UNIVERZA V MARIBORU
FAKULTETA ZA ELEKTROTEHNIKO,
RAČUNALNIŠTVO IN INFORMATIKO

Matej Zgubič

STATIČNI ANALIZATOR ZA PREVERJANJE KVALITETE PROGRAMSKE KODE

Diplomsko delo

Maribor, avgust 2015

STATIČNI ANALIZATOR ZA PREVERJANJE KVALITETE PROGRAMSKE KODE

Diplomsko delo

Študent:	Matej Zgubič
Študijski program:	univerzitetni študijski program
	Računalništvo in informacijske tehnologije
Mentor:	doc. dr. Tomaž Kosar
Lektoriranje:	Lektoriranje in jezikovno svetovanje,
	Jasmina Vajda Vrhunec, s. p.

Fakulteta za elektrotehniko,
računalništvo in informatiko
Smetanova ulica 17
2000 Maribor, Slovenija

Številka: E1053803
Datum in kraj: 01. 10. 2014, Maribor

Na osnovi 330. člena Statuta Univerze v Mariboru (Ur. l. RS, št. 46/2012)
izdajam

SKLEP O DIPLOMSKEM DELU

1. **Mateju Zgubiču,** študentu univerzitetnega študijskega programa RAČUNALNIŠTVO IN INFORMACIJSKE TEHNOLOGIJE, se dovoljuje izdelati diplomsko delo pri predmetu Prevajanje programskih jezikov.

2. **MENTOR:** **doc. dr. Tomaž Kosar**

3. **Naslov diplomskega dela:**
   STATIČNI ANALIZATOR Z PREVERJANJE KVALITETE PROGRAMSKE KODE

4. **Naslov diplomskega dela v angleškem jeziku:**
   STATIC ANALYSER FOR CHECKING THE QUALITY OF SOFTWARE CODE

5. Diplomsko delo je potrebno izdelati skladno z "Navodili za izdelavo diplomskega dela". Skladno s 7. členom *Pravilnika o postopku priprave in zagovora diplomskega dela na dodiplomskem študiju* je bilo odobreno podaljšanje roka za oddajo diplomskega dela do 30. 09. 2015. Diplomsko delo študent-ka odda v treh izvodih (dva vezana izvoda in en v spiralo vezan izvod) ter en izvod elektronske verzije v referatu za študentske zadeve.

Pravni pouk: Zoper ta sklep je možna pritožba na senat članice v roku 3 delovnih dni.

Dekan:
red. prof. dr. Borut Žalik

Obvestiti:
- kandidata,
- mentorja,
- odložiti v arhiv.

ZAHVALA

Zahvaljujem se mentorju za izkazano zaupanje, potrpežljivost in pomoč pri izdelavi diplomskega dela. Zahvaljujem se tudi staršem za podporo med študijem.

STATIČNI ANALIZATOR ZA PREVERJANJE KVALITETE PROGRAMSKE KODE

Ključne besede: statični analizator, kvaliteta programske kode, metrike, ANTLR.

UDK: 004.412(043.2)

Povzetek

V diplomskem delu je opisana statična analiza programske opreme. V prvem delu so opisani prednosti in slabosti, primeri uporabe, tehnike ter kategorije statične analize. V nadaljevanju so podrobneje opisane metrike za merjenje kvalitete programske kode, kot so ciklomatična kompleksnost, indeks vzdrževanja, Halsteadova kompleksnost, globina razreda v drevesu, število vrstic izvorne kode in druge. V zadnjem delu so opisani gramatika, leksikalna analiza, razpoznavanje, ANTLR in moja izdelava programa za merjenje metrik.

STATIC ANALYSER FOR CHECKING THE QUALITY OF SOFTWARE CODE

Key words: static analyser, quality of the source code, metrics, ANTLR

UDK: 004.412(043.2)

Abstract

This thesis describes the static analysis of a software. In the first part, the advantages and disadvantages, examples of use and categories of static analysis are listed. The next part contains a detailed description of metrics for measuring the quality of program codes, such as cyclomatic complexity, maintainability index, Halstead complexity, depth in tree, number of source code lines and others. In the last part, grammar, lexical analysis, parsing, ANTLR, and my making of a metrics measuring program are described.

KAZALO VSEBINE

1 UVOD 1
2 STATIČNA ANALIZA 2
2.1 PREDNOSTI IN SLABOSTI STATIČNE ANALIZE 2
2.2 PRIMERI UPORABE STATIČNIH ANALIZATORJEV V GOSPODARSTVU 4
2.3 KATEGORIJE STATIČNIH ANALIZATORJEV 4
3 TEHNIKE STATIČNE ANALIZE 5
3.1 PRETOK PODATKOV 5
3.2 GRAF NADZORA PRETOKA PODATKOV 6
4 METRIKE 8
4.1 UPORABA 8
4.2 TIPI METRIK 9
4.3 CIKLOMATIČNA KOMPLEKSNOST 10
4.4 INDEKS VZDRŽEVANJA 15
4.5 GLOBINA RAZREDA V DREVESU 17
4.6 ŠTEVILO VRSTIC IZVORNE KODE 17
4.7 OSTALE METRIKE 18
5 RAZUMEVANJE GRAMATIKE 21
5.1 LEKSIKALNA ANALIZA 23
5.2 RAZPOZNAVANJE 24
6 IZDELAVA STATIČNEGA ANALIZATORJA 25
6.1 OPIS PROBLEMA 25
6.2 UPORABLJENA ORODJA 25
6.2.1 ANTLR 25
6.2.2 DODATEK ANTLR ZA ECLIPSE 26
6.3 IZDELAVA 27
6.3.1 POMOŽNI RAZREDI 29
6.4 IZDELAVA UPORABNIŠKEGA VMESNIKA 31
7 UPORABA UPORABNIŠKEGA VMESNIKA 34
8 ZAKLJUČEK 38
9 VIRI IN LITERATURA 39

KAZALO SLIK

Slika 3.1: Prikaz vozlišč v grafu nadzora podatkov ... 6
Slika 3.2: Primer grafa nadzora podatkov [14] ... 7
Slika 4.1: Graf pretoka za primer ... 12
Slika 4.2: Prikaz indeksa vzdrževanja v orodju Visual Studiu ... 16
Slika 5.1: Sintaktično drevo za programski jezik Java (izrisano s pomočjo programa ANTLRWorks) ... 21
Slika 5.2: Primer treh različnih sintaktičnih dreves iz treh različnih programov z isto gramatiko [19] . 24
Slika 6.1: ''Railroad View'' za deklaracijo ''enum'' razreda ... 27
Slika 7.1: Postopek izbiranja datotek za analizo ... 34
Slika 7.2: Označevanje datotek in map za analizo ... 35
Slika 7.3: Prikaz analiziranih datotek ... 35
Slika 7.4: Prikaz rezultatov analize ... 35
Slika 7.5: Prikaz hierarhije razredov ... 36
Slika 7.6: Prikaz filtriranja rezultatov v tabeli ... 37

KAZALO TABEL

Tabela 2.1: Povprečna cena (v enotah) pri odpravljanju napak glede na čas njene vpeljave in zaznave [24] ... 3
Tabela 4.1: Mejne vrednosti ciklomatične kompleksnosti [21] ... 11
Tabela 4.2: Pomen parametrov pri izračunu Halsteadove kompleksnost ... 13
Tabela 4.3: Metrike pri Halsteadovi kompleksnosti in njene enačbe ... 13
Tabela 4.4: Seznam operatorjev za primer ... 14
Tabela 4.5: Seznam operandov za primer ... 14
Tabela 4.6: Opis ostalih metrik [25] ... 20
Tabela 5.1: Tokenizacija za primer ''sum = 3 + 2;''[6] ... 23

1 UVOD

Kvaliteta programske opreme je izjemno pomembna pri prodaji izdelka in ohranjanju dobrega imena podjetja. Že s samo enim slabo izdelanim izdelkom lahko škodujemo ugledu podjetja in s tem zmanjšamo prodajo prihodnjim izdelkom. Pri programski opremi je pomembno, da ne prihaja do napak, predvsem pri sistemih, kjer lahko napaka povzroči ogromno škodo (na primer pri bančnih sistemih). Napak na takšnih sistemih ne sme biti. Za pomoč pri izdelavi kvalitetne programske opreme si lahko pomagamo s statično analizo izvorne kode. S statično analizo preverjamo izvorno kodo brez zaganjanja programa. Z njo analiziramo, če so kakšne napake v izvorni kodi. To pa ni edina funkcionalnost statične analize. Z njo lahko določimo tudi kvantitativne vrednosti za določene meritve. Kvantitativne metode merjenja so zelo pomembne na vseh znanstvenih področjih. Cilj metrik je pridobiti objektivne, ponovljive in merljive rezultate, ki so lahko koristni pri zagotavljanju kvalitete programske opreme, zmogljivosti, odpravljanju napak, vodstvu in ocenjevanju stroškov. Uporabljajo se za iskanje napak v kodi, napovedovanje napak, napovedovanje uspešnosti projekta in napovedovanje tveganja projekta. Z njimi lahko izračunamo kompleksnost in prepoznamo težavnost vzdrževanja izvorne kode.

2 STATIČNA ANALIZA

Statična programska analiza je analiza programske opreme, ki se izvaja brez zaganjanja programa (analiza, ki se izvaja s pomočjo zagona programov, je znana kot dinamična analiza). V večini primerov gre za izvajanje analize nad izvorno kodo, v nekaterih primerih pa tudi nad njo. Analizo izvedemo s pomočjo avtomatiziranega orodja, lahko pa kodo pregledamo sami [36].

Ročni pregled kode je ena izmed najstarejših in najvarnejših metod za zaznavanje napak v kodi. Ta proces razkrije napake v kodi ali pa dele kode, ki bi lahko postali napake v prihodnosti. Od avtorja določenega odseka izvorne kode se pričakuje, da mu ni treba razlagati, kako določen del programa deluje. To mora biti takoj razvidno iz kode in komentarjev ob njej. Če ni tako, je treba kodo izboljšati. Ta način preverjanja kode deluje dobro, ker programerji dosti lažje opazimo napake v kodi nekoga drugega kot v naši lastni kodi.

Edina večja slabost tega načina pregledovanja kode je velika cena. Za ta način pregledovanja kode potrebujemo veliko programerjev in veliko časa [34].

2.1 PREDNOSTI IN SLABOSTI STATIČNE ANALIZE

Kot vsaka metodologija za zaznavanje napak ima tudi statična analiza dobre in slabe lastnosti. Pri statični analizi ni idealnih metod. Različne metode proizvedejo različne rezultate in kombinacija teh metod nam omogoča kar se da najboljšo kvaliteto naše programske opreme [34].

Prednosti

Glavna prednost statične analize je znaten prihranek denarja za odstranjevanje napak v kodi. Prej ko napako odkrijemo, manjša je cena za njeno odpravo.

Iz tabele 2.1 je razvidno, da odprava napak med testiranjem stane približno desetkrat več kot v fazi kodiranja.

Tabela 2.1: Povprečna cena (v enotah) pri odpravljanju napak glede na čas njene vpeljave in zaznave [24]

	Faza, v kateri smo napako zaznali				
Faza, v kateri smo napako ustvarili	Zahteve	Arhitektura	Konstrukcija	Sistemski test	Po izidu programa
Zahteve	1	3	5–10	10	10–100
Arhitektura	-	1	10	15	25–100
Konstrukcija	-	-	1	10	10–25

Naslednja prednost je pokritost celotne kode. Statični analizatorji preverijo tudi tiste dele kode, ki se redko uporabljajo. Teh delov kode ne moremo testirati prek drugih metod. Omogočajo nam iskanje napak v delih kode, kjer obravnavamo izjeme.

Statični analizatorji niso odvisni od prevajalnika in okolja, kjer zaganjamo program. Omogočajo nam iskanje skritih napak, ki bi se lahko odkrile šele nekaj let kasneje, na primer "napake nedefiniranega vedenja" (angl. undefined behavior errors). Takšne napake se lahko pojavijo pri menjavi prevajalnika in so značilne predvsem za jezika C in C++.

S pomočjo statičnih analizatorjev lahko hitro in enostavno odkrijemo tiskarske napake in napake, ki so nastale zaradi kopiranja in lepljenja delov kode. Zaradi teh napak lahko izgubimo veliko časa.

Slabosti

Glavna slabost statičnih analizatorjev je, da so po navadi slabi pri diagnosticiranju nesproščenega pomnilnika. Za zaznavo takšnih napak je treba program zagnati. Za tako diagnosticiranje se uporabljajo dinamični analizatorji.

Slabost statičnih analizatorjev je tudi to, da lahko zaznajo del kode, ki je povsem pravilna, kot napako. Te napake se imenujejo tudi lažno pozitivne napake [34].

2.2 PRIMERI UPORABE STATIČNIH ANALIZATORJEV V GOSPODARSTVU

Rastoča gospodarska uporaba statične analize se opazi pri preverjanju lastnosti programske opreme, ki se uporabljajo v računalniških sistemih, pomembnih za varnost in lociranje potencialno ranljive kode. Na primer naslednje gospodarske panoge uporabljajo statično analizo kot sredstvo za izboljšanje kakovosti programske opreme [34]:

1. medicina: ameriška administracija za prehrano in zdravila (FDA) jo uporablja za medicinsko programsko opremo;
2. nuklearna industrija: britanska organizacija za zdravje in varnost jo uporablja pri varnostnih sistemih reaktorjev;
3. letalstvo;
4. bančništvo;
5. informacijske tehnologije.

2.3 KATEGORIJE STATIČNIH ANALIZATORJEV

Statični analizatorji se glede na naloge delijo v tri kategorije [34]:

1. iskanje napak v programih;
2. nasveti glede oblikovanja kode (zamiki, število presledkov, tabulatorjev). Nekateri statični analizatorji nam omogočajo preverjanje skladnosti oblikovanja kode s pravili podjetja;
3. izračunavanje metrik. Metrike nam omogočajo določiti številsko vrednost za določeno lastnost programa ali specifikacij.

3 TEHNIKE STATIČNE ANALIZE

3.1 PRETOK PODATKOV

Analiza pretoka podatkov (angl. Data-flow analysis) je tehnika za zbiranje informacij o možnih nizov podatkov, izračunanih pri različnih točkah programa. Prevajalnik pogosto uporabi zbrane informacije med optimizacijo programa. Kanoničen primer analize pretoka podatkov je doseganje definicij. Preprost način za izvedbo analize pretoka podatkov v programu je priprava enačb pretoka podatkov za vsako vozlišče grafa nadzora pretoka podatkov, ki se jih rešiti z večkratnim lokalnim izračunavanjem izhoda iz danega vhoda pri vsakem vozlišču, dokler se cel sistem ne stabilizira, torej ko doseže fiksno točko. Ta splošni pristop je razvil Gary Kildall [10].

Primer:

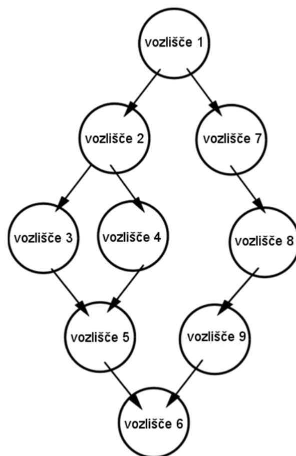
```
1.   if b==4 then
2.      a = 5;
3.   else
4.      a = 3;
5.   endif
6.
7.   if  a < 4 then
8.      ...
```

Dosežena definicija spremenljivke "a" v vrstici 7 je niz prirejanja vrednosti spremenljivki "a" v vrsticah 2 in 4 [10].

3.2 GRAF NADZORA PRETOKA PODATKOV

Graf nadzora podatkov v računalništvu je predstavitev vseh poti programa, ki jih lahko prepotuje program med izvajanjem. Je ključnega pomena za prevajalnike med optimizacijo in za orodja statične analize [7].

Je abstraktna predstavitev programske opreme z uporabo vozlišč, ki predstavljajo osnovne bloke kode. Vozlišče v grafu predstavlja blok kode, usmerjeni robovi pa predstavljajo pot od enega do drugega bloka. Če ima vozlišče samo izhodni rob, se imenuje vhodni blok. Če ima blok samo vhodni rob, se imenuje izhodni blok. Slika 3.1 prikazuje vozlišča v grafu nadzora podatkov. Vozlišče 1 je v tem primeru vhodni blok, vozlišče 6 pa izhodni blok [35].

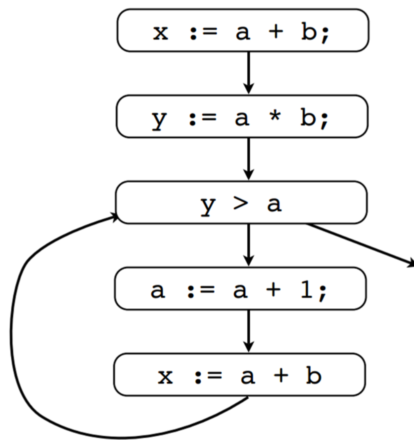


Slika 3.1: Prikaz vozlišč v grafu nadzora podatkov

Primer:

```
x := a + b;
y := a * b;
while (y > a) {
        a := a + 1;
        x := a + b
}
```

Slika 3.2 prikazuje potek pretoka podatkov z "while" zanko. Dokler je spremenljivka y večja od spremenljivke a, se a poveča za 1, x pa dobi vrednost vsote spremenljivk a in b.



Slika 3.2: Primer grafa nadzora podatkov [14]

.

4 METRIKE

Metrika programske opreme je merilo določene lastnosti nekega dela programa ali njegovih specifikacij. Kvantitativne metode merjenja so zelo pomembne na vseh znanstvenih področjih, zato se tudi v računalništvo vedno bolj trudijo za vpeljavo tega načina merjenja. Cilj je pridobiti objektivne, ponovljive in merljive rezultate [28].

Metrike se razlikujejo od preverjanja napak v tem, da ponujajo večje število informacij o naslednjih vidikih programske opreme [4]:

- kvaliteta programske opreme;
- urnik celotnega projekta;
- cena projekta programske opreme, ki vključuje vzdrževanje, raziskave in ostale stroške;
- velikost/kompleksnost sistema programske opreme.

4.1 UPORABA

Metrike se uporabljajo za pridobivanje objektivnih, ponovljivih meritev, ki so lahko koristne pri zagotavljanju kvalitete programske opreme, zmogljivosti, odpravljanju napak, vodstvu in ocenjevanju stroškov. Uporabljajo se za iskanje napak v kodi, napovedovanje napak, napovedovanje uspešnosti projekta in napovedovanje tveganja projekta [5].

4.2 TIPI METRIK

Metrike za zahteve programske opreme [29]**:**

- dokaj primitivne in predvidljive;
- funkcijske točke:
  - število vhodov in izhodov, uporabniških interakcij, zunanjih vmesnikov, uporabljenih datotek;
  - ocenitev kompleksnosti vseh funkcijskih točk in množenje z oteženim faktorjem (angl. weighting factor);
  - uporabljajo se za predvidevanje velikosti ali stroškov in za ocenitev produktivnosti projekta;
- število najdenih napak v zahtevah.

Produktivnost programerja [30]**:**

- ni možno meriti neposredno, saj je programska oprema nematerialna;
- če je slaba programska oprema narejena hitro, lahko daje vtis večje produktivnosti kot pa zanesljiva koda, ki je hkrati enostavna za vzdrževanje:
  - več ne pomeni vedno bolje;
  - lahko poveča stroške vzdrževanja sistema;
- pogoste meritve:
  - število napisanih vrstic na mesec;
  - število napisanih objektnih ukazov na mesec;
  - število strani napisane dokumentacije na mesec;
  - število napisanih in izvedenih testnih preizkusov na mesec.

Metrike za načrtovanje programske opreme [31]**:**

- število parametrov;
- število modulov;
- število klicanih modulov;
- povezanost podatkov;
- kohezija.

Vodstvene metrike

Tehnike za ugotavljanje stroškov [32]:

1. algoritmično modeliranje stroškov:
    - model temelji na podatkih iz zgodovine in predstavlja povezavo med neko metriko (po navadi število vrstic kode) in stroški projekta;
    - najbolj znanstvena metoda, ki pa ni nujno najnatančnejša;
2. profesionalna presoja;
3. ocena po analogiji: uporabno, kadar smo dokončali projekte iz iste domene.

4.3 CIKLOMATIČNA KOMPLEKSNOST

Ciklomatična kompleksnost je metrika programske opreme, ki jo je razvil Thomas J. McCabe leta 1976. Uporablja se za ugotavljanje kompleksnosti programa. Je kvantitativno merilo kompleksnosti programskih ukazov. Neposredno meri število neodvisnih poti čez izvorno kodo programa. Ciklomatično kompleksnost se izračuna s pomočjo grafa za nadzor pretoka programa. Lahko jo izračunamo tudi za posamezne funkcije, module, metode ali razrede znotraj programa [8].

Mejne vrednosti ciklomatične kompleksnosti posameznih kategorij, ki jih je vpeljal Inštitut inženirstva za programsko opremo (Software Engineering Institute), so prikazane v tabeli 4.1 [21].

Tabela 4.1: Mejne vrednosti ciklomatične kompleksnosti [21]

Ciklomatična kompleksnost	Ocenitev tveganja
1–10	Preprost modul z malo tveganja
11–20	Malo bolj zapleten modul z zmernim tveganjem
21–50	Kompleksen modul z velikim tveganjem
51 in več	Program, ki ga je težko testirati in predstavlja zelo veliko tveganje

Ciklomatično kompleksnost lahko izračunamo z naslednjo enačbo:

$$CC = E - N + P \tag{4.1}$$

Pri tem so [9]:

CC – ciklomatična kompleksnost;

E – število robov v grafu za nadzor pretoka;

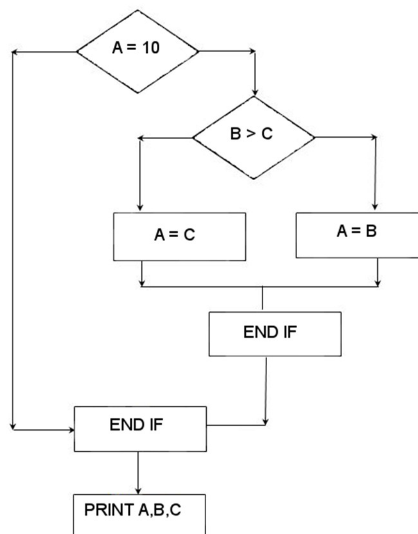
N – število vozlišč v grafu;

P – število vozlišč v grafu, ki imajo izhodno točko.

Primer:

```
IF A = 10 THEN
        IF B > C THEN
                A = B
        ELSE
                A = C
        ENDIF
ENDIF
Print A
Print B
Print C
```

Ciklomatična kompleksnost zgornjega primera se izračuna s pomočjo diagrama na sliki 4.1 in enačbe (4.1) na naslednji način: diagram prikazuje sedem vozlišč in osem robov. Iz tega sledi, da je ciklomatična kompleksnost enaka 3 (8 – 7 + 2 = 3) [9].



Slika 4.1: Graf pretoka za primer

Obstajata pa še dve drugi enačbi za izračun ciklomatične kompleksnosti:

$$CC = 1 + ifs + loops + cases \tag{4.2}$$

$$CC = 2 + ifs + loops + cases - returns \tag{4.3}$$

Pri tem so [22], [23]:

CC – ciklomatična kompleksnost;

Ifs – število IF-operatorjev;

Loops – število zank;

Cases – število vejitev pri SWITCH stavku (brez DEFAULT vejitve);

Returns – število RETURN operatorjev v funkciji.

HALSTEADOVA KOMPLEKSNOST

Halsteadova kompleksnost je metrika, ki jo je leta 1977 razvil Maurice Howard Halstead. Ta metrika je bila razvita za merjenje kompleksnosti programskega modula neposredno iz izvorne kode s poudarkom na računski kompleksnosti. Razvita je bila z namenom določanja kvantitativne vrednosti kompleksnosti, neposredno iz operatorjev in operandov v modulu [15], [16].

Izračun Halsteadove kompleksnosti

Tabela 4.2: Pomen parametrov pri izračunu Halsteadove kompleksnost

Parameter	Pomen
n_1	Število različnih operatorjev
n_2	Število različnih operandov
N_1	Število vseh operatorjev
N_2	Število vseh operandov

S pomočjo teh podatkov lahko izračunamo več različnih metrik, ki so opisane v tabeli 4.3.

Tabela 4.3: Metrike pri Halsteadovi kompleksnosti in njene enačbe

Metrika	Pomen	Enačba
n	Besednjak	$n_1 + n_2$
N	Velikost	$N_1 + N_2$
V	Volumen	$N * \log_2 n$
D	Težavnost	$\frac{n_1}{2} * \frac{N_2}{n_2}$
E	Napor	$V * D$
B	Napake	$\frac{V}{3000}$
T	Čas za implementacijo	$\frac{E}{k}$

V tabeli 4.3 je vrednost parametra *k* po navadi 18, lahko pa ga tudi, glede na pogoje, spremenimo [16].

Primer preproste funkcije (sortiranje števil po velikosti) [27]:

```
void sort(int *a, int n) {
	int i, j, t;

	if (n < 2) return;
	for (i = 0; i < n - 1; i++) {
		for (j = i + 1; j < n; j++) {
			if (a[i] > a[j]) {
				t = a[i];
				a[i] = a[j];
				a[j] = t;
			}
		}
	}
}
```

Tabela 4.4: Seznam operatorjev za primer

Operator	Število	Operator	Število
<	3	{	3
=	5	}	3
>	1	+	1
-	1	++	2
,	2	for	2
;	9	if	2
(	4	int	1
)	4	return	1
[]	6		

Tabela 4.5: Seznam operandov za primer

Operand	Število
0	1
1	2
2	1
a	6
i	8
j	7
n	3
t	3

Izračun:

$N_1 = 50$, $N_2 = 30$, $n_1 = 17$, $n_2 = 7$

$$n = n_1 + n_2 = 17 + 7 = 24$$

$$N = N_1 + N_2 = 50 + 30 = 80$$

$$V = N * \log_2 n = 80 * \log_2 24 \cong 366{,}8$$

$$D = \frac{n_1}{2} * \frac{N_2}{n_2} = \frac{17}{2} * \frac{30}{7} \cong 36{,}4$$

$$E = V * D = 366{,}8 * 36{,}4 \cong 13351{,}5$$

$$B = \frac{V}{3000} = \frac{366{,}8}{3000} \cong 0{,}122$$

$$T = \frac{E}{18} = \frac{13351{,}5}{18} = 741{,}75\ s \approx 12\ min$$

4.4 INDEKS VZDRŽEVANJA

Indeks vzdrževanja je metrika programske opreme, ki meri, kako enostavno je izvorno kodo programa vzdrževati in spreminjati. Indeks vzdrževanja se izračuna s pomočjo števila vrstic kode, ciklomatično kompleksnostjo in Halsteadovim volumnom (glej tabelo 4.3). Uporablja se v mnogih avtomatiziranih orodjih za merjenje metrik, vključno z orodjem Visual Studiom, ki uporablja lestvico od 0 do 100. Večja kot je vrednost, enostavnejše je vzdrževanje kode. Lahko uporabljamo tudi barvno lestvico za lažje prepoznavanje težav v kodi. Zelena barva predstavlja vrednosti od 20 do 100, rumena od 10 do 19, rdeča pa od 0 do 9. Na sliki 4.2 je prikazan indeks vzdrževanja kot ga izpiše Visual Studio [20].

Code Metrics Results

Filter: None Min: Max:

Hierarchy	Maintainability Index
3. b naloga (Debug)	78
_3.b_naloga	78
App	100
App()	100
DBConnect	56
danVTednuSLO(DateTime) : string	52
DBConnect()	100
pisiVDatoteko(string) : void	67
PoveziMe() : void	72
preveriAdmin(string, string) : bool	55
preveriGeslo(string, string) : bool	55
preveriUPIme(string) : bool	56
PridobiPodatkePosameznik(string) : Uporabnik	51
PridobiPodatkePosameznikGledeNaID(int) : Uporabnik	51
PridobiPodatkeSeznam() : List<Uporabnik>	50

Slika 4.2: Prikaz indeksa vzdrževanja v orodju Visual Studiu

Običajne enačbe za izračun indeksa vzdrževanja so:

- originalna enačba:

$$MI = 171 - 5.2\ln V - 0.23G - 16.2\ln L\,; \qquad (4.4)$$

- izpeljava enačbe, ki jo uporabljajo na Inštitutu inženirstva za programsko opremo:

$$MI = 171 - 5.2\log_2 V - 0.23G - 16.2\log_2 L + 50\sin\left(\sqrt{2.4\,C}\right); \qquad (4.5)$$

- izpeljava enačbe, ki jo uporablja Visual Studio:

$$MI = \max\left[0, 100\ \frac{171 - 5.2\ln V - 0.23G - 16.2\ln L}{171}\right]. \qquad (4.6)$$

Pri tem so [17]:

MI – indeks vzdrževanja (angl. maintainability index);

G – skupna vrednost ciklomatične kompleksnosti;

L – število vrstic kode;

C – odstotek vrstic s komentarji.

4.5 GLOBINA RAZREDA V DREVESU

Globina razreda v drevesu (angl. Depth in Tree (DIT)) izračunava, kako globoko v hierarhiji dedovanja je določen razred deklariran, kjer je DIT razdalja od razreda do korena drevesa dedovanja. V programskem jeziku Java je razred "java.lang.Object" končni nadrazred vseh razredov in je definiran z globino 0. Vsak razred, ki je neposredni podrazred tega razreda, ima vrednost globine 1. Naslednji podrazred tega podrazreda ima globino 2 in tako dalje.

Definicija globine razreda:

- vsi tipi vmesnikov imajo globino 1;
- razred "java.lang.Object" ima globino 0;
- vsi ostali razredi imajo globino za 1 večjo od njihovega neposrednega nadrazreda.

Globlje kot je razred v hierarhiji, večje število metod in spremenljivk podeduje, kar otežuje napoved njegovega vedenja. Razred postane bolj specializiran in lahko postane težavno razumeti sistem z veliko plastmi dedovanja. Kljub temu pa obstaja večja možnost ponovne uporabe določenih metod. Največja priporočena vrednost globine je 5, ker večja globina pomeni večjo zapletenost sistema [11].

4.6 ŠTEVILO VRSTIC IZVORNE KODE

Število vrstic izvorne kode (SLOC ali LOC) je metrika programske opreme, ki se uporablja za meritev velikosti programa s štetjem števila vrstic kode. Po navadi se uporablja za napovedovanje napora, ki je potreben tako za razvoj programa kot za oceno produktivnosti ali napora po končanem programiranju.

Metode merjenja

Obstajata dve glavni metodi merjenja: fizične vrstice kode (LOC) in logične vrstice kode (LLOC). Najpogostejša definicija merjenja fizičnih vrstic kode je število vseh vrstic v izvorni kodi, vključno s komentarji. Prav tako se štejejo prazne vrstice, razen če je v določenem odseku več kot 25 % praznih vrstic. V tem primeru se štejejo vrstice do 25 %, vse ostale vrstice se ne štejejo.

Logični način štetja vrstic poskuša meriti število stavkov, vendar se specifične definicije razlikujejo med posameznimi programskimi jeziki. Dosti lažje je narediti orodje, ki uporablja metodo fizičnih vrstic kot pa metodo logičnih vrstic [33].

Primer 1 [33]:

```
for (i = 0; i < 100; i += 1) printf("hello"); /* Koliko je vrstic kode? */
```

V tem primeru imamo:

- 1 fizično vrstico kode;
- 2 logični vrstici kode (stavek "for" in "printf");
- 1 vrstico s komentarjem.

Primer 2 [33]

Kodo v prejšnjem primeru lahko zapišemo tudi drugače:

```
for (i = 0; i < 100; i += 1)
{
    printf("hello");
} /* Koliko vrstic kode pa je zdaj? */
```

V tem primeru imamo:

- 4 fizične vrstice kode;
- 2 logični vrstici kode;
- 1 vrstico s komentarjem.

4.7 OSTALE METRIKE

Abstraktnost (A)

Ta linearna metrika meri, kako abstrakten je določen paket. Izračuna razmerje med abstraktnimi in konkretnimi razredi. Torej če je vrednost A = 0, to pomeni, da je paket v celoti konkreten, če pa je A = 1, to pomeni, da je paket v celoti abstrakten. Abstraktnost se izračuna s pomočjo enačbe (4.7).

$$A = \frac{Na}{Nc} \tag{4.7}$$

Pri tem so [2]:

A – abstraktnost;

Na – število abstraktnih razredov in vmesnikov v paketu;

Nc – število konkretnih razredov in vmesnikov v paketu.

Izvršljivi stavki (EXEC)

Ta metrika predstavlja število izvršljivih stavkov. To so stavki, ki zahtevajo eksplicitno akcijo. Ta metrika je objektivnejša od merjenja števila vrstic kode, saj ni odvisna od programerjevega stila pisanja kode [13].

Primer [13]:

```
int a = 13;
int b;
b = a + 4; // <- izvršljiv stavek
a++, b++; // <- dva izvršljiva stavka
if (b < 6) {
       System.out.println(// <- izvršljiv stavek
                  "Hello World");
}
```

DSQI (angl. design structure quality index)

DSQI je metrika arhitekturne zasnove, ki se uporablja za oceno strukture programske zasnove. Metriko so razvili v Združenih državah Amerike, v sistemskem centru za zračne sile. Vrednost DSQI je število med 0 in 1. Bližje kot je 1, boljše kvalitete je programska oprema. Metriko se splača uporabljati za primerjavo s prejšnjimi uspešnimi projekti [12].

Število tipov (NOT)

Ta metrika predstavlja število različnih razredov in vmesnikov na ravni razreda in paketa ter za celoten sistem. Večja kot je vrednost NOT, bolj je lahko zapleten sistem, kar pomeni, da je potencialnih interakcij med objekti več. To pa zmanjša razumljivost sistema, kar ga naredi težjega za testirati, razhroščevati in vzdrževati [26].

Tabela 4.6: Opis ostalih metrik [25]

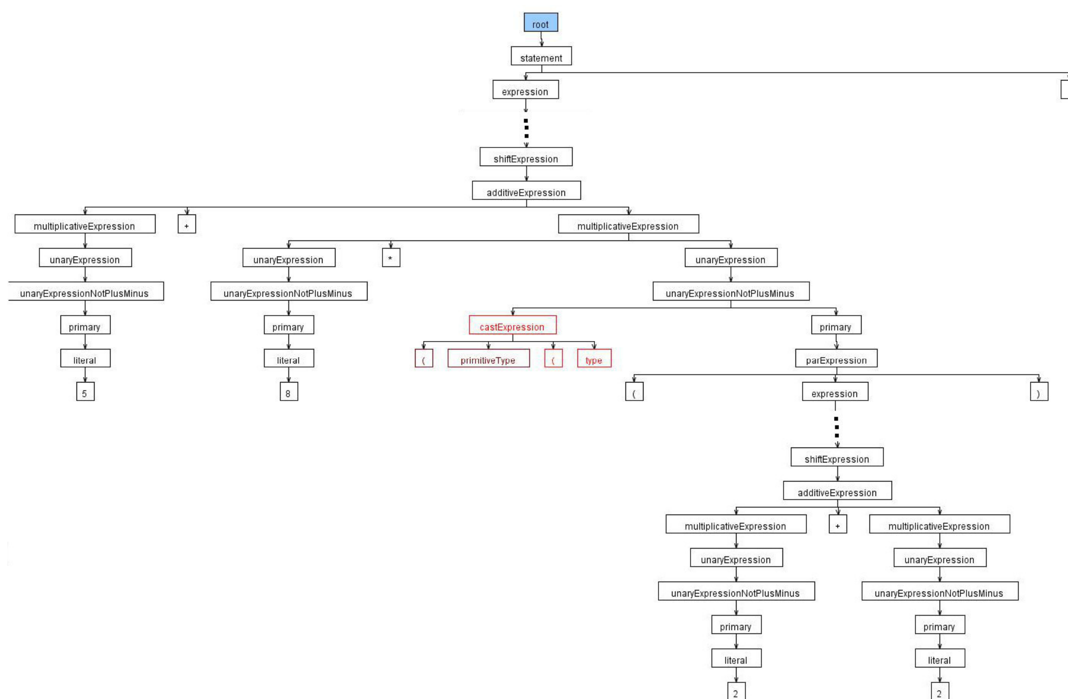
Ime	Opis
Dohodni sklop (angl. Afferent Coupling (Ca))	Število razredov iz drugih paketov, ki so odvisni od razredov v analiziranem paketu.
Odhodni sklop (angl. Efferent Coupling (Ce))	Število tipov iz analiziranega paketa, ki so odvisni od tipov iz drugih paketov.
Nestabilnost (angl. Instability) (I = Ce / (Ca + Ce))	Izračuna stabilnost/nestabilnost določenega paketa.
Otežene metode na razred (angl. Weighted Methods per Class)	Vsota vseh vrednosti ciklomatične kompleksnosti metod za določen razred.
Število otrok v drevesu (angl. Number of Children in Tree)	Število neposrednih podrazredov nekega razreda.
Odzivnost razreda (angl. Response for Class)	Število različnih metod in konstruktorjev, ki jih pokliče določen razred.

5 RAZUMEVANJE GRAMATIKE

Programski jezik je opisan s pomočjo semantike in sintakse. Semantika nam daje pomen določenega dela kode. Sintaksa daje programskemu jeziku strukturo. Za opis semantike obstaja veliko načinov, medtem ko se za opis sintakse večinoma uporablja ena tehnologija. Ta formalni način opisa se imenuje kontekstno prosta gramatika, kar pomeni, da je neodvisna od programskega jezika.

Gramatika nam omogoča transformirati program, ki je predstavljen kot linearni niz znakov ASCII v sintaktično drevo. Samo programi, ki so sintaktično pravilni, so lahko pretvorjeni na ta način. To drevo je glavna podatkovna struktura za prevajalnik ali interpreter. Po analizi tega drevesa ga lahko prevajalnik pretvori v strojno kodo [18].

Na sliki 5.1 je prikazan primer sintaktičnega drevesa za programski jezik Java. Uporabljen je bil preprost primer: "5 + 8 * (2 + 2);".



Slika 5.1: Sintaktično drevo za programski jezik Java (izrisano s pomočjo programa ANTLRWorks)

Dandanes se za glavno notacijo gramatike uporablja obrazec Backus-Naur (BNF). Ta notacija, ki jo je izumil John Backus, kasneje pa izboljšal Peter Naur, se je najprej uporabljala za opis sintakse programskega jezika Algol.

Gramatika BNF je definirana s štirimi elementi (T, N, P, S). Pomen teh elementov je naslednji:

- T predstavlja komplet terminalnih simbolov. So najmanjša enota sintakse. To so na primer: "for", "while", "+";
- N je komplet neterminalov. Neterminali niso del jezika. Lahko jih imenujemo tudi sintaktične spremenljivke;
- P predstavlja komplet produkcij;
- S je začetni simbol.

To je primer zelo preproste gramatike, ki prepozna aritmetične izraze. Z drugimi besedami, vsak program v tem preprostem jeziku predstavlja produkt ali seštevek simbolov 'a', 'b' in 'c' [18].

```
<exp> ::= <exp> "+" <exp>
<exp> ::= <exp> "*" <exp>
<exp> ::= "(" <exp> ")"
<exp> ::= "a"
<exp> ::= "b"
<exp> ::= "c"
```

To gramatiko lahko zapišemo tudi takole [18]:

```
<exp> ::= <exp> "+" <exp> | <exp> "*" <exp> | "(" <exp> ")" | "a" | "b" | "c"
```

5.1 LEKSIKALNA ANALIZA

V računalništvu je leksikalna analiza proces pretvarjanja niza znakov v niz terminalnih simbolov. Program ali funkcija, ki opravlja leksikalno analizo, se imenuje leksikalni analizator, lekser ali pregledovalnik. Leksikalni analizator se po navadi uporablja skupaj z razpoznavalnikom [6].

Terminalni simbol

Terminalni simbol je kategoriziran skupek teksta, po navadi sestavljen iz neločljivih nizov znakov (leksemov). Leksikalni analizator najprej prebere vse lekseme, jih kategorizira glede na funkcijo in jim določi pomen. Določanje pomena se imenuje tokenizacija (angl. tokenization) [6].

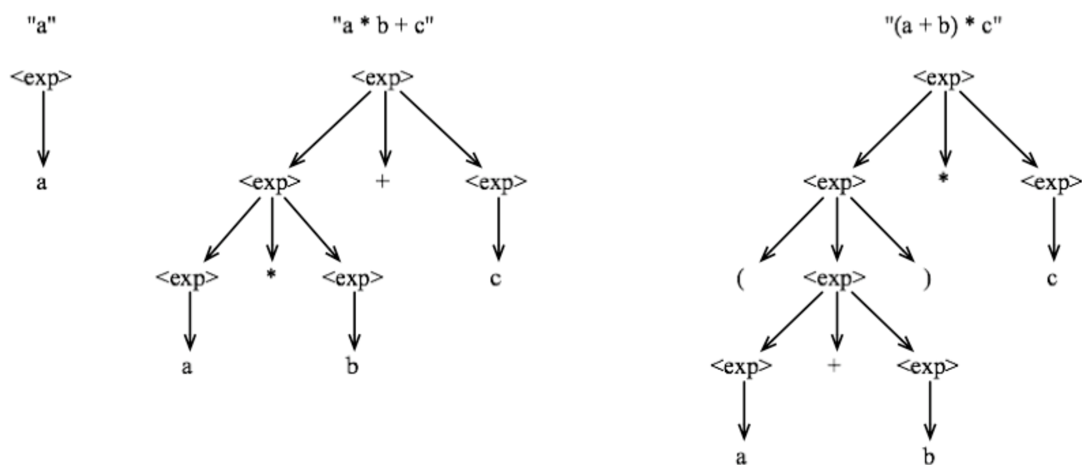
Tabela 5.1: Tokenizacija za primer "sum = 3 + 2;"[6]

Leksem	Terminalni simbol
sum	IDENT
=	ASSIGN_OP
3	NUMBER
+	ADD_OP
2	NUMBER
;	SEMICOLON

Tokenizaciji sledi razpoznavanje. Podatki se pretvorijo v podatkovne strukture za osnovno uporabo, interpretacijo ali prevajanje. Primer "sum = 3 + 2;" je prikazan v tabeli 5.1. Iz te enačbe pregledovalnik prepozna lekseme in jim določi terminalni simbol.

5.2 RAZPOZNAVANJE

Razpoznavanje je problem transformacije linearnega zaporedja znakov v sintaktično drevo. Dandanes obstaja veliko orodij, ki nam pomagajo pri tej nalogi. Vendar pa je bila ta naloga na začetku razvoja računalniške znanosti zelo težaven problem. To je bil eden izmed prvih in najbolj temeljnih izzivov, ki so jih morali pisci prevajalnikov premagati. Če je besedilo programa sintaktično pravilno, potem ga je mogoče pretvoriti v sintaktično drevo. Slika 5.2 prikazuje primer razpoznavanja znakov s sintaktičnim drevesom [19].



Slika 5.2: Primer treh različnih sintaktičnih dreves iz treh različnih programov z isto gramatiko [19]

6 IZDELAVA STATIČNEGA ANALIZATORJA

6.1 OPIS PROBLEMA

Za izdelavo statičnega analizatorja sem najprej potreboval gramatiko določenega jezika. V mojem primeru gre za programski jezik Java. Za razvojno okolje sem izbral Eclipse, za izgradnjo uporabniškega vmesnika pa okolje NetBeans IDE 8.0. Potreboval sem še ANTLR-dodatek za Eclipse, da sem lahko spreminjal gramatiko.

6.2 UPORABLJENA ORODJA

6.2.1 ANTLR

ANTLR je močno orodje, ki generira pregledovalnik in razpoznavalnik za branje, procesiranje, zaganjanje ali prevajanje strukturiranega teksta ali binarnih datotek. Pogosto se ga uporablja za akademske namene ter v industriji za grajenje vseh vrst jezikov in orodij. ANTLR je razvil profesor računalniških znanosti Terence Parr [1].

Za vhod ANTLR vzame gramatiko, ki definira jezik, in za izhod generira izvorno kodo. Trenutna verzija podpira dva programska jezika, to sta Java in C#. Jezik je definiran s pomočjo prostokontekstne gramatike.

ANTLR omogoča generiranje pregledovalnikov, razpoznavalnikov, sintaktičnih dreves in kombinacijo pregledovalnik–razpoznavalnik [3].

6.2.2 DODATEK ANTLR ZA ECLIPSE

Da sem lahko začel z delom, sem potreboval dodatek ANTLR za Eclipse, ki mi je omogočal avtomatsko generiranje pregledovalnika in razpoznavalnika iz dane gramatike. V mojem primeru je bila to gramatika za jezik Java, ki sem ga prilagodil za mojo nalogo. Uporabil sem orodje ANTLR IDE. Za delovanje dodatka sem potreboval tudi dodatek DLTK (angl. Dynamic Languages Toolkit).

Orodje ANTLR IDE nam omogoča avtomatsko generiranje pregledovalnika in razpoznavalnika iz gramatike. V gramatiko lahko na določenih mestih pišemo tudi Java kodo, ki jo ta dodatek potem doda na pravo mesto v pregledovalnik ali razpoznavalnik.

Primer:

```
classDeclaration
    :   normalClassDeclaration

    |   enumDeclaration {stRazredov++;
         podatki.add(new Razred((String)memory.get($enumDeclaration.text), 1));}
    ;
```

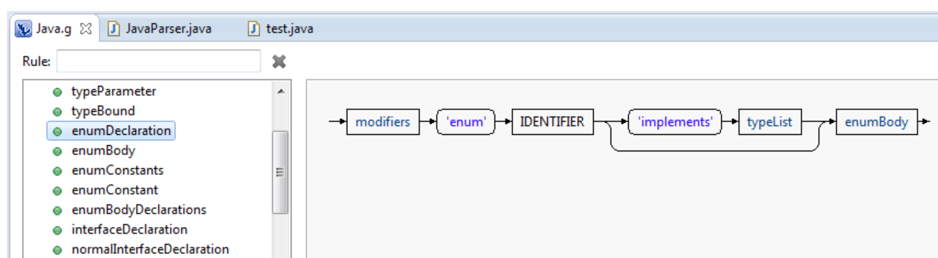
Iz tega izseka kode iz gramatike je razvidno, da pri deklaraciji razreda oziroma, še natančneje, pri deklaraciji "enum" razreda povečam števec "stRazredov" za 1 ter dodam ime in globino razreda (v tem primeru 1) v seznam "podatki". Ime razreda dobimo s pomočjo stavka "memory.get". Spremenljivka "Memory" je tipa HashMap. Da lahko ime spremenljivke sploh pridobimo, jo je treba vstaviti v spomin s stavkom "memory.put".

```
enumDeclaration
    :   {kjeSmo = "Razred"; zacetekRazreda = true;}
        modifiers
        ('enum'
        )
        IDENTIFIER
        ('implements' typeList
        )?
        enumBody
        {
          memory.put($enumDeclaration.text, new String($IDENTIFIER.text));
          {kjeSmo = ""; konecRazreda = true;}
        }
    ;
```

Iz primera lahko vidimo, da ime razreda pridobimo s spremenljivko "IDENTIFIER". V naslednjem izseku kode pa lahko vidimo del funkcije "enumDeclaration" v razpoznavalniku, ki jo avtomatsko generira orodje ANTLR IDE.

```
try {
        if ( state.backtracking>0 && alreadyParsedRule(input, 14) ) {return retval;}
        {if ( state.backtracking==0 ) {kjeSmo = "Razred"; zacetekRazreda = true;}
        pushFollow(FOLLOW_modifiers_in_enumDeclaration1107);
        modifiers();
        state._fsp--;
```

Orodje ima še eno zanimivo funkcionalnost, to je "Railroad View". S tem pogledom lahko analiziramo, kako si določeni elementi v jeziku sledijo (glej sliko 6.1).



Slika 6.1: "Railroad View" za deklaracijo "enum" razreda

6.3 IZDELAVA

Za statično analizo sem implementiral naslednje metrike: indeks vzdrževanja, ciklomatična kompleksnost, globina razreda, Halsteadova kompleksnost, približen čas za implementacijo metode/razreda v minutah, število vrstic.

Ciklomatična kompleksnost

Pri ciklomatični kompleksnosti sem uporabil enačbo (4.3).

```
statement
        forstatement
        {
          int tempStMetod = podatki.get(podatki.size() - 1
vgnezdenRazredNivo).getSeznamMetod().size();
          podatki.get(podatki.size() - 1 -
vgnezdenRazredNivo).getSeznamMetod().get(tempStMetod - 1).dodajLoop();
      …
```

V zgornjem izseku iz gramatike je viden primer, na kakšen način sem štel vse "for" zanke. Podobno sem naredil tudi za vse ostale zanke, "if" in "switch" stavke ter za "return" operatorje. Zbrane podatke sem nato lahko uporabil v enačbi za izračun ciklomatične kompleksnosti.

```
'if' parExpression statement ('else' statement {…})?
{
        …
        int tempStMetod = podatki.get(podatki.size() – 1
        vgnezdenRazredNivo).getSeznamMetod().size();
        podatki.get(podatki.size() - 1 -
        vgnezdenRazredNivo).getSeznamMetod().get(tempStMetod - 1).dodajIf();
        …
}
```

Halsteadova kompleksnost

Pri Halsteadovi kompleksnosti je bilo treba pridobiti vse operatorje in operande za določeno metodo oziroma razred.

```
forstatement
     {
       //halstead
       if (kjeSmo.equals("Razred")) {
         podatki.get(podatki.size() - 1 -vgnezdenRazredNivo).dodajOperator("for");
         podatki.get(podatki.size() - 1 - vgnezdenRazredNivo).dodajOperator("()");
         }
       else if (kjeSmo.equals("Metoda")){
         podatki.get(podatki.size() - 1 -
vgnezdenRazredNivo).getSeznamMetod().get(tempStMetod - 1).dodajOperator("for");
         podatki.get(podatki.size() - 1 -
vgnezdenRazredNivo).dodajOperator("for");
         podatki.get(podatki.size() - 1 -
vgnezdenRazredNivo).getSeznamMetod().get(tempStMetod - 1).dodajOperator("()");
         podatki.get(podatki.size() - 1 - vgnezdenRazredNivo).dodajOperator("()");
       }
     }
```

Zgornji primer prikazuje način, na katerega sem shranil operatorja "for" in "()" iz "for" zanke v seznam razredov. Če se razpoznavalnik trenutno nahaja v razredu, potem shrani operator med podatke o razredu, če se nahaja v metodi, pa ga shrani med podatke o razredu in metodi. Podatke o ostalih operatorjih in tudi operandih sem shranil na podoben način. Za izračun približnega časa za implementacijo metode/razreda sem uporabil enačbo, izpeljano iz Halsteadove kompleksnosti.

Indeks vzdrževanja

Za izračun indeksa vzdrževanja sem uporabil enačbo (4.6). Da lahko izračunamo indeks vzdrževanja, moramo najprej izračunati Halsteadov volumen in ciklomatično kompleksnost. Potrebujemo pa tudi število vrstic kode.

```
public void izracunMI(){
      MI = (int)Math.max(0, (171 - 5.2 * Math.log(V) - 0.23 * CC - 16.2 *
      Math.log(LOC)) * 100 / 171);
}
```

Globina razreda v drevesu

Pri določitvi globine razreda v drevesu sem si pomagal tako, da sem najprej ugotovil, če sploh gre za podrazred. Če to drži, se pregleda seznam shranjenih razredov in primerja z imenom razreda, iz katerega je ta razred podedovan. Ko najdemo pravi razred, je globina tega razreda za 1 višja od njegovega nadrazreda. Če ne gre za podrazred, je globina razreda enaka 1.

```
normalClassDeclaration
    :   {kjeSmo = "Razred"; zacetekRazreda = true;}
        modifiers  'class' IDENTIFIER
        (typeParameters
        )?
        ({kjeSmo = "";} 'extends' type {jePodrazred = true; tempNadrazred =
(String)memory.get($type.text);}
        )?
        ({kjeSmo = "";} 'implements' typeList
        )?
        {
          kjeSmo = "Razred";

          String imeRazreda = $IDENTIFIER.text;
          stRazredov++;

          if (jePodrazred) {
            int tempNivo = 1;
            for (int i = 0; i < podatki.size(); i++) {
              if (podatki.get(i).getIme().equals(tempNadrazred)){
                tempNivo = podatki.get(i).getNivo() + 1;
                podatki.get(i).dodajPodrazred(new Razred(imeRazreda, tempNivo));
              }
            }

            podatki.add(new Razred(imeRazreda, tempNivo));
          }
          else {
            podatki.add(new Razred(imeRazreda, 1));
          }
          jePodrazred = false;
         }
        classBody
        {
          memory.put($normalClassDeclaration.text, new String($IDENTIFIER.text));
          kjeSmo = "";
          konecRazreda = true;
        }
    ;
```

6.3.1 POMOŽNI RAZREDI

Pri izdelavi statičnega analizatorja sem še ustvaril nekaj razredov, ki so mi bili v pomoč pri shranjevanju podatkov o razredih in metodah ter tudi pri iskanju pravih datotek.

Eden izmed takih razredov je razred "Razred".

```
public class Razred {
	private String ime;
	private int nivo;
	private ArrayList<Metoda> seznamMetod;
	private ArrayList<Razred> seznamPodrazredov;
	private int CC; //ciklomatična kompleksnost
	private int LOC; //lines of code

	private int MI;

	//halstead
	private ArrayList<String> vsiOperatorji;
	private ArrayList<String> vsiOperandi;
	private int n1;
	private int N1;
	private int n2;
	private int N2;
	private int n;
	private int N;
	private double V;
	private double D;
	private double E;
	private double T;
	…
```

V tem razredu si shranimo podatke, kot so ime razreda, nivo (globina razreda v drevesu), seznam metod, seznam podrazredov, vse operatorje, vse operande in ostale stvari, ki so potrebne za izračun posameznih metrik.

Naslednji razred je razred "Metoda", kjer so shranjeni podatki o posameznih metodah v določenem razredu. V tem razredu so shranjene podobne stvari kot v prejšnjem razredu. Shranjeni so podatki o imenu metode in tipu ter seznam parametrov. Shranjeno je tudi število "if" stavkov, zank, "case" stavkov in "return" stavkov. Ti podatki so nujni za izračun ciklomatične kompleksnosti. Naslednji izsek iz kode pa prikazuje izračun Halsteadove kompleksnosti.

```
public void izracunHalstead(){
	try{
		HashSet tempHashSet = new HashSet(); //da dobim unikatne operatorje
		tempHashSet.addAll(vsiOperatorji);
		n1 = tempHashSet.size();
		N1 = vsiOperatorji.size();

		tempHashSet.clear();
		tempHashSet.addAll(vsiOperandi);

		n2 = tempHashSet.size();
		N2 = vsiOperandi.size();

		n = n1 + n2;
		N = N1 + N2;
		V = N * (Math.log10(N) / Math.log10(2));
		D = (n1 / 2) * (N2 / n2);
		E = D * V;
		T = E / 18;
	}
	catch(ArithmeticException ex){
		//System.out.println(ex.toString());
	}
}
```

V razredu "Parameter" so shranjeni podatki o parametrih določene metode. Shranjena sta dva podatka, to sta ime in tip parametra.

Naslednji pomemben razred je "DirectoryReader". Naloga tega razreda je poiskati vse datoteke s končnico ".java" v določeni mapi ali pa, če je vhod datoteka, preveriti, ali ima pravo končnico.

```
public class DirectoryReader {
    private static int spc_count = -1;
    private List<File> datoteke = new ArrayList<File>();

    private void Process(File aFile) {
        spc_count++;
        String spcs = "";
        for (int i = 0; i < spc_count; i++) {
            spcs += " ";
        }
        if (aFile.isFile() && Okno1.getFileExtension(aFile).equals("java") &&
                !aFile.getName().contains("BuildConfig") &&
!aFile.getName().contains("R.java")) {
            System.out.println(spcs + "[FILE] " + aFile.getName());
            datoteke.add(aFile);
        } else if (aFile.isDirectory()) {
            System.out.println(spcs + "[DIR] " + aFile.getName());
            File[] listOfFiles = aFile.listFiles();
            if (listOfFiles != null) {
                for (int i = 0; i < listOfFiles.length; i++) {
                    Process(listOfFiles[i]);
                }
            } else {
                System.out.println(spcs + " [ACCESS DENIED]");
            }
        }
        spc_count--;
    }

    public List<File> pozeni(String pot){
        File aFile = new File(pot);
        Process(aFile);
        return datoteke;
    }
}
```

6.4 IZDELAVA UPORABNIŠKEGA VMESNIKA

Za izdelavo uporabniškega vmesnika sem uporabil orodje NetBeans IDE 8.0. To orodje sem izbral zato, ker je preprosto za uporabo in sem svoj projekt zlahka vključil v uporabniški vmesnik.

V metodi za inicializacijo glavnega okna najprej odstranimo vse morebitne vnose in si nastavimo koren drevesnega seznama. Ustvarimo si tudi metodo, ki se sproži na miškin klik.

```
public Okno1() {
    initComponents();
    DefaultTreeModel model = (DefaultTreeModel) jTree1.getModel();
    DefaultMutableTreeNode root = (DefaultMutableTreeNode) model.getRoot();
    root.removeAllChildren();
    root.setUserObject("Vsi razredi");
    model.nodeChanged(root);
    jTree1.addMouseListener(new MouseAdapter() {
        public void mouseClicked(MouseEvent me) {
            doMouseClicked(me);
        }
    });
    model.reload(root);
}
```

Naslednja metoda se uporablja za napolnitev tabele z izračunanimi podatki o izvorni kodi. V prvem delu se preberejo izvorne kode iz vseh izbranih dokumentov.

```
//napolnitev tabele
private void napolniTabelo() {
  List<String> lines = new ArrayList<>();
  source = new ArrayList<>();

  try {
      //lines = readSmallTextFile(FILE_NAME);
      for (int i = 0; i < FILE_NAME.size(); i++) {
          lines = readSmallTextFile(FILE_NAME.get(i));
          src = "";

          for (int j = 0; j < lines.size(); j++) {
              if (!lines.get(j).contains("import") && !lines.get(j).contains("package")) {
                  src += lines.get(j) + "\n";
              }
          }
          source.add(src);
      }
  } catch (IOException e1) {
      e1.printStackTrace();
  }
```

Metoda se nadaljuje z ustvarjanjem spremenljivke "podatki", kjer so kasneje shranjeni vsi podatki o razredih, metodah, parametrih ipd. V tem delu se kličejo metode iz pregledovalnika in razpoznavalnika.

```
podatki = new ArrayList<>();
  for (int i = 0; i < source.size(); i++) {
      JavaLexer lexer1 = new JavaLexer(new ANTLRStringStream(source.get(i)));
      JavaParser parser1 = new JavaParser(new CommonTokenStream(lexer1));
      try {
          parser1.compilationUnit();
      } catch (RecognitionException e) {
           e.printStackTrace();
      }

      podatki = parser1.getPodatki();
  }
```

Preden podatke vstavimo v tabelo, je potrebno še izračunati koliko vrstic bo potrebno ustvariti za tabelo. Če smo izbrali prikaz samo določenega razreda, potrebujemo samo njegovo število metod, drugače seštejemo število vseh metod s številom razredov.

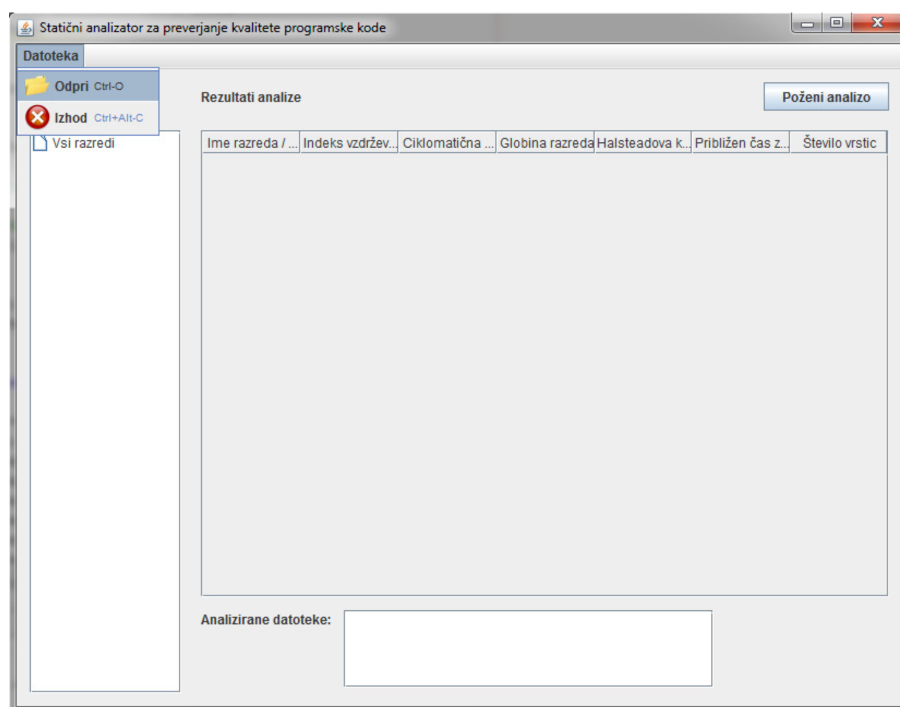
```
int steviloVsehVnosov = 0;
int min = 0;
int max = podatki.size();
if (izbranRazred != -1) {
    min = izbranRazred;
    max = min + 1;
}

for (int i = min; i < max; i++) {
    steviloVsehVnosov += 1 + podatki.get(i).getSeznamMetod().size();
}
```

Na koncu, vse zbrane podatke samo še vstavimo v tabelo.

```
Object o[][] = new Object[steviloVsehVnosov][7];

  int tempVrstica = 0;
  for (int i = min; i < max; i++) {
      o[tempVrstica][0] = podatki.get(i).getIme();
      o[tempVrstica][1] = podatki.get(i).getMI();
      o[tempVrstica][2] = podatki.get(i).getCC();
      o[tempVrstica][3] = podatki.get(i).getNivo();
      o[tempVrstica][4] = String.format("%.2f", podatki.get(i).getE());
      o[tempVrstica][5] = String.format("%.2f", podatki.get(i).getT() / 60);
      o[tempVrstica][6] = podatki.get(i).getLOC();

      tempVrstica++;
      for (int j = 0; j < podatki.get(i).getSeznamMetod().size(); j++) {
          Metoda tempMetoda = podatki.get(i).getSeznamMetod().get(j);
          o[tempVrstica][0] = "         " + tempMetoda.getIme() + " (";

          for (int k = 0; k < tempMetoda.getSeznamParametrov().size(); k++) {
              Parameter tempParameter = tempMetoda.getSeznamParametrov().get(k);
              o[tempVrstica][0] += tempParameter.getTip() + " " + tempParameter.getIme();

              if (k != tempMetoda.getSeznamParametrov().size() - 1) {
                  o[tempVrstica][0] += ", ";
              }
          }

          o[tempVrstica][0] += "): " + tempMetoda.getTip();
          o[tempVrstica][1] = tempMetoda.getMI();
          o[tempVrstica][2] = tempMetoda.getCC();
          o[tempVrstica][4] = String.format("%.2f", tempMetoda.getE());
          o[tempVrstica][5] = String.format("%.2f", tempMetoda.getT() / 60);
          o[tempVrstica][6] = tempMetoda.getLOC();

          tempVrstica++;
      }
  }

jTable1.setModel(new javax.swing.table.DefaultTableModel(
          o,
          new String[]{
              "Ime razreda / metode", "Indeks vzdrževanja", "Ciklomatična kompleksnost", "Globina
razreda", "Halsteadova kompleksnost", "Približen čas za implementacijo (v minutah)", "Število vrstic"
          }
  ));
```

7 UPORABA UPORABNIŠKEGA VMESNIKA

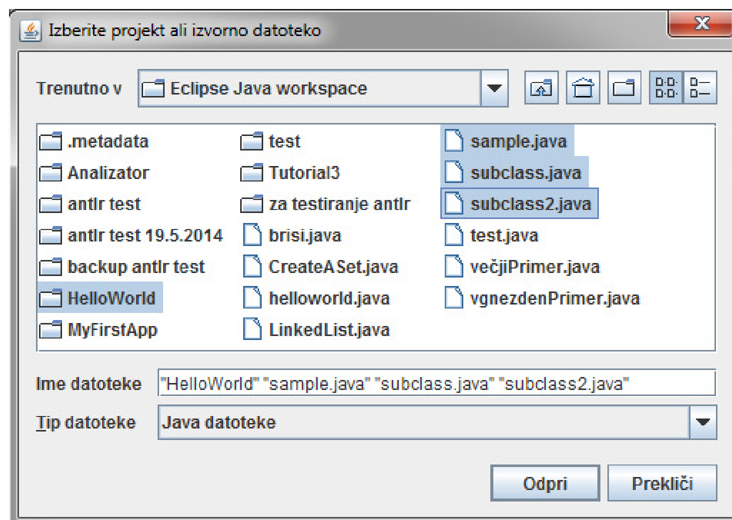
Da lahko začnemo z analiziranjem izvorne kode, zapisane v programskem jeziku Java, moramo najprej izbrati datoteke za analizo. Za izbiro datotek moramo najprej stisniti gumb "Datoteka" in nato gumb "Odpri". Možna je tudi bližnjica s tipkama "Ctrl" + "O". Postopek je viden na sliki 7.1.



Slika 7.1: Postopek izbiranja datotek za analizo

Lahko izberemo samo eno datoteko, lahko pa jih izberemo več. Če nočemo označiti vseh posameznih datotek, pa lahko označimo kar celotno mapo ali kombinacijo map in datotek. Algoritem bo poiskal vse datoteke s končnico "java" v vseh podmapah izbrane mape.

Ko izberemo vse datoteke in mape za analizo, kot je na primer prikazano na sliki 7.2, kliknemo gumb "Odpri".



Slika 7.2: Označevanje datotek in map za analizo

V okencu "Analizirane datoteke" se nam prikažejo poti vseh izbranih in najdenih datotek v podmapah izbranih map s končnico "java". To je vidno na sliki 7.3.

Slika 7.3: Prikaz analiziranih datotek

Da začnemo z analizo, moramo pritisniti na gumb "Poženi analizo". Program nato analizira vse izbrane datoteke in rezultate izpiše v tabeli (glej sliko 7.4).

Rezultati analize

Poženi analizo

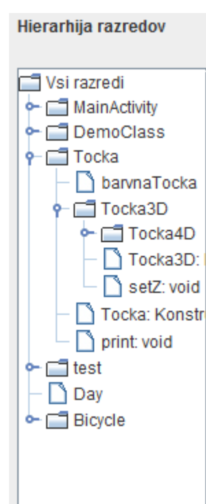
Ime razreda / metode	Indeks ...	Ciklomatič...	Globi...	Halsteado...	Približen čas za i...	Števi...
MainActivity	58	3	1	611,06	0,57	16
onCreate (Bundle savedInstance...	75	2		114,16	0,11	4
onCreateOptionsMenu (Menu m...	73	1		129,06	0,12	5
DemoClass	31	30	1	16951,51	15,70	94
DemoClass (): Konstruktor	76	2		31,02	0,03	5
DemoClass (int x, int c, int b, JI...	69	2		96,21	0,09	7
DemoClass (DemoClass otherD...	75	2		39,30	0,04	5
s1 (): void	82	1		0,00	0,00	3
i1 (): void	82	1		0,00	0,00	3
s2 (): void	75	1		39,30	0,04	5
i2 (): void	74	1		57,06	0,05	5
overloadTester (): void	61	2		614,32	0,57	11
overload (byte b): void	79	2		85,59	0,08	3

Slika 7.4: Prikaz rezultatov analize

Kot je razvidno iz slike 7.4, si stolpci sledijo v naslednjem vrstnem redu: najprej je prikazano ime razreda oziroma metode (metode so zamaknjene v desno), sledijo indeks vzdrževanja, ciklomatična kompleksnost, globina razreda (prikazana samo pri razredih, pri metodah je prazno polje), Halsteadova kompleksnost, približen čas za implementacijo (v minutah) in na koncu še število vrstic.

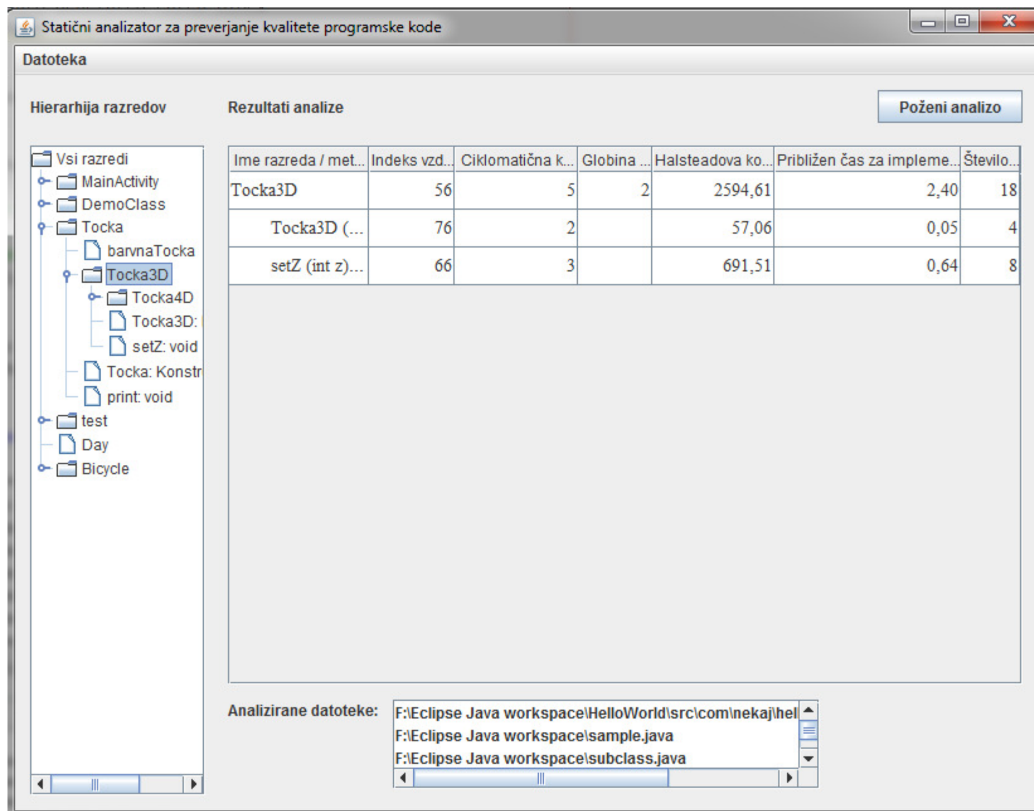
Na levi strani uporabniškega vmesnika imamo predstavljeno hierarhijo razredov. Iz te hierarhije je razvidno, kateri razred je kateremu razredu podrazred ali nadrazred. Med drugim lahko opazimo, katere metode spadajo pod kateri razred, kar je razvidno tudi iz tabele. Pri metodah je napisan še tip (void, int, double, string, konstruktor itd.).

Iz slike 7.5 tako lahko razberemo, da je v našem primeru "Tocka" neposredni nadrazred razreda "Tocka3D" in razreda "barvnaTocka". Razred "Tocka4D" pa je podrazred razreda "Tocka3D".



Slika 7.5: Prikaz hierarhije razredov

S klikom na določen razred ali metodo v hierarhiji razredov lahko filtriramo naše rezultate v tabeli. Če kliknemo na primer na razred "Tocka3D", se nam v tabeli prikažejo rezultati samo za ta razred. Isti rezultat dobimo, če kliknemo na katero koli metodo razreda "Tocka3D". Primer tega je viden na sliki 7.6.



Slika 7.6: Prikaz filtriranja rezultatov v tabeli

8 ZAKLJUČEK

Statična analiza je zelo pomembna pri kvaliteti programske opreme. Z njo ugotovimo morebitne napake in jih pravočasno odpravimo, kar je zelo pomembno, saj si s tem zmanjšamo morebitne stroške. Kvaliteto programske kode pa lahko tudi izmerimo s pomočjo različnih metrik – od preprostih, kot je število vrstic kode, do malo zahtevnejših, kot je Halsteadova kompleksnost. Na podlagi teh metrik se lahko določi kvaliteta kode ali pa čas, ki bi bil potreben za implementacijo takšne kode. Seveda pa se ne moremo zanašati samo na te izračune, saj je lahko določena izvorna koda zahtevnejša za implementacijo kot morda izgleda na prvi pogled. Za ocenitev kvalitete programske kode je dobro upoštevati tudi svojo osebno presojo, ki jo oblikujemo z izkušnjami in znanjem.

9 VIRI IN LITERATURA

[1] *About The ANTLR Parser Generator*. (2014). Pridobljeno 17. 10. 2014 iz ANTLR: http://www.antlr.org/about.html

[2] *Abstractness (A): Ratio of Abstract Classes and Interfaces in a Package*. (2014). Pridobljeno 7. 10. 2014 iz UNAK: http://staff.unak.is/andy/StaticAnalysis0809/metrics/a.html

[3] *ANTLR*. (2014). Pridobljeno 17. 10. 2014 iz Wikipedia, the free encyclopedia: http://en.wikipedia.org/wiki/ANTLR

[4] Boughton, A. (2014). *Software Metrics.* Pridobljeno 24. 9. 2014 iz Department of Computer Science, University of Colorado Boulder: http://www.cs.colorado.edu/~kena/classes/5828/s12/presentation-materials/boughtonalexandra.pdf, stran 3.

[5] Boughton, A. (2014). *Software Metrics.* Pridobljeno 24. 9. 2014 iz Department of Computer Science, University of Colorado Boulder: http://www.cs.colorado.edu/~kena/classes/5828/s12/presentation-materials/boughtonalexandra.pdf, stran 5.

[6] *Compiler Construction/Lexical analysis*. (2014). Pridobljeno 16. 10. 2014 iz WIKIBOOKS, Open books for an open world: http://en.wikibooks.org/wiki/Compiler_Construction/Lexical_analysis

[7] *Control flow graph*. (2014). Pridobljeno 11. 12. 2014 iz Wikipedia, the free encyclopedia: http://en.wikipedia.org/wiki/Control_flow_graph

[8] *Cyclomatic complexity*. (2014). Pridobljeno 24. 9. 2014 iz Wikipedia, the free encyclopedia: http://en.wikipedia.org/wiki/Cyclomatic_complexity

[9] *Cyclomatic Complexity*. (2014). Pridobljeno 24. 9. 2014 iz tutorialspoint: http://www.tutorialspoint.com/software_testing_dictionary/cyclomatic_complexity.htm

[10] *Data-flow analysis*. (2014). Pridobljeno 10. 12. 2014 iz Wikipedia, the free encyclopedia: http://en.wikipedia.org/wiki/Data-flow_analysis

[11] *Depth in Tree (DIT)*. (2014). Pridobljeno 2. 10. 2014 iz UNAK: http://staff.unak.is/andy/StaticAnalysis0809/metrics/dit.html

[12] *DSQI*. (2014). Pridobljeno 7. 10 2014 iz Wikipedia, the free encyclopedia: http://en.wikipedia.org/wiki/DSQI

[13] *Executable Statements (EXEC)*. (2014). Pridobljeno 7. 10. 2014 iz UNAK: http://staff.unak.is/andy/StaticAnalysis0809/metrics/exec.html

[14] Foster, J. (2011). *Dataflow analysis.* Pridobljeno 10. 12. 2014 iz Harvard, School of Engineering and Applied Sciences: http://www.seas.harvard.edu/courses/cs252/2011sp/slides/Lec02-Dataflow.pdf, stran 3.

[15] *Halstead complexity measures*. (2014). Pridobljeno 25. 9. 2014 iz Wikipedia, the free encyclopedia: http://en.wikipedia.org/wiki/Halstead_complexity_measures

[16] *Halstead Metrics*. (2014). Pridobljeno 25. 9. 2014 iz IBM Knowledge Center: http://www-01.ibm.com/support/knowledgecenter/SSSHUF_8.0.0/com.ibm.rational.testrt.studio.doc/topics/csmhalstead.htm

[17] *Introduction to Code Metrics*. (2014). Pridobljeno 30. 9. 2014 iz Radon: http://radon.readthedocs.org/en/latest/intro.html

[18] *Introduction to Programming Languages/Grammars*. (2014). Pridobljeno 15. 10. 2014 iz WIKIBOOKS, Open books for an open world: http://en.wikibooks.org/wiki/Introduction_to_Programming_Languages/Grammars

[19] *Introduction to Programming Languages/Parsing*. (2014). Pridobljeno 16. 10. 2014 iz WIKIBOOKS, Open books for an open world: http://en.wikibooks.org/wiki/Introduction_to_Programming_Languages/Parsing

[20] *Maintainability Index (MI)*. (2014). Pridobljeno 30. 9. 2014 iz ProjectCodeMeter: http://www.projectcodemeter.com/cost_estimation/help/GL_maintainability.htm

[21] *McCabe Cyclomatic Complexity*. (2012). Pridobljeno 10. 11. 2014 iz Klocwork: http://docs.klocwork.com/Insight-10.0/McCabe_Cyclomatic_Complexity

[22] *McCabe Cyclomatic Complexity*. (2012). Pridobljeno 24. 9. 2014 iz Klocwork: http://www.klocwork.com/products/documentation/current/McCabe_Cyclomatic_Compl exity

[23] *McCabe Cyclomatic Complexity*. (2012). Pridobljeno 10. 11. 2014 iz Klocwork: http://docs.klocwork.com/Insight-9.5/McCabe_Cyclomatic_Complexity

[24] McConnell, S. (2004). *Code Complete: A Practical Handbook of Software Construction, Second Edition Paperback.* Redmond, Washington: Microsoft Press, stran 29.

[25] *Metrics overview*. (2014). Pridobljeno 7. 10. 2014 iz UNAK: http://staff.unak.is/andy/StaticAnalysis0809/metrics/overview.html

[26] *Number of Types (NOT)*. (2014). Pridobljeno 7. 10. 2014 iz UNAK: http://staff.unak.is/andy/StaticAnalysis0809/metrics/not.html

[27] Serebrenik, A. (2010). *Software metrics (2).* Pridobljeno 25. 9. 2014 iz Technische Universiteit Eidnhoven: http://www.win.tue.nl/~aserebre/2IS55/2010-2011/10.pdf, strani 7–8.

[28] *Software metric*. (2014). Pridobljeno 24. 9. 2014 iz Wikipedia, the free encyclopedia: http://en.wikipedia.org/wiki/Software_metric

[29] *Software Metrics.* (2005). Pridobljeno 24. 9. 2014 iz MIT OPEN COURSEWARE, Massachusetts Institute of Technology: http://ocw.mit.edu/courses/aeronautics-and-astronautics/16-355j-software-engineering-concepts-fall-2005/lecture-notes/cnotes7.pdf, stran 17.

[30] *Software Metrics.* (2005). Pridobljeno 24. 9. 2014 iz MIT OPEN COURSEWARE, Massachusetts Institute of Technology: http://ocw.mit.edu/courses/aeronautics-and-astronautics/16-355j-software-engineering-concepts-fall-2005/lecture-notes/cnotes7.pdf, stran 18.

[31] *Software Metrics.* (2005). Pridobljeno 24. 9. 2014 iz MIT OPEN COURSEWARE, Massachusetts Institute of Technology: http://ocw.mit.edu/courses/aeronautics-and-astronautics/16-355j-software-engineering-concepts-fall-2005/lecture-notes/cnotes7.pdf, stran 21–23.

[32] *Software Metrics.* (2005). Pridobljeno 24. 9. 2014 iz MIT OPEN COURSEWARE, Massachusetts Institute of Technology: http://ocw.mit.edu/courses/aeronautics-and-astronautics/16-355j-software-engineering-concepts-fall-2005/lecture-notes/cnotes7.pdf, stran 24.

[33] *Source lines of code*. (2014). Pridobljeno 6. 10. 2014 iz ProjectCodeMeter: http://www.projectcodemeter.com/cost_estimation/help/GL_sloc.htm

[34] *Static code analysis*. (2014). Pridobljeno 17. 9. 2014 iz Viva 64: http://www.viva64.com/en/t/0046/

[35] *Static Code Analysis*. (2014). Pridobljeno 11. 12. 2014 iz OWASP: https://www.owasp.org/index.php/Static_Code_Analysis

[36] *Static program analysis*. (2014). Pridobljeno 22. 9. 2014 iz Wikipedia, the free encyclopedia: http://en.wikipedia.org/wiki/Static_program_analysis

Fakulteta za elektrotehniko,
računalništvo in informatiko
Smetanova ulica 17
2000 Maribor, Slovenija

IZJAVA O AVTORSTVU

Spodaj podpisani/-a Matej Zgubič

z vpisno številko E1053803

sem avtor/-ica diplomskega dela z naslovom: Statični analizator za preverjanje kvalitete programske kode

(naslov diplomskega dela)

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal/-a samostojno pod mentorstvom (naziv, ime in priimek)
  doc. dr. Tomaž Kosar
  in somentorstvom (naziv, ime in priimek)
  ____
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela.
- soglašam z javno objavo elektronske oblike diplomskega dela v DKUM.

V Mariboru,
dne 20. 8. 2015

Podpis avtorja/-ice:

____

Fakulteta za elektrotehniko,
računalništvo in informatiko
Smetanova ulica 17
2000 Maribor, Slovenija

IZJAVA O USTREZNOSTI ZAKLJUČNEGA DELA

Podpisani mentor :

doc. dr. Tomaž Kosar

(ime in priimek mentorja)

in somentor (eden ali več, če obstajata):

____

(ime in priimek somentorja)

Izjavljam (-va), da je študent

Ime in priimek: Matej Zgubič

Vpisna številka: E1053803

Na programu: Računalništvo in informacijske tehnologije

izdelal zaključno delo z naslovom:

Statični analizator za preverjanje kvalitete programske kode

(naslov zaključnega dela v slovenskem in angleškem jeziku)

Static analyser for checking the quality of software code

v skladu z odobreno temo zaključnega dela, Navodilih o pripravi zaključnih del in mojimi (najinimi oziroma našimi) navodili.

Preveril (-a, -i) in pregledal (-a, -i) sem (sva, smo) poročilo o plagiatorstvu.

Datum in kraj: 20. 8. 2015, Maribor — Podpis mentorja:

Datum in kraj: — Podpis somentorja (če obstaja):

UNIVERZA V MARIBORU

Fakulteta za elektrotehniko, računalništvo in informatiko

IZJAVA O ISTOVETNOSTI TISKANE IN ELEKTRONSKE VERZIJE ZAKLJUČNEGA DELA IN OBJAVI OSEBNIH PODATKOV DIPLOMANTOV

Ime in priimek diplomanta-tke: Matej Zgubič

Vpisna številka: E1053803

Študijski program: RAČUNALNIŠTVO IN INFORMACIJSKE TEHNOLOGIJE

Naslov diplomskega dela: Statični analizator za preverjanje kvalitete programske kode

Mentor: Tomaž Kosar

Somentor: ______

Podpisani-a Matej Zgubič **izjavljam, da sem za potrebe arhiviranja oddal elektronsko verzijo zaključnega dela v Digitalno knjižnico Univerze v Mariboru. Diplomsko delo sem izdelal-a sam-a ob pomoči mentorja. V skladu s 1. odstavkom 21. člena Zakona o avtorskih in sorodnih pravicah dovoljujem, da se zgoraj navedeno zaključno delo objavi na portalu Digitalne knjižnice Univerze v Mariboru.**

Tiskana verzija diplomskega dela je istovetna elektronski verziji, ki sem jo oddal za objavo v Digitalno knjižnico Univerze v Mariboru.

Zaključno delo zaradi zagotavljanja konkurenčne prednosti, varstva industrijske lastnine ali tajnosti podatkov naročnika: ______
ne sme biti javno dostopno do ______ **(datum odloga javne objave ne sme biti daljši kot 3 leta od zagovora dela).**

Podpisani izjavljam, da dovoljujem objavo osebnih podatkov vezanih na zaključek študija (ime, priimek, leto in kraj rojstva, datum diplomiranja, naslov diplomskega dela) na spletnih straneh in v publikacijah UM.

Datum in kraj: Maribor, 20.08.2015

Podpis diplomanta-tke: ______

Podpis mentorja ______
(samo v primeru, če delo ne sme biti javno dostopno):

Podpis odgovorne osebe naročnika in žig: ______
(samo v primeru, če delo ne sme biti javno dostopno)