Statička analiza koda na primjeru složene poslovne aplikacije

Solić, Matija

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU

ELEKTROTEHNIČKI FAKULTET

Sveučilišni studij

STATIČKA ANALIZA KODA NA PRIMJERU SLOŽENE POSLOVNE APLIKACIJE

Diplomski rad

Matija Solić

Osijek, 2016.

SADRŽAJ

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 na 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.

Drugo poglavlje opisuje teorijsku pozadinu statičke analize. Unutar ovog poglavlja objašnjena su osnovna načela koja omogućuju statičku analizu. Cilj poglavlja je detaljnije objasniti proces pomoću kojeg statička analiza dolazi do rezultata kako bi bilo moguće shvatiti nedostatke statičke analize. Nadalje, unutar trećeg poglavlja, opisani su alati otvorenog koda koji omogućuju statičku analizu Java koda. Opis alata sadrži popis mogućnosti alata (s obzirom na statičku analizu), specifičnosti alata te kratak pregled sučelja. Četvrto poglavlje opisuje proces provođenja statičke analize u navedenim alatima. Osim opisa samog postupka provođenja, opisan je i proces instaliranja alata. Posljednje, peto poglavlje, sadrži analizu rezultata analize. Sastoji se od dvije analize. Prva analiza opisuje rezultate korištenja alata za statičku analizu sa različitih stajališta. Cilj analize je pokazati učinkovitost korištenja analize u različitim stadijima razvoja. Cilj druge analize je pokazati koje su probleme alati pronašli, te u konačnici, pokazati način na koji su problemi rješeni.

Ovim putem želio bih zahvaliti ljudima iz tvrke Siemens Convergence Creators na uloženom trudu i vremenu.

2. UVOD U STATIČKU ANALIZU KÔDA

Statička analiza kôda predstavlja proces provjere kôda bez njegovog izvršavanja. Primjene statičke analize su razne. Današnji prevoditelji obavljaju jedan oblik statičke analize kôda – provjeru sintakse. Također, integrirana razvojna okruženja (engl. IDE) obavljaju statičku analizu kôda. Upotrebom refaktoriranja kôda ili formatiranja dokumenta, obavlja se statička analiza koja kod oblikuje prema svojim pravilima. Naprednije primjene statičke analize uključuju otkrivanje ili predviđanje potencijalnih grešaka u kodu te otkrivanje potencijalnih sigurnosnih propusta.

Turing je dokazao da je nemoguće analizirati program te sa sto postotnom sigurnošću tvrditi da će se program uspješno izvršiti. Takvu garanciju moguće je dobiti samo pokretanjem programa. Budući da alati za statičku analizu rade sa apstraktnim informacijama rezultati zaključeni nad apstraktnim informacijama ne moraju vrijediti i na stvarnom programu. Posljedica navedenog je mogućnost lažno-pozitivnih i lažno-negativnih upozorenja. Primjenom veće razine apstrakcije i boljom implementacijom pravila, smanjuje se broj pogrešnih upozorenja.

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

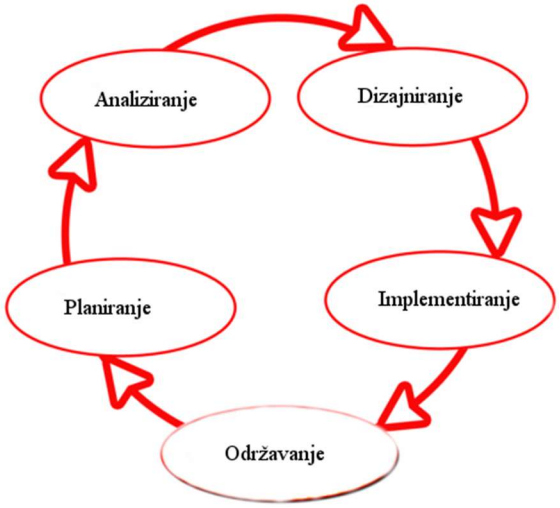
Prema [1], u '70. godinama prošlog stoljeća programeri su došli do zaključka, da za povećanje kvalitete programa, moraju uvesti neku metodu provjere kôda dok je projekt još u razvoju. Rezultat toga su službeni pregledi i inspekcije kôda u kojima skupina ljudi pregledava kod, liniju po liniju, te analizira 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

informacija, ne može se umoriti te svakoj liniji kôda pridodaje jednaku važnost – neće previdjeti nešto. Nažalost, računalo posjeduje veliku manu – ne može razmišljati. Ono može računati, vršiti operacije uspoređivanja, ali ne može zaključiti da li je kod ispravan kao što to može programer. Kako bi umanjili taj nedostatak, programeri su počeli pisati pravila koja upozoravaju na najčešće pogreške koje se mogu dogoditi. Pravila omogućavaju računalu da pronađe ili upozori na potencijalne ili stvarne pogreške u kodu te ih prikaže programeru. Teorija rada alata za statičku analizu objašnjena je u poglavlju 3.

2.1.1. Važnost statičke analize

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

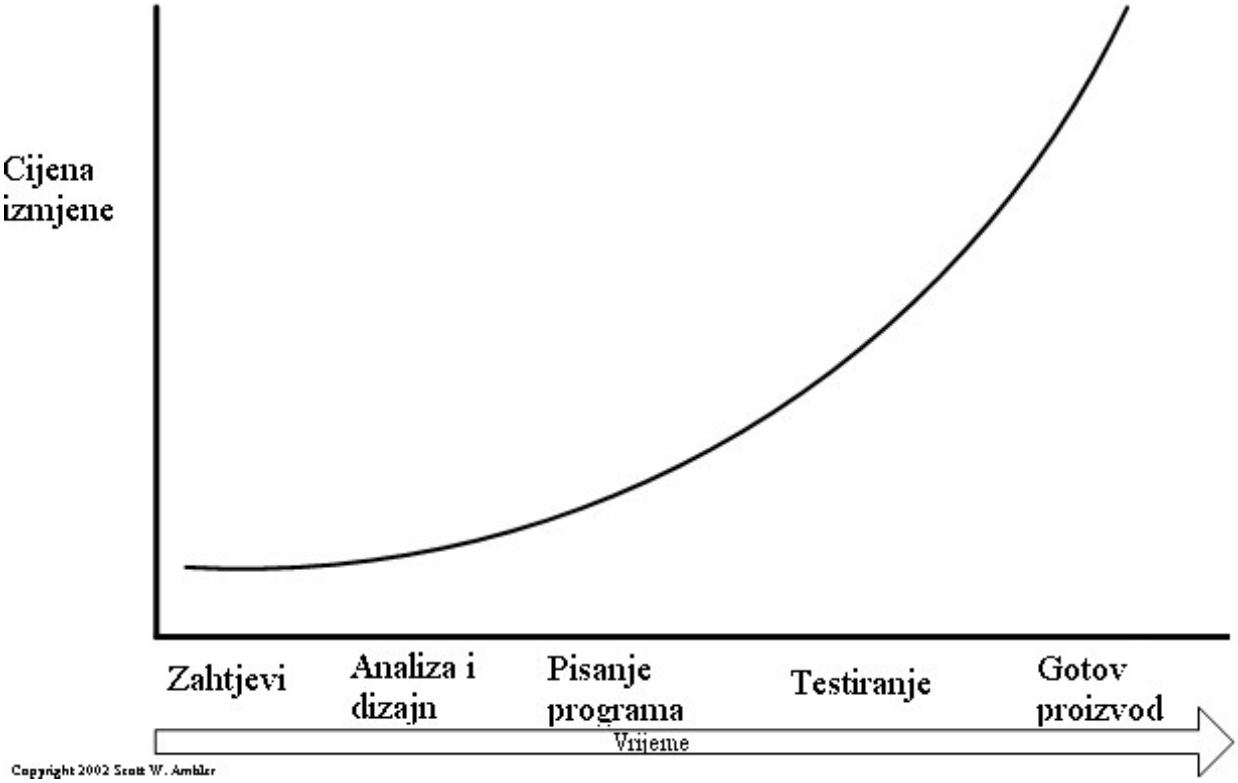


Slika 2.1. Ciklus razvoja programa

Kao što je prikazano Slika 2.1 razvoj svakog programa moguće je podijeliti na korake. Prvi korak je planiranje. Ono obuhvaća okupljanje tima, razvoj ideje te pisanje potrebne dokumentacije. Analiza obuhvaća definiranje potrebnih resursa i mogućnosti programa. Dizajn detaljno opisuje funkcionalnosti programa. U njemu je opisana poslovna logika, izgled aplikacije, te ostali dijelovi bitni za rad aplikacije. Dizajniranje je posljednji korak prije samog pisanja kôda, te je prvo mjesto u kojem statička analiza može pronaći pogrešku. Iako statička analiza može otkriti pogreške u dizajnu, ne može ih otkriti u fazi dizajniranja. Kao što je navedeno u uvodu, za analizu je potrebno posjedovati kod. Kako pisanje koda odvija odvija u fazi implemenetacije ona se smatra početnim i ključnim korakom primjene statičke analize. Posljednji korak ciklusa razvoja označava održavanje aplikacije – pronalaženje pogrešaka te

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 otkanjanja 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.



Slika 2.2 Ovisnost cijene popravke pogreške o vremenu

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 i nedostatke statičke analize lakše je uočiti usporednom statičke analize drugom, srodnom, analizom. Analiza koja najviše odgovara modelu statičke analize je analiza kôda koju obavlja programer. Nažalost, usporedba čovjeka i računala nema prevelikog smisla. Iz te usporedbe moguće je zaključiti da računalo nije sposobno razmišljati kao čovjek te da je čovjek sporiji i podložniji pogreškama. Pogodnija usporedba je između statičke i dinamičke analize kôda. Važno je za napomenuti da je teško usporediti statičku i dinamičku analizu, jer pokrivaju različita područja, tj. komplementarne su.

Prednosti statičke analize su:

- Prema [1], statička analiza može pronaći pogreške u ranom stadiju razvoja, prije nego se program prvi put pokrene. Ova činjenica omogućuje da programer, rano u procesu izrade, dobije povratne informacije o pogreškama ili lošem dizajnu. Dodatna pogodnost ove činjenice je da ranije otklanjanje pogreške višestruko smanjuje cijenu održavanja proizvoda.
- 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 nadogradivost. Ukoliko istraživači

pronađu 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.

- Prema [5], jednostavna je za upotrebu. Za razliku od dinamičke, nije potrebno pisati posebne testove. Dovoljno je odabrati razinu prikaza rezultata te pokrenuti provjeru. Prema [4], uobičajeno da se provjera vrši u određenom ciklusu, npr. jednom dnevno - tijekom pauze. Nakon pauze, programeri pogledaju rezultate te odlučuju o daljnjim koracima.

Nedostaci statičke analize:

- Pomoću statičke analize gotovo je nemoguće otkriti probleme koji nastaju međudjelovanjem s drugim sustavima. Izvor [4] navodi primjer da statička analiza ne može odrediti da li su parametri, koji se primaju od servera, izmijenjeni od treće strane, te ne može otkriti probleme koje takvi parametri mogu prouzročiti. Takav problem moguće je pronaći unutar komponenti koje rade s SQL bazom podataka, poslužiteljem ili uslugom.
- Drugi važan nedostatak statičke analize predstavljaju lažno-pozitivni i lažno-negativni izvještaji. Prema [5], lažno-pozitivni izvještaji predstavljaju dio kôda kojeg je alat za statičku analizu označio kao pogrešku. Ovi izvještaji oduzimaju vrijeme, te maskiraju prave probleme u sustavu. Velika količina lažno-pozitivnih izvješća navodi programera da prestane koristiti alat za statičku analizu čime se povećava nesigurnost programa. Lažno-negativna izvješća predstavljaju ozbiljniji problem. Lažno-pozitivna izvješća su iritantna i smetaju programeru, no lažno-negativna izvješća su prave pogreške u kodu koje alat nije prepoznao što povećava mogućnost da ih programer neće ispraviti.

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

Statičku analizu kôda moguće je napraviti nad izvornim kodom te nad strojnim kodom. Činjenica je da ti kodovi nisu jednaki, iako strojni kod proizlazi iz izvornog kôda. Izvor [4] navodi da izvorni kod sadrži dodatne informacije kao što su komentari, anotacije i sl. Iz strojnog

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

Prema [4], analiza izvornog kôda podrazumijeva analiziranje programa na način kako ga prevoditelj vidi. Pomoću analize izvornog kôda dobiva se veći broj informacija o programu jer izvorni kod sadrži detalje koji pomažu alatu da shvati što se točno događa. Izvor [4] kao primjer navodi sinkronizacijske primitive koje alatu govore da je programer u kodu vodio računa o problemu utrkivanja (npr. kod višenitnog programiranja). Dodatne informacije omogućuju da analiza bude provedena na semantički višoj razini. Prednost ovakvog pristupa je otkrivanje problema koje je teško otkriti analizom strojnog kôda. Izvor [4] kao primjer navodi problema preljeva *integer* spremnika, koji je u strojnom kodu predstavljen kao skup višestrukih *shift* i *add* operacija. Analiza izvornog kôda omogućuje pronalaženje pogrešaka na svim platformama na kojima se program izvodi. Također, analiza izvornog kôda omogućuje jednostavnije praćenje otkrivenog problema do njegovog izvora u kodu.

Prethodno spomenute činjenice predstavljaju prednosti analize nad izvornim kodom. Problem kod analize izvornog kôda predstavlja činjenica da izvorni kod često nije dostupan. Čest je slučaj da program koristi dodatne biblioteke ili se poziva na drugi program. Za utvrđenje ispravnosti programa potrebno je, osim provjere izvornog kôda samog programa, provjeriti i izvorni kod biblioteka ili drugih programa, što najčešće nije moguće. Izvor [4] navodi još jedan problem s analizom izvornog kôda - alat treba u potpunosti razumjeti sve jezike koji se koriste u izvornom kodu. Ukoliko alat ne razumije jezik, vjerojatno će taj dio kôda zanemariti, ne prepoznajući probleme u kodu. Analiza izvornog kôda ne može otkriti probleme koje prevoditelj može uvesti prevođenjem izvornog kôda u strojni.

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. Provođenje 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-pozitivnih 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 objedinjen 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
- Potvrđivanje 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 prijavljivanja 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 prijavljuju većinom su vezani uz čitljivost i sposobnost održavanja kôda. Alati za provjeru stila ne prijavljuju 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 prevođenju prijaviti pogrešku da vrijednosti *"green"* i *"blue"* nisu uzete u obzir.

```
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. Ispravljati pogreške na već započetom 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

Prema [4], alati za razumijevanje programa pomažu programeru razumjeti veću količinu kôda. Ove funkcionalnosti mogu se pronaći unutar integriranih razvojnih okruženja (engl. IDE). Izvor [4] kao primjer funkcionalnosti navodi "pronađi sva mjesta gdje se navedena metoda koristi" ili "pronađi mjesto gdje je varijabla deklarirana. Složenije primjene uključuju restrukturiranje kôda kao što je preimenovanje varijabli ili razdvajanje funkcije na više funkcija. Složeniji alati mogu dati uvid programeru kako program radi. Pomoću obrnutog inženjerstva alat može programeru prikazati kako kod zapravo funkcionira te kako se određeni dio kôda uklapa u cijeli program. Ovo se koristi kada je potrebno razumjeti što određeni dio kôda radi, a pripadajuća dokumentacija nije dostupna.

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 podudaraju. Provjera jednakosti često se koristi pri provjeri sklopovlja dok se rijetko koristi pri provjeri programa. Razlog tome je što programeri 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 privreizbornikh sigurnosnih svojstva. Privremena sigurnosna svojstva određuju slijed 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 programeri mogli provjeriti specifična svojstva za svoj program.

Prema [4], alat za provjeru svojstva se smatra sigurnim ukoliko uvijek prijavi problem, ako problem postoji. Posljedica ovoga je da alati za provjeru svojstva ne mogu imati lažno-negativne izvještaje. Kako bi alat mogao tvrditi da je provjera sigurna, program koji se pregledava mora zadovoljavati određene uvjete. Neki alati ne dozvoljavaju pokazivače na funkcije (engl. function pointers), drugi ne dozvoljavaju rekurziju ili rade s pretpostavkom da dva pokazivača nikad ne pokazuju na istu memorijsku adresu. Pri velikim projektima gotovo je nemoguće zadovoljiti navedene uvjete pa se uvjet da je provjera sigurna izostavlja.

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.

Alati za pronalaženje pogrešaka koriste slične algoritme kao alati za provjeru svojstva. Općenito govoreći, tendencija alata za pronalaženje pogrešaka je imati što manji broj lažno-pozitivnih izvještaja. Posljedica ove odluke je veći broj lažno-negativnih izvještaja. Idealan alat za pronalaženje pogrešaka je siguran u rezultat te vjeran primjeru. Drugim riječima, idealan alat za pronalaženje pogrešaka ne posjeduje lažno-negativne izvještaje te se sve prijavljene pogreške mogu poistovjetiti s primjerom opisanim u pravilu.

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.

Moderni alati su kombinacija alata za provjeru svojstava i alata za traženje pogrešaka. Alati za sigurnosnu provjeru koriste provjeru svojstava jer se velik broj sigurnosnih svojstava mogu izraziti kao programska svojstva. Izvor [4], kao primjer navodi provjeru preljeva spremnika. Pomoću provjere svojstava jednostavno je provjeriti da li će se preljev dogoditi. Potrebno je provjeriti da program ne pristupa memorijskoj lokaciji koja je izvan podjeljene memorije. Od alata za pronalaženje pogrešaka preuzeto je svojstvo pronalaženja već prije otkrivenih pogrešaka. Ovo svojstvo omogućava alatu da pronađe pogreške koje programer ponavlja jer nije svjestan sigurnosnog propusta. Alati za sigurnosnu provjeru ne mogu si priuštiti smanjenje broja lažno-pozitivnih izvještaja na račun više lažno-negativnih izvještaja. Nasuprot, alati za sigurnosnu provjeru imaju veći broj lažno-pozitivnih izvještaja kako bi bili sigurni da nemaju lažno-

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

Mjerne jedinice veličine koda prikazuju kvantitativne informacije o statistici koda. Primjer mjerne jedinice veličine koda je mjera broj linija koda (engl. Lines of code – LOC) . Mjerne jedinice veličine koda nisu detaljnije opisane u radu jer nemaju veliko značenje za provođenje statičke analize. U nastavku su opisane mjerne jedinice složenosti koda.

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:

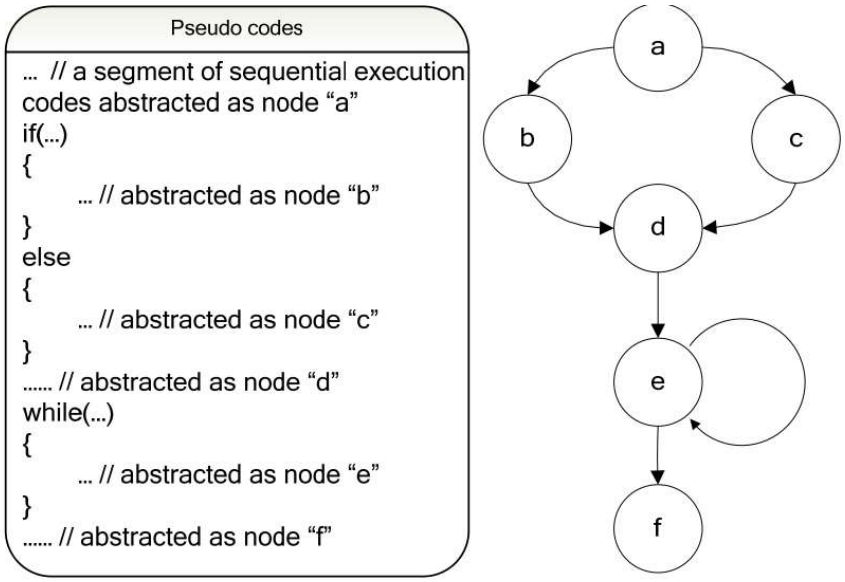
- Osnovna složenost
- Odabrana složenost (engl. selecting complexity)
- Slučajna složenost (engl. Incidental complexity)

Osnovna složenost određena je problemima koje program pokušava riješiti. Odabrana složenost određena je programskim jezikom te poteškoćama modeliranja i dizajniranja programa. Slučajna složenost određena je kvalitetom izvedbe programa, odnosno programerom. Najpoznatije mjere složenosti kôda su McCabe ciklomatska složenost te Halsteadova mjera složenosti. Današnji alati koriste ove dvije mjere zajedno te pomoću njih tvore vlastite jedinice koje opisuju neka specifična ponašanja programa.

McCabe ciklomatska složenost

Prema [6], ciklomatska složenost (engl. Cyclomatic Complexity Measures - CCM) smatra se najosnovnijom i najvažnijom mjernom jedinicom. Većina ostalih mjernih jedinca 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 graf 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 operanda. 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

Za izračun spomenih veličina HCM koristi parametre kao što su broj jedinstvenih operacija i operanda te ukupne brojeve operatora i operanda. Proces izračuna pojedinih veličina opisan je u [6]. Prednost HCM-a je što se jednostavno računa. Razlog tome je što HCM ne treba složenu analizu programske logike. Također, prednost je što analiza nije ograničena programskim jezikom te što može predvidjeti gustoću pogrešaka. Nedostatak HCM-a proizlazi iz činjenice da su izračuni temeljeni samo na složenosti toka podataka dok je kontrola toka zanemarena.

3. TEORIJA STATIČKE ANALIZE

U prethodnom poglavlju spomenute su poteškoće s kojima se računalo susreće pri analiziranju kôda. Kako bi računalo moglo otkriti problem ono mora razumjeti programski kod, kao što ga programer razumije. Također računalo mora razumjeti probleme koji se pojavljuju u kodu. Razumijevanje problema definirano je pravilima koja pišu programeri. Pravila su ograničena razumijevanjem kôda jer se oba sustava moraju bazirati na istom modelu. Postoji više modela koji se koriste pri statičkoj analizi te je svaki ima svoje prednosti i nedostatke. Općenito govoreći, alat za statičku analizu je napredniji i bolji ukoliko koristi više modela.

3.1. Programski modeli

Prvi korak pri izradi analize je je pretvorba kôda u programski model, u strukturu podataka koja predstavlja kod. Programski 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)

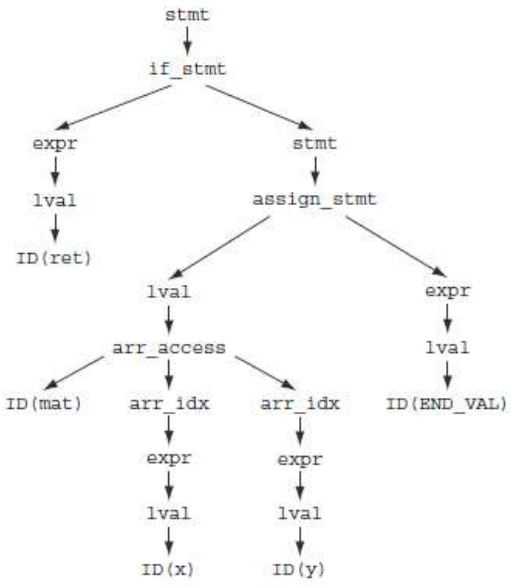
Jezični parser koristi kontekstno neovisnu gramatiku (engl. Context-Free Grammar - CFG) kako bi usporedio niz oznaka. Gramatika se sastoji od skupa produkcija koje opisuju simbole u jeziku. Primjer 3.2, preuzet iz [4], prikazuje skupinu produkcija koje su potrebne za parsanje testnog niza oznaka.

```
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 rasčlanjivanja.

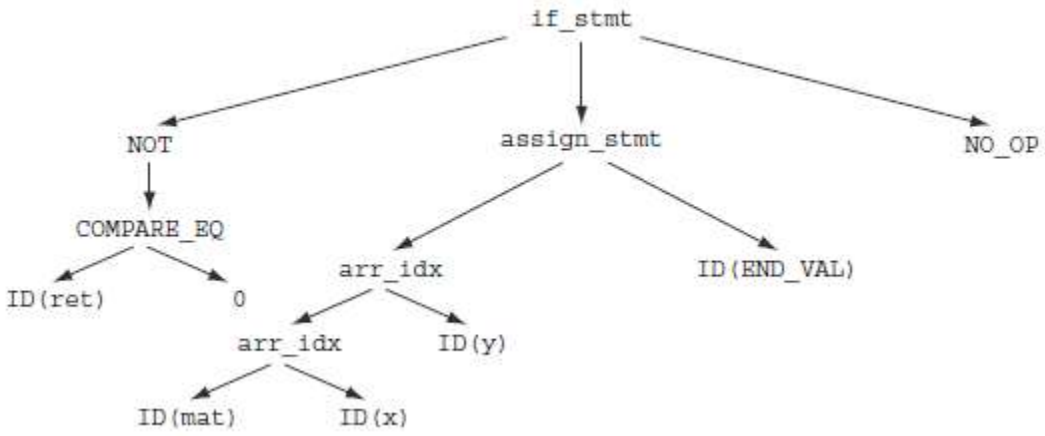
Slika 3.1 preuzeta iz [4], prikazuje stablo raščlanjivanja. Pomoću stabla parsiranja omogućava provjeru srednje složenih problema. Primjer provjere koju je moguće obaviti pomoću stabla parsiranja je analiza stila. Stablo parsiranja odgovara za stilsku provjeru jer je njegov opis najbliži izvornom kodu. Provođenje složene analize nad stablom parsiranja nije poželjno iz više razloga. Čvorovi stabla nastali su iz produkcijskih pravila koja često dodaju simbole kako bi olakšali parsanje i spriječili dvosmislenost, kompliciraući time razumijevanje stabla.



Slika 3.1. Stablo raščlanjivanja

3.1.3. Apstraktna sintaksa

Složene analize poželjno je raditi nad generaliziranim modelom koji ne sadrži detalje dobivene produkcijskim pravilima. Model koji posjeduje generaliziran opis kôda naziva se apstraktno sintaksno stablo (engl. Abstract syntax tree - AST). Model apstraktnog sintaksnog stabla je rezultat povezivanja kôda za generiranje stabla sa produkcijskim pravilima. Slika 3.2, preuzeta iz [4], prikazuje AST za Primjer 3.2. Raščlanjivanje .



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“* usporedbe se izričito uspoređuje s nulom.

Ovisno o potrebi, AST može sadržavati više ograničenih konstrukcija nego izvorni jezik. Izvor [4], za primjer navodi pretvaranje poziva metode u funkcijski poziv ili pretvaranje *for* i *do* petlje u *while* petlju. Ova tehnika pojednostavljivanja programa se naziva smanjivanje (engl. lowering). Bliski jezici, kao što su C I C++, mogu se pojednostaviti u isto AST stablo. Kod ovakvog pojednostavljivanja treba voditi računa o mogućem izobličenju programerove namjere. Jezici koji imaju sličnu sintaksu, kao što su C++ i Java, mogu imati zajedničke AST čvorove ali sigurno posjeduju AST čvorove koji su specifični za njihov jezik.

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

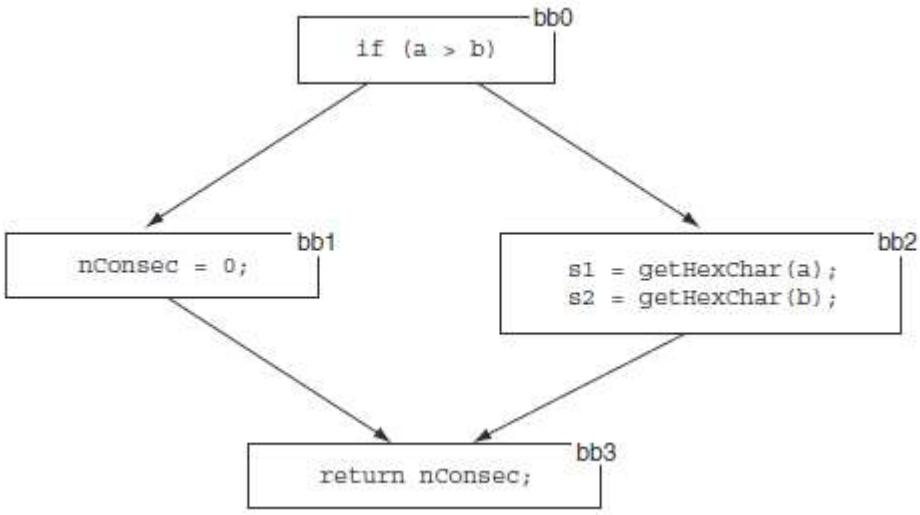
Brojni algoritmi koriste neku vrstu analize toka kako bi otkrili probleme. Postoji više vrsta analize toka, svaka analiza ima svoje specifičnosti, prednosti i nedostatke. Zajedničko svim analizama je da svoje analize baziraju nad AST-om. Rezultat analize toka je graf.

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.

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

Primjer 3.3. Pseudo kod za graf kontrole toka



Slika 3.3. Graf kontrole toka za primjer 3.3

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.

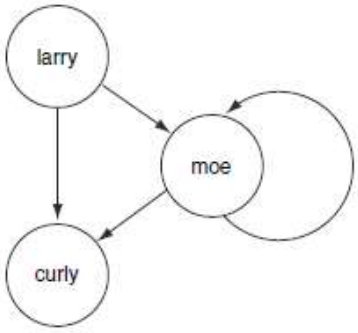
Graf pozivanja (engl. call graph)

Graf pozivanja predstavlja kontrolu toka između funkcija ili metoda. Ukoliko ne postoje pokazivači ili virtualne metode konstrukcija grafa pozivanja svodi se na traženje identifikatora funkcije u drugim funkcijama. Čvorovi grafa predstavljaju funkcije. Strelice u grafu predstavljaju mogućnost funkcije da pozove drugu. Primjer 3.4, preuzet iz [4], prikazuje tri funkcije koje se međusobno pozivaju. Kod je iskorišten za izradu grafa pozivanja. Slika 3.4 prikazuje graf pozivanja za Primjer 3.4. Graf pozivanja ne može analizirati module koji se dodaju dinamički, stoga treba voditi računa o mogućnosti da je graf nepotpun.

```
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

Cilj analize toka podataka je odrediti kako podaci putuju kroz program. Analiza toka podataka obavlja se nad kontrolnim grafom neke funkcije. Prolaskom kroz kontrolni graf, bilježe se lokacije na kojima podaci nastaju te gdje se koriste. Uobičajeno je da se pri izradi analize toka podataka funkcija prebaci u statični jednoznačni oblik (engl. Static Single Assignment - SSA). SSA oblik zahtjeva da se svaka varijabla samo jednom postavi te da je prije postavljanja deklarirana. Naknadno postavljanje tretira se kao nova varijabla. SSA oblik je pogodan za analizu toka podataka jer omogućava jednostavno određivanje podrijetla vrijednosti varijable. Primjer 3.5, preuzet iz [4], prikazuje pretvorbu izvornog kôda u SSA oblik. Vidljivo je kako SSA oblik pravi nove varijable za svaku novu vrijednost varijable.

```
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 koda u statički jednoznačni oblik

3.2. Algoritmi za provođenje statičke analize

U prethodnom poglavlju opisani su modeli čijim je korištenjem moguće otkriti neke tipove programskih pogrešaka. Prema [4] problem modela je što su sposobni prepoznati mogućnost pogreške, no ne mogu utvrditi da li će se pogreška stvarno pojaviti. Kako bi se smanjio broj netočnih pronalazaka, pri analizi se koriste kontekstno ovisi algoritmi. Kontekstno ovisni algoritmi mogu odrediti okolnosti i uvjete u kojima se kod izvršava, te mogu bolje procijeniti postoji li mogućnost da se neki problem dogodi. Primjer kontekstno osjetljivog algoritma je algoritam koji može odrediti da li će se, u uvjetima pri kojim se kod izvodi, dogoditi preljev spremnika.

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

Apstraktna interpretacija je teorija aproksimiranja matematičkih struktura. Apstraktna interpretacija pripada skupini lokalnih analiza. Prema [5], apstraktna interpretacija temelji se na zamjeni složenih elemenata sa manje detaljnom apstrakcijom kako bi bilo moguće izračunati svojstva programa. Pomoću apstraktne interpretacije moguće je otkriti *runtime* probleme, kao što su dijeljenje s nulom, preljev spremnika itd. Primjer realizacije apstraktne interpretacije je neovisna analiza toka (engl. flow insensitive) programa. Pri neovisnoj analizi toka, redoslijed izvršavanja nije bitan se te s tim jamči da su svi slučajevi izvršavanja uzeti u obzir. Prednost apstraktne interpretacije je uklanjanje potrebe za složenim analizama kontrole toka. Nedostatak apstraktne interpretacije je što u analizu uključuje redoslijed izvođenja koji se nikada neće dogoditi. Za optimalan rezultat poželjno je u analizu uključiti ovisnu analizu toka kako bi se smanjio utjecaj lažno pozitivnih izvještaja.

3.2.2. Transformacija suda (predikatna transformacija)

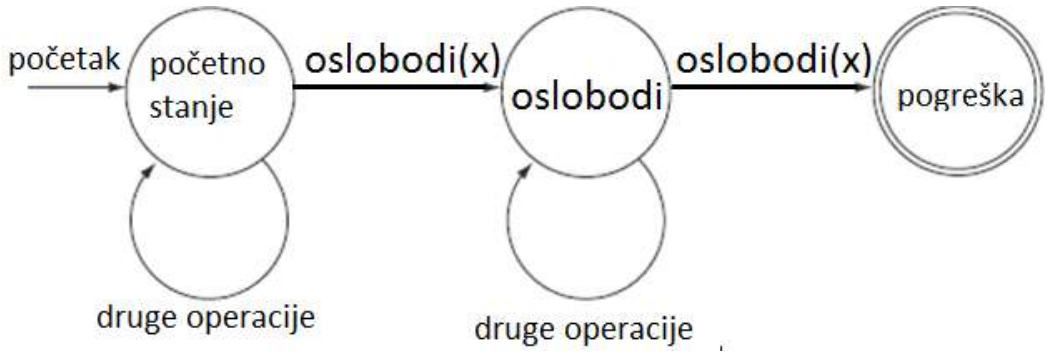
Sud u matematici, predstavlja tvrdnju koja mora nešto tvrditi, odnosno sud mora biti istinit ili lažan. Transformacija suda opisuje smisao programa kroz transformaciju krajnjeg stanja u početno stanje. Važan primjer funkcije koja rabi transformaciju suda je funkcija najslabijeg

početnog stanja. Najslabije početno stanje programa određuje se pronalaskom najmanjeg skupa uvjeta koji moraju biti proslijeđeni od pozivatelja programa kako bi se došlo u krajnje stanje.

3.2.3. Provjera modela sustava

Provjera modela je tehnika provjere valjanosti svojstva sustava koja imaju konačan broj stanja. Provjera modela omogućava povjeru dva opisa svojstva koji funkcionalno nisu jednaki. Tehnika provjere modela pripada skupini lokalnih analiza. Prvi korak provjere modela je pretvorba programa u automat s konačnim brojem stanja. Nad automatom, radi se provjera da li će, za određeno svojstvo sustava, automat preći u stanje pogreške. Ukoliko automat pređe u stanje pogreške, testirano svojstvo se smatra ranjivošću sustava. Izvor [4], kao primjer problema kojeg provjera modela može otkriti, navodi problem višestrukog oslobađanja memorije.

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 oslobođene memorije pokuša ponovo osloboditi memorija, prelazi se u stanje pogreške.



Slika 3.5. Automat s konačnim brojem stanja za problem višestrukog broja oslobađanja memorije

3.3. Pravila koja koriste alati za statičku analizu

Pravila označavaju što alat za statičku analizu treba prijaviti korisniku kao pogrešku. Pravila imaju jednaku važnost kao algoritmi. Algoritmi za statičku analizu obavljaju većinu posla ali pravila odlučuju što je pogreška. Pravila je moguće koristiti i za definiranje ponašanja programa koje nije obuhvaćeno samim programom kao što su biblioteke. Poželjno je da se pravila nalaze u vanjskom dijelu alata kako bi se pravila mogla dodavati te izmjenjivati bez modificiranja alata. Ovisno o algoritmu i tehnikama otkrivanja pogrešaka, pravila mogu raditi samostalno i u paru. Pri pisanju pravila važno je imati na umu da količina pravila ne jamči i kvalitetnije rezultate. Najčešće, veća količina pravila poveća broj lažno pozitivnih prijava te maskira prave probleme.

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 podijeli 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

Pronalaženje pogreške samo je dio posla koji alat za statičku analizu mora obaviti. Pronađene pogreške su beskorisne ukoliko ne postoji dobar način da ih se prezentira. Rezultat je potrebno prikazati tako da se može ocijeniti da li je problem kritičan, 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 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 pridaje 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 sintakse 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

Grupiranje i sortiranje rezultata

Grupiranjem i sortiranjem rezultata omogućuje se programeru da ukloni veliki broj neželjenih rezultata bez da pregleda svaki problem pojedinačno. Ovo je korisno jer omogućava isključivanje rezultata iz datoteka za koje programer zna da su valjane. Također je bitno da alat odredi prioritet problemima te ih tako sortirane prikaže programeru. Sortiranje prema prioritetu otklanja mogućnost zapostavljanja kritičnih problema te omogućava brže rješavanje. Alati za statičku analizu koriste dva podatka za određivanje prioriteta – ozbiljnost problema te povjerenje u rezultat. Ozbiljnost problema predstavlja „težinu“ problema. Povjerenje u rezultat predstavlja aproksimiranu vrijednost da je rezultat ispravan.

Otklanjanje neželjenih rezultata

Kako je napisano u uvodu u poglavlje, nije jednostavno pregledavati izvještaj koji sadrži gomilu jednakih rezultata. Programer će tolerirati takav problem par puta ali uzastopno pojavljivanje istog rezultata natjerati će programera da prestane koristiti alat. Zbog toga dobar alat za statičku analizu kôda mora posjedovati mehanizme za suzbijanje rezultata koji nisu poželjni. Ukoliko je mehanizam dobro napravljen, informaciju o suzbijanju će koristiti pri idućim analizama te će na taj način olakšati pregledavanje rezultata. Suzbijanje rezultata moguće je izvesti na više načina. Prvi način je dodavanje anotacija. Problem s ovim pristupom je što osoba koja provjerava kod često nema pravo i mijenjati izvorni kod tj. nema mogućnost dodati anotacije. Bolji pristup je generiranje oznaka koje sadrže imena funkcija i varijabli, bitne dijelove grafa kontrole toka te oznaku pravila kojeg suzbijaju.

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

napisan tako da sadrži primjer pogrešnog kôda s opisom pogreške kako bi programer lakše shvatio u čemu je problem. Opis utjecaja i važnosti sadrži informacije o mogućim posljedicama neispravljanja pogreške. Ovaj dio opisa treba objasniti programeru ozbiljnost problema te ga uvjeriti da ispravi problem. Naposljetku, opis treba sadržavati postupak kako provjeriti da li kod uistinu sadrži tu pogrešku.

4. OPIS ALATA ZA STATIČKU ANALIZU PROGRAMA

U ovom poglavlju opisani su korišteni alati za statičku analizu programa. Odabrani su FindBugs te PMD. Navedeni alati su odabrani zbog svoje popularnosti među razvijateljima programa te zbog svojih mogućnosti. Važno je napomenuti da navedeni alati nisu jedini alati za statičku analizu kôda. Postoje komercijalni alati koji obavljaju složenije, specifičnije i preciznije analize ali oni se ne uklapaju u ideologiju rada te s toga nisu uzeti u obzir. Osim alata za statičku analizu koda opisani su alati za produkciju i praćenje projekta - Jenkins i SonarQube. Jenkins posjeduje mogućnost integracije alata za statičku analizu, dok SonarQube sadrži alat za statičku analizu.

4.1. Alat FindBugs

FindBugs je alat otvorenog kôda koji omogućava statičku analizu Java programa. Razvijen je na sveučilištu Maryland. Prema [7], prva namjena FindBugsa bila je pronalaženje učestalih pogrešaka koje rade studenti. Burg, profesor zadužen za FindBugsov nastanak, je na temelju pogrešaka koje rade studenti napisao nekoliko pravila kako bi lakše pregledavao predane radove i kako bi shvatio što studentima nije jasno. Analizirajući izvorni kod samog JDK došao je do zaključka da i iskusni programeri rade početničke pogreške. FindBugs alat pisan je u Javi. U trenutku pisanja rada aktualna inačica FindBugs-a je 3.0. Inačica 3.0. podržava Javu 1.8. te za ispravan rad zahtijeva minimalnu inačicu JRE/JDK 1.7.

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 pronalazak 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.

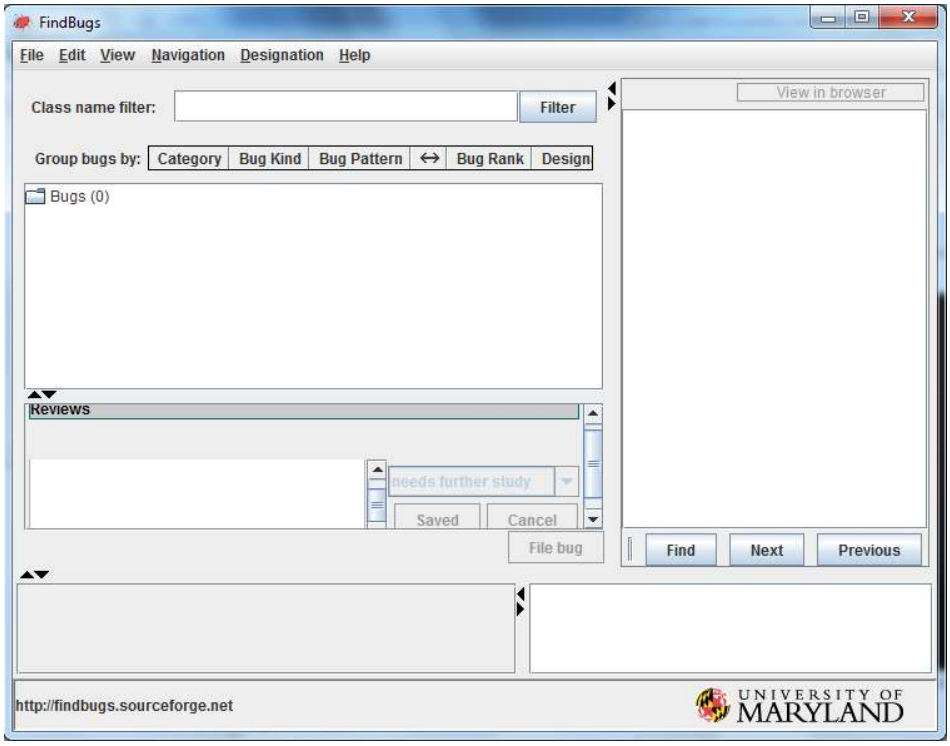
Općenito govoreći FindBugs obrasce pogrešaka dijeli u slijedeće kategorije :

- 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.



Slika 4.1 Grafičko sučelje 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 je alat otvorenog kôda koji omogućava statičku analizu Java programa. Značenje kratice PMD nije jasno definirano. Prema [8], autori tvrde da su ime PMD odabrali jer im se svidjela kombinacija slova. Definiranje značenja prepustili su zajednici koja je smislila par prijedloga kao što su Project Mess Detector, Project Meets Deadline ili Program of Mass Destruction.

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].

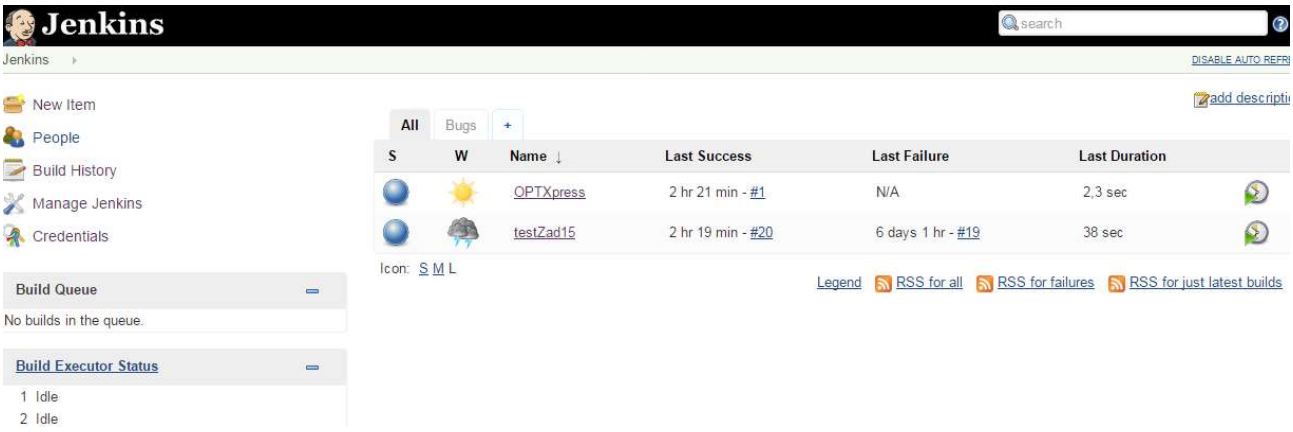
4.3. Alat Jenkins

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.

Jenkins je nastao iz alata Hudson. Oracleovom kupnjom Sun-a, izvornog razvijatelja Hudsona, skupina programera odlučila je nastaviti razvoj na alatu te ga i dalje izdavati kao alat otvorenog kôda. Kako ne bi imali problema, napustili su ime Hudson te alat nazvali Jenkins. Prema [9] Jenkins nudi puno više od osnovne ideje kontinuirane integracije. Zahvaljujući otvorenosti kôda, Jenkins je okupio veliku zajednicu koja razvija dodatke koji proširuju Jenksinsovu funkcionalnost. Zahvaljujući proširenjima, pomoću Jenkinsa je moguće obavljati statičku analizu, integrirati ga s drugim alatima te ga podesiti da obavlja razne zadatke ovisno o događaju. Druga prednost Jenkinsa je što podržava više programskih jezika kao što su .NET, Ruby, Groovy, PHP te Javu. Nadalje, Jenkins je jednostavan za korištenje. Sama instalacija na Windows platformi je automatizirana, te omogućuje korištenje Jenksinsa par minuta nakon preuzimanja. Pristup Jenkinsu omogućen je pomoću preglednika web stranica. Sučelje je jednostavno, intuitivno te bogato mogućnostima. Pomoću sučelja moguće je promijeniti postavke alata za izgradnju programa, instalirati dodatke te napraviti nove projekte. Dodatne informacije o Jenkinsu-u moguće je pronaći u izvoru [9].

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



Slika 4.2. Sučelje alata Jenkins

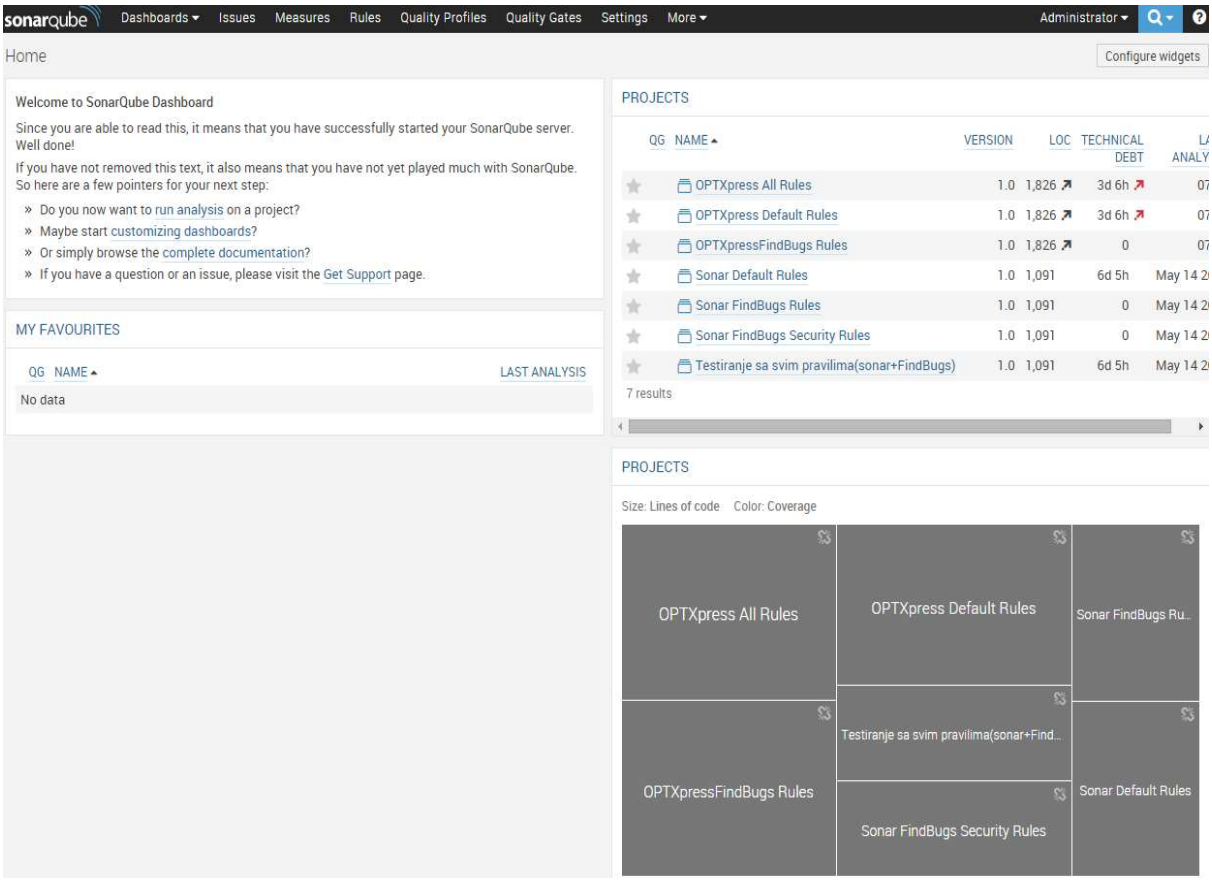
Instalaciju dodataka moguće je obaviti iz Jenkinsovog sučelja ili je moguće dodatke preuzeti s [10] te ih kopirati unutar *plugins* direktorija.

4.4. Alat SonarQube

SonarQube je alat otvorenog kôda namijenjen kontinuiranom testiranju. Kontinuirano testiranje pripada upravljanju kvalitetom kôda. Cilj kontinuiranog testiranja je povećanje kvalitete kôda tijekom razvoja. Temeljni koncept kontinuiranog testiranja je rano otkrivanje pogrešaka te popravak dok je cijena popravka niska. Prema [11] SonarQube posjeduje alate za statičku analizu kôda, prijavljivanje pogrešaka, alate za praćenje pogrešaka do izvora (engl. bug hunting) te mogućnost vraćanja prošlih inačica kôda. Funkcionalnost SonarQube-a moguće je dodatno unaprijediti pomoću proširenja koje zajednica razvija. Također, moguće ga je integrirati s Jenkinsom te na taj način upotpuniti priču o kontinuiranoj integraciji.

SonarQube je prilagodljiv i jednostavan za korištenje sa jednako jednostavnom instalacijom. Potrebno je preuzetu datoteku raspakirati na željenu lokaciju, te u bin direktoriju pokrenuti odgovarajuću datoteku, ovisno o operacijskom sustavu. Također, moguće je i instalirati servis koji će pokretati SonarQube pri pokretanju Windowsa. Osim SonarQube-a za analizu je potrebno koristi alat koji će obavljati analizu. Ukoliko se alat koristi lokalno moguće je postaviti SonarRunner, Sonarov alat koji obavlja analize.. Osim SonarRunner-a, moguće je koristiti alate za izgradnju projekta kao što su Ant ili Maven. Ukoliko se koriste alati za izgradnju projekta sve što je potrebno je postaviti definirani zadatak unutar *build.xml* datoteke (za ant) ili *pom.xml* (za maven). Slika 4.3 prikazuje SonarQube-ovu kontrolnu ploču. Prvi vrhu se nalazi izbornik koji omogućuje pristup svim dijelovima SonarQube-a. Glavni dio ekrana nudi informacije o svim projektima. S desne strane nalaze se kratke informacije o projektima, kao što je trenutno stanje, tehnički dug, broj linija kôda itd.

SonarQube analizu radi slično kao i ostali opisani alati. Pomoću pravila SonarQube pretražuje da li kod sadrži potencijalne pogreške. Za razliku od ostalih alata SonarQube može analizirati izvorni kod i strojni kod. Nadalje, SonarQube je moguće proširiti FindBugsom i PMD-om iako to nije nužno potrebno jer su SonarQubeova pravila pisana prema uzoru na FindBugsova i PMD-ova pravila. Pravila pojedinog alata su objedinjena u profile kvalitete. SonarQube posjeduje mogućnost pisanja vlastitih pravila te kombiniranja pravila od različitih alata u jednu cjelinu. Ovo je korisno ukoliko se želi provjeriti projekt sa više alata jer SonarQube može imati samo jedan aktivan profil kvalitete.



Slika 4.3. SonarQube sučelje

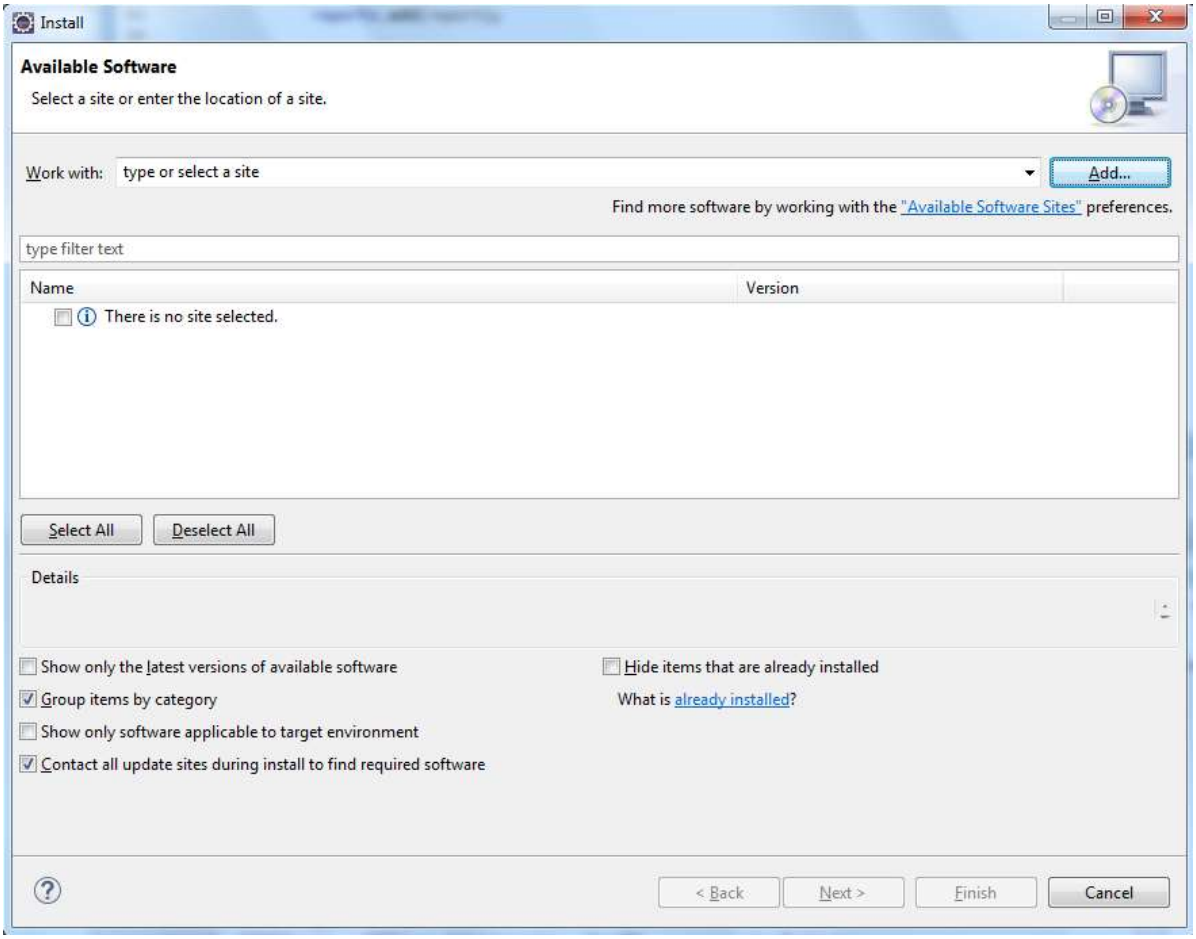
5. PROVOĐENJE TESTIRANJA NA KODU

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

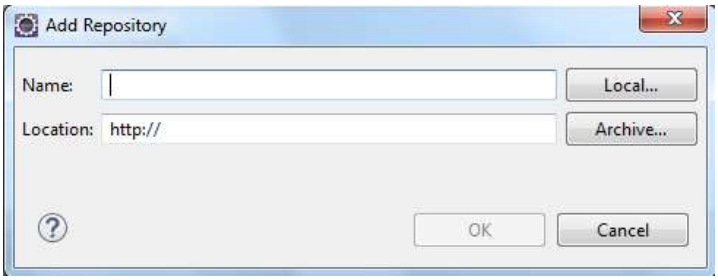
Integracija alata za statičku analizu omogućena je korištenjem dodataka. Za potrebe testiranja korišteni su dodaci PMD i FindBugs koje je potrebno instalirati pomoću Eclipsovog alata *Install New Software...* (izbornik Help).

Slika 5.1 prikazuje prozor za instalaciju dodataka. Potrebno je pritisnuti *Add…*. Klikom na *Add…* otvara se prozor za dodavanje repozitorija.



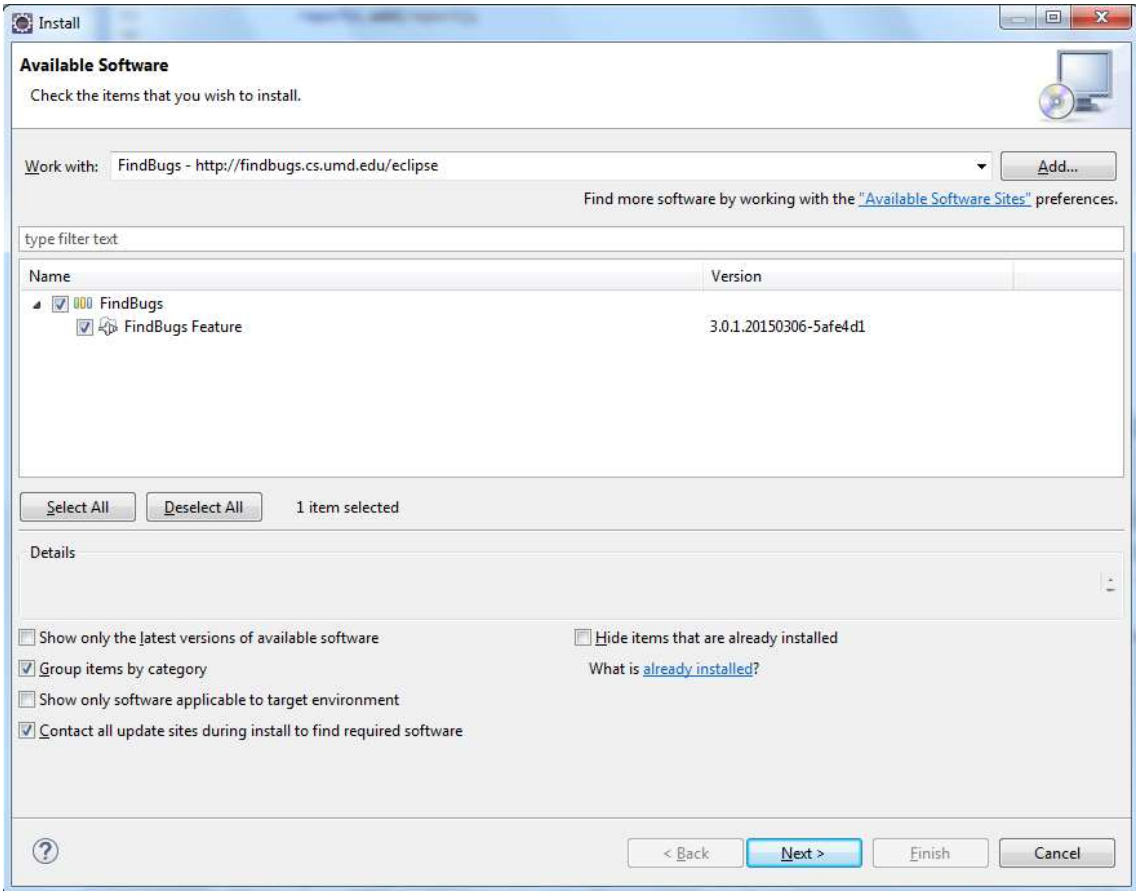
Slika 5.1. Prozor alata Install New Software

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



Slika 5.2. Dodavanje repozitorija dodatka

Nakon dodavanja repozitorija potrebno je unutar polja *„Work with:“* odabrati novo dodane repozitorije. Slika 5.3 prikazuje izgled prozora za instalaciju nakon što se repozitorij učita. Unutar navedenog prozora nalazi se popis dodataka koje je moguće instalirati. Nakon ovog koraka potrebno je prihvatiti licencne ugovore te će se dodaci instalirati. Nakon instalacije poželjno je ponovo pokrenuti Eclipse.



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 prevođenju 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

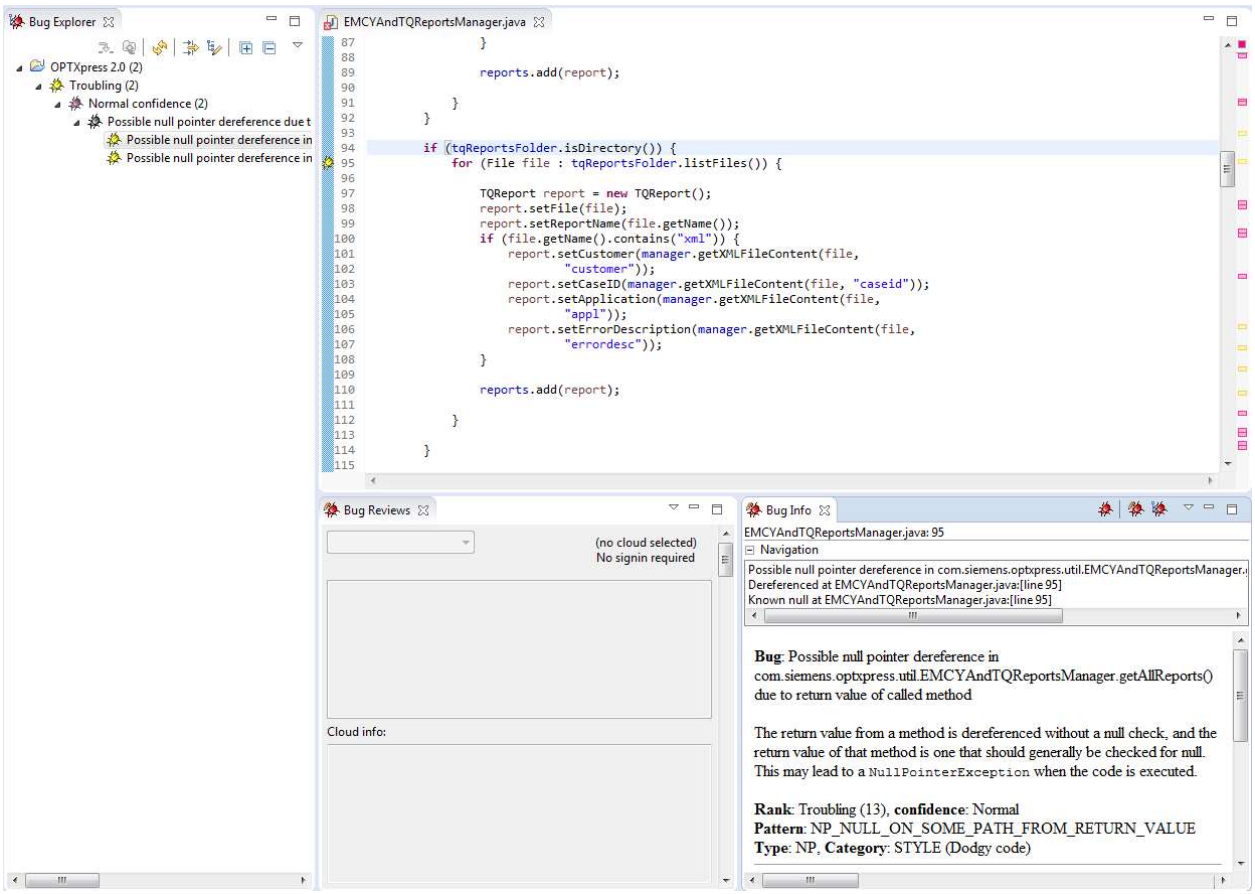
Pokretanje PMD-a obavlja se također pomoću izbornika. Potrebno je desnim klikom na projekt odabrati PMD izbornik te kliknuti na *Check Code*.

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

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

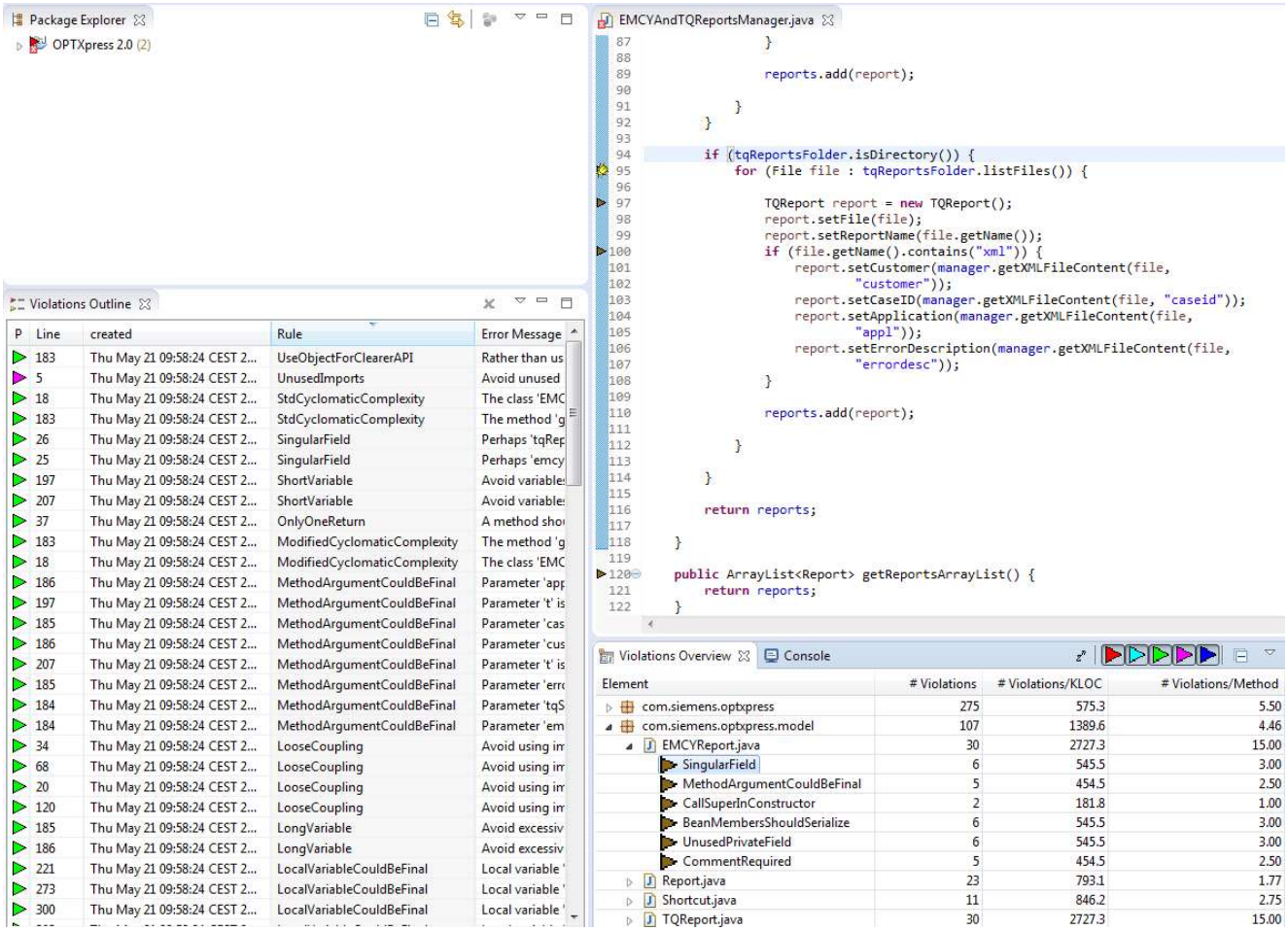
Slika 5.6 prikazuje FindBugs perspektivu.



Slika 5.6. FindBugs perspektiva

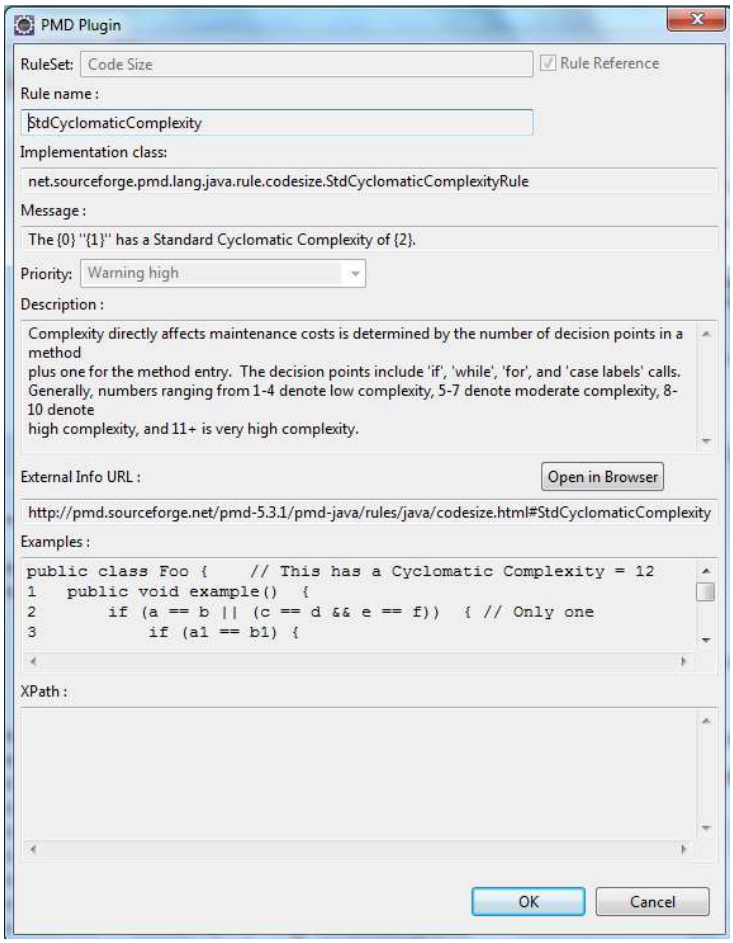
Perspektiva se pokreće pokretanjem FindBugs-a. Perspektivu čine tri dijela. Na lijevoj strani nalazi se *Bug Explorer*, popis pogrešaka koje su poredane po složenosti. Ostatak perspektive podijeljen je u dva dijela. Gornji dio je uređivač kôda. Donji dio sadrži opis pogreške. U opisu pogreške detaljno je opisana pogreška te je prikazan primjer. Perspektiva je podijeljena na četiri dijela. Lijevu stranu čine *Package Explorer* i *Violations Outline. Package Explorer* omogućuje pregledavanje sadržaja paketa. *Violations Outline* omogućuje kratak pregled pronađenih pogrešaka. Osim linije pogreške, pregled sadrži pravilo koje je otkrilo pogrešku te opis pogreške. Detaljniji opis moguće je dobiti desnim klikom na pogrešku te odabirom *Show details…* Desnu stranu čine uređivač kôda te *Violations Overview*. *Violations Overview* omogućuje pregled pogrešaka po klasama. Osim pregleda pogrešaka, *Violations Overview* nudi i mjerne jedinice vezane uz klase i pojedinu pogrešku.

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.



Slika 5.8. PMD- detaljan pregled pogreške

5.2. Podešavanje Jenksinsa

Integracija alata za statičku analizu omogućena je korištenjem dodataka. Za potrebe testiranja korišteni su dodaci PMD i FindBugs. Instalaciju dodataka moguće je obaviti iz Jenkinsovog sučelja ili je moguće dodatke preuzeti s [10] te ih kopirati unutar *plugins* direktorija.

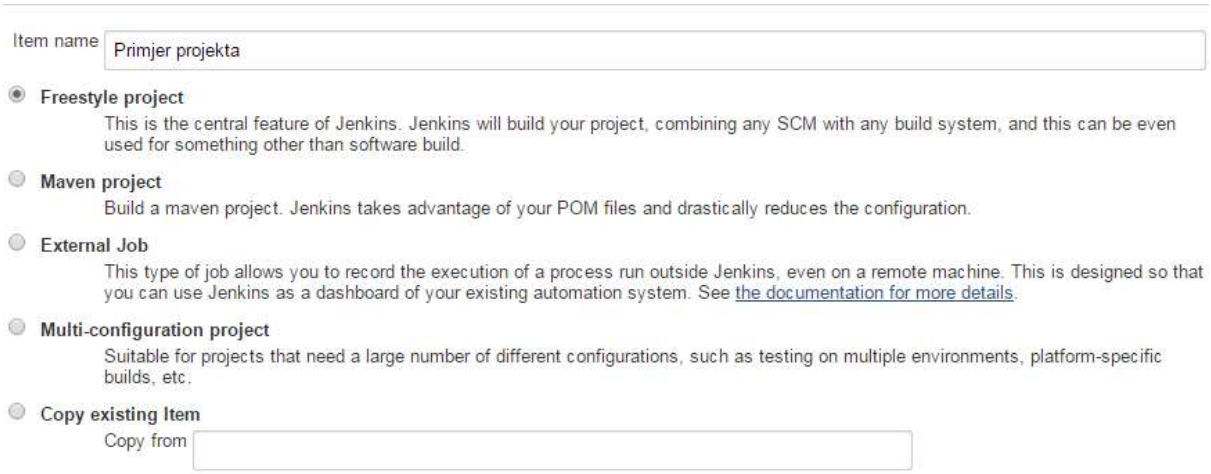
Za potrebe rada korišteni su slijedeći dodaci :

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

Nakon instaliranja dodataka potrebno je ponovno pokrenuti Jenkins. Najjednostavniji način ponovnog pokretanja je dodati */restart* na kraj poveznice.

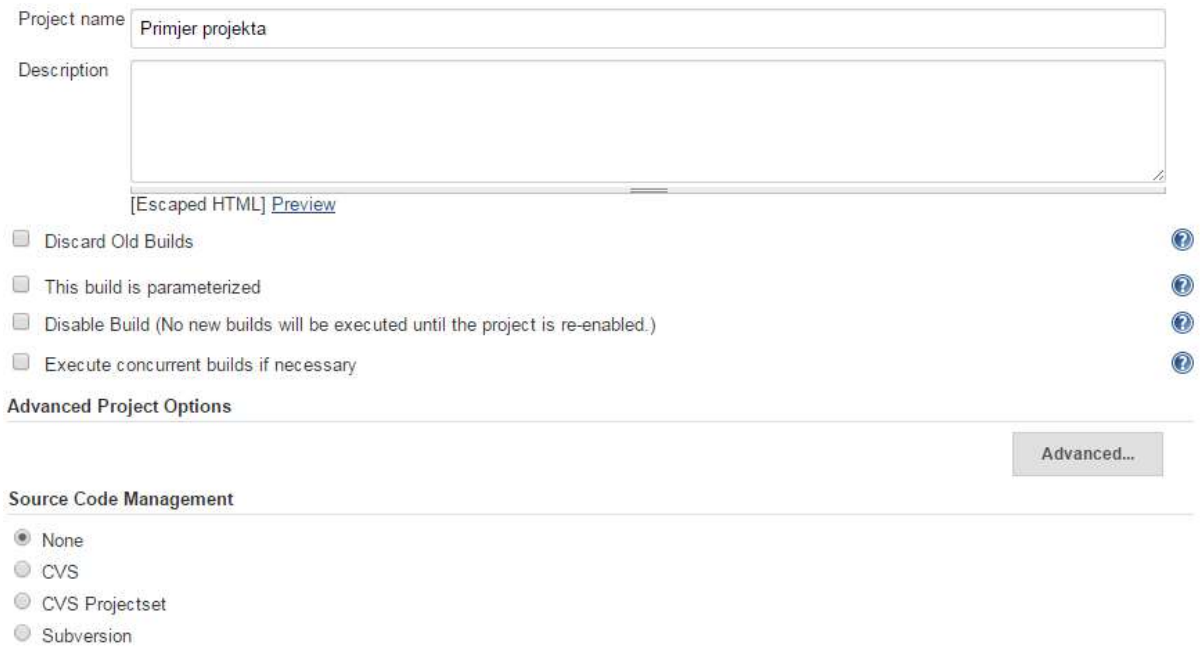
Idući korak podešavanja Jenkinsa je stvaranje projekta. Opcija za stvaranje novog prozora je *New item*. Moguće joj je pristupiti s početne stranice Jenkinsa, tj. s komandne ploče.

Slika 5.9 prikazuje stranicu za stvaranje Jenkins projekta. Pri izradi Jenkins projekta, moguće je birati između par tipova projekta. Odabir tipa projekta ovisi o stvarnom projektu za kojeg se radi kontinuirana integracija. Uobičajeno je napraviti *Freestyle* projekt jer pruža velik izbor podešavanja ili napraviti Maven projekt ukoliko se koristi Maven alat. Maven projekt je jednostavniji za korištenje jer nije nužno podešavati većinu parametara. Zbog potrebe spajanja sa sustavom za upravljanje kodom napravljen je Freestyle projekt.



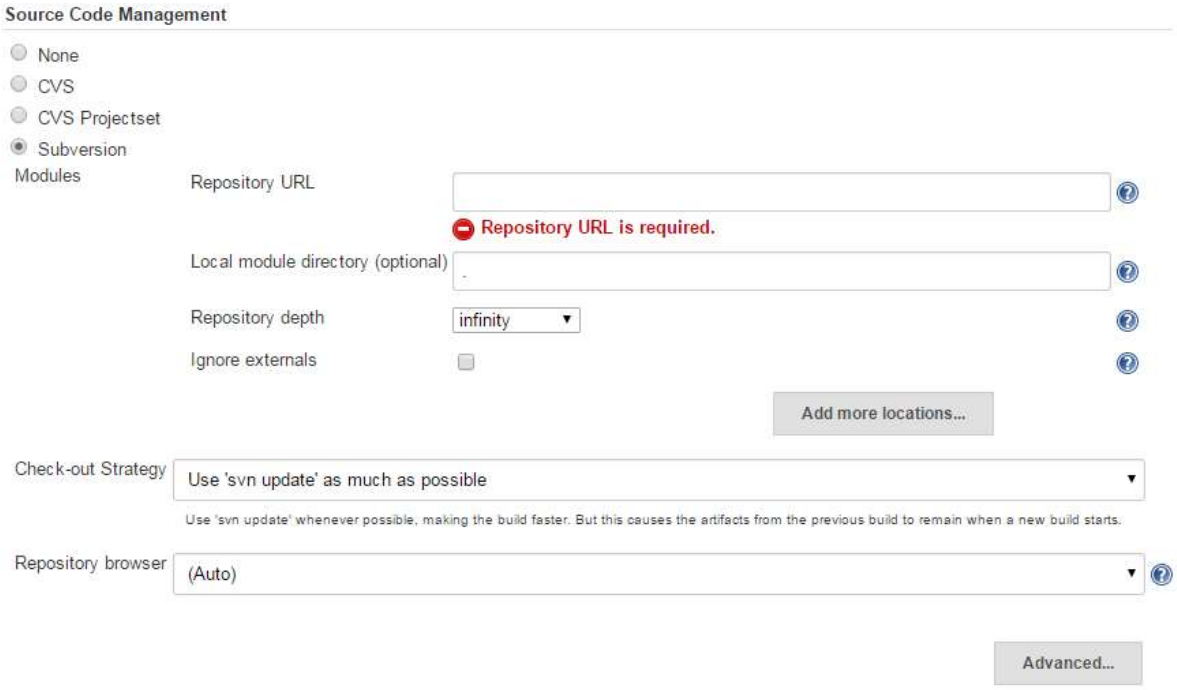
Slika 5.9. Stvaranje Jenksins projekta

Postavke projekta su prikazane su pomoću više slika kako bi se povećala čitljivost. Slika 5.10. Prikazuje opće postavke projekta kao što su ime projekta, opis projekta te opcije za odbacivanje prethodnih inačica ili dodavanje parametara izgradnje projekta. Osim općih postavki Slika 5.10. prikazuje i opcije kontrole izvornog kôda.



Slika 5.10. Opće postavke Freestyle projekta

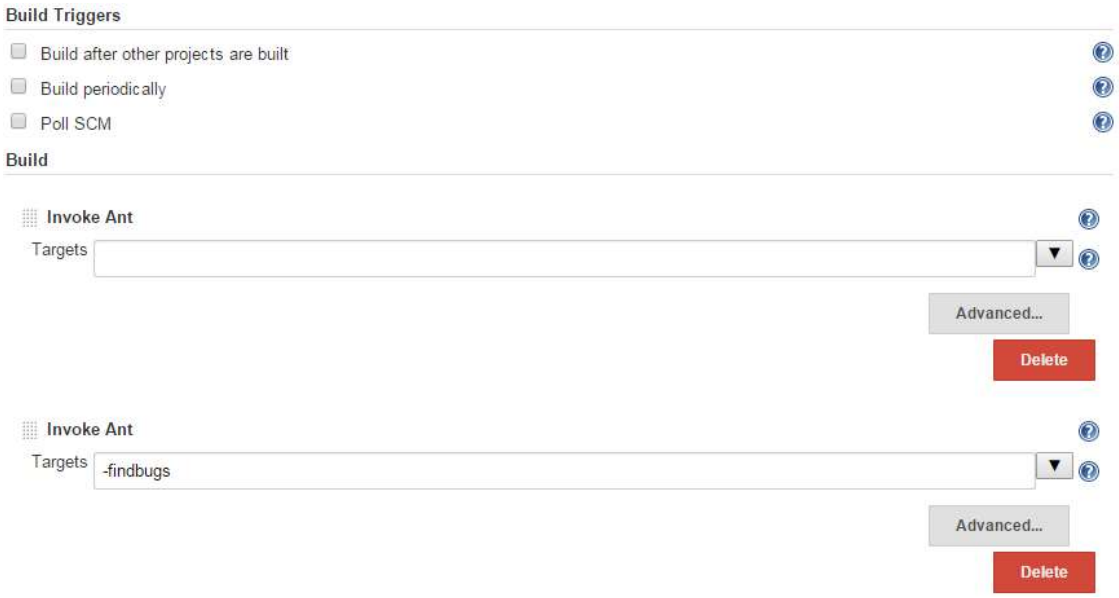
Budući da testni projekt koristi *subversion* kontrolu izvornog kôda, potrebno je Jenkins projekt podesiti da radi sa *subversionom*. Slika 5.11 prikazuje opcije kontrole izvornog kôda.



Slika 5.11. Podešavanje kontrole izvornog kôda

Polje *Repository URL* nužno je popuniti. Ukoliko navedeni repozitorij posjeduje autentifikaciju, Jenkins će prijaviti da spajanje nije bilo uspješno te zatražiti da se unesu korisnički podaci. Nakon unošenja ispravnih podataka Jenkins će javiti da se je spajanje uspješno.

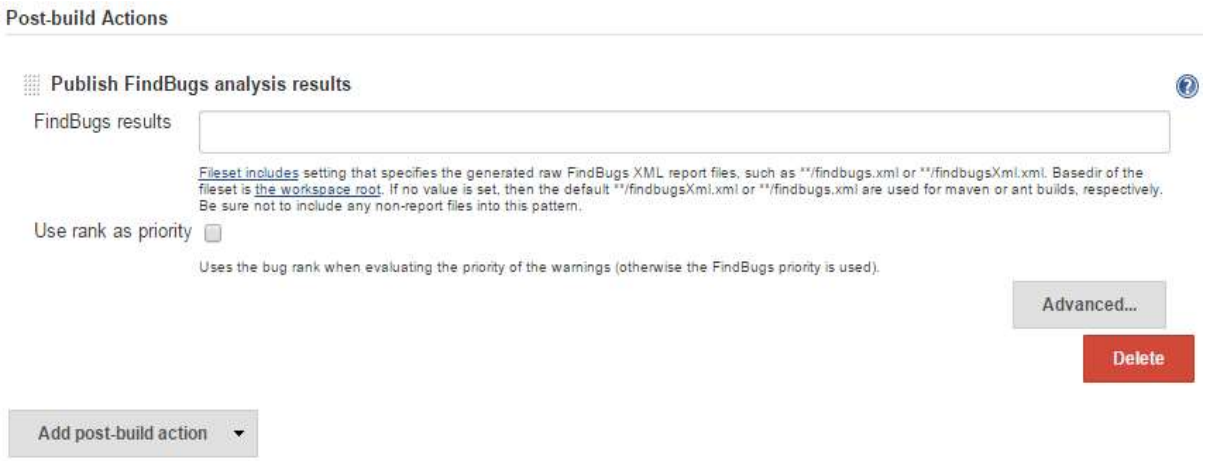
Potrebno je postaviti alat za izgradnju projekta. Jenkins omogućuje izgradnju projekta pomoću Ant, Maven, Linux naredbenog prozora ili Windows naredbenog prozora. Osim same izgradnje, moguće je podesiti okidače koji će sami pokrenuti izgradnju projekta. Slika 5.12. prikazuje postavke izgradnje projekta za alat Ant te postavke okidača za automatsku izgradnju projekta.



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 *Configurate*.

Budući da se pri izgradnji koristi Ant alat, potrebno je napraviti Ant build datoteku. Ant build datoteka sadrži informacije o projektu, potrebnim bibliotekama te alatima. Osim općih informacija Ant build datoteka može sadržavati i Ant zadatke. Ant build datoteku moguće je generirati pomoću Eclipse-a. Unutar generirane Ant build datoteke potrebno je upisati Ant zadatke.

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="${findbugs.home}"
        effort="max"
       output="xml"
       outputFile="findbugs.xml"
        >
                <sourcePath path="${basedir}/src/" />
                <class location="${basedir}/bin/" />

  </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 zadatka. Ovako napisan Ant zadatak izvršiti će se ukoliko

postoje *ANT_HOM*E te *FINDBUGS_HOME* sistemske varijable. U suprotnom, potrebno ih je stvoriti ili u Ant build datoteke upisati lokaciju FindBugs-a.

5.3. Podešavanje SonarQube-a

Za razliku od Jenkinsa, SonarQube je moguće koristiti nakon same instalacije. Jedini preduvjet korištenja je posjedovanje alata koji će obaviti analizu. Analizu je moguće obaviti pomoću SonarRunnera, samostalnog alata za obavljanje analize ili ju je moguće odraditi pomoću alata za izgradnju projekta. Budući da Jenkins koristi Ant, logično je postaviti SonarQube da koristi Ant za analizu projekta. Kako bi to bilo moguće, potrebno je napisati Sonar zadatak.

Primjer 5.2. prikazuje Ant zadatak za SonarQube.

```
<!-- 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>

   <!-- Execute the SonarQube analysis -->
   <sonar:sonar />
</target>
```

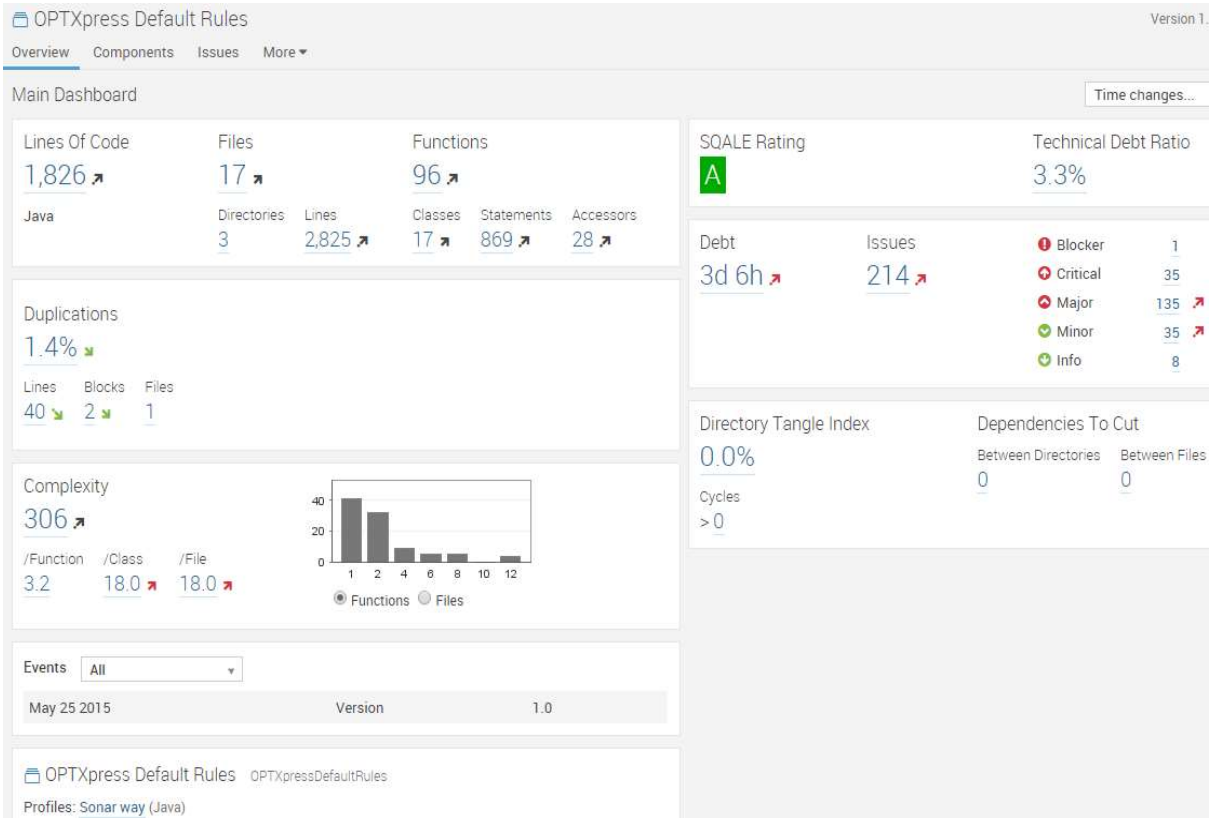
Primjer 5.2. SonarQube ant zadatak

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.



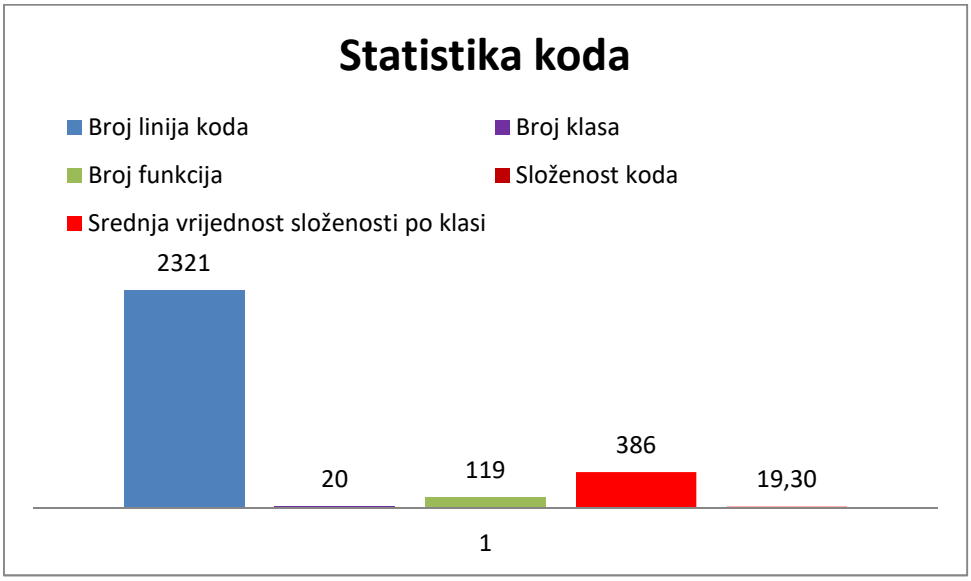
Slika 5.14. Upravljačka ploča projekta

Upravljačka ploča nudi opće informacije o projektu kao što su informacije o veličini projekta, te stanju projekta. Stanje projekta određeno je informacijama kao što je tehnički dug, broj duplikata te složenost kôda. Tehnički dug predstavlja vrijeme koje je potrebno potrošiti na ispravljanje grešaka. Tehnički dug određen je brojem grešaka te njihovom složenosti. Složenost kôda je zapravo ciklomatska složenost. Osim informacija o kodu, na dnu stranice, nalazi se informacija pomoću kojeg profila su rezultati prikupljeni. Ukoliko nije drugačije navedeno, zadani profil je *Sonar way*. Klikom na profil, moguće je promijeniti profil. Na ovaj način moguće je analizu obaviti sa ručno zadanim skupom pravila. Pri idućoj analizi, SonarQube će koristiti novi profil, ali će rezultate uspoređivati s prethodnim inačicama koje su analizirate s drugim profilom.

6. ANALIZA DOBIVENIH REZULTATA

Analiza je rađena na komercijalnom kodu. Budući da se kod nalazi na vanjskom repozitoriju, postojala je mogućnost da alati ne rade sa zadnjom inačicom kôda. Problem je riješen izradom Jenkins projekta koji se povezuje s vanjskim repozitorijem i sinkronizira kod sa lokalnim. Ostali alati postavljeni su da koriste Jenksinsov radni prostor. Potpoglavlje 6.1 opisuje korištenje statičke analize u različitim dijelovima razvoja.

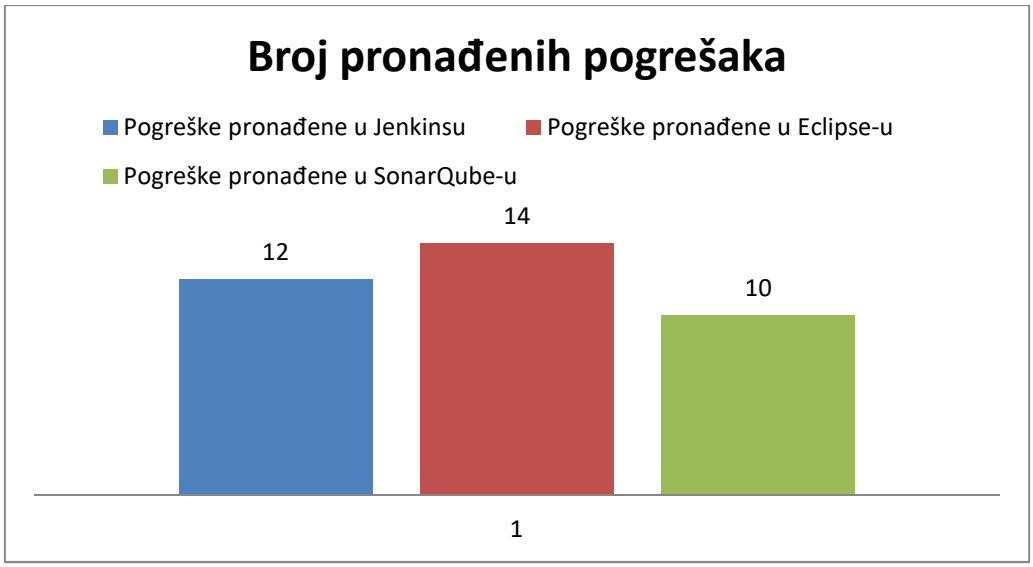
Za pravilno razumijevanje analize potrebne su informacije o kodu. Značajne informacije o kodu prikazane su Slika 6.1. Statistika kôda



Slika 6.1. Statistika kôda

Iz Slika 6.1. Statistika kôda vidljivo je da je analiza rađena na manjem projektu. Također, moguće je zaključiti da projekt ima srednje visoku složenost jer je prosječna složenost po klasi 19. Povećanjem projekta vjerojatno je da će prosječna složenost po klasi prijeći preporučenu vrijednost od 25, te je preporučljivo obaviti refaktoriranje 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.

Ime pogreške	**Složenost**	**Broj pronađenih pogrešaka**
Nekorištena vrijednost	Normal	1
Nepročitana vrijednost	Normal	5
Pisanje u statičku varijablu	High	2
NULL pointer dereference	High	2
Ukupno		10

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.

Budući da je cilj ove analize usporediti te tri različite perspektive, potrebno je podesiti programe da pokazuju što sličnije rezultate. S tim na umu, svi programi su podešeni da koriste FindBugs pravila. Kako bi se uskladili rezultati, rezultati SonarQube-a su korišteni kao referentni. SonarQube je izabran jer posjeduje najviše mogućnosti i pruža veliku prilagodljivost.

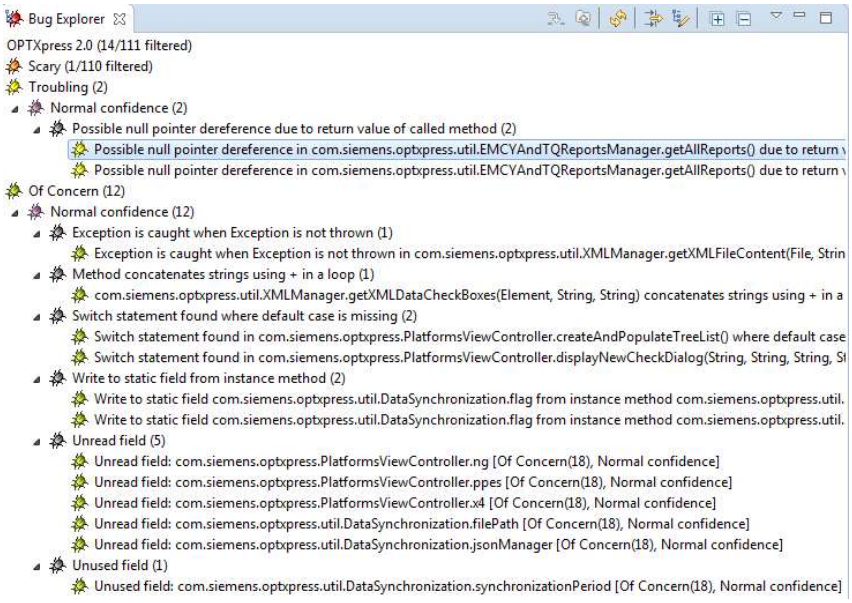
Analiza je rađena na manjem Java projektu koji je (u vrijeme pisanja ovoga rada) u implementacijskoj fazi i mijenjao se svakodnevno.

6.1.1. Perspektiva programera

Kao primjer programeove perspektive uzet je rad unutar Eclipse razvojnog okruženja. Eclipse je alat u kojem programer provodi najviše vremena. Eclipse posjeduje brojne perspektive koje olakšavaju programeru promatranje kôda. Pri ovoj analizi korištena je FindBugs perspektiva. Opis FindBugs perspektive nalazi se u potpoglavlju 4.1. Iz opisa FindBugs perspektive moguće je zaključiti da je cilj perspektive prikazati programeru gdje je pogreška, koliko je ozbiljna

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

Slika 6.3 prikazuje otkrivene pogreške unutar Eclipse-a. Zbog čitljivosti i preglednosti, umjesto potpune FindBugs perspektive, prikazani su samo *Bug Explorer* te *Bug Info* prozor.



Slika 6.3. Prozor Bug Explorera

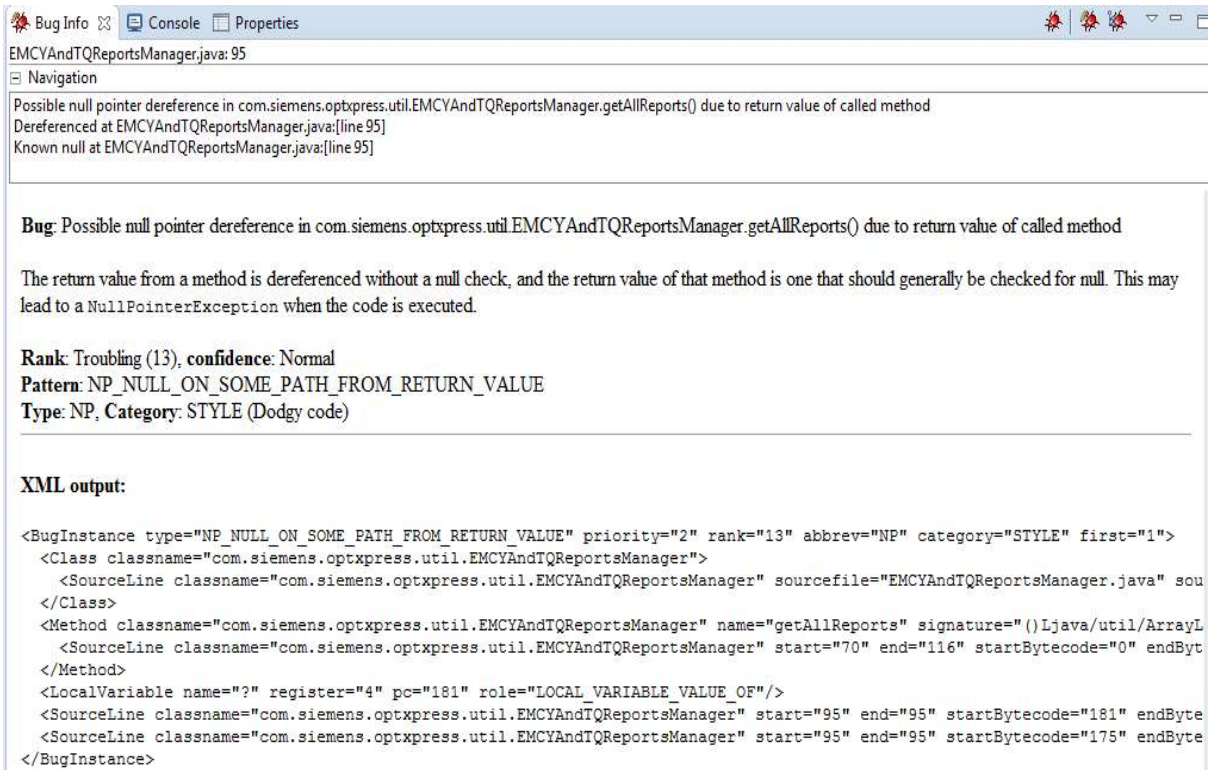
Pogreške su podijeljenje u skupine prema problematičnosti. Pogreške u skupini Scary zanemarene su iz razloga što predstavljaju upozorenja da FindBugs ne može pronaći strojni kod JavaFX biblioteke. Pogreške koje su ocijenjene kao najkritičnije nalaze se na vrhu popisa, dok su pogreške najmanje složenosti na dnu popisa. Otvaranjem pojedine skupine dolazi se do pogrešaka.

Slijedeći korak programerove perspektive uključuje proučavanje prijavljenih problema. Kao primjer odabrana je *Null pointer* pogreška. Klikom, unutar *Bug Explorera*, na pogrešku *Possible null pointer derefrence* uređivač izvornog kôda pokazat će kod za koji je prijavljena pogreška.

Slijedeći kod označen je kao null pointer pogreška :

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

Za lakše razumijevanje problema FindBugs posjeduje prozor sa dezaljim opisom problema. Slika 6.4 prikazuje informacije koje FindBugs posjeduje za „null pointer derefrence“ problem.



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 pogrešaka.

Iz navedenog kôda nije moguće odrediti hoće li se iznimka *NullPointerException* dogoditi ali je moguće potvrditi da objekt *tqReportsFolder* može imati vrijednost *NULL*. Kako bi odredili da li postoji mogućnost da se iznimka dogodi potrebno je pregledati cijeli kod. Pregledavanjem je utvrđeno da u navedenoj funkciji ne postoji provjera da li je objekt *NULL*. Za ispravak problema potrebno je prije rada s objektom napraviti provjeru da li je objekt jednak *NULL*.

6.1.2. Perspektiva voditelja projekta

Kao primjer perspektive voditelja projekta uzet je rad sa SonarQube alatom. SonarQube omogućava širi,ali i detaljnjiji 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 statistika kôda, složenost kôda, tehnički dug, broj pogrešaka, omjer dupliciranog kôda te

informacije o događajima. Iz opisa je vidljivo da upravljačka ploča nudi velik broj informacija. Upravljačka ploča prikazana je slikom Slika 5.14. Upravljačka ploča projekta. Zbog čitljivosti i preglednosti analizirati će se samo dijelovi vezani uz prikaz pogreške.

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

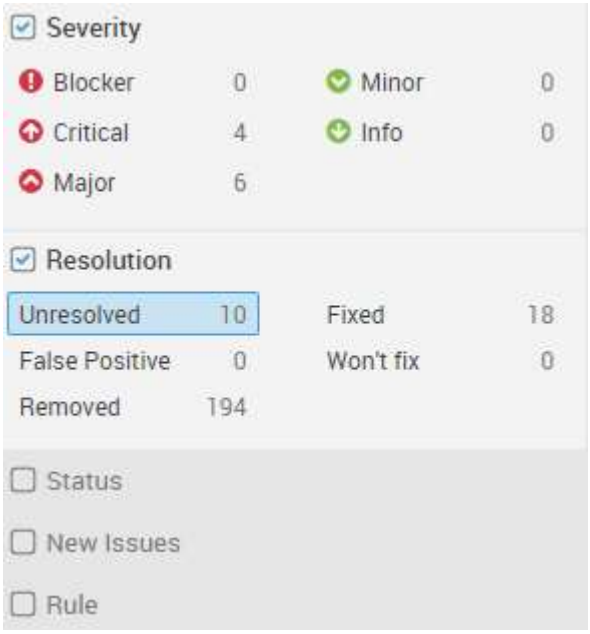
Slika 6.6. prikazuje pronađene pogreške. SonarQube pronađene pogreške raspoređuje prema složenosti.

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.

Nakon upoznavanja sa upravljačkom pločom moguće je prijeći na analiziranje pogrešaka. Budući da je prikaz pogrešaka složen,prikazan je u dijelovima. Prikaz pogrešaka podijeljen je u dva dijela. Prvi dio predstavlja izbornik koji omogućuje filtriranje pogrešaka prema određenim kriterijima. Drugi dio prikaza je popis pogrešaka koje zadovoljavaju kriterije.

Slika 6.7. Prikazuje izbornik za odabir prikaza pogrešaka.



Slika 6.7. Odabir pogreške

Pogreške je moguće odabrati prema slijedećim kriterijima :

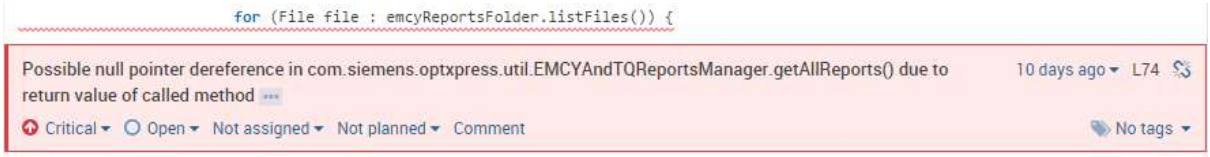
- Složenost
- Riješenost
- Status
- Nove pogreške
- Pravilo
- Tag
- Modul
- Direktorij
- Datoteka
- Osoba koja je pronašla pogrešku
- Osoba kojoj je greška dodijeljena
- Jeziku
- Planu rada

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

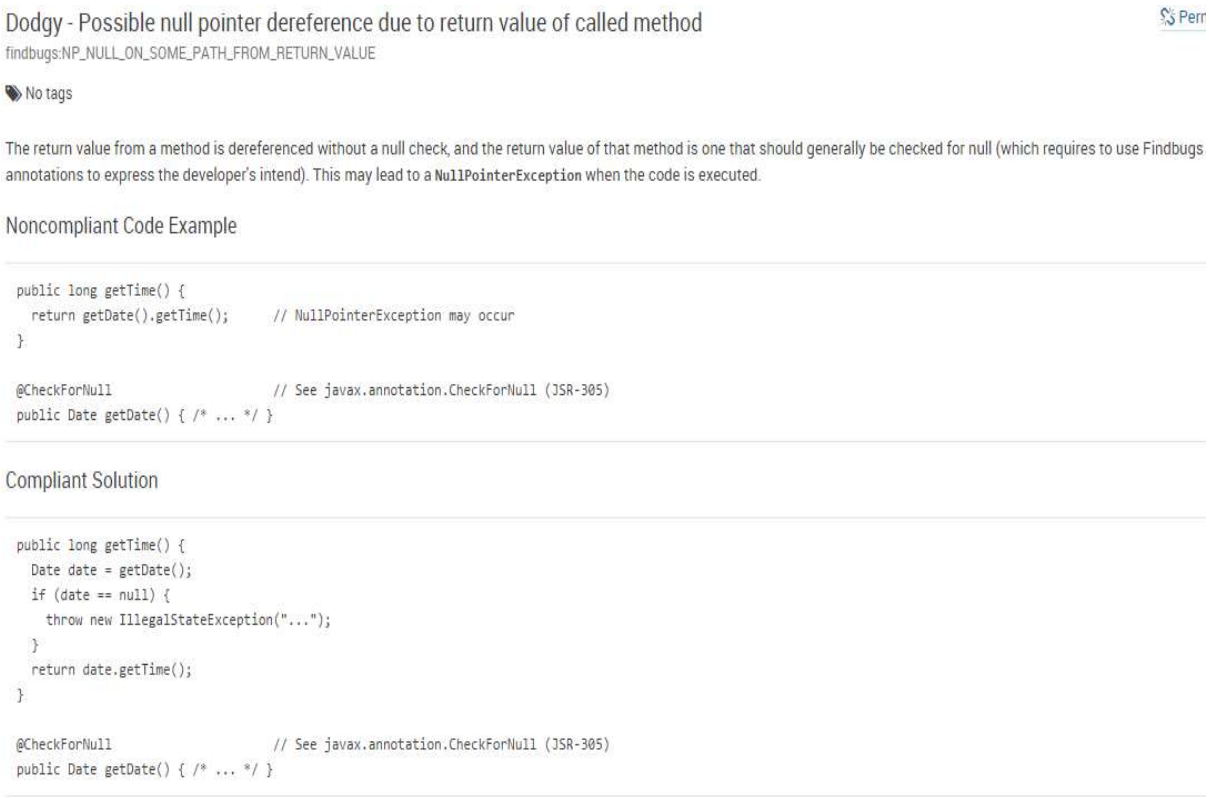
Slika 6.9. prikazuje način na koji SonarQube prijavljuje pogrešku. Unutar opisa moguće je vidjeti ime klase u kojoj je pogreška pronađena ta kratak opis pogreške.



Slika 6.9. Analiza pogreške

Ukoliko programer vjeruje da je otkrivena pogreška stvarna pogreška ili da nije pogreška, on može pogrešku potvrditi ili je prijaviti kao lažno-pozitivan izvještaj. Navedene stvari omogućene su izbornicima prikazanim Slika 6.9. Prvi izbornik predstavlja složenost pogreške. Drugi izbornik služi za potvrdu statusa pogreške. Pogrešku je moguće potvrditi, proglasiti riješenom, proglasiti lažno-pozitivnom ili ju je moguće označiti kao pogrešku koja se neće rješavati. Treći izbornik označava osobu kojoj je pogreška dodijeljena. Četvrti, pretposljednji, izvornik označava plan kojemu popravak pripada. Posljednji izbornik omogućava ostavljanje komentara uz pogrešku.

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

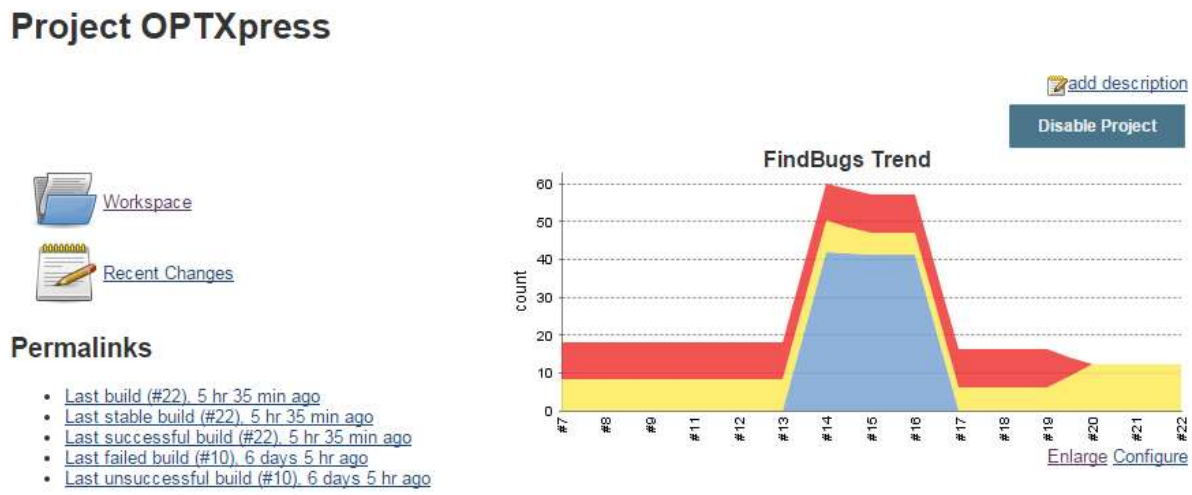
U opisu pogreške navedeno je, da je pri pozivu objekta bez prethodne provjere da li je objekt jednak *NULL*, može doći do *NullPointer* iznimke. Ostatak slike prikazuje dva kôda. Prvi kod predstavlja loš kod koji će aktivirati pravilo. Drugi kod pokazuje kako treba ispraviti loš kod.

Ostale mogućnosti objašnjenje su u poglavlju 5.3.

6.1.3. Perspektiva vodstva

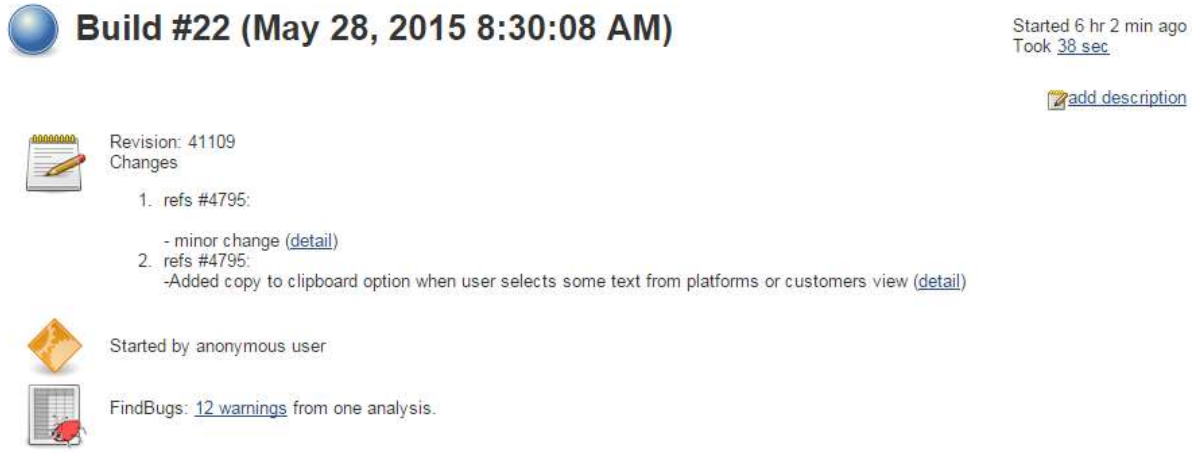
Kao primjer perspektive vodstva, uzet je rad sa Jenkins alatom. Jenkins omogućuje sažet pregled svih projekata koje on izgrađuje. Osnovna inačica Jenkinsa nudi informacije o trendu izgradnje projekta – koliko posljednjih izgradnji je završilo pogreškom. Sam Jenkins nema mnogo mogućnosti kontrole kvalitete ali zahvaljujući dodacima moguće je pratiti pogreške koje se događaju. Osnovno sučelje Jenkinsa prikazano je Slika 4.2 te se neće dodatno analizirati. Za analizu je zanimljivije sučelje projekta.

Slika 6.11 prikazuje kontrolnu ploču projekta. Iz slike je vidljivo da Jenkins veliku pažnju posvećuje praćenju statistike za svaku izgradnju projekta. Na lijevoj strani slike vidimo opciju za pristupanje radnom direktoriju projekta te opciju za pregled nedavnih promjena. Ispod ovih opcija nalaze se karakteristične inačice izgrađenih projekta. Karakteristične su po uspješnosti. Jenkins nudi poveznice za posljednju izgradnju, posljednju stabilnu izgradnju, posljednju uspješnu izgradnju te posljednju neuspješnu izgradnju. Stabilna izgradnja zadovoljava kriterije kvalitete kôda (osim što je uspješna). Desna strana slike prikazuje trend kretanja pogrešaka po inačicama izgradnje. Krivac za skok pogrešaka od inačica 13. do 17. je korištenje različitih pravila i postavki analize.



Slika 6.11. Kontrolna ploča projekta

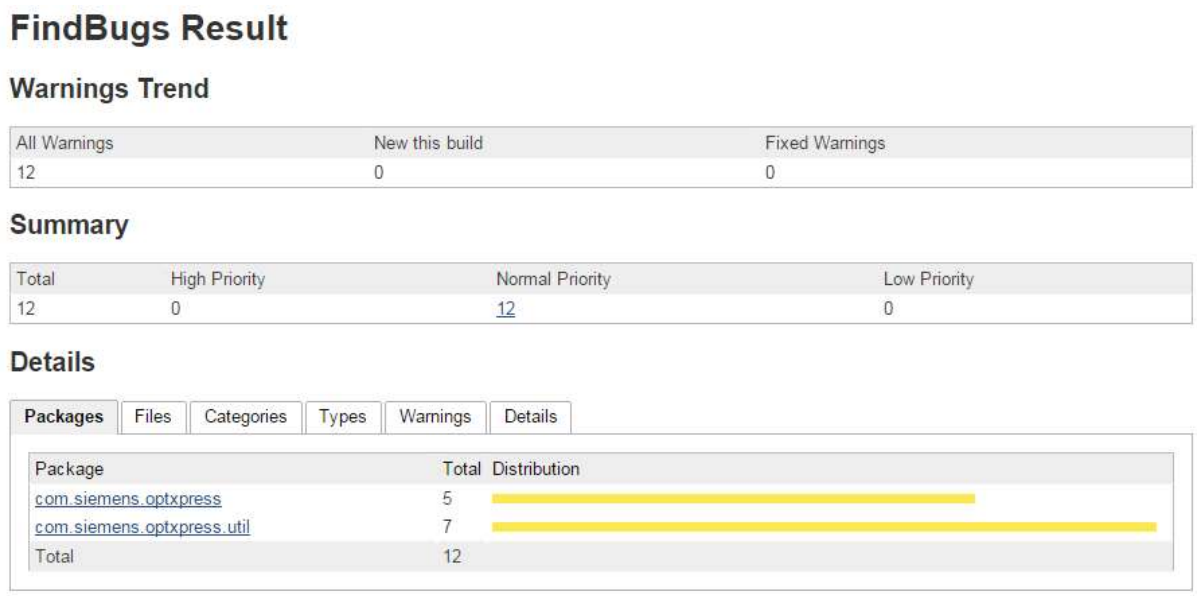
Za detaljniji prikaz pogrešaka potrebno je odabrati inačicu izgradnje. Slika 6.12. prikazuje Jenkinsov izvještaj izgradnje za posljednju inačicu. Unutar izvještaja nalaze se informacije o izmjenama, ime osobe koja je pokrenula izgradnju te poveznica na FindBugs izvještaj.



Slika 6.12. Izvještaj posljednje izgradnje

Kako je Jenkins spojen sa sustavom za kontrolu kôda, preko njega preuzima informacije o izmjenama kôda. Značajnija informacija, za ovu analizu, je informacija o broju pogrešaka koje je FindBugs pronašao. Klikom na FindBugs poveznicu moguće je dobiti detaljniju analizu problema.

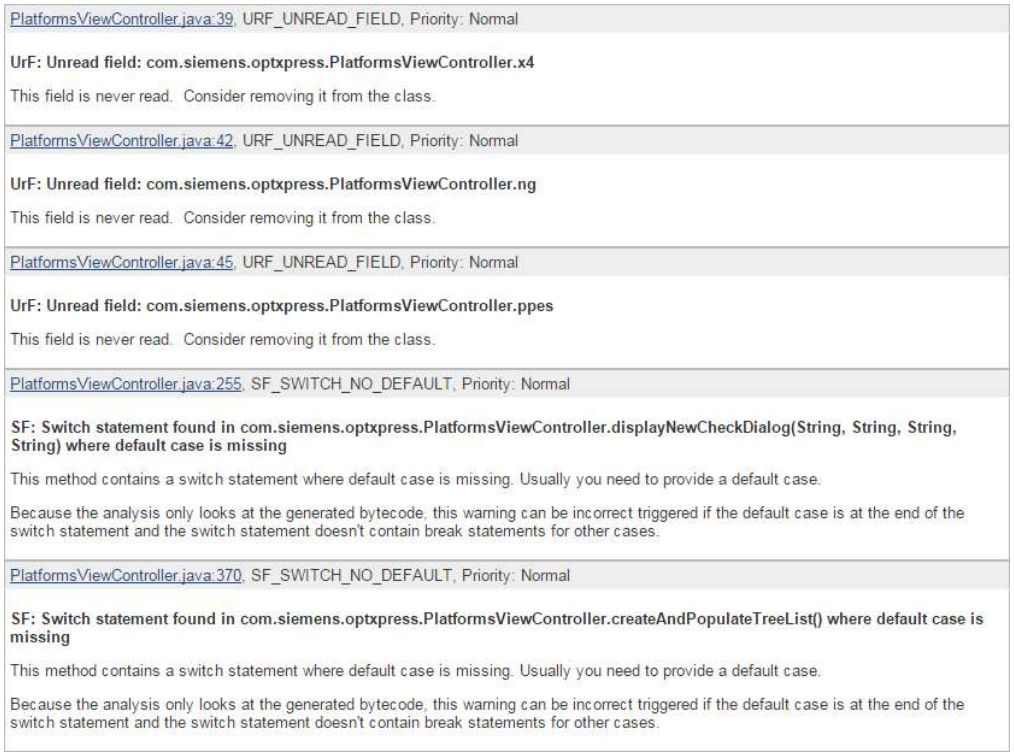
Slika 6.13. prikazuje FindBugs izvještaj.



Slika 6.13. FindBugs izvještaj

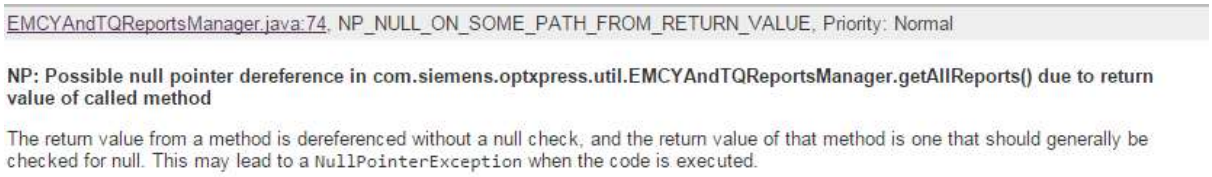
FindBugs izvještaj nudi informacije o količini grešaka. Prva tablica sadrži podatke o ukupnom broju pogrešaka, novonastalim pogreškama te o pogreškama koje su ispravljene u odnosu na prethodnu inačicu. Druga tablica sadrži podatke o složenosti pogrešaka. Poslijedna, treća, tablica sadrži detaljniji opis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

Slika 6.14 prikazuje detaljan popis pogrešaka. Radi preglednosti prikazan je samo dio. Svaka pogreška sadrži ime i opis problema.



Slika 6.14. Detaljan prikaz grešaka

Slika 6.15 prikazuje Jenksinsov način izvještavanja Null pointer *derefrence* pogreške. Za razliku od SonarQube-a, Jenkins ne nudi primjer pogreške. Klikom na ime pogreške moguće je proučiti problematičnu liniju kôda. Izvorni kod nudi identičan opis pogreške kao što je prikazano slikom 6.15.



Slika 6.15. Prikaz pogreške Null pointer derefrence

6.2. Detaljna analiza projekta

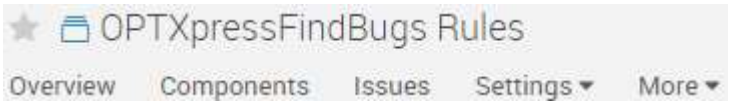
Za analizu su odabrana FindBugs, SonarQube i PMD pravila za statičku analizu kôda. Detaljne analize sa FindBugs i SonarQube pravilima izrađene su pomoću SonarQube-a, a razlozi njegova odabira navedeni su u prethodnom poglavlju. Cilj ovih analiza je demonstrirati SonarQube-ove mogućnosti. Za razliku od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

Analiza projekta obavljena je pomoću FindBugs i SonarQubeovih pravila. Analiza FindBugsovih rezultata nadovezuje se na analizu iz poglavlja 6. Nakon provođenja obje analize, rezultate je potrebno usporediti te izvesti zaključak kako unaprijediti kod.Na kraju, detaljna analiza istog projekta biti će napravljena i pomoću PMD pravila unutar Eclipse razvojnog okruženja.

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

Izbornik *„Overview“* spomenut je u potpoglavlju 6.2.1, gdje su objašnjeni njegovi osnovni elemnti. Ostali elementi objašnjeni su u nastavku. *„Components“* izbornik omogućava pristup datotekama i direktorijima analiziranog projekta. Pomoću njega je moguće pregledati izvorni kod. *„Issues“* je izbornik u kojemu programer provodi najviše vremena. *„Issues“* izbornik zadužen je za prikaz pogrešaka. *„Settings“* izbornik sadrži postavke izbornika. Pomoću njega je moguće promjeniti općenite postavke projekta, promjeniti profil kvalitete, postaviti mjere kvalitete i sl. *„More“* izbornik sadrži alate za usporedbu projekta, pregled pogrešaka i sl.

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 dupliciranom kodu

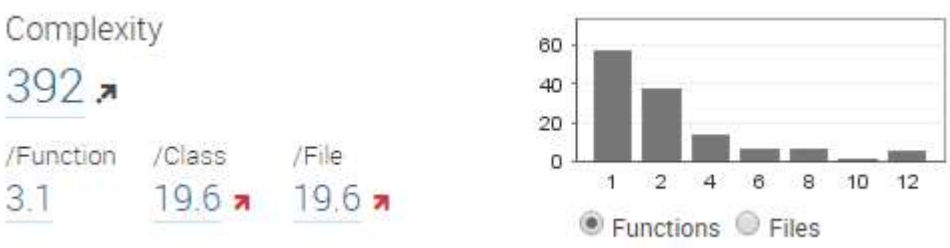
Klikom na broj linija dobivaju se informacije u kojim klasama su otkrivena ponavljanja. Odabirom pojedine klase, otvara se preglednik kôda u kojemu je moguće vidjeti problematične linije.

Slika 6.18 prikazuje isječak iz SonarQubeovog preglednika koda. Na slici je prikazan način označavanja ponavljanja. Narančasta boja pokraj linija označava da je riječ o ponavljanju. Prelaskom preko linije dobiva se informacija s kojim dijelom kôda se linija podudara.

Slika 6.18. Prikaz dupliciranih linija

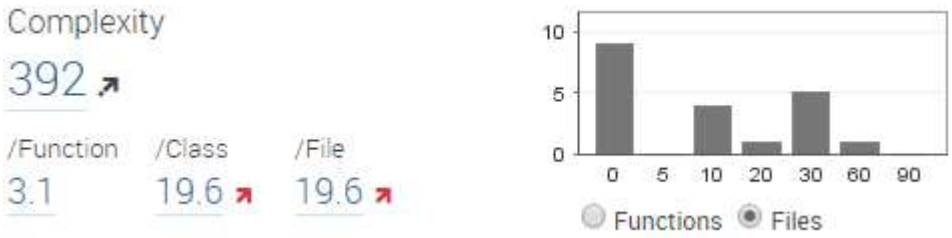
SonarQube, pri prikazu složenosti, nudi prikaz složenosti funkcija ili prikaz složenosti datoteka. Svaki prikaz nudi informacije o ukupnoj složenosti projekta kao i informacije o srednjoj složenosti funkcija, klasa i datoteka. Statistika složenosti prikazana je slikama Slika 6.19 i Slika 6.20. Kao što je napisano u poglavlju 6, složenost je na granici preporučenih vrijednosti te, kako je vidljivo iz slika, složenost ima tendenciju rasta. Dodatne informacije o složenosti pojedinih komponenti mogu se dobiti klikom na pojedinu vrijednost.

Slika 6.19 prikazuje složenost funkcija. Ordinata grafa predstavlja broj funkcija dok apscisa predstavlja složenost. Iz grafa možemo zaključiti da postoji veći broj jednostavnih funkcija.



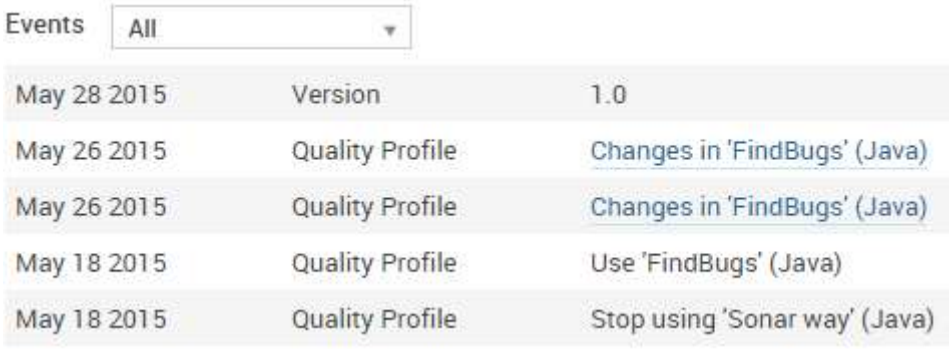
Slika 6.19. Prikaz složenosti funkcija

Slika 6.20 prikazuje složenost datoteka. Ordinata grafa predstavlja broj datoteka dok apscisa predstavlja složenost. Iz grafa možemo zaključiti da postoji potreba za uvođenjem novih klasa jer postoji veći broj složenih klasa.



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.

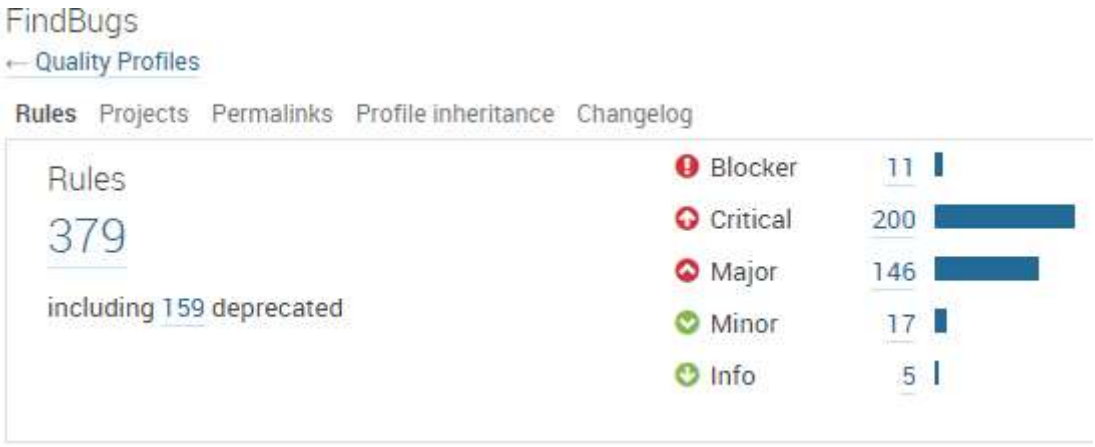


Slika 6.21. Prikaz događaja

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.

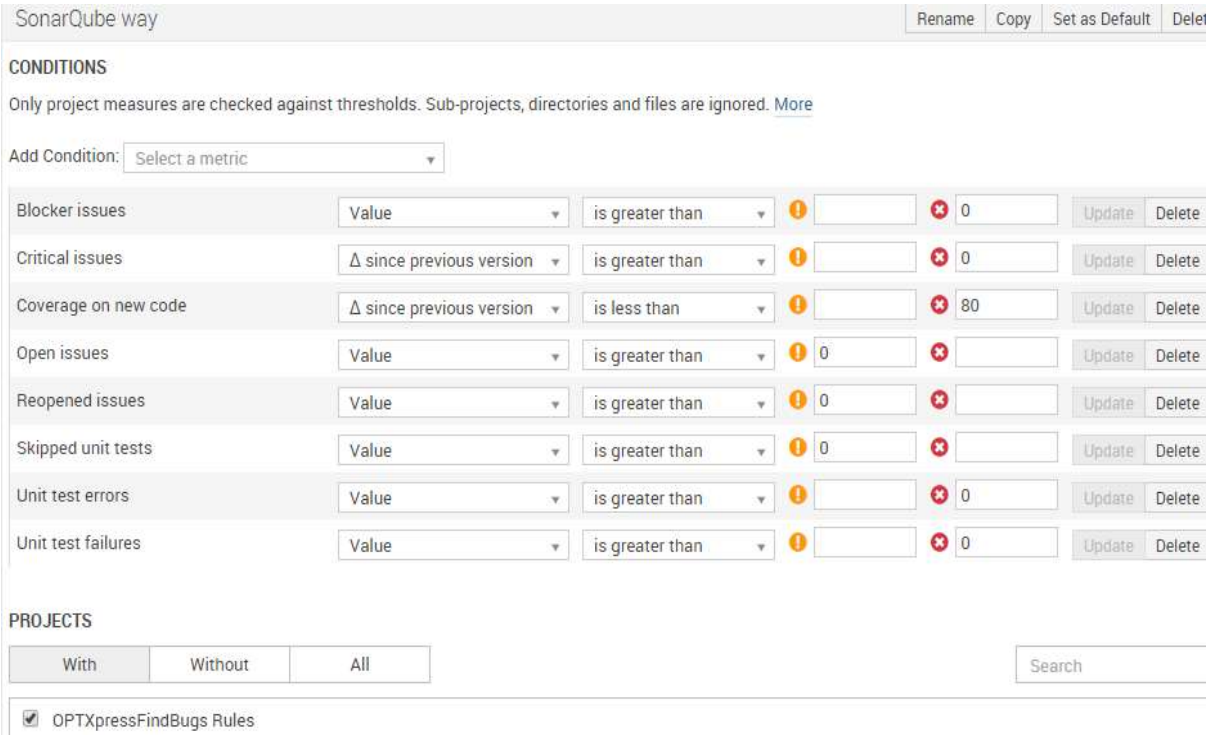
Slika 6.24. Ocjena kvalitete SQALE

Slika 6.25 prikazuje statistiku zapletenosti datoteka. Zapletenost datoteka je mjera uzajamne ovisnosti datoteka.

Slika 6.25. Zapletenost datoteka

Osim izrade plana popravaka SonarQube omogućuje automatsku provjeru kvalitete. Provjera kvalitete omogućena je primjenom kontrolnih stanica. Kontrolne stanice predstavljaju skup pravila koje projekt mora zadovoljiti kako bi bio proglašen prolaznom ocjenom.

Slika 6.26 prikazuje pravila kontrolne stanice. Pravila su zapravo granične vrijednosti mjernih jedinica koje projekt ne smije preći. Kao što je vidljivo na slici pravila je moguće prilagoditi. Također je moguće dodati vlastita pravila.



Slika 6.26. Pravila kontrolne stanice

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

Tablica 6.2 predstavlja pregled pronađenih pogrešaka pomoću FindBugs pravila.

Ime pogreške	**Složenost**	**Broj pronađenih pogrešaka**
Povjerenje u zadano šifriranje (encoding)	Visoka	12
NULL pointer dereference	Kritična	2
Metoda ne provjerava vraćenu vrijednost kod metoda koje bacaju iznimke	Visoka	2
Spremljena vrijednost varijable se ne koristi	Kritična	1
Ukupno		17

Tablica 6.2. Popis pronađenih pogrešaka pomoću FindBugs pravila

Kako bi projekt analizirali pomoću SonarQube-ovih pravila potrebno je podesiti profil kvalitete. Profil kvalitete moguće je podesiti preko „Settings“ izbornik-a.

U prilogu P.6.1 nalazi se tablica pronađenih 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

pronašla je 1698 problema dok je SonarQube-ova analiza pronašla 266. Ova činjenica ne znači da je PMD analiza kvalitetnija nego samo pokazuje da je SonarQube-ova analiza „pročišćena“ od određenih lažno-pozitivnih pogrešaka. Nadalje, SonarQube određene pogreške ne navodi kao pogreške već o njima izvještava u obliku statistike kôda. Primjer ovoga je ciklomatska složenost.

6.2.2. PMD analiza unutar Eclipse razvojnog okruženja

Za razliku od prethodnih analiza pomoću SonarQube alata, PMD analiza rađena je unutar Eclipse-a. PMD analiza prijavljuje probleme vezane uz nepoštivanje dogovorenih standarda ili daje preporuke vezane uz dobru praksu kodiranja. Za razliku od ostalih alata, PMD je prijavio velik broj problema. Budući da je analiza rađena na projektu u tijeku, očekivano je da će broj kršenja dogovorenih standarda biti još i veći jer programer nije upoznat s time. Prilog P.6.2. prikazuje popis pogrešaka koje je PMD pronašao. Ova analiza prije svega pokazuje programerovu upućenost u standarde jezika i kodiranja. Najčešća pogreška je kršenje *Law of Demeter pravila*. *Law of Demeter* pravilo glasi da određeni član treba imati ograničeno znanje o ostalim članovima, tj. treba znati samo za svoje najbliže članove.

6.3. Ispravljanje pogrešaka

Kako bi odrađene analize imale svrhu potrebno je, na temelju njih, započeti proces ispravljanja pogrešaka. Ovisno o broju pogrešaka, složenosti ispravaka te o dostupnom vremenu, proces ispravljanja pogrešaka može biti dugotrajan i naporan. Zbog toga je poželjno napraviti listu prioriteta ispravaka. U prioritetnu listu ispravaka nužno je uključiti probleme koji imaju veliki utjecaj na kvalitetu programa. Za određivanje prioriteta možemo se poslužiti preporukom, ali moramo biti svjesni činjenice da različiti alati različito ocjenjuju složenost. Promotrimo ovo na primjeru. Analize SonarQube-a i PMD-a prijavile su problem da postoje privatne varijable koje se ne koriste. SonarQube je ovaj problema ocijenio sa visokom složenosti dok je PMD složenost ocijenio normalnom. Nekorištene varijable nisu beznačajan problem, one zauzimaju resurse koji bi mogli koristiti drugi procesi, ali ukoliko pogledamo ostale probleme, sigurno ćemo pronaći problem koji više zaslužuje veći prioritet od “brisanja nekorištenih linija kôda”. Dakle, pri izradi liste prioriteta poželjno je proučiti probleme te napraviti vlastitu listu prioriteta. Za izradu liste prioriteta korištena je tablica iz priloga P.6.1.

Tablica 6.3 prikazuje popis grešaka koje je poželjno što prije ispraviti.

Ime pogreške	Složenost	Broj pronađenih pogrešaka
Rukovatelj iznimkama treba sačuvati izvornu iznimku	Kritična	30
Throwable.printStackTrace(...) se ne treba pozivati	Kritična	9
Metode trebaju imati manju složenost	Visoka	5
Prilikom provjere jednakosti String se treba nalaziti s lijeve strane	Visoka	45
Deklaracija treba koristiti Java kolekcije kao sučelje kao što je List nego specifične implementirane klase kao što je "LinkList"	Visoka	29
Nepoželjno je duboko ugnježđivanje izjava za kontrolu toka kao što su "if", "for", "while", "switch" and "try"	Visoka	19

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 neinicijaliziranim 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

[1] Ivo Gomes, Pedro Morgado, Tiago Gomes,Rodrigo Moreira, "An overview on the Static Code Analysis approach in Software Development".

[2] "Program development lifecycle", Wikipedia, URL: http://en.wikipedia.org/wiki/Systems_development_life_cycle#/media/File:SDLC-Maintenance-Highlighted.png, pristup ostvaren 15.4.2015.

[3] "Dynamic code analysis vs static code analysis", Computer Weekly, URL: http://www.computerweekly.com/answer/Dynamic-code-analysis-vs-static-analysis-source-code-testing, pristup ostvaren 20.4.2015.

[4] Brian Chess, Jacob West, "Secure Programming with Static Code Analysis".

[5] Jean-Louis Boulanger, Jean Souyris, David Delmas and Stéphane Duprat, "Static Analysis : The Abstract Interpretation".

[6] Sheng Yu, Shijie Zhou, "A Survey on Metric of Software Complexity".

[7] "FindBugs tool description", FindBugs, URL: http://findbugs.sourceforge.net/bugDescriptions.html, pristup ostvaren 12.5. 2015.

[8] "PMD tool description", PMD, URL: http://pmd.sourceforge.net/pmd-4.3.0/rules/index.html, pristup ostvaren 15.5.2015.

[9] J. F. Smart, "Jenkins The Definitive Guide".

[10] "Jenkins plugins tool description", Jenkins, URL: https://updates.jenkins-ci.org/download/plugins/, pristup ostvaren 18.5.2015.

[11] "Sonarqube tool description", Sonarqube, URL: http://www.sonarqube.org/, pristup ostvaren 18.5.2015.

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

[13] Gogul Balakrishnan and Thomas Reps, "WYSINWYX: What You See Is Not What You eXecute".

[14] "Cost of code change", Agile modeling, URL: http://www.agilemodeling.com/essays/costOfChange.htm, pristup ostvaren 11.5.2015.

[15] "FindBugs Eclipse plugin description", FindBugs, URL: http://findbugs.cs.umd.edu/eclipse/, pristup ostvaren 20.5.2015.

[16] "PMD Eclipse plugin", PMD, URL: http://sourceforge.net/projects/pmd/files/pmd-eclipse/update-site/, pristup ostvaren 20.5.2015.

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.

ŽIVOTOPIS

Matija Solić rođen je 11.2.1992. u Virovitici. Godine 2006. , nakon završene osnovne škole „Ivana Brlić-Mažuranić" upisuje Tehničku školu u Virovitici. Godine 2010. upisuje preddiplomski studij računarstva na Elektrotehničkom fakultetu u Osijeku. Godine 2013., nakon završetka preddiplomskog studija, upisuje sveučilišni diplomski studij računarstva, smjer Procesno računarstvo.

Ime i prezime

_____ (potpis)

PRILOZI

P.6.1. Popis pronađenih pogrešaka pomoću SonarQube pravila

Ime pogreške	Složenost	Broj pronađenih pogrešaka
Prilikom provjere jednakosti String se treba nalaziti s lijeve strane	Visoka	45
Nekorištene privatne metode treba ukloniti	Visoka	41
Rukovatelj iznimkama treba sačuvati izvornu iznimku	Kritična	30
Deklaracija treba koristiti Java kolekcije kao sučelje (kao što je List) ne specifične implementirane klase (kao što je "LinkList")	Visoka	29
Tab znakovi se ne trebaju koristiti	Niska	22
Nepoželjno je duboko ugnježđivanje naredbi za kontrolu toka kao što su "if", "for", "while", "switch" and "try"	Visoka	19
String oznake trebaju biti jedinstvene	Niska	18
Nekorištene privatne varijable treba ukloniti	Visoka	15
Throwable.printStackTrace(...) se ne treba pozivati	Kritična	9
Potrebno je ukloniti nepotrebne zagrade oko izraza	Visoka	9
"TODO" tagove treba implementirati	Info	7
Ugniježđeni dijelovi kôda ne trebaju biti prazni	Visoka	6
Nekorištene importe treba ukloniti	Niska	6
Metode trebaju imati manju složenost	Visoka	5
Standardni izlaz se ne treba koristiti za bilježenje informacija(logging)	Visoka	5
Ukupno		266

P.6.2. Popis pronađenih pogrešaka pomoću PMD pravila

Ime pogreške	Složenost	Broj pronađenih pogrešaka
Dodjeljivanje vrijednosti pri operaciji (usporedbi)	Normal	1
Klasa treba imati barem jedan konstruktor	Normal	9
Izbjegavati hvatanje generičke iznimke	Normal	8
Izbjegavati hvatanje NullPointer iznimke	Normal	1
Izbjegavati hvatanje Trowable iznimke	Normal	1
Izbjegavati iste vrijednosti varijabli	Normal	16
Izbjegavati instanciranje objekata u petlji	Normal	21
Izbjegavati korištenje vrijednosti u if uvjetu	Normal	5
Izbjegavati PrintStackTrace	Normal	9
Izbjegavati bacanje čistih iznimki	High	2
Bean članovi bi trebali biti serijalizirani	Normal	117
Dobra je praksa pozvati super u konstruktoru	Normal	4
Ugniježđeni if uvjeti se mogu kombinirati	Normal	2
Potreban komentar	Normal	196
Prevelik komentar	Normal	6

Zbunjujući uvjet	Normal	9
Ciklomatska složenost	Normal	8
Dataflow analiza	Low	67
Default paket	Normal	9
Izbjegavati poziv System.exit funkciji	Normal	1
Izbjegavati korištenje niti u J2EE	Normal	1
Prazan try catch blok	Normal	4
Prazan if blok	Normal	5
Veliki broj importa	Normal	1
Predugačke metode	Normal	3
Deklaracija varijabli treba se nalaziti na početku klase	Normal	39
God klasa	Normal	2
If uvjet treba imati blok	Normal	9
ImmutableField	Normal	72
JUnit4Test treba koristiti Test anotaciju	Normal	2
Law of demeter	Normal	404
Lokalna varijabla može biti final	Normal	281
Predugo ime varijable	Normal	37
Izbjegavati koristiti ArrayList kao implementacijski tip. Koristiti sučelje umjesto Arraylist.	Normal	12
Argumenti metode mogu biti konačni	Normal	106
Izmijenjena ciklomatska složenost	Normal	7
Npath složenost	Normal	2
Metoda treba imati samo jednu return izjavu	Normal	20
Prilikom provjere jednakosti String se treba nalaziti s lijeve strane	Normal	28
Suvišno inicijaliziranje polja	Normal	3
Prekratko ime klase	Normal	1
Prekratko ime varijable	Normal	15
Pojednostaviti logički izraz	Normal	1
Varijabla može biti lokalna	Normal	21
Standardna Ciklomatska složenost	Normal	8
Switch petlja treba imati default	Normal	3
Ne koristiti SystemPrintln	High	5
Previše varijabli	Normal	2
Previše metoda	Normal	4
Ne komentiran prazni konstruktor	Normal	4
Nepotreban konstruktor	Normal	3
Nepotrebne zagrade	Normal	9
Nekorišteni parametar	Normal	3
Nekorišteni import	Normal	6
Nekorištena privatna varijabla	Normal	14
Nekorištena privatna metoda	Normal	41
Koristiti objekte za čišći API	Normal	2
Koristiti valjani ClassLoader	Normal	4
Koristiti String Buffer za dopunjavanje stringa	Normal	5

Koristiti pomoćne klase	Normal	1
Koristiti programske argumente (varargs)	Normal	1
Nepotrebne zagrade	Normal	12
Poštivati konvenciju o imenovanju varijabli	High	3
Ukupno		1698