mjerne jedinice statičke analize**

Programske mjerne jedinice su kvantitativna mjerila koliko računalni sustav ili proces posjeduje neko svojstvo. Koriste se za određivanje troškova razvoja projekta, određivanje toka razvoja te provjeru kvalitete kôda. Za statičku analizu važne su mjerne jedinice za provjeru kvalitete kôda. Pomoću njih programeri odlučuju o prioritetu popravke kôda, raspodjeli resursa i sl.

Mjerne jedinice za provjeru kvalitete kôda moguće je podijeliti u dvije skupine:

- Mjerne jedinice složenosti kôda
- Mjerne jedinice veličine kôda


2.5.1. **Mjerne jedinice složenosti kôda**

Mjere složenosti programa ključan su dio programskih mjernih jedinica. Mjere složenosti programa pokazatelj su kvalitete izvornog kôda. Prema [6] složenost kôda se može podijeliti u tri skupine:

- Osnovna složenost
- Odabrana složenost (engl. selecting complexity)
- Slučajna složenost (engl. Incidental complexity)

McCabe ciklomatska složenost

Prema [6], ciklomatska složenost (engl. Cyclomatic Complexity Measures - CCM) smatra se najosnovnijom i najvažnijom mjernom jedinicom. Većina ostalih mjernih jedinica izvedene su pomoću ciklomatske složenosti. CCM predstavlja mjeru složenosti odluka strukturnog grafa. CCM se računa pomoću grafa kontrole toka. Broj CCM-a predstavlja broj nezavisnih tokova izvođenja programa.

Slika 2.3 prikazuje grafi kontrole toka za pseudo kod. Čvorovi grafa predstavljaju sekvencijalne blokove kôda dok strelice pokazuju smjer kretanja programa. Pomoću ovog grafa kontrole toka moguće je izračunati CCM. CCM se računa kao razlika u broju usmjerenih strelica i broja čvorova te se razlici doda dva. CCM za navedeni primjer iznosi tri.

Slika 2.3. Graf kontrole toka

Prednost CCM-a je što vjerno prikazuje vezu između gustoće pogrešaka i složenosti kôda. Poželjno je da CCM bude manji od 10 te ne veći od 20. Ukoliko je CCM blizu 100, program je toliko kompliciran da je vjerojatnost 60% da će pri ispravku pogreške nastati nova pogreška.

Nedostaci CCM-a proizlaze iz činjenice da CCM zanemaruje tok podataka u programu. CCM će za potpuno sekvencijalni kod koji se sastoji od više linija izračunati složenost jednaku jednoj liniji kôda. Drugi nedostatak je što CCM ne uzima u obzir složenost ugniježđenih kodova, čija je složenost veća. Nadalje, CCM sve vrste kontrole toka boduje jednako, ne raspoznaje razliku između „if“ i „case“ uvjeta.
**Halsteadova mjera složenosti**

Halsteadova mjera složenosti (engl. Halstead Complexity Measures - HCM) koristi znanstvene metode kako bi analizirala značajke i strukturu programa. HCM se računa na temelju broja operatora i operandi. Operatori su simboli unutar izraza koji označavaju radnju nad operandima. Operandi su osnovne logičke jedinice. HCM mjeri logičku veličinu programa.

Pomoću HCM-a moguće je izračunati slijedeće veličine:

- Očekivanu veličinu programa
- Veličinu programa
- Veličinu rječnika
- Obujam
- Težinu
- Potreban trud za pisanje kôda
- Procjena broja pogrešaka
- Potrebno vrijeme za pisanje kôda

3. TEORIJA STATIČKE ANALIZE


3.1. Programske modeli

Prvi korak pri izradi analize je pretvorba kôda u programske model, u strukturu podataka koja predstavlja kod. Programske model ovisi o vrsti analize koja se provodi, ali općenito govoreći, alati za statičku analizu se ponašaju kao prevoditelji.

3.1.1. Leksička analiza

Prvi korak pri pretvorbi izvornog kôda je stvaranje niza oznaka (engl. Token) pri čemu se uklanjaju manje bitne značajke programa kao što su razmaci i komentari. Stvaranje ovog niza oznaka naziva se leksička analiza. Pravila koja čine leksičnu analizu koriste regularne izraze kako bi otkrili oznake.

Primjer 3.1, preuzet iz [4], prikazuje rezultat leksičke analize nad C kodom. Iz navedenog primjera moguće je zamijetiti da je većina oznaka u potpunosti određena samo sa tipom oznaka dok je ID oznaka potreban dodatan podatak – ime identifikatora. Kako bi kasnije dobili korisne povratne informacije oznake trebaju posjedovati barem još jednu informaciju, informaciju o lokaciji unutar izvornog kôda.

```
C kod:
if (ret) // probably true
mat[x][y] = END_VAL;
```

Kod nakon leksičke analize:
```
IF LPAREN ID(ret) RPAREN ID(mat) LBRACKET ID(x) RBRACKET LBRACKET ID(y) RBRACKET EQUAL ID(END_VAL) SEMI
```

**Primjer 3.1. Leksička analiza C koda**
3.1.2. Raščlanjivanje (parsing)


| stmt := if_stmt | assign_stmt |
| if_stmt := IF LPAREN expr RPAREN stmt |
| expr := lval |
| assign_stmt := lval EQUAL expr SEMI |
| lval = ID | arr_access |
| arr_access := ID arr_index+ |
| arr_idx := LBRACKET expr RBRACKET |

Primjer 3.2. Raščlanjivanje oznaka

Parser provodi raščlanjivanje tako što uspoređuje oznake iz niza s produkcijskim pravilima. Ukoliko je svaki simbol vezan sa simbolom iz kojeg je izveden moguće je napraviti stablo raščlanjivanja.


![Slika 3.1. Stablo raščlanjivanja](image-url)
3.1.3. Apstraktna sintaksa


Slika 3.2. Apstraktno sintaksno stablo

Ukoliko AST usporedimo sa stablom parsiranja, prikazano na Slika 3.1, moguće je primijetiti razlike. Vidljivo je da je AST jednostavniji od stabla parsiranja jer posjeduje manje grana. Iako jednostavniji, AST posjeduje grane koje se ne nalaze u kodu - praznu "else" granu (NO_OP). Također, uvjet "if" uspoređuje se izričito s nulom.


3.1.4. Semantička analiza

Kako bi alat mogao vršiti semantičku analizu nad kodom mora posjedovati AST i tablicu simbola. Tablica simbola sadrži popis svih identifikatora u programu. Tablica za svaki
identifikator ima spremljen tip i pokazivač na njegovu deklaraciju. Tablica simbola pravi se usporedno s AST stablom.

Semantička analiza je bitna za objektno orijentirano programiranje jer oznaka tipa objekta označava koje sve metode objekt može pozvati. Pomoću semantičke analize alat može utvrditi preopterećivanje operatora u C-u ili implementaciju neke klase u Javi. Također, ova značajka omogućuje provjeru strukture programa.

3.1.5. Analiza toka programa


**Graf kontrole toka**

Graf kontrole toka čine čvorovi. Čvorovi predstavljaju niz instrukcija koji će se uvijek izvršiti, počevši od prve instrukcije sve do posljednje. Strelice (engl. edges) u grafu kontrole toka su usmjereni i predstavljaju tok programa između čvorova. Strelice koje pokazuju u suprotnom smjeru predstavljaju moguću petlju. Primjer 3.3 preuzet iz [4], prikazuje jednostavan kod koji sadrži dvije grane. Kod je iskorišten za izradu grafa kontrole toka. Slika 3.3 prikazuje graf kontrole toka za Primjer 3.3.

```pseudo
if (a > b) {
    nConsec = 0;
} else {
    s1 = getHexChar(1);
    s2 = getHexChar(2);
} return nConsec;
```

**Primjer 3.3. Pseudo kod za graf kontrole toka**
Važno je napomenuti da ovaj graf, zbog čitljivosti i razumijevanja, koristi izvorni kod dok graf kontrole toka konstruira čvorove na temelju AST-a. Čvorovi imaju oznaku $bb0$ do $bb3$ i predstavljaju naredbe koje se moraju izvršiti. Sve strelice u grafu su usmjerene u istom smjeru i predstavljaju tok izvođenja programa. Iz grafa je vidljivo da ne postoje petlje jer su sve strelice okrenute u istom smjeru.

**Graf pozivanja (engl. call graph)**


```c
int larry(int fish) {
  if (fish) {
    moe(1);
  } else {
    curly();
  }
}
int moe(int scissors) {
  if (scissors) {
    curly();
    moe(0);
  } else {
```
Curly();
{
    int curly() {
        /* empty */
    }
}

Primjer 3.4. Pseudo kod za graf pozivanja

Slika 3.4. Graf pozivanja

Analiza toka podataka


Izvorni kod:
sum = sum + delta;
sum = sum & top;
y = y + (z<<4)+k[0] ^ z+sum ^ (z>>5)+k[1];
y = y & top;
z = z + (y<<4)+k[2] ^ y+sum ^ (y>>5)+k[3];
z = z & top;
SSA oblik:
sum2 = sum1 + delta1;
sum3 = sum2 & top1;
y2 = y1 + (z1<<4)+k[0]1 ^ z1+sum3 ^ (z1>>5)+k[1]1;
y3 = y2 & top1;
z2 = z1 + (y3<<4)+k[2]1 ^ y3+sum3 ^ (y3>>5)+k[3]1;
z3 = z2 & top1;

Primjer 3.5. Pretvorba izvornog kôda u statički jednoznačni oblik
3.2. **Algoritmi za provođenje statičke analize**


Prema [4], svaki algoritam se sastoji od dvije analize: intraproceduralne analize te interproceduralne analize. Intraproceduralna analiza analizira sve dijelove kôda pojedinačno dok interproceduralna analiza analizira interakciju između dijelova. Zbog sličnosti imena, u literaturi se češće koriste nazivi lokalna analiza za intraproceduralnu te globalna analiza za interproceduralnu analizu. Primjer lokalne analize je graf kontrole toka dok je primjer globalne analize graf pozivanja.

3.2.1. **Apstraktna interpretacija**


3.2.2. **Transformacija suda (predikatna transformacija)**

Sud u matematici, predstavlja tvrdnju koja mora nešto tvrditi, odnosno sud mora biti istinit ili lažan. Transformacija suda opisuje smisao programa kroz transformaciju krajnjeg stanja u početno stanje. Važan primjer funkcije koja rabi transformaciju suda je funkcija najslabijeg
početnog stanja. Najslabije početno stanje programa određuje se pronalaskom najmanjeg skupa uvjeta koji moraju biti prošlijeđeni od pozivatelja programa kako bi se došlo u krajnje stanje.

3.2.3. Provjera modela sustava


Slika 3.5, preuzeta iz [4], prikazuje rezultat pretvorbe kôda u automat s konačnim brojem stanja. Iz slike je vidljivo da je dozvoljeno samo jedno oslobađanje memorije. Ukoliko se iz stanja oslobodene memorije pokuša ponovo osloboditi memorija, prelazi se u stanje pogreške.

3.3. Pravila koja koriste alati za statičku analizu

Pravila, kao i spomenute tehnike, moguće je podijeliti u skupine. Uobičajeno je pravila klasificirati u skupine prema vrsti problema ili prema metodi koju koriste.

Generalizirajući, pravila prema vrsti problema mogu se podijeliti na:

- Pravila za provjeru sintakse
- Pravila za provjeru tipa podataka
- Pravila za provjeru sigurnosti

Podjelu pravila prema metodama nije potrebno ponovno navoditi jer su pravila spomenuta kao primjeri unutar opisa metoda.

3.4. Prikaz rezultata analize


Ukoliko alat važnost pridodaje broju problema a ne recimo složenosti, programer će većinu vremena provesti na analiziranju veće količine manjih problema te će kritične probleme zapostaviti. Za pretpostaviti je da veća skupina manjih problema predstavljaju sintaksne probleme, koje će programer zanemariti jer ne želi trošiti dragocjeno vrijeme preformulirajući kod. Naposljetku, zbog zatrpanosti upozorenjima, programer će početi zanemarivati sva upozorenja koje alat generira.

Kako bi se izbjegle ovakve situacije poželjno je da prikaz rezultata posjeduje ove značajke:

- Grupiranje i sortiranje rezultata
- Otklanjanje neželjenih rezultata
- Objašnjenje važnosti rezultata

Pronalaženje pogreške samo je dio posla koji alat za statičku analizu mora obaviti. Prema [4], pronađene pogreške su beskorisne ukoliko ne postoji dobar način da ih se prezentira. Rezultat je potrebno prikazati tako da se može ocijeniti kritičnost problema, te da može procijeniti kako popraviti taj problem. Također treba voditi računa o broju rezultata koje alat prikazuje. Ukoliko alat važnost pridodaje broju problema a ne recimo složenosti, programer će većinu vremena provesti na analiziranju veće količine manjih problema te će kritične probleme zapostaviti. Za pretpostaviti je da veća skupina manjih problema predstavljaju sintaksne probleme, koje će
programer zanemariti jer ne želi trošiti dragocjeno vrijeme preformulirajući kod. Naposljetku, zbog zatranosti upozorenjima, programer će početi zanemarivati sva upozorenja koje alat generira.

Kako bi se izbjegle ovakve situacije poželjno je da prikaz rezultata posjeduje ove značajke:

- Grupiranje i sortiranje rezultata
- Otklanjanje neželjenih rezultata
- Objašnjenje važnosti rezultata

**Grupiranje i sortiranje rezultata**


**Otklanjanje neželjenih rezultata**


**Objašnjenje važnosti rezultata**

Dobar opis pogreške sadrži opis problema, opis utjecaja i važnosti problema te korake kako se problem može dogoditi. Opis problema mora biti detaljan i jasan. Poželjno je da opis problema
4. OPIS ALATA ZA STATIČKU ANALIZU PROGRAMA


4.1. Alat FindBugs


FindBugs omogućuje pronalaženje pogrešaka u Java programima. Analiza se provodi nad strojnim kodom. S tim na umu, FindBugs posjeduje sve prednosti i nedostatke alata koji analizu obavljaju nad strojnim kodom. Više o ovome napisano je u potpoglavlju 2.2. Temeljno načelo FindBugsa je pronalažak obrazaca pogrešaka, a ne samo pogrešku. Stoga je FindBugs opremljen za pronalaženje jednostavnih pogrešaka koje se ponavljaju kao što su obrasci otkrivanja referenciranja NULL pokazivača, otkrivanje beskonačnih petlji, ne zatvaranje streamova itd.

Općenito govoreći FindBugs obrasce pogrešaka dijelju u slijedeće kategorije:

- Loša praksa
- Točnost kôda
- Sigurnosni propusti
- Višenitna korektnost kôda
- Brzina izvođenja
Popis svih obrazaca pogrešaka te opis pogreške moguće je pročitati u izvoru [7]. Također, FindBugs je moguće proširiti vlastitim pravilima, te je moguće izvještaje analiza spremiti u XML ili tekstualnu datoteku.

FindBugs je moguće koristiti unutar razvojnog okruženja kao što su Eclipse i NetBeans ili ga pokrenuti samostalno. Ukoliko se pokreće samostalno moguće je birati između GUI izvedbe ili izvedbe iz komandne linije.

Slika 4.1 Grafičko sučelje FindBugs-a prikazuje grafičko sučelje samostalnog FindBugs-a.

Za razliku od samostalne izvedbe, pri integraciji FindBugsa s razvojnim okruženjem nije potrebno instalirati samostalni FindBugs. Dovoljno je preuzeti dodatak za razvojno okruženje. Dodatak posjeduje u sebi samostalni FindBugs u obliku komandne linije koji se izvršava pokretanjem analize. Rad sa integriranom inačicom je jednostavan jer se oslanja na mogućnosti integriranog razvojnog okruženja da korisniku prenese informacije. Proces integracije i rada s FindBugsom opisan je u potpoglavlju 5.1. Dodane informacije o FindBugsu moguće je pronaći u izvoru [7].
4.2. Alat PMD


PMD analizu provodi nad izvornim kodom. S tim na umu, PMD posjeduje sve prednosti i nedostke alata koji analizu obavljaju nad izvornim kodom. PMD analizu obavlja pomoću apstraktnog sintaksnog stabla (AST). Za razliku od FindBugs-a, PMD ne koristi provjeru toka podataka. Rezultat ovih činjenica je veći broj lažno-negativnih izvještaja. Kako bi se smanjila količina izvještaja koju programer mora provjeriti, preporuka je analizu provoditi u više koraka. Preporuka je početi s osnovnim setom pravila, te nakon što se provjere izvješća, povećavati broj pravila.

Općenito PMD može otkriti slijedeće skupine problema:
- Moguće pogreške
- Neaktivni kod
- Ne optimiziran (engl. suboptimal) kod
- Prekomplicirane izraze
- Duplicirani kod

Popis svih skupina problema moguće je pročitati u izvoru [8].

PMD je moguće koristi samostalno pomoću naredbene linije ili ga je moguće integrirati unutar razvojnog okruženja. Ukoliko se koristi samostalno, uobičajeno je integrirati PMD sa alatom za izgradnju projekta kao što su Ant ili Maven. Također, PMD je moguće proširiti vlastitim pravilima, te je moguće izvještaje analiza spremiti u XML ili tekstualnu datoteku. Dodatne informacije o PMD-u moguće je pronaći u izvoru [8].

4.3. Alat Jenkins

Jenkins je alat otvorenog kôda namijenjen kontinuiranoj integraciji i izgradnji projekta. Za definiranje mogućnosti Jenksinsa potrebno je prvo definirati kontinuiranu integraciju. Prema [9], cilj kontinuirane integracije je smanjenje rizika i davanje brže povratne informacije programeru. Kontinuirana integracija, u najjednostavnijem obliku, predstavlja proces praćenja promjene u inačice projekta te reagiranja na promjenu. Pri pronalasku promjene, sustav za kontinuiranu integraciju može pokrenuti prevođenje i testiranje programa. Ukoliko se pogreška dogodi,
sustav za kontinuiranu integraciju obavještava programera o problemu. Napredni sustavi za kontinuiranu integraciju mogu pratiti kvalitetu kôda na način da računaju mjerne jedinice te vode računa da tehnički dug (vrijeme potrebno da se isprave pronađene pogreške) ne prijeđe zadanu vrijednost.


Slika 4.2. prikazuje Jenksinovu kontrolnu ploču koja je ujedno i početna stranica Jenksinsa koja sadrži ključne informacije o projektima. Osim informacija o projektima kontrolna ploča omogućuje pristup postavkama Jenksinsa.

![Slika 4.2. Sučelje alata Jenkins](image)

Slika 4.2. Sučelje alata Jenkins

Instalaciju dodataka moguće je obaviti iz Jenksinovog sučelja ili je moguće dodatke preuzeti s [10] te ih kopirati unutar plugins direktorija.
4.4. Alat SonarQube


Slika 4.3. SonarQube sučelje
5. PROVODENJE TESTIRANJA NA KODU

U ovom poglavlju opisano je podešavanje te rad sa alatima unutar Eclipse, Jenkins te SonarQube alata.

5.1. Podešavanje alata za rad s Eclipse-om

Integracija alata za statičku analizu omogućena je korištenjem dodataka. Za potrebe testiranja korišteni su dodaci PMD i FindBugs koje je potrebno instalirati pomoću Eclipseovog alata \textit{Install New Software}... (izbornik Help).

Slika 5.1 prikazuje prozor za instalaciju dodataka. Potrebno je pritisnuti \textit{Add}.... Klikom na \textit{Add}... otvara se prozor za dodavanje repozitorija.

![Slika 5.1. Prozor alata Install New Software](image)

Slika 5.2. Dodavanje repozitorija dodatka prikazuje prozor za dodavanje repozitorija. Unutar prozora potrebno je upisati ime po kojemu će Eclipse prepoznati repozitorij te web adresu repoziroija.
Slika 5.2. Dodavanje repozitorija dodatka


Slika 5.3. Odabir dodataka

PMD i FindBugs mogućnosti moguće je podesiti pomoću Preferences prozora. Postavke FindBugs-a nalaze se unutar Java skupine dok su PMD-ove postavke smještene u glavnom prozoru. Za potrebe rada postavke PMD-a i FindBugsa nisu mijenjanje.
Kako bi mogli koristiti PMD na nekom projektu potrebno ga je uključiti unutar projektnih postavki. FindBugs nije potrebno posebno uključivati ali je moguće namjestiti da se pokreće pri svakom prevodenju projekta.Ukoliko nije drugačije postavljeno alate je potrebno ručno pokrenuti. Pokretanje FindBugs-a obavlja se desnim klikom na projekt, odabirom FindBugs izbornika te klikom na FindBugs.

Slika 5.4 prikazuje FindBugs-ov izbornik. Osim samog pokretanja FindBugs-a izbornik omogućuje spremanje ili učitavanje rezultata analize.

![Slika 5.4. FindBugs izbornik](image)

Pokretanje PMD-a obavlja se također pomoću izbornika. Potrebno je desnim klikom na projekt odabrati PMD izbornik te kliknuti na Check Code.

Slika 5.5. prikazuje PMD-ov izbornik. Osim samog pokretanja PMD-a izbornik omogućuje spremanje izvještaja, poništavanje pogrešaka te pokretanje alata za provjeru kopiranja.

![Slika 5.5. Izbornik alata PMD](image)

Za lakši rad oba alata posjeduju vlastitu perspektivu. Prilikom pokretanja alata, Eclipse prebacuje perspektivu u perspektivu alata.

Slika 5.6 prikazuje FindBugs perspektivu.
Slika 5.6. FindBugs perspektiva


Slika 5.7 prikazuje PMD perspektivu.
Slika 5.7. PMD perspektiva

Slika 5.8 prikazuje Show details... prozor za pravilo StdCyclomaticComplexity. Prozor sadrži ime klase u kojoj je napisano pravilo, detaljan opis problema te primjer kako bi programer lakše shvatio problem.
5.2. **Podešavanje Jenksinsa**


Za potrebe rada korišteni su slijedeći dodaci:

- Ant plugin
- Checkstyle plugin
- FindBugs plugin
- PMD plugin
- Static Analysis Collector Plugin
- Static Analysis Utilities

Idući korak podešavanja Jenkansa je stvaranje projekta. Opcija za stvaranje novog prozora je *New item*. Moguće joj je pristupiti s početne stranice Jenkansa, tj. s komandne ploče.


Slika 5.9. Stvaranje Jenkns projekt

Slika 5.10. Opće postavke Freestyle projekta

Budući da testni projekt koristi subversion kontrolu izvornog kôda, potrebno je Jenkins projekt podesiti da radi sa subversionom. Slika 5.11 prikazuje opcije kontrole izvornog kôda.

Slika 5.11. Podešavanje kontrole izvornog kôda


Slika 5.12. Postavke izgradnje projekta

Pri izgradnji projekta nužno je prvo pozvati Ant bez dodatnih argumenta kako bi projekt izgradio. Drugi Ant poziv koristi se za pozivanje Ant zadataka koje želimo izvršiti. Pri pozivu Ant-a moguće je navesti više zadataka koji će se izvršiti slijedno.

Jenkins posjeduje mogućnost izvršavanja određenih zadataka nakon što se izgradnja projekta završi. Zadaci mogu biti razni, ovisno o instaliranim proširenjima, ali uobičajeno je pokrenuti zadatke za prikaz rezultata ili zadatke za slanje obavijesti korisnicima. Slika 5.13 prikazuje FindBugs zadatak koji će se izvršiti nakon izgradnje projekta.
Slika 5.13. Postavke izvršavanja zadataka nakon izgradnje projekta

Nakon postavljanja postavki potrebno je kliknuti na Save. Ovime je završen proces postavljanja projekta. Navedene postavke moguće je promijeniti odabirom projekta te klikom na poveznicu Configure.


Kako bi napravljeni Jenkins projekt radio, potrebno je napraviti Ant task koji će pozvati FindBugs. Primjer 5.1. FindBugs ant zadatak prikazuje FindBugs Ant zadatak.

```
<taskdef name="findbugs" classname="edu.umd.cs.findbugs.anttask.FindBugsTask"/>
<target name="findbugs">
  <findbugs home="${findbugs.home}">
    <sourcePath path="${basedir}/src"/>
    <class location="${basedir}/bin"/>
    <effort="max"/>
    <output="xml"/>
    <outputFile="findbugs.xml"/>
  </findbugs>
</target>
```

Primjer 5.1. FindBugs ant zadatak

Oznaka taskdef definira zadatak imena findbugs. Oznaka target definira Ant zadatak dok oznaka findbugs definira uvjete findbugs zadataka. Ovako napisan Ant zadatak izvršiti će se ukoliko...
 postoje $ANT_HOME$ te $FINDBUGS_HOME$ sistemskе varijable. U suprotnom, potrebno ih je stvoriti ili u Ant build datoteke upisati lokaciju FindBugs-a.

### 5.3. Podešavanje SonarQube-a


Primjer 5.2. prikazuje Ant zadatak za SonarQube.

```xml
<!-- Define the SonarQube project properties -->
<property name="sonar.projectKey" value="OPTXpressDefaultRules" />  
<property name="sonar.projectName" value="OPTXpress Default Rules" />  
<property name="sonar.projectVersion" value="1.0" />  
<property name="sonar.language" value="java" />  
<property name="sonar.sources" value="${basedir}/src/" />  
<property name="sonar.java.binaries" value="${basedir}/bin/" />

<!-- Define the SonarQube target -->
<target name="sonar">  
  <taskdef uri="antlib:org.sonar.ant" resource="org/sonar/ant/antlib.xml">  
    <!-- Update the following line, or put the "sonar-ant-task-*.jar" file in your "$HOME/.ant/lib" folder -->  
    <classpath path="C:/sonarqube/ant-lib/sonar-ant-task.jar" />  
  </taskdef>  
  <sonar:sonar />  
</target>
```

**Primjer 5.2. SonarQube ant zadatak**

U Primjer 5.2 oznake `property` definiraju varijable koje su nužne za rad SonarQube-a. Svojstvo `sonar.projectName` definira ime projekta dok svojstvo `sonar.projectKey` definira ključ raspoznavanja projekta. Ime projekta ne mora biti jedinstveno, ali ključ raspoznavanja treba biti jedinstven. Oznaka `target` definira ant zadatak imena sonar. Oznaka `taskdef` definira sonar zadatak te sadrži lokaciju ant sonar biblioteke. Lokacija biblioteke ne mora se navoditi ukoliko se biblioteka nalazi unutar Ant direktorija.

Analiza se pokreće pomoću naredbe ant sonar. Pri pokretanju analize, Ant provjerava dostupnost Sonar servera, te započinje analizu. Naposljetku, rezultati se zapisuju u bazu podataka. Ukoliko projekt postoji, rezultati će se zabilježiti kao novo izvođenje te će se ažurirati statistika projekta.
Ukoliko ne postoji projekt, SonarQube će napraviti projekt te zabilježiti rezultate za prvo izvođenje. Nakon što je analiza gotova moguće ju je vidjeti pristupanjem SonarQubeovom sučelju. Osim samog pregleda analize, moguće je promijeniti postavke projekta.

Slika 5.14. prikazuje upravljačku ploču projekta.

**Slika 5.14. Upravljačka ploča projekta**

6. ANALIZA DOBIVENIH REZULTATA


Za pravilno razumijevanje analize potrebne su informacije o kodu. Značajne informacije o kodu prikazane su Slika 6.1. Statistika kôda

<table>
<thead>
<tr>
<th>Statistika kôda</th>
</tr>
</thead>
<tbody>
<tr>
<td>Broj linija kôda</td>
</tr>
<tr>
<td>Broj klasa</td>
</tr>
<tr>
<td>Broj funkcija</td>
</tr>
<tr>
<td>Složenost kôda</td>
</tr>
<tr>
<td>Srednja vrijednost složenosti po klasi</td>
</tr>
</tbody>
</table>

Slika 6.1. Statistika kôda

Alati su postavljeni na istu preciznost da bi lakše vidjeli iste pogreške u različitim alatima. Slika 6.2. prikazuje rezultat analize projekta FindBugsom.
Slika 6.2. Usporedba korištenja FindBugs-a

Zanimljivo je da postoji odstupanje u alatima iako su postavljeni na istu preciznost te da prijavljuju istu razinu pogrešaka. Budući da su alati prijavili slične pogreške, vjerojatni razlog odstupanja je korištenje različitih metoda analiziranja strojnog kôda.

Tablica 6.1. prikazuje referentne pogreške koje su bile zajedničke u svim rezultatima. Tablica sadrži ime pogreške, kritičnost te broj pronađenih pogrešaka.

<table>
<thead>
<tr>
<th>Ime pogreške</th>
<th>Složenost</th>
<th>Broj pronađenih pogrešaka</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nekorištena vrijednost</td>
<td>Normal</td>
<td>1</td>
</tr>
<tr>
<td>Nepročitana vrijednost</td>
<td>Normal</td>
<td>5</td>
</tr>
<tr>
<td>Pisanje u statičku varijablu</td>
<td>High</td>
<td>2</td>
</tr>
<tr>
<td>NULL pointer dereference</td>
<td>High</td>
<td>2</td>
</tr>
<tr>
<td>Ukupno</td>
<td></td>
<td>10</td>
</tr>
</tbody>
</table>

Tablica 6.1. Referentne pogreške

Za pogreške navedene unutar Tablica 6.1, u nastavku poglavlja, opisan je način korištenja alata za statičku analizu iz perspektivu programera, voditelja projekta te vodstva (višeg menadžmenta).

Analiza iste pogreške u različitim okruženjima/perspektivama pokazala je da SonarQube pruža najbolje mogućnosti pronalaženja, izvještavanja i opisa pogreške. Osim navedenih mogućnosti SonarQube pruža mogućnost interakcije sa pogreškom te nudi bitne statistike o kodu te njegovoj kvaliteti.
6.1. Analiza korištenja statičke analize u različitim dijelovima razvoja projekta

Cilj ove analize je prikazati korištenje statičke analize na različitim razinama projekta. Ideja je prikazati korištenje alata za statičku analizu iz perspektive programera, perspektive voditelja projekta te perspektive vodstva (višeg menadžmenta).

Programerova perspektiva odnosi se razvojno okruženje. Ona predstavlja najuži pogled na razvojni proces jer programeri najčešće rade na određenom dijelu ili dijelovima projekta, ne vodeći računa o ostalim dijelovima. Alat koji predstavlja ovu skupinu je Eclipse integrirano razvojno okruženje. Unutar ove perspektive, alati za statičku analizu se koriste za pronalaženje i otklanjanje pogrešaka na onim dijelovima na kojima programer radi.

Perspektiva voditelja projekta pruža pogled na cijeli projekt, stoga alat (ili više njih) za statičku analizu moraju voditi računa o cijelom projektu. Alat koji predstavlja ovu skupinu je SonarQube. Osim prijave pogrešaka, alat pruža informacije o kvaliteti kôda, dajući do znanja vođi tima da li kvaliteta kôda opada ili raste.

Perspektiva vodstva odnosi se na pogled na sve projekte. Alati koji predstavljaju ovu skupinu su SonarQube i Jenkins. Alati za statičku analizu moraju posjedovati mogućnost praćenja više projekata te prikaz kvalitete za sve projekte.


Analiza je rađena na manjem Java projektu koji je (u vrijeme pisanja ovoga rada) u implementacijskoj fazi i mijenjao se svakodnevno.

6.1.1. Perspektiva programera

pogreška te mu objasniti zašto je navedeno pogreška. Navedene stvari omogućuju prozori Bug Explorer i Bug Info.


Slika 6.3. Prozor Bug Explorera


Slijedeći kod označen je kao null pointer pogreška :

```java
if (tqReportsFolder.isDirectory()) {
    for (File file : tqReportsFolder.listFiles()) {
```

Slika 6.4. Bug Info prozor

U opisu problema piše da se pozivanjem objekta, bez provjere da li je objekt postoji, može se dogoditi iznimka oblika `NullPointerException`. Osim opisa problema, Bug Info sadrži i XML izvještaj o pogrešaku.


6.1.2. Perspektiva voditelja projekta

Kao primjer perspektivne voditelja projekta uzet je rad sa SonarQube alatom. SonarQube omogućava širi, ali i detaljniji pogled na projekt. poglavlja 4.4 i 5.3 nude kratak opis programa i njegovih mogućnosti. Za ovu analizu zanimljive su SonarQube-ove mogućnosti vođenja statistike projekta, otkrivanja pogrešaka te planiranje popravaka.

U poglavlju 5.3 opisana je upravljačka ploča SonarQube-a pomoću koje se može otvoriti upravljačka ploča projekta. Upravljačka ploča projekta prikazuje informacije o kodu kao što su statistike kôda, složenost kôda, tehnički dug, broj pogrešaka, omjer dupliciranog kôda te

Slika 6.5 prikazuje statistiku o veličini kôda. Detaljniju statistiku moguće je dobiti klikom na informaciju. Iz priložene statistike može se zaključiti da je analiza rađena na manjem projektu.

Slika 6.5. Statistika kôda
Slika 6.6. prikazuje pronađene pogreške. SonarQube pronađene pogreške raspoređuje prema složenosti.

Slika 6.6. Popis pogrešaka
Informacija Debt predstavlja vrijednost tehničkog duga izraženu u radnim satima. Infomacija Issues predstavlja broj problema koji su analizom pronađeni. Ostatak slike prikazuje broj problema razvstan u skupine prema kritičnosti. Klikom na broj pogrešaka ili pojedinu skupinu dobivamo informacije o pogreškama.


Slika 6.7. Prikazuje izbornik za odabir prikaza pogrešaka.
Slika 6.7. Odabir pogreške

Pogreške je moguće odabrati prema slijedećim kriterijima:

- Složenost
- Riješenost
- Status
- Nove pogreške
- Pravilo
- Tag
- Modul
- Direktorij
- Datoteka
- Osoba koja je pronašla pogrešku
- Osoba kojoj je greška dodijeljena
- Jeziku
- Planu rada

Svaki od navedenih kriterija posjeduje vlastite opcije prema kojima je moguće odabrati pogreške. Kako je cilj prikazati sve ne riješene pogreške odabran je kriterij Resolution te opcija Unresolved.

Slika 6.8. Prikazuje listu pogrešaka koje zadovoljavaju kriterij Unresolved. Odabirom strelice uz pojedinu pogrešku, otvara se izvorni kod u kojemu je označena pogreška.
Slika 6.8. Lista pogrešaka

Slika 6.9. prikazuje način na koji SonarQube prijavljuje pogrešku. Unutar opisa moguće je vidjeti ime klase u kojoj je pogreška pronađena ta kratak opis pogreške.

Slika 6.9. Analiza pogreške

Za potvrđivanje postojanja pogreške poželjno je proučiti pogrešku. Detaljan opis pogreške moguće je prikazati klikom na ikonu sa tri točke u sebi. Slika 6.10 Prikazuje SonarQube-ov način opisivanja pogreške.

![Slika 6.10. Opis Null pointer dereference pogreške](image)


Ostale mogućnosti objašnjenje su u poglavlju 5.3.

### 6.1.3. Perspektiva vodstva


Slika 6.11. Kontrolna ploča projekta


Slika 6.12. Izvještaj posljednje izgradnje

54

Slika 6.13. prikazuje FindBugs izvještaj.

**FindBugs Result**

<table>
<thead>
<tr>
<th>Warnings Trend</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>All Warnings</td>
<td>New this build</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Summary</th>
<th>Total</th>
<th>High Priority</th>
<th>Normal Priority</th>
<th>Low Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>0</td>
<td>12</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Details</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Packages</td>
<td>Files</td>
</tr>
<tr>
<td>com.siemens.optexpress</td>
<td>5</td>
</tr>
<tr>
<td>com.siemens.optexpress.util</td>
<td>7</td>
</tr>
<tr>
<td>Total</td>
<td>12</td>
</tr>
</tbody>
</table>

Slika 6.13. **FindBugs izvještaj**

FindBugs izvještaj nudi informacije o količini grešaka. Prva tablica sadrži podatke o ukupnom broju pogrešaka, novonastalim pogreškama te o pogreškama koje su ispravljene u odnosu na prethodnu inačicu. Druga tablica sadrži podatke o složenosti pogrešaka. Posljednja tablica sadrži detaljniji opis problema. Posljednja tablica omogućuje pregledavanje pogrešaka prema paketima, datotekama, kategoriji, tipu te nudi pregled grešaka i detaljan pregled.

Slika 6.14. Detaljan prikaz grešaka


Slika 6.15. Prikaz pogreške Null pointer dereference

6.2. Detaljna analiza projekta

Za analizu su odabrana FindBugs, SonarQube i PMD pravila za statičku analizu kôda. Detaljne analize sa FindBugs i SonarQube pravilima izrađene su pomoću SonarQube-a, a razlozi njegova odabira navedeni su u prethodnom poglavlju. Cilj ovih analiza je demonstrirati SonarQube-ove mogućnosti. Za razliku od analize u prethodnom poglavlju, unutar ove analize korištene su različite skupine pravila kako bi se zaključilo koja pravila daju bolje i korisnije rezultate.

6.2.1. SonarQube analiza pomoću FindBugs pravila

Rezultati FindBugs analize opisani su u poglavlju 6. U ovom poglavlju opisane su ostale mogućnosti koje SonarQube posjeduje. Opis mogućnosti uključuje informacije sa kontrolne ploče te detalje na koje kontrolna ploča pokazuje.

Slika 6.16 prikazuje izbornik projekta. „Overview“ izbornik omogućuje prikaz osnovnih informacija o projektu.

**Slika 6.16. Izbornik projekta**


U nastavku je opisan „Overview“ prikaz. „Overview“ prikaz omogućava pristup većini ključnih elemenata te sadrži elemente kao što su prikaz pogreške, promjena pravila, pristup pravilima i sl.

Slika 6.17 prikazuje statistiku o dupliciranom kodu. SonarQube je pronašao 95 linija koje su ponovljene. Strelica uz brojku broj linija upućuje da je broj dupliciranih linija porastao.
Slika 6.17. Informacije o dupligranom kodu
Klikom na broj linija dobivaju se informacije u kojim klasama su otkrivena ponavljanja. Odabirom pojedine klase, otvara se preglednik kôda u kojemu je moguće vidjeti problematične linije.


Slika 6.18. Prikaz dupliciranih linija


Slika 6.19. Prikaz složenosti funkcija

Slika 6.20. Prikaz složenosti datoteka projekta

Slika 6.21 prikazuje popis svih događaja koji su vezani za project. SonarQube bilježi događaje kao što su promjena inačice, promjena pravila te promjenu kvalitete.

Slika 6.21. Prikaz događaja

Slika 6.22 prikazuje profil pomoću kojeg je napravljena analiza. Klikom na profil dobivaju se informacije o profilu.

Slika 6.22. Profil analize

Slika 6.23 prikazuje detalje FindBugs profila. Prikazan je broj pravila koje profil sadrži te njihova složenost. Klikom na pojedinu komponentu dobivaju se informacije o pravilima koja pripadaju toj skupini.
Slika 6.23. Profil FindBugs
Slika 6.24 prikazuje SonarQube-ov način ocjenjivanja kvalitete. SQALE ocjena bazira se na procjenjivanju tehničkog duga, mjere koja označava koliko vremena je potrebno utrošiti na ispravak pogrešaka.

Slika 6.24. Ocjena kvalitete SQALE
Slika 6.25 prikazuje statistiku zapletenosti datoteka. Zapletenost datoteka je mjera uzajamne ovisnosti datoteka.

Slika 6.25. Zapletenost datoteka

Slika 6.26. Pravila kontrolne stanice

Za uspoređivanje rezultata FindBugs i SonarQube analize potrebno je dokumentirati pronađene pogreške.

Tablica 6.2 predstavlja pregled pronadjenih pogrešaka pomoću FindBugs pravila.

<table>
<thead>
<tr>
<th>Ime pogreške</th>
<th>Složenost</th>
<th>Broj pronadjenih pogrešaka</th>
</tr>
</thead>
<tbody>
<tr>
<td>Povjerenje u zadano šifriranje (encoding)</td>
<td>Visoka</td>
<td>12</td>
</tr>
<tr>
<td>NULL pointer dereference</td>
<td>Kritična</td>
<td>2</td>
</tr>
<tr>
<td>Metoda ne provjerava vraćenu vrijednost kod metoda koje bacaju iznimke</td>
<td>Visoka</td>
<td>2</td>
</tr>
<tr>
<td>Spremljena vrijednost varijable se ne koristi</td>
<td>Kritična</td>
<td>1</td>
</tr>
<tr>
<td><strong>Ukupno</strong></td>
<td></td>
<td><strong>17</strong></td>
</tr>
</tbody>
</table>

Tablica 6.2. Popis pronadenih pogrešaka pomoću FindBugs pravila


U prilogu P.6.1 nalazi se tablica pronadenih pogrešaka pomoću SonarQube pravila. Iz priloga P.6.1 može se zaključiti da su SonarQubeova pravila brojnija te općenita. Prema opisu problema, navede pogreške karakterizirali bi kao lošu praksu. Rezultati analize pokazali su da SonarQube prijavljuje slične oblike pogrešaka kao PMD. Razlog tome je činjenica da su SonarQube-ova pravila nastala iz PMD-ovih te FindBugs-ovih pravila. Iako PMD i SonarQube analiza obuhvaćaju slične probleme, analize se uvelike razlikuju. PMD analiza

6.2.2. PMD analiza unutar Eclipse razvojnog okruženja


6.3. Ispravljanje pogrešaka


Tablica 6.3 prikazuje popis grešaka koje je poželjno što prije ispraviti.
<table>
<thead>
<tr>
<th>Ime pogreške</th>
<th>Složenost</th>
<th>Broj pronađenih pogrešaka</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rukovatelj iznimkama treba sačuvati izvornu iznimku</td>
<td>Kritična</td>
<td>30</td>
</tr>
<tr>
<td>Throwable.printStackTrace(...) se ne treba pozivati</td>
<td>Kritična</td>
<td>9</td>
</tr>
<tr>
<td>Metode trebaju imati manju složenost</td>
<td>Visoka</td>
<td>5</td>
</tr>
<tr>
<td>Prilikom provjere jednakosti String se treba nalaziti s lijeve strane</td>
<td>Visoka</td>
<td>45</td>
</tr>
<tr>
<td>Deklaracija treba koristiti Java kolekcije kao sučelje kao što je List nego specifične implementirane klase kao što je &quot;LinkList&quot;</td>
<td>Visoka</td>
<td>29</td>
</tr>
<tr>
<td>Nepoželjno je duboko ugnježđivanje izjava za kontrolu toka kao što su &quot;if&quot;, &quot;for&quot;, &quot;while&quot;, &quot;switch&quot; and &quot;try&quot;</td>
<td>Visoka</td>
<td>19</td>
</tr>
</tbody>
</table>

**Tablica 6.3. Prioritet ispravljanja**

Samo ime pogreške otkriva na koji način je potrebno ispraviti pogrešku, tako da nije potrebno detaljno opisivati probleme. Potrebno je voditi računa o refaktoriranju kôda, koje je nužno kako bi se smanjila složenost metoda. Refaktoriranje može uvesti nove pogreške ali i potpunosti ukloniti već pronađene probleme te mu, zbog navedenog, treba pristupiti s odgovarajućom dozom ozbiljnosti.
7. ZAKLJUČAK

Kako god se trendovi u industriji programske podrške mijenjali, neke stvari su konstatne, ne zastarijevaju i vrijede za sve metodologije razvoja. Jedna od takvih je kvaliteta izrađenog programa. U ovom radu pokazana je mogućnost statičke analize s ciljem povećanja kvalitete programskog koda. U radu je pokazana primjena dva različita tipa alata za statičku analizu, PMD-a te FindBugs-a. Alat PMD je pokazao kako se statička analiza može koristiti za povećanje čitljivosti, preglednosti, pronalaženje jednostavnih pogrešaka te smanjenje složenosti. Alat FindBugs pokazao je kako se statička analiza može koristiti za pronalaženje propusta, kao što su rad sa neinicializiranim vrijednostima, nekorištene varijable, nepravilni pozivi metoda te za pronalaženje jednostavnih sigurnostnih propusta kao što je prelijevanje spremnika. Također, pokazano je kako se statička analiza može primijeniti na različitim razinama razvojnog procesa. Nju je moguće uključiti u razvojno okruženje, proizvodno okruženje ili je integrirati u specijalizirane alate za rukovođenje projekta kao što je SonarQube. Uključenjem statičke analize u više razina razvoja, povećava se vjerojatnost otkrivanja pogrešaka što rezultira uštedom resursa.

Iz navedenoga se može zaključiti da statička analiza treba biti obvezan dio svakog razvoja programa. Važno je da se statička analiza postupno uključi u svakodnevni rad kako ona ne bi postala svojevrstan teret, jer se, u tom slučaju, nakon nekog vremena odustane od njegove primjene ili se svede na formalnost koja ne služi svrsi.
LITERATURA


[12] Christel Baier and Joost-Pieter Katoen, "Principles of Model Checking".


SAŽETAK

U ovom radu opisana je primjena statičke analize na jednostavnom komercijalnom projektu. Naprije su opisane prednosti i nedostaci statičke analize te njena moguća primjena. U glavnom dijelu rada pokazana je upotreba statičke analize pomoću alata FindBugs i PMD za pronalaženje pogrešaka i loših praksi pisanja koda. Također, u radu je opisana primjena statičke analize iz perspektive programera, voditelja projekta te perspektive višeg managmenta. Pokazano je kako je primjenom statičke analize moguće smanjiti broj pogrešaka te povećati pouzdanost i sigurnost programa.

Ključne riječi: ispravljanje pogrešaka, Java, održavanje programskog koda, pronalaženje pogrešaka, statička analiza koda.

ABSTRACT

This paper describes application of static code analysis on a simple commercial project. Firstly, paper describes advantages, disadvantages and applicable cases of using static code analysis. Main chapter of paper describes application of static code analysis for finding bugs and bad practices by using tools such as FindBugs and PMD. Furthermore, this paper describes application of static code analysis in programmer’s perspective, project management perspective and upper management perspective. It has been shown that by using static code analysis it is possible to reduce number of bugs and increase reliability and security of the program.

Keywords: bug correction, code maintenance, Java, finding bugs, static code analysis.

Ime i prezime

____________________ (potpis)
**PRILOZI**

### P.6.1. Popis pronađenih pogrešaka pomoću SonarQube pravila

<table>
<thead>
<tr>
<th>Ime pogreške</th>
<th>Složenost</th>
<th>Broj pronađenih pogrešaka</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prilikom provjere jednakosti String se treba nalaziti s lijeve strane</td>
<td>Visoka</td>
<td>45</td>
</tr>
<tr>
<td>Nekorištene privatne metode treba ukloniti</td>
<td>Visoka</td>
<td>41</td>
</tr>
<tr>
<td>Rukovatelj iznimkama treba sačuvati izvornu iznimku</td>
<td>Kritična</td>
<td>30</td>
</tr>
<tr>
<td>Deklaracija treba koristiti Java kolekcije kao sučelje (kojoj je List) ne specifične implementirane klase (kojoj je &quot;LinkList&quot;)</td>
<td>Visoka</td>
<td>29</td>
</tr>
<tr>
<td>Tab znakovi se ne trebaju koristiti</td>
<td>Niska</td>
<td>22</td>
</tr>
<tr>
<td>Nepoželjno je duboko ugnježdavanje naredbi za kontrolu toka kao što su &quot;if&quot;, &quot;for&quot;, &quot;while&quot;, &quot;switch&quot; and &quot;try&quot;</td>
<td>Visoka</td>
<td>19</td>
</tr>
<tr>
<td>String oznake trebaju biti jedinstvene</td>
<td>Niska</td>
<td>18</td>
</tr>
<tr>
<td>Nekorištene privatne varijable treba ukloniti</td>
<td>Visoka</td>
<td>15</td>
</tr>
<tr>
<td>Throwable.printStackTrace(...) se ne treba pozivati</td>
<td>Kritična</td>
<td>9</td>
</tr>
<tr>
<td>Potrebno je ukloniti nepotreban za grad koju je &quot;if&quot;, &quot;for&quot;, &quot;while&quot;, &quot;switch&quot; and &quot;try&quot;</td>
<td>Visoka</td>
<td>9</td>
</tr>
<tr>
<td>“TODO” tagove treba implementirati</td>
<td>Info</td>
<td>7</td>
</tr>
<tr>
<td>Ugnježdeni dijelovi kôda ne trebaju biti prazni</td>
<td>Visoka</td>
<td>6</td>
</tr>
<tr>
<td>Nekorištene importe treba ukloniti</td>
<td>Niska</td>
<td>6</td>
</tr>
<tr>
<td>Metode trebaju imati manju složenost</td>
<td>Visoka</td>
<td>5</td>
</tr>
<tr>
<td>Standardni izlaz se ne treba koristiti za bilježenje informacije(logging)</td>
<td>Visoka</td>
<td>5</td>
</tr>
<tr>
<td><strong>Ukupno</strong></td>
<td></td>
<td>266</td>
</tr>
</tbody>
</table>

### P.6.2. Popis pronađenih pogrešaka pomoću PMD pravila

<table>
<thead>
<tr>
<th>Ime pogreške</th>
<th>Složenost</th>
<th>Broj pronađenih pogrešaka</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dodjeljivanje vrijednosti pri operaciji (usporedbi)</td>
<td>Normal</td>
<td>1</td>
</tr>
<tr>
<td>Klasa treba imati barem jedan konstruktor</td>
<td>Normal</td>
<td>9</td>
</tr>
<tr>
<td>Izbjegavati hvatanje generičke iznimke</td>
<td>Normal</td>
<td>8</td>
</tr>
<tr>
<td>Izbjegavati hvatanje NullPointerException iznimke</td>
<td>Normal</td>
<td>1</td>
</tr>
<tr>
<td>Izbjegavati hvatanje Throwable iznimke</td>
<td>Normal</td>
<td>1</td>
</tr>
<tr>
<td>Izbjegavati iste vrijednosti varijabli</td>
<td>Normal</td>
<td>16</td>
</tr>
<tr>
<td>Izbjegavati instanciranje objekata u petlji</td>
<td>Normal</td>
<td>21</td>
</tr>
<tr>
<td>Izbjegavati korištenje vrijednosti u if uvjetu</td>
<td>Normal</td>
<td>5</td>
</tr>
<tr>
<td>Izbjegavati PrintStackTrace</td>
<td>Normal</td>
<td>9</td>
</tr>
<tr>
<td>Izbjegavati bacanje čistih iznimki</td>
<td>High</td>
<td>2</td>
</tr>
<tr>
<td>Bean članovi bi trebali biti serijalizirani</td>
<td>Normal</td>
<td>117</td>
</tr>
<tr>
<td>Dobra je praksa pozvati super u konstruktor</td>
<td>Normal</td>
<td>4</td>
</tr>
<tr>
<td>Ugniježdeni if uvjeti se mogu kombinirati</td>
<td>Normal</td>
<td>2</td>
</tr>
<tr>
<td>Potreban komentar</td>
<td>Normal</td>
<td>196</td>
</tr>
<tr>
<td>Prevelik komentar</td>
<td>Normal</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>Normal</td>
<td>9</td>
</tr>
<tr>
<td>Zbunjujući uvjet</td>
<td>Normal</td>
<td>9</td>
</tr>
<tr>
<td>Ciklomatska složenost</td>
<td>Normal</td>
<td>8</td>
</tr>
<tr>
<td>Dataflow analiza</td>
<td>Low</td>
<td>67</td>
</tr>
<tr>
<td>Default paket</td>
<td>Normal</td>
<td>9</td>
</tr>
<tr>
<td>Izbjegavati poziv System.exit funkciji</td>
<td>Normal</td>
<td>1</td>
</tr>
<tr>
<td>Izbjegavati korištenje niti u J2EE</td>
<td>Normal</td>
<td>1</td>
</tr>
<tr>
<td>Prazan try catch blok</td>
<td>Normal</td>
<td>4</td>
</tr>
<tr>
<td>Prazan if blok</td>
<td>Normal</td>
<td>5</td>
</tr>
<tr>
<td>Veliki broj importa</td>
<td>Normal</td>
<td>1</td>
</tr>
<tr>
<td>Predugačke metode</td>
<td>Normal</td>
<td>3</td>
</tr>
<tr>
<td>Deklaracija varijabli treba se nalaziti na početku klase</td>
<td>Normal</td>
<td>39</td>
</tr>
<tr>
<td>God klasa</td>
<td>Normal</td>
<td>2</td>
</tr>
<tr>
<td>If uvjet treba imati blok</td>
<td>Normal</td>
<td>9</td>
</tr>
<tr>
<td>ImmutableField</td>
<td>Normal</td>
<td>72</td>
</tr>
<tr>
<td>JUnit4Test treba koristiti Test anotaciju</td>
<td>Normal</td>
<td>2</td>
</tr>
<tr>
<td>Law of demeter</td>
<td>Normal</td>
<td>404</td>
</tr>
<tr>
<td>Lokalna varijabla može biti final</td>
<td>Normal</td>
<td>281</td>
</tr>
<tr>
<td>Predugo ime varijable</td>
<td>Normal</td>
<td>37</td>
</tr>
<tr>
<td>Izbjegavati koristiti ArrayList kao implementacijski tip. Koristiti sučelje umjesto ArrayList.</td>
<td>Normal</td>
<td>12</td>
</tr>
<tr>
<td>Argumenti metode mogu biti konačni</td>
<td>Normal</td>
<td>106</td>
</tr>
<tr>
<td>Izmijenjena ciklomatska složenost</td>
<td>Normal</td>
<td>7</td>
</tr>
<tr>
<td>Npath složenost</td>
<td>Normal</td>
<td>2</td>
</tr>
<tr>
<td>Metoda treba imati samo jednu return izjavu</td>
<td>Normal</td>
<td>20</td>
</tr>
<tr>
<td>Prilikom provjere jednakosti String se treba nalaziti s lijeve strane</td>
<td>Normal</td>
<td>28</td>
</tr>
<tr>
<td>Suvišno inicijaliziranje polja</td>
<td>Normal</td>
<td>3</td>
</tr>
<tr>
<td>Prekratko ime klase</td>
<td>Normal</td>
<td>1</td>
</tr>
<tr>
<td>Prekratko ime varijable</td>
<td>Normal</td>
<td>15</td>
</tr>
<tr>
<td>Pojednostaviti logički izraz</td>
<td>Normal</td>
<td>1</td>
</tr>
<tr>
<td>Varijabla može biti lokalna</td>
<td>Normal</td>
<td>21</td>
</tr>
<tr>
<td>Standardna Ciklomatska složenost</td>
<td>Normal</td>
<td>8</td>
</tr>
<tr>
<td>Switch petlja treba imati default</td>
<td>Normal</td>
<td>3</td>
</tr>
<tr>
<td>Ne koristiti SystemPrintLn</td>
<td>High</td>
<td>5</td>
</tr>
<tr>
<td>Previše varijabli</td>
<td>Normal</td>
<td>2</td>
</tr>
<tr>
<td>Previše metoda</td>
<td>Normal</td>
<td>4</td>
</tr>
<tr>
<td>Ne komentiran prazni konstruktor</td>
<td>Normal</td>
<td>4</td>
</tr>
<tr>
<td>Nepotreban konstruktor</td>
<td>Normal</td>
<td>3</td>
</tr>
<tr>
<td>Nepotrebne zagrade</td>
<td>Normal</td>
<td>9</td>
</tr>
<tr>
<td>Nekorišteni parametar</td>
<td>Normal</td>
<td>3</td>
</tr>
<tr>
<td>Nekorišteni import</td>
<td>Normal</td>
<td>6</td>
</tr>
<tr>
<td>Nekorištena privatna varijabla</td>
<td>Normal</td>
<td>14</td>
</tr>
<tr>
<td>Nekorištena privatna metoda</td>
<td>Normal</td>
<td>41</td>
</tr>
<tr>
<td>Koristiti objekte za čišći API</td>
<td>Normal</td>
<td>2</td>
</tr>
<tr>
<td>Koristiti valjani ClassLoader</td>
<td>Normal</td>
<td>4</td>
</tr>
<tr>
<td>Koristiti String Buffer za dopunjavanje stringa</td>
<td>Normal</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>Normal</td>
<td>1</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>--------</td>
<td>---</td>
</tr>
<tr>
<td>Koristiti pomoćne klase</td>
<td>Normal</td>
<td>1</td>
</tr>
<tr>
<td>Koristiti programske argumente (varargs)</td>
<td>Normal</td>
<td>1</td>
</tr>
<tr>
<td>Nepotrebe zagrade</td>
<td>Normal</td>
<td>12</td>
</tr>
<tr>
<td>Poštivati konvenciju o imenovanju varijabli</td>
<td>High</td>
<td>3</td>
</tr>
<tr>
<td><strong>Ukupno</strong></td>
<td></td>
<td>1698</td>
</tr>
</tbody>
</table>