Zbornik radova Fakulteta tehničkih nauka, Novi Sad

UDK: 004.41
DOI: https://doi.org/10.24867/03BE24Letic

DETEKCIJA I ANALIZA DUPLICIRANOG JAVASCRIPT KODA UPOTREBOM JSINSPECT ALATA

DETECTION AND ANALYSIS OF DUPLICATED JAVASCRIPT CODE USING JSINSPECT TOOL

Tamara Letić, *Fakultet tehničkih nauka, Novi Sad*

Oblast – ELEKTROTEHNIKA I RAČUNARSTVO

Kratak sadržaj – *U radu je opisan JavaScript jezik, njegovo trenutno mesto u svetu softverskog inženjerstva, kao i njegov potencijal u budućnosti. Pokrivena je tema kvaliteta koda kao uvod i potreba za postojanjem alata za njegovu statičku analizu. Detaljno je obrađena problematika pojave dupliciranog koda i istražen je već postojeći alat za njegovu detekciju kroz implementaciju veb aplikacije u koju je alat integrisan i analizu rezultata dobijenih prilikom analize.*

Ključne reči: *Java Script, kvalitet koda , statička analiza, duplirani kod, JSInspect*

Abstract *– This paper describes the JavaScript language, its current place in the world of software engineering, as well as its potential in the future. The theme of the quality of the code is covered as an introduction and the need for the existence of a tool for its static analysis. The problem of the duplicate code appearance has been elaborated in detail, and an already existing tool for its detection has been explored through the implementation of the web application into which the tool is integrated and analysis of the results obtained during the analysis.*

Keywords: *Java Script, code quality, static analysis, duplicate code, JSInspect*

1. UVOD

Ovaj projekat ima za cilj da ukratko uvede pojavu kvaliteta koda kroz dinamički otkucani jezik kao što je JavaScript, da analizira rad algoritma za pretragu - Apstraktno sintaksno stablo i alat JSInspect, kroz poređenje sa postojećim algoritmom i alatom za pronalaženje dupliciranog koda. Da definiše dobar skup metrika i problematičnih delova kodova koji su specifični za JavaScript jezik.

Da bi se procenio alat za analizu implementirana je veb aplikacija kroz koju je alat integrisan tako da je omogućen korisnički interfejs, obuhvata se i obrađuje velika kolekcija JavaScript koda prikupljenih iz projekata otvorenog koda i iz zbirke popularnih veb stranica.

NAPOMENA:
Ovaj rad proistekao je iz master rada čiji mentor je bio dr Darko Čapko vanr.prof.

2. JAVASCRIPT JEZIK I KVALITET SOFTVERSKOG KODA

JavaScript je objektno orijentisan programski jezik. Kao skriptni jezik, dizajniran je za kontrolu i automatizaciju postojećeg sistema. Ovaj postojeći sistem naziva se host okruženjem. Host okruženje otkriva objekte sa propertijima i funkcijama kojima se može pristupiti iz JavaScript-a. JavaScript jezik ne pruža nikakav ugrađeni mehanizam za generički ulaz ili izlaz, već se oslanja na objekte koje host okruženje obezbeđuje za komunikaciju sa hostom ili sa eksternim sistemima.

2.1 Code smells

Termin "kvalitet softvera" je skup više različitih ideja. Relevantne teme istraživanja u oblasti kvalitet softvera su identifikacija "smrada koda", određenih anti-paterna koji ukazuju na nizak kvalitet koda i "klonova koda", identične ili slične fragmente koda koji se javljaju na više lokacija u okviru istog projekta. Prisustvo smrada i klonova koda često ukazuje na to da će refaktorizacija koda ili redizajniranje programa koristiti sveukupnom kvalitetu softvera. Donekle povezano, takođe postoji velika količina posla u određivanju i definisanju 'metrika kodova' kao sredstvo za kvantifikaciju različitih aspekata izvornog koda.

Tehnike statičke analize mogu se koristiti za otkrivanje smrada koda, pronalaženje kodnih klonova i izračunavanje metrika koda prilikom pisanja softvera, pomažući programerima da spreče često pravljene greške. Ove tehnike postaju naročito korisne kada su programerima obezbeđeni alati za automatsko analiziranje njihovog koda i pomoć pri pisanju softvera, integrisanog u njihov IDE ili kao deo kontinuiranog sistema integracije.

2.2 Code clones

Softverski inženjeri govore o kod klonovima kada nailaze na više fragmenata koda (npr. datoteke, moduli, funkcije, blokovi) koji su potpuno ili delimično identični. Klonovi kodova se kreiraju kad god programer kopira određeni deo koda i *paste*-uje ga negde drugde, eventualno napravi neke male modifikacije nakon toga. Klonovi kodova takođe mogu da se "pojave" kada, na primer, dve osobe nezavisno napišu veoma sličnu klasu ili metodu u drugom fajlu. Ponekad je njihovo postojanje legitimno, ali u mnogim slučajevima, ovi klonovi su neželjeni i, sa određenom refaktorizacijom i redizajniranjem koda, ih treba pažljivo spajati nazad u jednu jedinstvenu celinu.

2.2.1 JavaScript klonovi koda

Ne postoji razlog zašto različite tehnike otkrivanja dupliciranog koda ne bi funkcionisale kod JavaScript-a. U praksi, međutim, čini se da postoji malo istraživanja koja se eksplicitno usmeravaju na klonove u JavaScript kodu.

Synytskyy, Cordy i Dean [1] opisuju pristup za pronalaženje kloniranog koda na veb stranicama koje se fokusiraju na HTML fragmente, ali koje se bave i *inline* i eksternim JavaScript kodom (preko <script> oznaka bez ili sa atributom src). Oni priznaju da koriste veoma jednostavan algoritam detekcije klona, jer otkrivanje klona nije glavni cilj njihovog istraživanja. Čini se da su uključene datoteke i *inline* kod blokovi tretirani kao pojedinačni fragmenti koda između kojih se otkrivaju klonovi.

Calefato [2] je objavio članak o detekciji kloniranih funkcija u veb aplikacijama. Oni tretiraju JavaScript funkcije kao fragmente koda, najmanjim jedinicama poređenja. Kandidati za klonove koda se detektuju pronalaskom duplikata imena funkcija u bazi koda. Oni tvrde da klonove JavaScript koda kreiraju programeri koji kopiraju i *paste*-uju funkcije iz jednog fajla u drugi, eventualno menjajući telo funkcije, ali zadržavajući ime funkcije netaknuto. Ovi kandidati za klonirane funkcije se ručno upoređivani i klasifikovani.

Softver PMD sadrži copy/paste detektor, skraćeno kao CPD. Koristi algoritam koji podudaranja stringova upoređujući string hash vrednosti i radi na leksičkom nivou. Uključeni CPD u verziji 4.3 PMD sadrži podršku za JavaScript (sa opcijom ECMAScript jezika), ali to nije izloženo u svim delovima interfejsa.

CPD takođe omogućava korisnicima da dodaju podršku za nove jezike pomoću dodataka, u obliku definicije jezika i kreatora tokena. Pronađena su dva open-source projekta koja koriste ovu metodu za analizu JavaScript koda: jedan *standalone* JavaScript modul i drugi modul kao deo Sonar open-source platforme za upravljanje kvalitetom koda.

Komercijalni ECMAScript CloneDR alat koristi sintaksičke metode za detekciju klona pomoću AST-ova. Ira Baxter, koja je napisala tekst "Detekcija klonova koristeći apstraktna sintaksna stabla" [3], je CTO kompanije koja nudi ECMAScript CloneDR alat.

3. STRUKTURA ALATA JSINSPECT I INTEGRACIJA KROZ VEB BAZIRANO REŠENJE

U prethodnom delu ovog rada je opisano zbog čega je došlo do potrebe za pojavom statičke analize koda, kao i o problemima takve analize jezika kao što je JavaScript. U delu 2.2.1 su pomenuti neki postojeći alati za pronalazak dupliciranog koda, ovo poglavlje baviće se novijim open-source alatom JSInspect. Zbog potrebe korisničkog prikaza ovakvih alata, koji se inače pokreću isključivo preko konzole, kreirana je veb aplikacija koja će obezbediti lak unos parametara potrebnih za analizu, kao i prikaz rezultata.

3.1. Opis kreirane veb aplikacije

U cilju olakšavanja korisničkog iskustva sa alatom JSInspect i sličnim alatima napravljena je veb aplikacija. Server je osmišljen tako da se pridržava RESTFul principa. Pomoću Express *framework*-a koji se oslanja na Node.js kreiran je Web API.

Potrebno je naglasiti da se podaci ne čuvaju na serveru, interni serverski model služi isključivo za obradu zahteva. Server prima objekat opcija i na osnovu prosleđenih opcija vrši obradu zahteva. Nakon obrade formira čitav model i JSON objekat vraća kao odgovor.

Zbog obezbeđivanja fleksibilnosti servera za podršku integracije različitih alata za pretragu dupliciranog koda bilo je potrebno kreirati adaptere. Cilj adaptera je da kada stigne takav zahtev da zadovoljava inerni serverski model, vrši mapiranje pristiglih opcija radi dobijanja formata opcija za odabrani alat. Poziva se funkcija alata koja vrši analizu i vraća rezultat. Taj rezultat ponovo prolazi kroz adapter i kao takav dolazi kao odgovor na klijent.

Zbog klijentske potrebe pregleda strukture odabranog foldera u obliku stabla na serveru je implementiran kontroler. On na osnovu relativne ili apsolutne putanje proveri sve fajlove koji se nalaze u traženom direktorijumu, kreira strukturu stabla i vrati klijentu.

Da bi klijentu bilo omogućeno da vidi fajlove u kojima se nalaze duplikati takođe je obezbeđeno da na odabir odredjenog objekta rezultata analize, server na taj zahtev odgovara sadržajem tražene datoteke.

Klijentski deo aplikacije je pisan u JavaScript programskom jeziku, koristeći React biblioteku.

Inicijalno je postavljen testni projekat čija je struktura izlistana na levoj strani, ako korisnik želi da promeni folder za analizu može to uraditi preko opcije **File -> Open folder** preko navigacije na vrhu aplikacije. Selekcijom nekog čvora strukture i klikom na dugme **Inspect folder** bira se putanja nad kojom će alat vršiti pretragu i otvara se modalni dijalog radi prikupljanja preostalih neophodnih informacija za analizu.

Prvo polje dijaloga predstavlja combobox preko kog korisnik odlučuje koji alat za analizu želi da koristi. Spram odabranog alata se renderuju različita polja za unos ne bi li se prikupile infornacije koje taj sprecifični alat zahteva.

Nakon popunjavanja i validacije polja za izabrani alat korisnik ima mogućnost da zahtev za analizom šalje na server ili da odustane od akcije.

3.2. JSInspect alat

JSInspect je alat za detekciju dupliciranog koda sa podrškom za JavaScript ekosistem: ES6, JSX i Flow. On koristi AST za pravljenje strukture izvornog koda. To znači da je kod predstavljen preko stabla, gde svaki čvor predstavlja konstrukciju kao što je blok izjava, deklaracija promenljivih itd. Zbog ovoga, JSInspect bi trebalo da bude dobar u pronalaženju strukturno sličnog koda.

Kroz alat je pružena mogućnost da se specificira *treshold* koji određuje najmanji podskup čvorova za analizu. Ovo će identifikovati kod sa sličnom strukturom, baziranom na AST tipovima čvorova. Inicijalno, on traži čvorove sa podudarajućim identifikatorima i literalima za copy/paste orijentisanu detekciju, ali ovo može biti isključeno. Zbog konteksta, identifikatori uključuju imena varijabli, metoda, propertija itd. Dok literale predstavljaju stringovi, brojevi i ostalo.

Alat prihvata listu putanja za parsiranje i vraća ispis svih pronađenih podudaranja. Kroz sve direktorijume na prosleđenim putanjama se prolazi rekurzivno i samo .js i .jsx fajlovi bivaju analizirani. Takođe, postoji mogućnost da se eksplicitno proslede putanje koje uključuju drugačije ekstenzije.

3.2.1 Parsiranje

Apstraktno sintaksno stablo (AST) je reprezentacija apstraktne sintaktičke strukture izvornog koda napisanog u programskom jeziku. Svaki čvor stabla predstavlja konstrukciju koja se javlja u izvornom kodu. Sintaksa je apstraktna u smislu da ne predstavlja bas svaki detalj koji se javlja u pravoj sintaksi, nego samo strukturne detalje, vezane za sadržaj. Npr. grupisane zagrade su implicitne u strukturi stabla, a sintaktička konstrukcija kao što je if-uslov-then izraz može biti obeležena kao jedan čvor sa tri grane.

Ovo pravi razliku između apstraktno sintaksnih stabala i konkretnih sintaksnih stabala, tradicionalno korišćenih stabala za parsiranje, koja tipično kreira parser prilikom prevođenja izvornog koda i kompajliranja. Nakon kreiranja, dodatne informacije se dodaju u AST kao proizvod rekurzivne obrade, npr. kontekstualne analize.

AST se naširoko koristi u kompajlerima kako bi se predstavila struktura programskog koda. [4]

JSInspect alat nema svoj parser, već koristi postojeći Babylon, JavaScript parser koji se koristi u Babel-u. Babylon generiše AST prema Babel AST formatu. On je zasnovan na ESTree spec sa sledećim devijacijama:

- **Literal** token se zamenjuje sa StringLiteral, NumericLiteral, BooleanLiteral, NullLiteral, RegExpLiteral
- **Property** token se zamenjuje sa ObjectProperty i ObjectMethod
- **MethodDefinition** se zamenjuje sa ClassMethod
- **Program** i **BlockStatement** sadrži dodatno directives polje sa Directive i DirectiveLiteral
- **ClassMethod**, **ObjectProperty** i **ObjectMethod** vrednosti propertija se dovode u čvor glavne metode

3.2.2 Analiza

U ovom delu biće opisan način rada alata od dobijanja sadržaja .js fajlova u obliku niza AST objekata do vraćanja rezultata analize.

Glavna klasa alata Inspector, koja nasleđuje EventEmitter, kroz konstruktor kreira novu instancu koja prima niz putanja i opcije sa bilo kojom kombinacijom postojećih opcija. Instanca emituje tri event-a: *start, match i end*.

Run je metoda koja pokreće inspekciju na datim putanjama, obezbeđenim u konstruktoru. Na samom početku emituje *start event*, nakon čega radi parsiranje sadržaja svakog fajla sa prosleđenih putanja i pretvara ga u niz AST objekata.

Za svaki AST objekat se radi prolaženje kroz njegovo stablo i pravi se niz čvorova koji zadovoljavaju *threshold* prosledjen *Inspector*-u preko opcija. Kad se pronađu svi takvi čvorovi, konačan niz se prosleđuje dalje metodi koja generiše ključ baziran na kombinovanim tipovima svakog prosleđenog čvora. Niz stavlja u drugi niz na mesto tog ključa u mapi. Čvorovi se *update*-uju da bi se održala referenca na sve njihove pojave u mapi.

Sledeći korak prolazi kroz ključeve na kojima se čuvaju različiti čvorovi. Ključ koji sadrži niz od više od jednog upisa označava mogući pogodak. Čvorovi se zatim grupišu ako je podudaranje identifikatora moguće. Rezultati podudaranja u releventnim čvorovima se uklanjaju iz bilo kakvih budućih rezultata. Ovo odsecanje obezbeđuje da je samo najveći zajednički roditelj uključen u set podudaranja. Kraj analize označava ispaljivanjem *end event*-a.

Alat vraća niz objekata koji sadrže imena fajla, *start, end* i linije koje su povezane sa svim instancama pogotka. Taj niz dalje ima mogućnost da se konvertuje u neki od ponuđenih i spomenutih reportera.

3.2.3 Reporting

Osim inicijalnog reportera, dostupni su i JSON i stil PMD CPD XML reporteri. U JSON obliku su primenjeni uvlačenje i formatiranje. Dalje, properti id koji je dostupan u ovim reporterima je koristan za parsiranje rezultata od strane automatskih skripti kako bi se utvrdilo da li se duplicirani kod promenio između *build*-ova.

Korisniku se rezultati vraćaju kao niz objekata koji u sebi sadrže po jedno podudaranje, odnosno pronađeni duplikat u dva fajla koji ga sadrže. Kako bi se ovakvi rezultati pregledno prikazali, preko kreirane aplikacije, osmišljeno je da se iz liste dobijenih objekata, klikom otvaraju oba fajla koja taj objekat obuhvata, a njihov duplikat označava posvetljavanjem linija na kojima se nalazi. Sumirani rezultati u vidu broja klonova, duplikata, pretraženih fajlova, procenta dupliciranosti koda i broja dupliciranih linija predstavljeni su komponentom u donjem levom uglu aplikacije.

4. EVALUACIJA RADA ALATA ZA PRONALAŽENJEDUPLICIRANOG KODA

Kako bi performanse alata JSInspect bile zaista evaluirane korišćen je još jedan alat za pronalaženje dupliciranog koda JSCPD, koji radi na potpuno drugačiji način. JSCPD alat radi na principu najsličnijem Rabin-Karpovom algoritmu [5]. Na osnovu tokena kreira heš vrednosti. Prvo prepozna sve fajlove koji bi trebalo da budu pročitani. Zatim prolazi kroz svaki fajl, čita liniju po liniju iz koje prepoznaje tokene. Nakon što su prepoznati tokeni, u svakom koraku iterator se pomera za jedan token. Uzima se onoliko tokena, u zavisnosti od parametra koji je pristigao kao minimalan broj tokena koji se prepoznaje kao duplikat. Zatim se od tih tokena kreira heš vrednost, koja se poredi sa mapom koja sadrži sve dotadašnje heš vrednosti. Ukoliko takav heš ne postoji, u mapu se dodaje novi heš. Heš funkcija je kreirana tako da može da razlikuje ključne reči od identifikatora. Ukoliko se neke dve funkcije ponašaju isto, sa različitim imenima varijabli, njihov heš kod biće identičan.

4.1 Prikupljanje

U svrhu analize rada ovih alata bilo je potrebno pribaviti par projekata sa velikim brojem JavaScript fajlova. Ovaj zadatak je lako urađen zbog velike količine *open-source* projekata na internetu.

Tabela 1:*Projekti korišćeni za analizu i njihove veličine*

Projekat	Veličina	Broj fajlova (total/JS)	Broj linija
Projekat1	12KB	4/4	84
Projekat2	21.8MB	1611/1600	314.575
Projekat3	473MB	15.001/6977	523.120

4.2 Analiza i rezultati

S obzirom na to da alati rade na potpuno drugačiji način prvo je trebalo utvrditi pod kojim parametrima vraćaju slične rezultate. U ovu svrhu analiziran je prvo folder od 7 fajlova, od kojih je 4 sa .js ekstenzijom.
Merenja su vršena sa default-nim parametrima oba alata i sa minimalnim parametrima (JSInspect: threshold = 10, JSCPD: minLines = 5, minTokens = 5) za koje je utvrđena najveća sličnost u vraćenim rezultatima. Rezultati merenja su prikazani tabelarno za svaki projekat pojedinačno, pokretana su oba alata za svaki odabrani ulaz. Vreme obuhvata period od klika na dugme modalnog dijaloga do krajnjeg prikaza rezultata na veb stranici. Broj fajlova predstavlja broj fajlova u kojima su pronađeni duplikati, a broj linija broj onih linija koje duplikati obuhvataju.

Tabela 2: *Sumiranje rezultata za Projekat1*

Alat	**Parametri**	**Vreme [ms]**	**Broj fajova**	**Broj linija**
JSInspect	10	310	4	127
JSCPD	5/5	750	4	69
JSInspect	30	310	0	0
JSCPD	5/70	560	3	42

Tabela 3: *Sumiranje rezultata za Projekat2*

Alat	**Parametri**	**Vreme [min:sec]**	**Broj fajova**	**Broj linija**
JSInspect	10	01:50	693	124.717
JSCPD	5/5	01:50	763	150.003
JSInspect	30	02:02	350	55.670
JSCPD	5/70	02:05	273	41.036

Tabela 4: *Sumiranje rezultata za Projekat3*

Alat	**Parametri**	**Vreme [min:sec]**	**Broj fajova**	**Broj linija**
JSInspect	10	03:49	1475	198.031
JSCPD	5/5	03:01	1533	260.195
JSInspect	30	02:26	846	85.727
JSCPD	5/70	01:48	665	62.814

Nakon obavljenih merenja u predstavljenim tabelama se vidi da su alati slični što se tiče vremena izvršavanja, međutim tokom rada nad velikim projektima primećena je velika mana JSInspect alata što se tiče zauzeća memorije. JSCPD alat je nad svakim projektom radio bez problema na default-nom memorijskom zauzeću od 512MB koje Node.js nudi, međutim, zbog prethodno pomenutog problema JSInspect alata, memorija za izvršavanje aplikacije je proširena na 4GB. Daljim tumačenjem dobijenih rezultata, primećena je mnogo veća preciznost pronadjenih klonova, kao i veći broj pogodaka kod JSCPD alata. Iz prethodne tri tabele se vidi da je JSInspect alat skoro uvek vraćao veći broj dupliranih linija koda, ovo se dešavalo zbog toga što njegova podudaranja često obuhvataju iste linije koda unutar istog fajla, što je još jedna mana ovog alata. Zaključak analize, na osnovu svih prethodno iznetih stavki, bi bila preporuka da se JSInspect alat koristi isključivo na manjim projektima, dok kod velikih projekata bi bilo korisnije upotrebiti analizu JSCPD alata.

5. ZAKLJUČAK

U ovom radu ukratko je napravljen uvod u JavaScript programski jezik kako bi se ukazalo na to kako je došlo do povećanja problema kakav je pojava dupliciranog koda, a spram toga i kako je došlo do pojave sve većeg broja alata za statičku analizu koda. Za dalju analizu odabran je alat čiji je cilj nalaženje klonova koda u isključivo JavaScript fajlovima, pa je spram toga opisan ukratko i njegov način rada. Jedan deo rada je posvećen i AST predstavi koda kao izrazito važnom delu ovakve analize. Kreirana je veb aplikacija u cilju integracije više alata za statičku analizu ne bi li se napravilo poređenje performansi alata kreiranih na različit način, primenom različitih algoritama, ali sa istim, zajedničkim ciljem: nalaženjem dupiliciranog koda.

6. LITERATURA

[1] N.Synytskyy, J.R.Cordy, and T.Dean.“Resolution of static clones in dynamic web pages”. In: Web Site Evolution, 2003. Theme: Architecture. Proceedings. Fifth IEEE International Workshop on. IEEE, 2003.

[2] F. Calefato, F. Lanubile, and T. Mallardo. “Function clone detection in web applications: A semiautomated approach”. In: Journal of Web Engineering 3 , pp. 3–21, 2004.

[3] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant’Anna, and Lorraine Bier. “Clone Detection Using Abstract Syntax Trees”. In: Proceedings of the International Conference on Software Maintenance. ICSM’98. Washington, DC, USA: IEEE Computer Society, 1998.

[4] Douglas Thain, “The Abstract syntax tree”, In: Compilers and language design, https://www3.nd.edu/~dthain/courses/cse40243/fall2016/chapter6.pdf , 2016.

[5] Robert Sedgewick, Kevin Wayne, “Algorithms, 4th Edition” , 2011.

Kratka biografija

Tamara Letić rođena je 05.05.1994. u Novom Sadu, Srbiji. Završila je Osnovnu školu “Jovan Jovanović Zmaj” u Rumi, posle čega upisuje Gimnaziju “Stevan Puzić”, društveno-jezički smer. Fakultet tehničkih nauka, smer Elektroenergetski softverski inženjering, upisuje školske 2013/2014. godine. Osnovne studije je završila 2017.godine, nakon čega je upisala master studije, smer Primenjeno softversko inženjerstvo.