UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Miha Anžiček

UREJEVALNIK JAVANSKE KODE

DIPLOMSKO DELO NA
VISOKOŠOLSKEM STROKOVNEM ŠTUDIJU

MENTOR: doc. dr. Boštjan Slivnik

Ljubljana, 2011

Univerza
v Ljubljani

Fakulteta za računalništvo
in informatiko

Tržaška 25
1000 Ljubljana, Slovenija
telefon: 01 476 84 11
faks: 01 426 46 47
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 00534/2010
Datum: 04.10.2010

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **MIHA ANŽIČEK**

Naslov: **UREJEVALNIK JAVANSKE IZVORNE KODE**
JAVA SOURCE CODE FORMATTER

Vrsta naloge: Diplomsko delo visokošolskega strokovnega študija

Tematika naloge:

Sestavite enostaven urejevalnik javanske izvorne kode. Natančno opišite vse njegove vgrajene funkcije, hkrati pa tudi pomankljivosti in s tem povezane možnosti za razširitve. Primerjajte stil, ki ga boste vgradili v svoj urejevalnik, z obstoječimi stili urejanja izvorne kode.

Mentor:
doc. dr. Boštjan Slivnik

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

Dekan:
prof. dr. Nikolaj Zimic

Univerza
v Ljubljani

Fakulteta za računalništvo
in informatiko

Tržaška 25
1000 Ljubljana, Slovenija
telefon: 01 476 84 11
faks: 01 426 46 47
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 00534/2010
Datum: 04.10.2010

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **MIHA ANŽIČEK**

Naslov: **UREJEVALNIK JAVANSKE IZVORNE KODE**
JAVA SOURCE CODE FORMATTER

Vrsta naloge: Diplomsko delo visokošolskega strokovnega študija

Tematika naloge:

Sestavite enostaven urejevalnik javanske izvorne kode. Natančno opišite vse njegove vgrajene funkcije, hkrati pa tudi pomankljivosti in s tem povezane možnosti za razširitve. Primerjajte stil, ki ga boste vgradili v svoj urejevalnik, z obstoječimi stili urejanja izvorne kode.

Mentor:
doc. dr. Boštjan Slivnik

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

Dekan:
prof. dr. Nikolaj Zimic

I Z J A V A O A V T O R S T V U

diplomskega dela

Spodaj podpisani/-a Miha Anžiček,

z vpisno številko 63040006,

sem avtor/-ica diplomskega dela z naslovom:

Urejevalnik javanske kode

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal/-a samostojno pod mentorstvom (naziv, ime in priimek)

  doc. dr. Boštjan Slivnik

  in somentorstvom (naziv, ime in priimek)

- so elektronska oblika diplomskega dela, naslov (slov., angl.) ter povzetek (slov., angl
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki »Dela FRI«.

V Ljubljani, dne _____ Podpis avtorja/-ice: _____

ZAHVALA

Rad bi se zahvalil vsem, ki so me podpirali na moji poti do diplome. Na prvem mestu bi se zahvalil staršem, ki so mi stali ob strani ob premagovanju ovir ter seveda zahvala tudi vsem kolegom in kolegicam na fakulteti, profesorjem ter mentorju.

KAZALO

POVZETEK

V diplomski nalogi je opisan program za urejanje sintaksno pravilne javanske kode. Program se poganja iz ukazne vrstice z dvema parametroma, ki določata datoteko, z neurejeno kodo ter zaželeno maksimalno dolžino vrstic, ki pa ni obvezna. Deluje izključno za javansko kodo ter po samem zagonu ne potrebuje več uporabnikove interakcije. Pokriva vse osnovne funkcije urejevalnika, kot so pravilno postavljanje v novo vrstico, zamiki, presledki, itd. ter tudi lomljenje predolgih vrstic glede na podano maksimalno dolžino (če ni podana, se vzame privzeta dolžina 40 znakov). Izpisana koda je sintaksno pravilna in pripravljena za nadaljnjo uporabo.

KLJUČNE BESEDE

Java

Urejevalnik kode

Lomljenje vrstic

Ukazna vrstica

Predolge vrstice

ABSTRACT

The goal of this thesis is proper ordering of Java syntax code. The program is being run from command prompt with two parametres. First one includes Java source code in which Java syntax is in dissordered form and the second one includes maximum line length . The second parameter is not necessary. It works only for Java syntax and after being started it does not need user interaction. It covers all basic code parser functions like getting into a new line, proper spacing, etc. and it also includes line breaking of overlength lines according to the given maximum line length. If maximum line length is not given, default value is used (40 characters). Printed code is syntactically correct and is prepared for further use.

KEY WORDS

Java

Code parser

Line breaking

Command prompt

Overlength lines

1. UVOD

V današnjem času je uporaba urejevalnikov kode postala že nekaj samoumevnega. Vsako moderno razvojno okolje in urejevalnik teksta ima že vgrajeno funkcijo, ki avtomatsko uredi izvorno kodo, da ta postane razumljiva ter da so posamezni segmenti izvorne kode jasno in nedvoumno vidni. S tem programerju npr. omogoči hiter pregled ravnokar naložene kode, ki se je po možnosti pri prenosu s spleta spremenila v nerazpoznavno obliko. To mu prihrani veliko časa, če je koda zelo obsežna (kar pogosto tudi je), ki ga lahko porabi za nujnejša opravila. Ker pa se zapisi programov različnih programerjev med sabo razlikujejo, ima večina razvojnih okolij tudi možnost prilagajanja nastavitev za urejanje izvorne kode. Ta sega od dolžine odmikov med posameznimi stavki, spuščanja ustreznega števila praznih vrstic vse do nastavljanja želene dolžine in stila lomljenja predolgih vrstic. Ravno zato so nekateri odprtokodni projekti predlagali standardizirane stile urejanja kode. Tu bi posebej izpostavil GNU stil in BSD stil. Vendar pa se kljub uvedbi standardov olepševanje kode verjetno ne bo nikoli popolnoma standardiziralo. Vedno bodo pomembnejše posamezne želje in navade posameznikov, da določen kos izvorne kode uredijo po svoje. Zaradi tega pride do pogostega problema, da se zaradi stila enega programerja na prvi pogled zdi koda nerazumljiva drugemu programerju. Prav tako se stili urejanja kode med sabo zelo razlikujejo glede na posamezni programski jezik. Posebej bi izpostavil razliko med postopkovnimi in nepostopkovnimi programskimi jeziki.

V nekaj naslednjih razdelkih je prikazano, kako so se razvijali stili in predvsem v kakšnih podrobnostih se med seboj razlikujejo. Razlike niso velike in se nanašajo predvsem na postavljanje zamikov in zavitih oklepajev. V tem trenutku je najbolj razširjen Bannerjev stil, ki prevladuje v večini internetne dokumentacije in tematskih strokovnih knjigah, vendar pa se tudi tu najdejo odstopanja. En tak primer je, da je zaviti zaklepaj brez zamika in se nahaja v istem stolpcu kot deklaracija. Vendar pa je pri tem potrebno opozoriti, da noben izmed teh stilov ni obvezujoč in se jih lahko uporablja po lastnih željah in presoji. Glavni cilj teh stilov je predvsem to, da se nekako uskladi globalna oblika kodiranja, ker gre pri programiranju za proces in postopke, ki morajo saj v osnovi biti dostopni in razumljivi vsakomur, ne glede na nacionalnost in ne kot oblika omejevanja pri izražanju svoje programske logike.

1.1 WHITESMITH-OV STIL

Za Whitesmithov stil je značilno, da so začetni zaviti oklepaji v novi vrstici in da so zamiki poravnani z zamiki stavkov[1]:

```
public void compare(int x, int y)
      {
      while(x == y)
            {
            System.out.println("equal");
            x++;
            }
      System.out.println("finished");
      }
```

Slika 1 – Whitesmithov stil

1.2 HORSTMANN-OV STIL

Horstmannov stil se od Whitesmihtovega razlikuje v tem, da so zaviti oklepaji poravnani z začetkom deklaracij funkcij in začetkom vejitvenih struktur ter da je prvi stavek v isti vrstici kot začetni zaviti oklepaj[2] (prikazano na naslednji strani):

```
while(true)
{       int i = 0;
        i++;
        if(j > i)
        {       j--;
                System.out.print("Incremented!");
        }
}
```

Slika 2 – Horstmannov stil

1.3 BANNER-JEV STIL

Bannerjev stil postavi začetni zaklepaj v isto vrstico, v kateri se nahaja deklaracija funkcije ter zaklepaj ima isti zamik kot vsi pripadajoči stavki[3]:

```
public int add (int a, int b)  {
      int c = a + b;
      if (c > 10)  {
          return c;
      }
      else  {
          return 0;
      }
}
```

Slika 3 – Bannerjev stil

1.4 JAVANSKA KODERSKA KONVENCIJA

S strani podjetja Sun Microsystems, ki je avtor Jave, je bila izdana tudi posebna dokumentacija, ki predpisuje uradna pravila urejanja javanske kode s strani razvijatelja samega. Razlogi za izdajo teh pravil so sledeči:

- redko kateri program je razvit in vzdrževan s strani samo enega avtorja;
- koderske konvencije izboljšajo berljivost programov in omogočijo programerjem, da razumejo novo kodo hitreje;
- če se izvorna koda izdaja kot produkt, mora biti dobro urejena, pregledna in ustrezno zapakirana.

V njej so opredeljena pravila za vse segmente kodiranja od deklaracij spremenljivk, imenovanja spremenljivk in datotek do postavljanja komentarjev in predlog za pravilno in optimalno kodiranje določenih segmentov kode[4].

1.4.1 KONČNICE

Najprej bi omenil predpise za končnice javanskih datotek z izvorno kodo. Končnica za datoteko z javansko izvorno kodo je .**java**, medtem ko je končnica za javansko byte kodo **.class**.

1.4.2 ZAČETEK RAZREDA

Vsak javanski razred vsebuje en javni razred ali vmesnik. Vsak začetek razreda ali vmesnika mora po konvenciji izgledati tako in v točno takšnem zaporedju:

- začetni komentarji (slika 4);

```
/*
 * Classname
 *
 * Version information
 *
 * Date
 *
 * Copyright notice
 */
```

Slika 4 – Primer pravilnega stila začetnih komentarjev

- stavki, ki označujejo uvoz in pakiranje (slika 5);

```
package diplomska;

import java.io.*;
```

Slika 5 – Pravilni stil stavkov za uvoz in pakiranje

- deklaracija razreda ali vmesnika (slika 6).

```
public class CodePrinter
```

Slika 6 – Pravilni stil deklaracije razreda ali vmesnika

1.4.3 LOMLJENJE VRSTIC

Predpisano je, da dolžina vrstic ne sme biti daljša od 80 znakov, veljajo pa različna pravila za lomljenje različnih izrazov:

- lomljenje klicev metod (slika 7);

```
getBiggestPlanet(innerPlanet.getMercury(), innerPlanet.getEarth()
       outerPlanet.getJupiter(), outerPlanet.getSaturn();
```

Slika 7 – Pravilni stil lomljenje klicev metod

- lomljenje aritmetičnih izrazov (slika 8);

```
sum = getMaxSize() + length + depth - getFirstHeight()
      * secondHeight() + arrayNumber;
```

Slika 8 – Pravilni stil lomljenja aritmetičnega izraza

- lomljenje pogojnih stavkov (slika 9).

```
if ((ch=='<')||(ch=='>')
      ||(ch=='+')||(ch=='-')
      ||(ch=='%')||(ch=='{')) {
    System.out.println(ch);
}
```

Slika 9 – Pravilni stil lomljenja pogojnega stavka

1.4.4 KOMENTARJI

Javanski programi imajo lahko 2 vrsti komentarjev: implementacijski (/*…*/ in //) komentarji in dokumentacijski komentarji(/**…*/). Implementacijski komentarji so namenjeni komentiranju kode in komentarjev o specifični implementaciji, medtem ko dokumentacijski komentarji opisujejo specifikacije kode. Namen komentarjev je, da podrobneje razložijo kodo in dajo dodatne informacije, ki niso neposredno vidne iz same izvorne kode. Komentarji se ločijo na štiri stile:

- blokovni komentarji (slika 10);

```
/*
 * Here is a block comment
 */
```

Slika 10 – Pravilni stil blokovnega komentarja

- enovrstični komentarji (slika 11);

```
if (true) {
    /*Handle the condition*/
    start();
}
```

Slika 11 – Pravilni stil enovrstičnega komentarja

- zaključni komentarji (slika 12);

```
if (a == 2) {
     return true;   /*special case*/
}
else {
    return isPrime(a);   /*works only for odd a*/
}
```

Slika 12 – Pravilni stil zaključnega komentarja

- zaključno-vrstični komentarji (slika 13).

```
if (foo > 0) {
    foo = getSize(foo)  //explain why here
}
```

Slika 13 – Pravilni stil zaključno-vrstičnega komentarja

1.4.5 DEKLARACIJE

Priporočena je ena deklaracija na vrstico(slika 14), pri tem pa se moramo izogibati deklaracijam različnih tipov v eni vrstici(slika 15).

```
int a;  //pravilno
int b;
```

Slika 14 – Pravilni stil deklaracij spremenljivk v eni vrstici

```
int c; int d; //nepravilno
```

Slika 15 – Nepravilni stil deklaracij spremenljivk

Pomembno je tudi, da so lokalne spremenljivke inicializirane tam, kjer so deklarirane! Deklaracije je potrebno postavljati na začetek blokov (katerekoli kos kode, ločen z { in }, slika 16).

```
public void myMethod() {
    int x = 0 + getSize();
    if (x > 0) {
        int y = start();
    }
}
```

Slika 16 – Pravilni stil inicializacije deklaracij spremenljivk

Pri deklaracijah razredov in vmesnikov je potrebno upoštevati nekatera pravila (slika 17)

- ne sme biti presledka med imenom metode in (, ki predstavlja začetek parametrov;
- oklepaj { se pojavi na koncu iste vrstice, v kateri je tudi deklaracija;
- zaklepaj } se nahaja v novi vrstici.

```
Class MyClass extends Object {
      int ivar1;
      int ivar2;

      MyClass(int i, int j) {
            ivar1 = i;
            ivar2 = j;
      }
    }
```

Slika 17 – Pravilni stil deklaracije razreda in pripadajočih metod

1.4.6 STAVKI

Za zapis in oblikovanje stavkov obstajajo naslednja pravila:

- vsaka vrstica more vsebovati natančno en stavek(slika 18);

```
argv++;
argv--;
```

Slika 18 – Pravilno razporejanje stavkov

- Povratni stavek z vrednostjo mora vračati vrednost, ki je kolikor je možno očitna in jasno razvidna(slika 19);

```
return;
return myDisk.size();
return (size ? size : defaultSize);
```

Slika 19 – Pravilen stil povratnega stavka

- pravilen stil pogojnih in pogojno-pogojnih stavkov je prikazan na slikah 20 in 21;

```
if (
    i > 3( {
    i--;
} else if (
    i < 0) {
    i++;
} else {
    i = 0;
}
```

Slika 20 – Pravilen stil pogojnega stavka

```
if (
    i > 3( {
    i--;
} else {
    i++;
}
```

Slika 21 – Pravilen stil pogojno-pogojnega stavka

- For stavek mora imeti obliko, kot jo prikazuje slika 22(na naslednji strani);

```
for (
    int i = 0;
    i < 5;
    i++) {
    i = get(len());
}
```

Slika 22 – Pravilen stil for stavka

- predpisano obliko while stavka prikazuje slika 23

```
while (
    i > 0 {
    System.out.println(i);
    i++;
}
```

Slika 23 – Pravilen stil while stavka

- do stavek mora imeti obliko prikazano na sliki 24 (na naslednji strani);

```
do {
    i > 0 {
    System.out.println(i);
    i++;
} while (
    i > 0);
```

Slika 24 – Pravilen stil do stavka

- Switch stavek (slika 25);

```
switch (
    x) {
    case 1:
    System.out.println("1");
        break;
    case 2:
    System.out.println("2");
        break;
    default:
    System.out.println("Error");
        break;
}
```

Slika 25 – Pravilen stil switch stavka

- pravilno obliko try-catch stavka prikazuje slika 26, le-ta pa lahko vsebuje tudi ukaz finally (slika 27).

```
try {
    File dat = new File("test.txt");
} catch (Exception e) {
    e.printStackTrace();
}
```

Slika 26 – Pravilen stil try-catch stavka

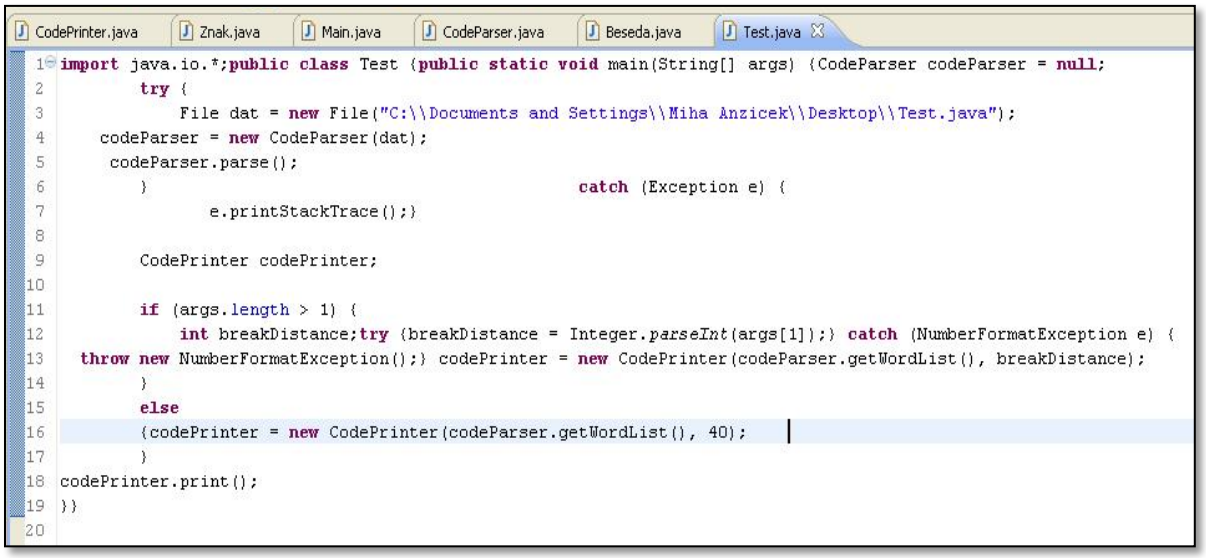
```
try {
    File dat = new File("test.txt");
} catch (Exception e) {
    e.printStackTrace();
} finally {
    System.out.print("Finished");
}
```

Slika 27 – Pravilen stil try-catch stavka z ukazom finally

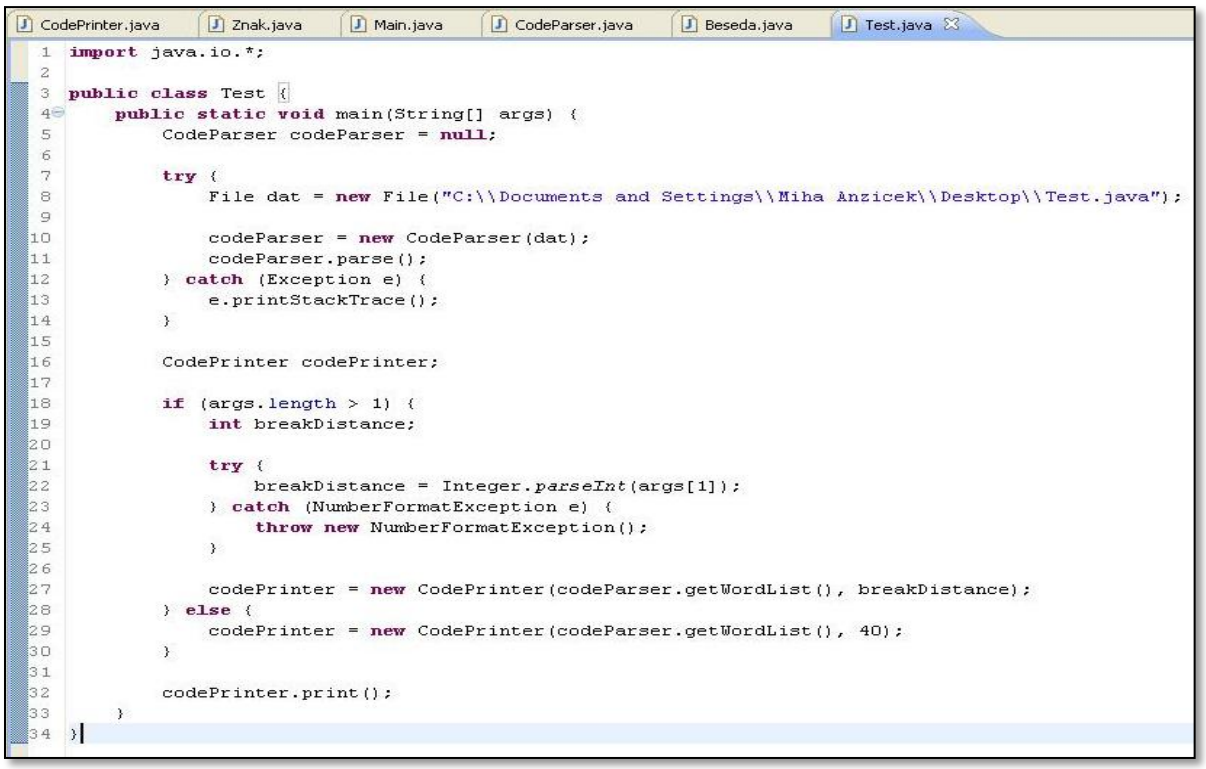
1.5 JINDENT

Prikazal bi primer uporabe urejevalnika javanske kode v modernem razvojnem okolju Eclipse. Uporabil sem razširitev za Eclipse, imenovano Jindent[5], ki je profesionalno orodje za urejanje, olepševanje in izpisovanje javanske kode. To orodje uporablja najzmogljivejše algoritme urejanja in deluje po najnovejših standardih javanske koderske konvencije.

Koda pred ureditvijo je prikazana na sliki 28, koda po ureditvi pa na sliki 29:



Slika 28 - Koda pred ureditvijo v okolju Eclipse



Slika 29 – Koda po ureditvi z orodjem Jindent

Na sliki 30 se lepo vidi stil, ki popolnoma ustreza najnovejšim standardom. Stil, ki je uporabljen zgoraj, je privzet, da pa se ga popolnoma prilagoditi. Lepo prikazuje zamike in prehajanja v novo vrstico, posebej pa bi izpostavil zavite oklepaje, ki se nahajajo v isti vrstici kot deklaracija strukture, kar nekako ustreza že prej omenjenemu BSD stilu. To pravilo se uporablja v veliki večini orodij in pri mnogih posameznikih, vendar pa jaz osebno uporabljam način, da je zaviti oklepaj v novi vrstici. To nekako ustreza tudi že prej omenjenemu GNU stilu. Iz tega izhaja tudi dejstvo, da moja diplomska uporablja način, ki postavlja zaviti oklepaj v novo vrstico.

2. ZASNOVA FUNKCIJ UREJEVALNIKA

Urejevalnik pretvarja sintaksno pravilno javansko kodo iz neurejene oblike v urejeno, pri čemer se obdrži sintaksna pravilnost. Vsebuje samo osnovne funkcije, ki predstavljajo nekakšen minimalen standard vsakega urejevalnika. S tem se ohranja preprostost ter učinkovitost urejanja. Namenjen je programerjem in vsem tistim, ki rabijo hiter in preprost pregled strukture naključne kode v nerazumljivi obliki. Urejevalnik ne vsebuje grafičnega vmesnika, ampak se poganja iz ukazne vrstice. Kot parameter se mu poda datoteka z neurejeno kodo ter drugi želen parameter, ki določa maksimalno dovoljeno dolžino vrstice. Če ta parameter ni podan, se privzame dolžina 40 znakov. Sama urejena koda se izpiše v sami konzoli, da je uporaba orodja kar najbolj učinkovita.

2.1 VSEBINA FUNKCIJ

Osnovne funkcije obsegajo:

- **Pravilno prehajanje v nove vrstice (slika 30)**. Med najpomembnejše in najbolj osnovne funkcije spada postavljanje posameznih stavkov in struktur v novo vrstico. To se v osnovi izvaja nad podpičji, ki v programskem jeziku Java pomenijo konec stavka oz. strukture. S tem damo olepšani kodi osnovni izgled, ki da nekakšen osnovni pregled nad izvorno kodo. Posamezne stavke je možno v večini primerov že razbrati, vendar pa se še vedno ne vidi pravi logični pomen teh stavkov ter sama njihova pripadnost metodam in algoritmom.

```
int a;  //pravilno
int b;

int c; int d;  //nepravilno
```

Slika 30 – Pravilno prehajanje v nove vrstice

- **Pravilno postavljanje zamikov pripadnosti funkcijam in algoritmom (slika 31).** Naslednja zelo pomembna funkcija je delanje zamikov že postavljenih vrstic in stavkov glede na njihovo pripadnost posameznih metodam oziroma njihovo gnezdenje v posameznih strukturah. Tu se že pokaže logika posameznih stavkov, njihova ugnezdena pripadnost ter njihovo natančno zaporedje. S tem dobi programer dokaj natančen pregled nad samo izvorno kodo, njeno osnovno urejenostjo ter na prvi pogled lahko da grobo oceno, kaj je neko osnovno logično bistvo te izvorne kode. To mu omogoči, da lahko na prvi pogled izlušči, kaj določena izvorna koda počne ter ugotovi, če je npr. to sploh tisto, kar on potrebuje.

```
    try  //nepravilno
    {
    breakDistance =  integer.parseInt(args[1]);
    }
    catch (NumberFormatException e)
    {
throw new NumberFormatException();
    }

    try  //pravilno
    {
        breakDistance =  integer.parseInt(args[1]);
    }
    catch (NumberFormatException e)
    {
        throw new NumberFormatException();
    }
```

Slika 31 – Pravilno postavljanje zamikov

- **Pravilno postavljanje presledkov pri posebnih znakih (primerjave, izrazi,...).** Najprej omenimo znake, ki so namenjeni primerjanju dveh vrednosti oziroma predstavljajo logične operatorje. Funkcija je zmožna ločevati posebne znake, ki predstavljajo en znak (>, <, `&`, ...) ali so sestavljeni iz večih znakov (<=, ==, `&&`, ...). V ustreznih pogojnih in vejitvenih stavkih se presledki med, po in pred njimi postavijo tako, da je vloga in pomen posebnega znaka jasno vidna in nedvoumna. To omogoči pregled in prikaže natančen pomen pogojev v posameznih vejitvenih stavkih in programerju jasno in nazorno pokaže zaporedje izvajanja stavkov ob določenih pogojih. Tu bi tudi omenil primer posebnega znaka `!`, ki v logičnem smislu pomeni negacijo pogoja. Funkcija pred omenjeni znak postavi presledek, vendar pa ga za njim ni. To spada pod standard izpisovanja tega znaka in ne predstavlja napake. Izhaja pa iz logike, da se znak nanaša samo na obstoječ klic metode oziroma logičnega pogoja, ker ga logično zanika in ne predstavlja primerjave dveh vrednosti oz. logične izbire med dvema pogojema. V tej funkcionalnosti je zajeto tudi postavljanje presledkov med aritmetičnimi izrazi. Tudi te znake je funkcija možna razločevati na enoznačne (+, −, ...) in tiste, ki so sestavljeni iz več znakov (++, −−, ...). S tem je omogočeno jasno razbiranje aritmetičnih izrazov. Tu bi rad izpostavil znaka + in `*`, ker imata lahko oba več pomenov. Skupno jima je, da se uporabljata kot osnovna aritmetična izraza. V tem primeru se presledek postavi pred in po posebnem znaku, `*` pa se lahko uporablja tudi v primeru tako imenovanih `import` stavkov. Sami stavki imajo pomen uvažanja in uporabe zunanjih paketov razredov in `*` na koncu pove, da naj uvozi vso vsebino izbranega paketa. V tem primeru presledka pred njo ni. Lahko pa se uporablja tudi pri označevanju več vrstičnih komentarjev, vendar o tem nekoliko kasneje. + pa se lahko uporablja tudi za lepljenje nizov znotraj stavkov za izpisovanje.

```
a = 3 + 2;        //pravilno
if((i < 0) && !isBollean)
    return false;

a=3+2;            //nepravilno
if((i<0)&& ! isBollean)
    return false;
```

Slika 32 – Pravilno postavljanje presledkov pri posebnih znakih

- **Lomljenje predolgih vrstic glede na podano največjo dovoljeno dolžino (slika 32).** Maksimalna dolžina vrstice za lomljenje se nastavi že kot argument pri zagonu programa, vendar pa ni obvezna. Če je ni, se uporabi privzeta vrednost 40 znakov. Samo lomljenje poteka po pogojih, ki se nahajajo v določeni vrstici. Tu mislim predvsem logične operatorje, po potrebi pa se vrstice lomijo tudi po pikah. Logičnim operatorjem se določijo prioritete, in sicer operator z najnižjo prioriteto se uporabi najprej. Tako dosežemo, da sam izpis lomljenje vrstice ohranja uporabniku razumljiv izgled. Pri lomljenju sem moral upoštevati nekatere posebne primere kot je npr. predolgo ime spremenljivke.

```
private boolean isArithmeticChar(String znak)  //pravilno
{
    if (znak.equals("*") || znak.equals("/")
        || znak.equals("-") || znak.equals("+")
            || znak.equals("||") || znak.equals("&&"))
    {
        return true;
    }

    return false;
}

private boolean isArithmeticChar(String znak)  //nepravilno
{
    if (znak.equals("*") || znak.equals("/") || znak.equals("-") ||
znak.equals("+")
                || znak.equals("||")
    || znak.equals("&&"))
    {
        return true;
    }

    return false;
}
```

Slika 33 – Pravilno lomljenje predolge vrstice

Urejevalnik pa ne vsebuje nekaterih drugih pogostih funkcij, ki jih kot standard jemljejo profesionalna orodja. Glavni razlog za to je implementacija, ki je preveč kompleksna. Glavne izmed teh funkcij so sledeče:

- raznovrstnih predlog za optimizacijo in »pametnejšo« rabo posameznih segmentov kode;
- ne označuje odvečnih spremenljivk in importov;
- ne barva kode;
- ne ustreza popolnoma standardiziranim stilom urejanja kode. Najbolj očitno odstopanje se pojavlja pri pisanju zavitih oklepajev (slika 33).

```
if(true) { //po standardu
    //do something
}

if(true)  //moj stil
{
    //do something
}
```

Slika 34 – Primerjava pisanja zavitih oklepajev

Omenil bi tudi komentarje, ki se tudi ne izpisujejo po konvencijah, ampak v standardnem formatu (tam kjer se pojavijo tudi so) in so brez posebne obdelave. Ravno tako pa tudi ne obstaja raba praznih vrstic med posameznimi metodami ipd..

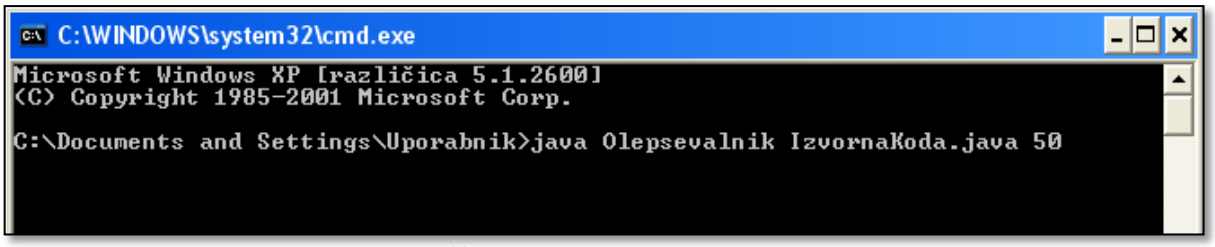
3. REALIZACIJA UREJEVALNIKA

3.1 UVOD

Urejevalnik prebere podano datoteko, ki vsebuje neurejeno javansko izvorno kodo. Nato to datoteko razbije na posamezne besede ter jim takoj določi potrebno dolžino zamika (pripadnost funkcijam, različnin vejitvenim stavkom, ...) in zaporedno vrstico, v kateri se bo posamezna beseda izpisala. Kot takšna se beseda shrani v seznam besed. Nato se posamezne besede po vrsti berejo iz seznama in se ustrezno izpisujejo na mestih, ki smo jih določili prej. Po potrebi se posamezni stavki tudi lomijo.

3.2 ZAGON IN PRIPRAVA

V osnovi je urejevalnik realiziran z zagonom iz ukazne vrstice. Ob zagonu lahko sprejme 2 parametra, od katerih je prvi obvezen. Prvi predstavlja datoteko, ki vsebuje neurejeno izvorno kodo, drugi pa je izbirni in predstavlja maksimalno dovoljeno dolžino vrstic. Če le-tega argumenta ni, se uporabi privzeta vrednost, in sicer 40 znakov na vrstico. Primer zagona je prikazan na sliki 34.



Slika 35 – Primer zagona programa

Ko se program požene, se drugi parameter takoj pretvori iz argumenta v spremenljivko razreda `File`, ki se potem poda razredu `CodeParser`, le-ta pa je zadolžen za razbijanje podane vsebine datoteke na posamezne besede.

```
File dat = new File(args[0]);
codeParser = new CodeParser(dat);
codeParser.parse();
```

Pri tem se pokliče konstruktor, v katerem se nastavi podana datoteka.

```
private File toParse;

public CodeParser(File toParse)
{
    this.toParse = toParse;
}
```

3.3 RAZBIRANJE BESED

Sledi klic metode `parse()`, ki dejansko začne razčlenjevati vsebino datoteke. V grobem je logika takšna, da se iz datoteke bere vrstica za vrstico ter se pri tem ta vrstica razčlenjuje na posamezne besede. Te besede se shranjujejo v razred `Beseda` in se shranjujejo v `ArrayList`.

```
private ArrayList<Beseda> wordList = new ArrayList<Beseda>();
```

Razred `Beseda` je objekt, ki določa osnovne značilnosti posamezne besede. Vsaka beseda je opredeljena s samim nizom, ki predstavlja vrednost besede, razdaljo, ki predstavlja njen odmik od začetka vrstica glede na pripadnost metodi/algoritmu/gnezdnemu stavku ter vrstico, v katero se mora ta beseda izpisati.

```
private String beseda;
private int distance;
private int line;

public Beseda()
{
    distance = 0;
    line = 0;
}

public Beseda(String beseda)
{
    this.beseda = beseda;
    distance = 0;
    line = 0;
}

public Beseda(String beseda, int value, int line)
{
    this.beseda = beseda;
    this.distance = value;
    this.line = line;
}
```

Vsebuje tudi metode za pridobivanje ter naknadno nastavljanje vrednosti predstavljenih zgoraj.

```
public String getBeseda()
{
    return beseda;
}

public void setBeseda(String newBeseda)
{
    beseda = newBeseda;
}

public void setValue(int newValue)
{
    distance = newValue;
}
```

```
public void setLine(int newLine)
{
    line = newLine;
}

public int getDistance()
{
    return distance;
}

public int getLine()
{
    return line;
}
```

Ko se v metodi `parse()` prebere posamezna vrstica, se v njej vsi tabulatorji zamenjajo s presledki. S tem preprečimo nadaljne težave pri razčlenjevanju datoteke in morebitne napake pri izpisovanju vsebine.

```
while((str = read.readLine()) != null)
{
    pos = 0;
    str = str.replace('	', ' ');
```

Nato se začne razčlenjevanje posamezne prebrane vrstice. V tej vrstici se bere znak za znakom ter se sproti preverja, za kakšen znak gre. Za to uporabljam posebno metodo, ki testira posamezne znake, če ti pripadajo rezerviranim besedam v javi. To pomeni, da ti znaki s sabo nosijo nekatere specifične funkcionalnosti in se jih ne sme obravnavati kot posamezne naključne znake. Metoda, ki to preverja, je sledeča:

```
private boolean isSpecialChar(char ch)
{
    if((ch=='<')||(ch=='>')||(ch=='+')||(ch=='-')||(ch=='%')||(ch=='(')||(ch==')')
            ||(ch=='!')||(ch=='*')||(ch=='/')||(ch==';')||(ch==':')||(ch=='=')
            ||(ch==',')||(ch=='(')||(ch==')')||(ch=='[')||(ch==']')||(ch=='?')
            ||(ch=='|') || (ch=='&'))
        return true;

    return false;
}
```

To so predvsem znaki, ki predstavljajo začetek ali konec izrazov, konec stavka ter logične operatorje. Posebej pa jih moramo obravnavati zaradi njihovega kasnejšega pravilnega izpisovanja ali ustreznega prehoda v novo vrstico. Preden pa se dejansko preverjajo posamezni znaki, se še preveri, ali trenutna vrstica vsebuje rezervirano besedo “for”. Ta beseda predstavlja poseben strukturiran stavek, kateremu je specifična uporaba podpičij v njegovi glavi. Tu pa nastane težava, ker se mora na teh dvopičjih onemogočiti prehod v novo vrstico.

```
if(string.equals("for") && !lineComment && (str.charAt(pos) == ' ' || str.charAt(pos) == '('))
{
    containsFor = true;
    wordList.add(new Beseda(string, distance, line));
    string = "";

    if(str.charAt(pos) == '(')
    {
        wordList.add(new Beseda("(", distance, line));
        oklepaji++;
    }
}
```

Tu (slika zgoraj) se mora prav tako preverjati, ali se trenutno nahajamo znotraj komentarja (`!lineComment`), ker izpisovanje besed znotraj komentarjev ne sme spreminjati vsebine. Šele na to dejansko sledi preverjanje posameznih znakov:

```
else if(isSpecialChar(str.charAt(pos)))
{
    if(!string.equals(""))
    {
        wordList.add(new Beseda(string, distance, line));
    }
```

Pred samim dodajanjem posebnih znakov se preverja, če se v spremenljivki `string` nahaja beseda. Če se je pojavil posebni znak, se ta obravnava kot nova beseda in trenutna vrednost spremenljivke se shrani v seznam prebranih besed. Sama spremenljivka `string` predstavlja trenutno besedo, kateri se sproti dodajajo znaki in se shrani v zgoraj omenjenem primeru. Šele na to sledi klic posebne metode, ki je namenjena shranjevanju posebnih znakov. Pred samim klicom te metode se preverja, če se slučajno posebni znaki, ki predstavljajo prehajanje v novo vrstico, pojavljajo znotraj niza ali so označeni kot znak. V tem primeru se dodajo kot normalni znaki, v nasprotnem pa kot posebni.

```
if((str.charAt(pos) == ';'
    || str.charAt(pos) == '{'
        || str.charAt(pos) == '}') && (pos-1 >= 0) && str.charAt(pos-1)=='\''
            && (pos+1 <str.length()) && str.charAt(pos-1)=='\'')
{
    wordList.add(new Beseda(str.charAt(pos) + "", distance, line));
}
else
{
    if((str.charAt(pos) == ';'
        || str.charAt(pos) == '{'
            || str.charAt(pos) == '}') && isString)
    {
        wordList.add(new Beseda(str.charAt(pos) + "", distance, line));
    }
    else
    {
        addSpecialChar(str.charAt(pos));
    }
}
```

Posebna metoda pa je potrebna zato, ker so nekateri posebni znaki sestavljeni iz več posebnih znakov. Kot primer vzemimo znak '='. Ta znak predstavlja prirejanje vrednosti neki spremenljivki. Vendar pa lahko iz tega znaka izpeljemo še '==' in '=<', ki pa predstavljata logična operatorja "enaka vrednost" ter "večji ali enak". V tej metodi se torej preverja še znak, ki sledi trenutnemu prebranemu znaku. Če le-tega ni, se beseda shrani kot '=', v nasprotnem primeru kot '==' oziroma '=<'. Celotna metoda izgleda takole:

```
private void addSpecialChar(char c)
{
    switch(c)
    {
        case ';':
            wordList.add(new Beseda(";",distance,line));

            if(!containsFor)
            {
                line++;
            }
            break;
        case ':':
            wordList.add(new Beseda(":",distance,line));
            break;
        case '?':
            wordList.add(new Beseda("?",distance,line));
            break;
        case '%':
            wordList.add(new Beseda("%",distance,line));
            break;
        case '+':
            pos = preglej("+",c,pos);
            break;
        case '/':
            pos = preglej("/",'=',pos);
            break;
        case '*':
            pos = preglej("*",'=',pos);
            break;
        case '-':
            pos = preglej("-",c,pos);
            break;
        case '&':
            pos = preglej("&",c,pos);
            break;
        case '|':
            pos = preglej("|",c,pos);
            break;
        case '=':
            pos = preglej("=",c,pos);
            break;
        case '!':
            pos = preglej("!",'=',pos);
            break;
```

```
            case '<':
                pos = preglej("<",'=',pos);
                break;
            case '>':
                pos = preglej(">",'=',pos);
                break;
            case '(':
                wordList.add(new Beseda("(",distance,line));
                break;
            case ')':
                wordList.add(new Beseda(")",distance,line));
                break;
            case '[':
                wordList.add(new Beseda("[",distance,line));
                break;
            case ']':
                wordList.add(new Beseda("]",distance,line));
                break;
            case ',':
                wordList.add(new Beseda(",",distance,line));
                break;
            case '{':
                if(!string.equals(""))
                {
                    wordList.add(new Beseda(string,distance,line));
                }

                containsFor = false;
                line++;
                wordList.add(new Beseda("{", distance, line));
                line++;
                distance++;
                break;
            case '}':
                line++;
                distance--;
                wordList.add(new Beseda("}", distance, line));
                line++;
                break;
            default:
                break;
        }
    }
```

Ta metoda pravilno kliče metode za preverjanje dvojnega posebnega znaka glede na podan znak, če je to seveda potrebno. Dvojne znake dodaja metoda `preglej(String s, char c, int i)`, ki izgleda takole:

```
private int preglej(String s, char c, int i)
{
    int temp = i;
    temp++;
    Beseda besT = new Beseda(s,distance,line);

    if(temp < str.length())
    {
        if(str.charAt(temp) == c)
        {
            besT.setBeseda(besT.getBeseda() + c);
            i = temp;
            wordList.add(besT);
        }
        else if(str.charAt(temp) == '/')
        {
            lineComment = true;
            besT.setBeseda(besT.getBeseda() + '/');
            i = temp;
            wordList.add(besT);
        }
        else
            wordList.add(besT);
    }
    else
        wordList.add(besT);

    return i++;
}
```

Ta metoda vrača tip `int`(celo število), ki predstavlja naslednjo pozicijo. To je pomebmo v primeru, če je naslednji znak del večznakovnega posebnega znaka ter se v tem primeru želimo izogniti dvakratnemu primerjanju istega znaka. Ta metoda je tudi zelo važna v primeru preverjanja znakov, ki predstavljajo komentar. Tu nastane problem, ker enovrstični komentarji večinoma ne vsebujejo podpičja in se vrstica ne prelomi. Zato sem tu vpeljal boolean spremenljivko `lineComment`, ki nekako označuje, kdaj se nahajamo znotraj tega komentarja. Sam prelom pa je implementiran tako, da ko se v seznam besed vstavi sam komentar, se za njim doda beseda “\n”, ki ob izpisu naredi novo vrstico.
To izvede naslednji stavek:

```
if(lineComment && (pos >= str.length()))
{
    wordList.add(new Beseda(string, distance, line));
    wordList.add(new Beseda("\n", distance, line));
    string = "";
    lineComment = false;
}
```

Na tej točki se tudi zaključi algoritem za razbijanje besed ter se začne samo izpisovanje besed.

3.4 IZPISOVANJE

Takoj za metodo `parse()` se kliče nov konstruktor razreda `CodePrinter`. Ta razred je namenjen izpisovanju prebranih besed v pravilni obliki. Vsebuje javno metodo `print()`, ki vsebuje glavni algoritem za izpisovanje ter množico drugih metod, katere pa niso javne in se lahko kličejo samo znotraj razreda.

```
if(args.length > 1)
{
    int breakDistance;

    try
    {
        breakDistance = Integer.parseInt(args[1]);
    }
    catch(NumberFormatException e)
    {
        throw new NumberFormatException();
    }
    codePrinter = new CodePrinter(codeParser.getWordList(), breakDistance);
}
else
{
    codePrinter = new CodePrinter(codeParser.getWordList(), 40);
}

codePrinter.print();
```

Preden se dejansko kliče metoda `print()`, se preveri, če obstaja 2. argument, ki določa maksimalno dovoljeno dolžino vrstic pri izpisovanju. Če ga ni, je iz zgornje kode razvidno, da se uporabi privzeta vrednost 40. Ta vrednost se poda v konstruktor skupaj s tabelo vseh besed, ki jih je prebrala metoda `parse()`. Konstruktor izgleda tako:

```
public CodePrinter(Object[] wordTable, int breakDistance)
{
    this.wordTable = wordTable;
    this.breakDistance = breakDistance;
}
```

Sama metoda `print()` prebira posamezne besede iz podane tabele prebranih besed ter jih skupaj dodaja v seznam, tako da sproti sestavlja trenutno vrstico za izpis. Trenutno vrstico predstavlja seznam:

```
private ArrayList<Beseda> currentLine;
```

Na začetku algoritma se prebere prva beseda ter se določi spremenljivka `line`, ki jo uporabljamo za primerjavo, katere besede pripadajo isti vrstici. Prav tako se iz te besede prebere dolžina zamika trenutne vrstice, na podlagi katere se nato “sestavi” odmik, ki je vstavljen na začetek seznama (na naslednji stran).

```
for(int i = 0; i<wordTable.length; i++)
{
    Beseda b1 = (Beseda)wordTable[i];

    int line = b1.getLine();
    int distance = b1.getDistance();
    String disStr = setSpace(distance);
    currentLine.add(new Beseda(disStr));
```

Za sestavljanje odmika se uporabi metoda `setSpace(int distance)`. Ta skupaj "zlepi" tolikšno število presledkov, kot je to podano v argumentu. Uporabljam privzete odmike, katere sem določil sam in znašajo 4 znake presledkov. Metoda izgleda takole:

```
private String setSpace(int distance)
{
    String str = "";
    int i = 0;

    while(i < distance)
    {
        str = str + "    ";
        i++;
    }

    return str;
}
```

Nato sledi zanka, ki prebira besede iz tabele besed toliko časa, dokler imajo vse besede zapisano isto vrednost vrstice, kot je shranjena v spremenljivki `line`. Če posamezna beseda ustreza temu pogoju, se pokliče posebna metoda, ki mora ponovno preverjati, ali je ta beseda posebni znak, in ki dejansko poskrbi, da se doda v seznam. Ta metoda se imenuje `buildLine`. Metoda `buildLine`:

```
private void buildLine(int n, Beseda b2, Beseda b1)
{
    if(isSpecialChar(b2))
    {
        currentLine.add(b2);
    }
    else
    {
        if(currentLine.size()-2 < 0)
        {
            currentLine.add(new Beseda(" "));
            currentLine.add(b2);
        }
        else
        {
            if(isPreviousSpecial(currentLine.size()-n))
            {
                currentLine.add(b2);
            }
```

```
            else
            {
                if(isDoubleChar(b2))
                {
                    currentLine.add(b2);
                }
                else
                {
                    currentLine.add(new Beseda(" "));
                    currentLine.add(b2);
                }
            }
        }
    }
}
```

Znotraj te metode se najprej preveri, če gre za običajno enoznakovno posebno besedo. To stori metoda `isSpecialChar(Beseda b)`, ki izgleda takole:

```
private boolean isSpecialChar(Beseda b2)
{
    if(b2.getBeseda().equals(";") || b2.getBeseda().equals("(")
        || b2.getBeseda().equals(")") || b2.getBeseda().equals("[")
            || b2.getBeseda().equals("]") || b2.getBeseda().equals(".")
                || b2.getBeseda().equals("*") || b2.getBeseda().equals("<")
                    || b2.getBeseda().equals(">") || b2.getBeseda().equals(","))
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

Če podana beseda ustreza enoznakovnemu znaku, se ta doda v seznam, ki predstavlja vrstico za izpis. V primeru, da beseda ne ustreza omenjenemu pogoju, se izvede novi pogojni stavek, ki preveri, če je prejšnja beseda posebni znak. To izvede s klicem metode `isPreviousSpecial(int pos)`. To se nanaša na znake, kot so oklepaji in znaki, ki predstavljajo začetek komentarja. Smisel te metode pa je, da se pri izpisovanju pravilno postavljajo presledki med posebnimi znaki. Metoda:

```
private boolean isPreviousSpecial(int pos)
{
    Beseda b = currentLine.get(pos);
    if(b.getBeseda().equals("(") || b.getBeseda().equals("[")
        || b.getBeseda().equals(".") || b.getBeseda().equals("//")
            || currentLine.get(pos).getBeseda().equals("\n"))
    {
        return true;
    }

    return false;
}
```

Če pa tudi ta pogoj ne drži, se še preveri, če je beseda slučajno dvojni posebni znak ++ ali --. Tudi to se preveri zaradi pravilne postavitve presledkov pri izpisovanju.

```
private boolean isDoubleChar(Beseda b)
{
    if(b.getBeseda().equals("++") || b.getBeseda().equals("--"))
    {
        return true;
    }

    return false;
}
```

To zaporedje se ponavlja, dokler ne preberemo besede, ki se nahaja v novi vrstici. V tem primeru se zanka prekine ter se izvede pogojni stavek, ki preveri, če je vrstica predolga. To naredimo s klicem metode `getCurrentLineCharSize()`. Metoda se nahaja spodaj:

```
private int getCurrentLineCharSize()
{
    int size = 0;
    for(int j = 0; j < currentLine.size(); j++)
    {
        Beseda b = currentLine.get(j);
        if(!areOnlySpaces(b))
        {
            size = size + b.getBeseda().length();
        }
    }
    return size;
}
```

Metoda se sprehodi skozi seznam, ki predstavlja vse besede v trenutni vrstici ter sešteje posamezne znake teh besed. Med štetjem pa se moramo izogniti presledkom, ki predstavljajo zamik glede na gnezdenje, kar storimo s klicem metode `areOnlySpaces(Beseda b)`. Koda te metode se nahaja spodaj:

```
private boolean areOnlySpaces(Beseda b)
{
    String str = b.getBeseda();
    int i = 0;
    while(i < str.length())
    {
        if(str.charAt(i) != ' ')
        {
            return false;
        }
        i++;
    }
    return true;
}
```

Pri primerjanju dolžine se uporabi morebitni drugi argument, ki se določi ob klicu samega programa. V primeru, da je vrstica predolga se kliče metoda `breakCurrentLine()`, ki

razbije vrstico na segmente, ki so manjši od maksimalne dovoljene dolžine ter izpiše vsakega posebej v novo vrstico. Če pa vrstica ni predolga, sledi takojšen izpis vrstice v metodi `printCurrentLine()`.

```
if(getCurrentLineCharSize() > breakDistance)
{
    breakCurrentLine();
}
else
{
    printCurrentLine(currentLine);
    currentLine.clear();
}
```

Po izpisu vrstice se celoten postopek ponovi, dokler se ne sprehodimo skozi celotno tabelo znakov, ki so bili prebrani iz datoteke.

V primeru, da je vrstica predolga, se v metodi `breakCurrentLine()` naprej preberejo vsi logični operatorji iz trenutne vrstice. Nato se sortirajo po prioriteti, katero računamo sproti za vsako besedo posebej. Začetek `breakCurrentLine()` izgleda takole:

```
private void breakCurrentLine()
{
    getCharsFromList(currentLine);
    Collections.sort(priorityCharList);
```

Vse logične operatorje preberem v metodi `getCharsFromList(ArrayList<Beseda> list)`. V tej metodi se sprehodimo skozi celotno trenutno vrstico ter preverjamo, če je trenutna beseda logični operator. Pri tem moramo paziti, da se operator ne nahaja znotraj niza ali znaka. Zato vpeljemo 2 `boolean` spremenljivki `isString` ter `isChar`, ki se nastavita na `true`, če se nahajamo znotraj niza ali znaka. Kot je že bilo omenjeno, je to pomembno, ker se del kode, ki se nahaja znotraj niza, ne sme spreminjati. Druga spremenljivka pa je namenjena temu, da se izognemu obravnavanju znakov znotraj znaka kot posebne znake. Del kode, ki skrbi za to:

```
if(!b.getBeseda().equals("'\\''")
    && !(firstChar == '\'') && !(lastChar == '\''))
{
    if(firstChar == '"')
    {
        if(!isString)
        {
            isString = true;
        }
        else
        {
            isString = false;
        }
    }
    else if(firstChar == '\'')
    {
```

```
        if(!isChar)
        {
            isChar = true;
        }
        else
        {
            isChar = false;
        }
    }
    else if(b.getBeseda().equals("("))
    {
        priority++;
    }
    else if(b.getBeseda().equals(")"))
    {
        priority--;
    }
    else if((isArithmeticChar(b.getBeseda())
            || b.getBeseda().equals("="))
                && !isString && !isChar)
    {
        priorityCharList.add(new Znak(getCharPriorityValue(
                b.getBeseda()) + priority, i, b.getBeseda()));
    }
```

Pogoja, ki preverjata, če je trenutna beseda oklepaj ali zaklepaj, sta del kode, ki skrbi za sprotno izračunavanje prioritete posameznega logičnega operatorja. O tem nekoliko kasneje. Če noben izmed naštetih pogojev ne drži, se izvede pogoj, ki dejansko preverja, če je beseda operator. Ta metoda se imenuje `isArithmeticChar(String znak)` in je sledeča:

```
private boolean isArithmeticChar(String znak)
{
    if(znak.equals("*") || znak.equals("/")
        || znak.equals("-") || znak.equals("+")
            || znak.equals("||") || znak.equals("&&")
                || znak.equals("&") || znak.equals("|")
                    || znak.equals("^") || znak.equals("%")
                        || znak.equals("<") || znak.equals("<=")
                            || znak.equals(">=") || znak.equals(">")
                                || znak.equals("==") || znak.equals("!="))
    {
        return true;
    }

    return false;
}
```

Če ta pogoj drži, se beseda doda v seznam logičnih operatorjev. Ta seznam vsebuje elemente razred `Znak` in je definiran na sledeč način:

```
private ArrayList<Znak> priorityCharList;
```

`Znak` je objekt, ki je namenjen izključno shranjevanju informacij o logičnih operatorjih. Vsebuje naslednje podatke: sam znak (npr. '=='), prioriteta ter pozicija. Prioriteta pomeni pomembnost posameznega operatorja glede na ostale logične operatorje. Pozicija pa predstavlja mesto, na katerem se nahaja posamezni operator znotraj trenutne vrstice. Konstruktor objekta `Znak` izgleda takole:

```
private int priority;
private int position;
private String znak;

public Znak(int priority, int position, String znak)
{
    this.priority = priority;
    this.position =  position;
    this.znak = znak;
}
```

Objekt seveda vsebuje metode, ki določajo/vračajo vrednosti, ki se nastavljajo v konstruktorjih. Omenil bi tudi, da implementira vmesnik `Comparable`:

```
public class Znak implements Comparable<Znak>
```

To je potrebno zaradi primerjanja vrednosti, ki se uporablja pri sortiranju seznama. Posledica implementacije je, da moramo implementirati metodo `compareTo(Znak znak)`, ki vrača sledeče vrednosti: 0, če sta vrednosti enaki; negativno, če je podana vrednost večja od trenutne in pozitivno, če je podana vrednost manjša od trenutne. Še sama metoda:

```
public int compareTo(Znak znak)
{
    if(znak.getPriority() == priority)
    {
        return 0;
    }
    else if(znak.getPriority() > priority)
    {
        return -1;
    }
    else
    {
        return 1;
    }
}
```

Če je trenutna beseda logični operator, se ta doda v seznam operatorjev z naslednjio metodo:

```
priorityCharList.add(
    new Znak(getCharPriorityValue(b.getBeseda()) + priority, i, b.getBeseda()));
```

Za izračun prioritete operatorja se uporablja metoda `getCharPriorityValue(String znak)`. Ta metoda vrne vrednost, ki pripada določenemu operatorju, in sicer glede na to, kateri operator močneje veže dva pogoja. Poleg tega se vrnjeni vrednosti prišteje še vrednost `priority`, ki predstavlja dodano vrednost glede na število oklepajev znotraj katerih je vgnezden operator. Metoda izgleda takole (na naslednji strani):

```
private int getCharPriorityValue(String znak)
{
    if(znak.equals("*") || znak.equals("/") || znak.equals("%"))
    {
        return 3;
    }
    else if(znak.equals("-") || znak.equals("+"))
    {
        return 2;
    }
    else if(znak.equals("||") || znak.equals("&&")
            || znak.equals("|") || znak.equals("&")
                || znak.equals("^") || znak.equals("<")
                    || znak.equals("<=") || znak.equals(">=")
                        || znak.equals(">") || znak.equals("==")
                            || znak.equals("!="))
    {
        return 1;
    }
    else
    {
        return 0;
    }
}
```

Na tej točki se tudi zaključi postopek dodajanja operatorja. Ko so poiskani in dodani vsi operatorji, se njihov seznam sortira po vrsti. Izpisovanje vrstic se začne tako, da naprej preverimo, če je bil najden samo en operator. Tu se vrstica prelomi točno na tistem operatorju, ne glede na posledice, kar pomeni, da mogoče izpis ne bo pravilen v lepotnem smislu, ampak bo ohranil sintaksno pravilnost, kar je najbolj pomembno. Vrstica se prelomi na dva dela, vsakemu posebej se doda odmik ter oba se zstrezno izpišeta.

Implementacija:

```
if(priorityCharList.size() == 1)
{
    printCurrentLine(currentLine.subList(
            0, priorityCharList.get(0).getPosition()));
    currentLine.subList(0,
            priorityCharList.get(0).getPosition()).clear();
    currentLine.add(0, new Beseda(setSpace(
            currentLine.get(0).getDistance() + 1)));
    printCurrentLine(currentLine.subList(0, currentLine.size()));
    currentLine.clear();
}
else
{
```

Po tem preverjanju se začne dejanski algoritem za lomljenje vrstice. Le-ta se sprehodi po seznamu operatorjev od začetka do konca. Osnovna logika algoritma je, da lepi skupaj dele vrstice do posamezne pozicije prebranega operatorja in sproti preverja dolžino zlepljene vrstice. Pri tem si tudi zapomnimo pozicijo zadnjega izpisanega znaka. Algoritem se začne tako, da prebere trenutni znak. Nato preverimo, če je razlika med pozicijo trenutno prebranega

znaka ter pozicijo nazadnje izpisanega znaka večja od največje dovoljene dolžine. Če je, se najprej doda zamik. Za določanje dolžine zamika ponovno uporabimo metodo `setSpace(int distance)`, katero smo opisali že zgoraj. Takoj za tem se pokliče metoda `printCurrentLine()`, ki izpiše trenutno vrstico. O tej metodi nekoliko kasneje. Sledi metoda, ki izprazni del vrstice, ki je bil pravkar izpisan, iz seznama. Ta hrani celotno predolgo vrstico. Če se zgornji pogoj ne izpolni, se preverja, če je dolžina še preostalega dela glavne vrstice manjša od najvišje dovoljene dolžine vrstice. V primeru izpolnitve pogoja se ponovno najprej vstavi presledek, nato pa preostanek vrstice izpiše ter njena vsebina izprazni in algoritem se konča. Če pa se tudi ta pogoj ne izpolni, se samo nastavi spremenljivka `lineLength`, ki opredeljuje dolžino trenutno sestavljene vrstice. Algoritem v celoti je sledeč:

```
while(i < priorityCharList.size() && !currentLine.isEmpty())
{
    Znak z = priorityCharList.get(i);
    if(z.getPosition() - printedPosition >= breakDistance)
    {
        if(lineLength > 0)
        {
            currentLine.add(0, new Beseda(setSpace(
                    currentLine.get(0).getDistance() + spaceDistance)));
            int tralala = lineLength - printedPosition;
            printCurrentLine(currentLine.subList(0, tralala));
            currentLine.subList(0, lineLength+1 - printedPosition).clear();
            printedPosition = priorityCharList.get(i-1).getPosition();
            lineLength = 0;
            spaceDistance++;
        }
        else
        {
            lineLength = z.getPosition();
        }
    }
    else
    {
        if(currentLine.size() <= breakDistance)
        {
            currentLine.add(0, new Beseda(setSpace(
                    currentLine.get(0).getDistance() + spaceDistance)));
            printCurrentLine(currentLine.subList(0, currentLine.size()));
            currentLine.clear();
        }
        else
        {
            lineLength = z.getPosition();
        }
    }

    i++;
}
```

Po algoritmu sledi še en pogojni stavek, ki izpiše preostanek vrstice, če se algoritem sprehodi skozi celoten seznam operatorjev in pri tem ne izpiše celotne vsebine. Pogojni stavek je videti takole:

```
if(!currentLine.isEmpty())
{
    currentLine.add(0, new Beseda(
            setSpace(currentLine.get(0).getDistance() + spaceDistance)));
    printCurrentLine(currentLine);
    currentLine.clear();
}
```

Tukaj se metoda za lomljene vrstice tudi konča.

Na koncu pa preostane samo še metoda za izpisovanje vrstic, ki je namenjena izpisovanju tekoče vrstice in tudi tekoče lomljene vrstice. Metoda kot argument sprejema seznam besed, ki predstavljajo trenutno vrstico ter jih izpisuje glede na pomen posamezne besede. Najprej definiramo tri spremenljivke, in sicer `isString`, `isChar` ter `wasString`. Spremenljivka `isString` se nastavi na vrednost `true`, kadar se nahajamo znotraj niza. To je pomembno zaradi pravilnega izpisovanja besed znotraj nizov ter njim pripadajočih presledkov in presledkov, ki se nahajajo takoj za in pred nizom. Spremenljivka `wasString` označuje, kadar se nahajamo takoj za izpisanim nizom, spremenljivka `isChar` pa, kadar se nahajamo v znaku. Tudi to je pomembno zaradi pravilnega izpisa presledkov. Takoj za omenjenimi deklaracijami sledi algoritem , ki se sprehodi skozi celoten seznam. Najprej se preverja, če je prebrani znak “\n”. Ta znak označuje prehod v novo vrstico, pomemben pa je, ker mu moramo pred izpisom določiti odmik. Primer iz kode:

```
private void printCurrentLine(List<Beseda> line)
{
    boolean isString = false;
    boolean isChar = false;
    boolean wasString = false;

    for(int i=0;i<line.size();i++)
    {
        Beseda b = (Beseda)line.get(i);

        if(i-1 >= 0 && line.get(i-1).getBeseda().equals("\n"))
        {
            int j = 0;
            while(j < line.get(i-1).getDistance())
            {
                System.out.print("    ");
                j++;
            }
        }
```

Nato sledijo pogojni stavki, ki preverjajo, če se nahajamo znotraj niza ali znaka in posledično nastavljajo vrednosti zgoraj opisanih spremenljivk. Ti pogojni stavki izgledajo takole:

```
if(b.getBeseda().length() > 0)
{
    if(b.getBeseda().charAt(0) == '"'
        || b.getBeseda().charAt(b.getBeseda().length()-1) == '"')
    {
        if(!(b.getBeseda().charAt(0)
            == '"' && b.getBeseda().charAt(b.getBeseda().length()-1)
                == '"' && b.getBeseda().length() == 2))
        {
            {
                if(isString)
                {
                    isString = false;
                    wasString = true;
                }
                else
                {
                    isString = true;
                    wasString = false;
                }
            }
        }
    }

    if(!b.getBeseda().equals("'\\''")
        && !(b.getBeseda().charAt(0) == '\'')
            && !(b.getBeseda().charAt(b.getBeseda().length()-1) == '\''))
    {
        if(b.getBeseda().charAt(0) == '\'' || b.getBeseda().equals(""))
        {
            if(isChar)
            {
                isChar = false;
            }
            else
            {
                isChar = true;
            }
        }
    }
}
```

Spremenljivke se ustrezno nastavljajo glede na pogoje, ki preverjajo, če je znak za začetek niza na začetku besede, na koncu besede in če je beseda sestavljena samo iz tega znaka ter če smo znotraj znaka ter enako za primer znaka.

Za tem sledi pogojni stavek, ki preverja, če je trenutna beseda presledek. Ta pogojni stavek je namenjen izpisovanju presledkov samo v določenih primerih. To pomeni, da če je trenutna beseda presledek, ni nujno, da bo ta presledek tudi izpisan. Začetni pogoj preverja, če

predhodnji stavek ni '!'. To je pomembo, ker se pred '!' ne izpisuje presledek! Sledijo pogoji, ki določajo, kdaj se presledek izpiše. Ti pogoji so:

- znak pred trenutno ter po trenutni besedi ne sme biti znak za začetek in konec niza;
- znak pred trenutno besedo mora biti znak za začetek niza ter celotna beseda pred trenutno besedo mora vsebovati več kot en znak;
- znak za trenutno besedo mora biti znak za konec niza ter celotna beseda za trenutno besedo mora vsebovati več kot en znak;
- znak spredaj ne sme biti '!'.

Z_naštetimi pogoji se želimo izogniti nepravilnemu izpisovanju presledkov za in pred znaki, ki označujejo niz ali znak ter stremimo k ohranjanju pravilnega izpisa besed znotraj niza. Izsek iz kode s pogoji:

```
if(!temp1.getBeseda().equals("!"))
{
    if(temp2.getBeseda().length() > 0
        && temp2.getBeseda().charAt(0) != '.' && !isString)
    {
        if(temp1.getBeseda().length() > 0
                && temp1.getBeseda().charAt(0) != '"'
                    && temp2.getBeseda().length() > 0
                        && temp2.getBeseda().charAt(temp2.getBeseda().length()-1)
                            != '"')
        {
            System.out.print(b.getBeseda());
        }
        else if(temp1.getBeseda().length() > 0
                && temp1.getBeseda().charAt(0)
                == '"' && temp1.getBeseda().length() > 1)
        {
            System.out.print(b.getBeseda());
        }
        else if(temp2.getBeseda().length() > 0
                && temp2.getBeseda().charAt(temp2.getBeseda().length()-1)
                    == '"' && temp2.getBeseda().length() > 1)
        {
            System.out.print(b.getBeseda());
        }
    }
}
else if(temp1.getBeseda().equals("!") && isString)
{
    System.out.print(b.getBeseda());
}
```

Če pa prebrana beseda ni presledek, se ta normalno izpiše. Vendar pa se pred njenim izpisom nahaja še nekaj pogojnih stavkov. Prvi preverja, če je predhodni znak '+' ter če se ne nahajamo znotraj niza. Ta pogojni stavek je pomemben, ker ima sam znak '+' lahko več pomenov in v primeru da gre za navaden '+' predenj postavi presledek. Naslednji pogojni stavek pa je namenjen pravilnemu izpisovanju znakov v primeru, da se nahajamo znotraj znaka. Še vzorec kode samega izpisa (na naslednji strani):

```
else
{
    if(i-1>=0 && line.get(i-1).getBeseda().length()>0)
    {
        if(line.get(i).getBeseda().equals("+")
            && !isString && wasString
                && !line.get(i-1).getBeseda().equals(" "))
        {
            System.out.print(" ");
        }
    }

    if(!b.getBeseda().equals(" "))
    {
        System.out.print(b.getBeseda());
    }
    else if(b.getBeseda().equals(" ")
            && line.get(i-1).getBeseda().charAt(0)
                == '\'' && line.get(i+1).getBeseda().charAt(0) == '\'')
    {
        System.out.print(b.getBeseda());
    }
}
```

4. ZAKLJUČEK

Urejevalnik javanske kode, ki je predstavljen v mojem diplomskem delu, pokriva vse osnovne funkcije, ki naj bi jih vsak urejevalnik kode imel že kot privzete. Pri samem programiranju nisem imel večjih težav, omenil pa bi nekaj manjših, ki sem jih tekom programiranja tudi uspešno rešil. Imel sem probleme s presledki okoli znakov, ki imajo znotraj javanske sintakse več pomenov (to zadeva predvsem '+' in '*') ter tudi s presledki in besedami znotraj nizov ter komentarjev. Tu je bilo treba paziti, da sintaksna pravila za metode znotraj niza ali komentarja niso veljala za rezervirane besede, ker se format teh besedil ne sme spreminjati. Nekaj manjših težav pa se je pojavilo tudi pri programiranju lomljenja predolgih vrstic.

Opozoril bi tudi še na nekatere težave, ki se pojavljajo ob uporabi. Tu mislim na občasne presledke. Ti presledki v določenih primerih niso pravilni (uporaba generikov). Prav tako so tudi mogoče težave pri nekaterih lomljenih vrsticah.

Od točke, na kateri sem zaključil moje diplomsko delo, pa so možnosti za razširitve in izboljšave praktično neomejene. Pod izboljšave spada optimizacija določenih algoritmov za urejanje, razbiranje in izpisovanje besed. Od razširitev bi omenil barvanje ključnih metod, spremembe vrste pisave za določene tipe spremenljivk ali metod, samodejno označevanje pripadnosti stavkov posameznim metodam, itd., za kar pa bi bilo potrebno implementirati tudi grafični vmesnik. Omenil bi tudi nekatere pametne funkcije, kot so samodejne predloge novih metod posameznega objekta, pametno popravljanje programerjeve logike, predloge za optimalno rabo deklaracij in posameznih metod in pa seveda možnosti celotnega nastavljanja pravil urejanja (presledki, odmiki, itd.).

5. PRILOGE

Priloga A: izvorna koda celotnega programa pred ureditvijo z diplomskim programom

***Razred Znak*:**

```
public class Znak implements Comparable<Znak> {private int priority; private int position; private String
znak; public Znak(int priority, int position, String znak) { this.priority = priority; this.position =
position; this.znak = znak; } public int getPriority() { return priority; } public int getPosition() {
return position; } public String getZnak() { return znak; } public int compareTo(Znak znak) {
if(znak.getPriority() == priority) { return 0; } else if(znak.getPriority() > priority) { return -1; } else
{ return 1; } } }
```

***Razred Main*:**

```
import java.io.*; public class Main { public static void main(String[] args) { CodeParser codeParser =
null; try { File dat = new File(args[0]); codeParser = new CodeParser(dat); codeParser.parse(); }
catch(Exception e) { e.printStackTrace(); } CodePrinter codePrinter; if(args.length > 1) { int
breakDistance; try { breakDistance = Integer.parseInt(args[1]); } catch(NumberFormatException e) { throw
new NumberFormatException(); } codePrinter = new CodePrinter(codeParser.getWordList(), breakDistance); }
else { codePrinter = new CodePrinter(codeParser.getWordList(), 40); } codePrinter.print(); } }
```

***Razred Beseda*:**

```
public class Beseda { private String beseda; private int distance; private int line; public Beseda() {
distance = 0; line = 0; } public Beseda(String beseda) { this.beseda = beseda; distance = 0; line = 0; }
public Beseda(String beseda, int value, int line) { this.beseda = beseda; this.distance = value; this.line
= line; } public String getBeseda() { return beseda; } public void setBeseda(String newBeseda) { beseda =
newBeseda; } public void setValue(int newValue) { distance = newValue; } public void setLine(int newLine) {
line = newLine; } public int getDistance() { return distance; } public int getLine() { return line; }
public String toString() { return beseda + ' ' + distance + ' ' + line; } }
```

***Razred CodeParser*:**

```
import java.io.*; import java.util.ArrayList; public class CodeParser { private File toParse; private
ArrayList<Beseda> wordList = new ArrayList<Beseda>(); private String str; private int distance = 0; private
int line = 0; private String string = ""; private boolean lineComment = false; private int oklepaji = 0;
private int pos; private boolean containsFor = false; public CodeParser(File toParse) { this.toParse =
toParse; } public void parse() { try { BufferedReader read = new BufferedReader(new FileReader(toParse));
pos = 0; boolean isString = false; boolean isChar = false; while((str = read.readLine()) != null) { pos =
0; str = str.replace(' ', ' '); while(pos < str.length()) { if(str.charAt(pos) == '"' && str.charAt(pos-1)
!= '\'' && str.charAt(pos+1) != '\'') { if(isString) { isString = false; } else { isString = true; } }
if(str.charAt(pos) == '\'') { if(isChar) { isChar = false; } else { isChar = true; } }
if(string.equals("for") && !lineComment && (str.charAt(pos) == ' ' || str.charAt(pos) == '(')) {
containsFor = true; wordList.add(new Beseda(string, distance, line)); string = ""; if(str.charAt(pos) ==
'(') { wordList.add(new Beseda("(", distance, line)); oklepaji++; } } else
if(isSpecialChar(str.charAt(pos))) { if(!string.equals("")) { wordList.add(new Beseda(string, distance,
line)); } if(str.charAt(pos) == '(' && containsFor) { oklepaji++; } else if(str.charAt(pos) == ')' &&
containsFor) { oklepaji--; } string = ""; if((str.charAt(pos) == ';' || str.charAt(pos) == '{' ||
str.charAt(pos) == '}') && (pos-1 >= 0) && str.charAt(pos-1)=='\'' && (pos+1 <str.length()) &&
str.charAt(pos-1)=='\'') { wordList.add(new Beseda(str.charAt(pos) + "", distance, line)); } else {
if((str.charAt(pos) == ';' || str.charAt(pos) == '{' || str.charAt(pos) == '}') && isString) {
wordList.add(new Beseda(str.charAt(pos) + "", distance, line)); } else { addSpecialChar(str.charAt(pos)); }
} } else if(str.charAt(pos) == ' ') { if(!string.equals("") && !string.equals(" ")) { wordList.add(new
Beseda(string, distance, line));
string = ""; } } else if(str.charAt(pos) != ' ') { string = string + str.charAt(pos); } pos++; }
if(lineComment && (pos >= str.length())) { wordList.add(new Beseda(string, distance, line));
wordList.add(new Beseda("\n", distance, line)); string = ""; lineComment = false; } } read.close(); }
catch(Exception e) { e.printStackTrace(); } } private void addSpecialChar(char c) { switch(c) { case ';':
wordList.add(new Beseda(";",distance,line)); if(!containsFor) { line++; } break; case ':': wordList.add(new
Beseda(":",distance,line)); break; case '?': wordList.add(new Beseda("?",distance,line)); break; case '%':
wordList.add(new Beseda("%",distance,line)); break; case '+': pos = preglej("+",c,pos); break; case '/':
pos = preglej("/",'=',pos); break; case '*': pos = preglej("*",'=',pos); break; case '-': pos = preglej("-
",c,pos); break; case '&': pos = preglej("&",c,pos); break; case '|': pos = preglej("|",c,pos); break; case
'=': pos = preglej("=",c,pos); break; case '!': pos = preglej("!",'=',pos); break; case '<':
pos = preglej("<",'=',pos); break; case '>': pos = preglej(">",'=',pos); break; case '(': wordList.add(new
Beseda("(",distance,line)); break; case ')': wordList.add(new Beseda(")",distance,line)); break; case '[':
wordList.add(new Beseda("[",distance,line)); break; case ']': wordList.add(new Beseda("]",distance,line));
break; case ',': wordList.add(new Beseda(",",distance,line)); break; case '{': if(!string.equals("")) {
wordList.add(new Beseda(string,distance,line)); } containsFor = false; line++; wordList.add(new Beseda("{",
distance, line)); line++; distance++; break; case '}': line++; distance--; wordList.add(new Beseda("}",
```

```
distance, line)); line++; break; default: break; } } private int preglej(String s, char c, int i) { int
temp = i; temp++; Beseda besT = new Beseda(s,distance,line); if(temp < str.length()) { if(str.charAt(temp)
== c) { besT.setBeseda(besT.getBeseda() + c); i = temp; wordList.add(besT); } else if(str.charAt(temp) ==
'/') { lineComment = true; besT.setBeseda(besT.getBeseda() + '/');
i = temp; wordList.add(besT); } else wordList.add(besT); } else wordList.add(besT); return i++; } private
boolean isSpecialChar(char ch) { if((ch=='<')||(ch=='>')||(ch=='+') ||(ch=='-')||(ch=='%')||(ch=='{')
||(ch=='}')||(ch=='!')||(ch=='*') ||(ch=='/')||(ch==';')||(ch==':') ||(ch=='=')||(ch==',')||(ch=='(')
||(ch==')')||(ch=='[')||(ch==']') ||(ch=='?')||(ch=='|') || (ch=='&')) { return true; } return false; }
public void printWordList() { Beseda b; int i=0; while(i < wordList.size()) { b = (Beseda)wordList.get(i);
System.out.println("beseda: "+b); i++; if(i > wordList.size()) break; } } public Object[] getWordList() {
Object[] b = wordList.toArray(); return b; } }
```

***Razred CodePrinter*:**

```
import java.util.*; public class CodePrinter { private Object[] wordTable; private ArrayList<Beseda>
currentLine; private ArrayList<Znak> priorityCharList; private int breakDistance; public
CodePrinter(Object[] wordTable, int breakDistance) { this.wordTable = wordTable; this.breakDistance =
breakDistance; } public void print() { currentLine = new ArrayList<Beseda>(); for(int i = 0;
i<wordTable.length; i++) { Beseda b1 = (Beseda)wordTable[i]; int line = b1.getLine(); int distance =
b1.getDistance(); String disStr = setSpace(distance); currentLine.add(new Beseda(disStr));
currentLine.add(b1); do { Beseda b2 = null; if(i+1 < wordTable.length) { b2 = (Beseda)wordTable[i+1];
if(b2.getLine() == line) { i++; Beseda tmp = (Beseda)wordTable[i-1]; if(tmp.getBeseda().equals("/") ||
tmp.getBeseda().equals("*")) { if(b2.getBeseda().equals("/") || b2.getBeseda().equals("*")) {
currentLine.add(b2); } else { buildLine(2, b2, tmp); } } else { buildLine(1, b2, tmp); } } } else break;
if(b2 != null) { if(b2.getLine()!= line)
break; } }while(true); if(getCurrentLineCharSize() > breakDistance) { breakCurrentLine(); } else {
printCurrentLine(currentLine); currentLine.clear(); } } } private int getCurrentLineCharSize() { int size =
0; for(int j = 0; j < currentLine.size(); j++) { Beseda b = currentLine.get(j); if(!areOnlySpaces(b)) {
size = size + b.getBeseda().length(); } } return size; } private boolean areOnlySpaces(Beseda b) { String
str = b.getBeseda(); int i = 0; while(i < str.length()) { if(str.charAt(i) != ' ') { return false; } i++; }
return true; } private void buildLine(int n, Beseda b2, Beseda b1) { if(isSpecialChar(b2)) {
currentLine.add(b2); } else { if(currentLine.size()-2 < 0) { currentLine.add(new Beseda(" "));
currentLine.add(b2); } else { if(isPreviousSpecial(currentLine.size()-n)) { currentLine.add(b2); } else {
if(isDoubleChar(b2)) { currentLine.add(b2); } else { currentLine.add(new Beseda(" ")); currentLine.add(b2);
} } } } } private boolean isSpecialChar(Beseda b2) {
if(b2.getBeseda().equals(";") || b2.getBeseda().equals("(") || b2.getBeseda().equals(")") ||
b2.getBeseda().equals("[") || b2.getBeseda().equals("]") || b2.getBeseda().equals(".") ||
b2.getBeseda().equals("*") || b2.getBeseda().equals("<") || b2.getBeseda().equals(">") ||
b2.getBeseda().equals(",")) { return true; } else { return false; } } private String setSpace(int distance)
{ String str = ""; int i = 0; while(i < distance) { str = str + " "; i++; } return str; } private void
printCurrentLine(List<Beseda> line) { boolean isString = false; boolean isChar = false; boolean wasString =
false; for(int i=0;i<line.size();i++) { Beseda b = (Beseda)line.get(i); if(i-1 >= 0 && line.get(i-
1).getBeseda().equals("\n")) { int j = 0; while(j < line.get(i-1).getDistance()) { System.out.print(" ");
j++; } } if(b.getBeseda().length() > 0) { if(b.getBeseda().charAt(0) == '"' ||
b.getBeseda().charAt(b.getBeseda().length()-1) == '"') { if(!(b.getBeseda().charAt(0) == '"' &&
b.getBeseda().charAt(b.getBeseda().length()-1)
== '"' && b.getBeseda().length() == 2)) { { if(isString) { isString = false; wasString = true; } else {
isString = true; wasString = false; } } } } if(!b.getBeseda().equals("'\\'") && !(b.getBeseda().charAt(0)
== '\'') && !(b.getBeseda().charAt(b.getBeseda().length()-1) == '\'')) { if(b.getBeseda().charAt(0) == '\''
|| b.getBeseda().equals("")) { if(isChar) { isChar = false; } else { isChar = true; } } } }
if((b.getBeseda().equals(">") || b.getBeseda().equals("<")) && !isString && !isChar) { System.out.print("
"); } if(b.getBeseda().equals(" ") && !isChar) { if(i-1 >= 0 && i+1 < line.size()) { Beseda temp1 =
line.get(i-1); Beseda temp2 = line.get(i+1); if(!temp1.getBeseda().equals("!")) {
if(temp2.getBeseda().length() > 0 && temp2.getBeseda().charAt(0) != '.' && !isString) {
if(temp1.getBeseda().length() > 0 && temp1.getBeseda().charAt(0) != '"' && temp2.getBeseda().length() > 0
&& temp2.getBeseda().charAt(temp2.getBeseda().length()-1) != '"') { System.out.print(b.getBeseda()); }
else if(temp1.getBeseda().length() > 0 && temp1.getBeseda().charAt(0) == '"' && temp1.getBeseda().length()
> 1) { System.out.print(b.getBeseda()); } else if(temp2.getBeseda().length() > 0 &&
temp2.getBeseda().charAt(temp2.getBeseda().length()-1) == '"' && temp2.getBeseda().length() > 1) {
System.out.print(b.getBeseda()); } } } else if(temp1.getBeseda().equals("!") && isString) {
System.out.print(b.getBeseda()); } } } else { if(i-1>=0 && line.get(i-1).getBeseda().length()>0) {
if(line.get(i).getBeseda().equals("+") && !isString && wasString && !line.get(i-1).getBeseda().equals(" "))
{ System.out.print(" "); } } if(!b.getBeseda().equals(" ")) { System.out.print(b.getBeseda()); } else
if(b.getBeseda().equals(" ") && line.get(i-1).getBeseda().charAt(0) == '\'' &&
line.get(i+1).getBeseda().charAt(0) == '\'') { System.out.print(b.getBeseda()); } } } System.out.println();
} private boolean isPreviousSpecial(int pos) { Beseda b = currentLine.get(pos);
if(b.getBeseda().equals("(") || b.getBeseda().equals("[") || b.getBeseda().equals(".") ||
b.getBeseda().equals("//") ||currentLine.get(pos).getBeseda().equals("\n")) { return true; } return false;
} private boolean isDoubleChar(Beseda b) { if(b.getBeseda().equals("++") || b.getBeseda().equals("--")) {
return true; } return false; } private void breakCurrentLine() { getCharsFromList(currentLine);
Collections.sort(priorityCharList); int i = 0; int lineLength = 0; int printedPosition = 0; int
spaceDistance = 0; if(priorityCharList.size() == 1) { printCurrentLine(currentLine.subList( 0,
priorityCharList.get(0).getPosition())); currentLine.subList(0,
priorityCharList.get(0).getPosition()).clear(); currentLine.add(0, new Beseda(setSpace(
currentLine.get(0).getDistance() + 1))); printCurrentLine(currentLine.subList(0, currentLine.size()));
currentLine.clear(); } else { while(i < priorityCharList.size() && !currentLine.isEmpty()) { Znak z =
priorityCharList.get(i);
if(z.getPosition() - printedPosition >= breakDistance) { if(lineLength > 0) { currentLine.add(0, new
Beseda(setSpace( currentLine.get(0).getDistance() + spaceDistance))); int tralala = lineLength -
printedPosition; printCurrentLine(currentLine.subList(0, tralala)); currentLine.subList(0, lineLength+1 -
printedPosition).clear(); printedPosition = priorityCharList.get(i-1).getPosition(); lineLength = 0;
spaceDistance++; } else { lineLength = z.getPosition(); } } else { if(currentLine.size() <= breakDistance)
{ currentLine.add(0, new Beseda(setSpace( currentLine.get(0).getDistance() + spaceDistance)));
printCurrentLine(currentLine.subList(0, currentLine.size())); currentLine.clear(); } else { lineLength =
z.getPosition(); } } i++; } if(!currentLine.isEmpty()) { currentLine.add(0, new
Beseda(setSpace(currentLine.get(0).getDistance() + spaceDistance))); printCurrentLine(currentLine);
```

```
currentLine.clear(); } } } private void getCharsFromList(ArrayList<Beseda> list) { int priority = 0;
boolean isString = false;
boolean isChar = false; priorityCharList = new ArrayList<Znak>(); for(int i=0; i<list.size();i++) { Beseda
b = list.get(i); char firstChar = 'c'; char lastChar = 'c'; if(b.getBeseda().length() > 0) { firstChar =
b.getBeseda().charAt(0); lastChar = b.getBeseda().charAt(b.getBeseda().length()-1); }
if(!b.getBeseda().equals("'\\''") && !(firstChar == '\'') && !(lastChar == '\'')) { if(firstChar == '"') {
if(!isString) { isString = true; } else { isString = false; } } else if(firstChar == '\'') { if(!isChar) {
isChar = true; } else { isChar = false; } } else if(b.getBeseda().equals("(")) { priority++; } else
if(b.getBeseda().equals(")")) { priority--; } else if((isArithmeticChar(b.getBeseda()) ||
b.getBeseda().equals("=")) && !isString && !isChar) { priorityCharList.add(new Znak(getCharPriorityValue(
b.getBeseda()) + priority, i, b.getBeseda())); } if(b.getBeseda().length() > 0) {
if(b.getBeseda().charAt(0) == '"' && b.getBeseda().charAt(b.getBeseda().length()-1) == '"') { isString =
false; } } } } }
private boolean isArithmeticChar(String znak) { if(znak.equals("*") || znak.equals("/") || znak.equals("-")
|| znak.equals("+") || znak.equals("||") || znak.equals("&&") || znak.equals("&") || znak.equals("|") ||
znak.equals("^") || znak.equals("%") || znak.equals("<") || znak.equals("<=") || znak.equals(">=") ||
znak.equals(">") || znak.equals("==") || znak.equals("!=")) { return true; } return false; } private int
getCharPriorityValue(String znak) { if(znak.equals("*") || znak.equals("/") || znak.equals("%")) { return
3; } else if(znak.equals("-") || znak.equals("+")) { return 2; } else if(znak.equals("||") ||
znak.equals("&&") || znak.equals("|") || znak.equals("&") || znak.equals("^") || znak.equals("<") ||
znak.equals("<=") || znak.equals(">=") || znak.equals(">") || znak.equals("==") || znak.equals("!=") ||
znak.equals("=")) { return 1; } else { return 0; } } }
```

Priloga B: primer urejanja na izvorni kodi diplomske

Razred Main:

```
import java.io.*;
public class Main
{
    public static void main(String[] args)
    {
        CodeParser codeParser = null;
        try
        {
            File dat
                = new File(args[0]);
            codeParser
                = new CodeParser(dat);
            codeParser.parse();
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
        CodePrinter codePrinter;
        if(args.length > 1)
        {
            int breakDistance;
            try
            {
                breakDistance
                    = Integer.parseInt(args[1]);
            }
            catch(NumberFormatException e)
            {
                throw new NumberFormatException();
            }
            codePrinter
                = new CodePrinter(codeParser.getWordList(), breakDistance);
        }
        else
        {
            codePrinter
                = new CodePrinter(codeParser.getWordList(), 40);
        }
        codePrinter.print();
    }
}
```

***Razred Znak*:**

```
public class Znak implements Comparable < Znak >
{
    private int priority;
    private int position;
    private String znak;
    public Znak(int priority, int position, String znak)
    {
        this.priority = priority;
        this.position = position;
        this.znak = znak;
    }
    public int getPriority()
    {
        return priority;
    }
    public int getPosition()
    {
        return position;
    }
    public String getZnak()
    {
        return znak;
    }
    public int compareTo(Znak znak)
    {
        if(znak.getPriority() == priority)
        {
            return 0;
        }
        else if(znak.getPriority() > priority)
        {
            return - 1;
        }
        else
        {
            return 1;
        }
    }
}
```

Razred Beseda:

```
public class Beseda
{
    private String beseda;
    private int distance;
    private int line;
    public Beseda()
    {
        distance = 0;
        line = 0;
    }
    public Beseda(String beseda)
    {
        this.beseda = beseda;
        distance = 0;
        line = 0;
    }
    public Beseda(String beseda, int value, int line)
    {
        this.beseda = beseda;
        this.distance = value;
        this.line = line;
    }
    public String getBeseda()
    {
        return beseda;
    }
    public void setBeseda(String newBeseda)
    {
        beseda = newBeseda;
    }
    public void setValue(int newValue)
    {
        distance = newValue;
    }
    public void setLine(int newLine)
    {
        line = newLine;
    }
    public int getDistance()
    {
        return distance;
    }
    public int getLine()
    {
        return line;
    }
    public String toString()
    {
        return beseda + ' ' + distance + ' ' + line;
    }
}
```

Razred CodeParser:

```
import java.io.*;
import java.util.ArrayList;
public class CodeParser
{
    private File toParse;
    private ArrayList < Beseda > wordList = new ArrayList < Beseda >();
    private String str;
    private int distance = 0;
    private int line = 0;
    private String string = "";
    private boolean lineComment = false;
    private int oklepaji = 0;
    private int pos;
    private boolean containsFor = false;
    public CodeParser(File toParse)
    {
        this.toParse = toParse;
    }
    public void parse()
    {
```

```
try
{
    BufferedReader read
        = new BufferedReader(new FileReader(toParse));
    pos = 0;
    boolean isString = false;
    while((str = read.readLine()) != null)
    {
        pos = 0;
        str = str.replace('  ', ' ');
        while(pos < str.length())
        {
            if(str.charAt(pos)
                == '"'&&str.charAt(pos-1)!='\'')
            {
                if(isString)
                {
                    isString = false;
                }
                else
                {
                    isString = true;
                }
            }
            if(string.equals("for") && !lineComment &&(str.charAt(pos) == ' ' || str.charAt(pos) == '('))
            {
                containsFor = true;
                wordList.add(new Beseda(string, distance, line));
                string = "";
                if(str.charAt(pos) == '(')
                {
                    wordList.add(new Beseda("(", distance, line));
                    oklepaji++;
                }
            }
            else if(isSpecialChar(str.charAt(pos)))
            {
                if(!string.equals(""))
                {
                    wordList.add(new Beseda(string, distance, line));
                }
                if(str.charAt(pos) == '(' && containsFor)
                {
                    oklepaji++;
                }
              else if(str.charAt(pos) == ')' && containsFor)
              {
                  oklepaji--;
              }
              string = "";
              if((str.charAt(pos) == ';' || str.charAt(pos) == '(' || str.charAt(pos)
                  == ')') &&(pos - 1 >= 0) && str.charAt(pos - 1) == '\'')
              {
                  wordList.add(new Beseda(str.charAt(pos)
                      + "", distance, line));
              }
              else
              {
                  if((str.charAt(pos) == ';' || str.charAt(pos) == '(' || str.charAt(pos) == ')') && isString)
                  {
                      wordList.add(new Beseda(str.charAt(pos)
                          + "", distance, line));
                  }
                  else
                  {
                      addSpecialChar(str.charAt(pos));
                  }
              }
            }
            else if(str.charAt(pos) == ' ')
            {
                if(!string.equals("")
                    && !string.equals(""))
                {
                    wordList.add(new Beseda(string, distance, line));
                    string = "";
                }
            }
```

```
                else if(str.charAt(pos) != ' ')
                {
                    string = string + str.charAt(pos);
                }
                pos++;
            }
            if(lineComment &&(pos >= str.length()))
            {
                wordList.add(new Beseda(string, distance, line));
                wordList.add(new Beseda("\n", distance, line));
                string = "";
                lineComment = false;
            }
        }
        read.close();
    }
    catch(Exception e)
    {
        e.printStackTrace();
    }
}
private void addSpecialChar(char c)
{
    switch(c)
    {
        case ';' : wordList.add(new Beseda(";", distance, line));
        if(!containsFor)
        {
            line++;
        }
        break;
        case ':' : wordList.add(new Beseda(":", distance, line));
        break;
        case '?' : wordList.add(new Beseda("?", distance, line));
        break;
        case '%' : wordList.add(new Beseda("
            %", distance, line));
        break;
        case '+' : pos = preglej("+", c, pos);
        break;
        case '/' : pos = preglej("/", '=', pos);
        break;
        case '*' : pos = preglej("*", '=', pos);
        break;
        case '-' : pos = preglej("-", c, pos);
        break;
        case '&' : pos = preglej("&", c, pos);
        break;
        case '|' : pos = preglej("|", c, pos);
        break;
        case '=' : pos = preglej("=", c, pos);
        break;
        case '!' : pos = preglej("!", '=', pos);
        break;
        case '<' : pos = preglej(" <", '=', pos);
        break;
        case '>' : pos = preglej(" >", '=', pos);
        break;
        case '(' : wordList.add(new Beseda("(", distance, line));
        break;
        case ')' : wordList.add(new Beseda(")", distance, line));
        break;
        case '[' : wordList.add(new Beseda("[", distance, line));
        break;
        case ']' : wordList.add(new Beseda("]", distance, line));
        break;
        case ',' : wordList.add(new Beseda(",", distance, line));
        break;
        case '{' : if(!string.equals(""))
        {
            wordList.add(new Beseda(string, distance, line));
        }
        containsFor = false;
        line++;
        wordList.add(new Beseda("{", distance, line));
```

```
            line++;
            distance++;
            break;
            case '}' : line++;
            distance--;
            wordList.add(new Beseda("}", distance, line));
            line++;
            break;
            default : break;
        }
    }
    private int preglej(String s, char c, int i)
    {
        int temp = i;
        temp++;
        Beseda besT = new Beseda(s, distance, line);
        if(temp < str.length())
        {
            if(str.charAt(temp) == c)
            {
                besT.setBeseda(besT.getBeseda() + c);
                i = temp;
                wordList.add(besT);
            }
            else if(str.charAt(temp) == '/')
            {
                lineComment = true;
                besT.setBeseda(besT.getBeseda() + '/');
                i = temp;
                wordList.add(besT);
            }
            else wordList.add(besT);
        }
        else wordList.add(besT);
        return i++;
    }
    private boolean isSpecialChar(char ch)
    {
        if((ch == '<') ||(ch == '>') ||(ch == '+')
            ||(ch == '-') ||(ch == '%') ||(ch == '(')
                ||(ch == ')') ||(ch == '!') ||(ch == '*')
                    ||(ch == '/') ||(ch == ';') ||(ch == ':')
                        ||(ch == '=') ||(ch == ',')
                            ||(ch == '(') ||(ch == ')') ||(ch == '[')
                                ||(ch == ']') ||(ch == '?') ||(ch
                                    == '|') ||(ch == '&'))
        {
            return true;
        }
        return false;
    }
    public void printWordList()
    {
        Beseda b;
        int i = 0;
        while(i < wordList.size())
        {
            b =(Beseda) wordList.get(i);
            System.out.println("beseda:"+b);
            i++;
            if(i > wordList.size()) break;
        }
    }
    public Object[] getWordList()
    {
        Object[] b = wordList.toArray();
        return b;
    }
}
```

Razred CodePrinter:

```
import java.util.*;
public class CodePrinter
{
    private Object[] wordTable;
    private ArrayList < Beseda > currentLine;
    private ArrayList < Znak > priorityCharList;
    private int breakDistance;
    public CodePrinter(Object[] wordTable, int breakDistance)
    {
        this.wordTable = wordTable;
        this.breakDistance = breakDistance;
    }
    public void print()
    {
        currentLine = new ArrayList < Beseda >();
        for(int i = 0; i < wordTable.length; i++)
        {
            Beseda b1 =(Beseda) wordTable[i];
            int line = b1.getLine();
            int distance = b1.getDistance();
            String disStr = setSpace(distance);
            currentLine.add(new Beseda(disStr));
            currentLine.add(b1);
            do
            {
                Beseda b2 = null;
                if(i + 1 < wordTable.length)
                {
                    b2 =(Beseda) wordTable[i + 1];
                    if(b2.getLine() == line)
                    {
                        i++;
                        Beseda tmp =(Beseda) wordTable[i - 1];
                        if(tmp.getBeseda().equals("/") || tmp.getBeseda().equals("*"))
                        {
                            if(b2.getBeseda().equals("/") || b2.getBeseda().equals("*"))
                            {
                                currentLine.add(b2);
                            }
                            else
                            {
                                buildLine(2, b2, tmp);
                            }
                        }
                     }
                     else
                     {
                         buildLine(1, b2, tmp);
                     }
                  }
               }
               else break;
               if(b2 != null)
               {
                   if(b2.getLine() != line) break;
               }
           }
           while(true);
           if(getCurrentLineCharSize()
                > breakDistance)
           {
               breakCurrentLine();
           }
```

```
        else
        {
            printCurrentLine(currentLine);
            currentLine.clear();
        }
    }
}
private int getCurrentLineCharSize()
{
    int size = 0;
    for(int j = 0; j < currentLine.size(); j++)
    {
        Beseda b = currentLine.get(j);
        if(!areOnlySpaces(b)) size = size + b.getBeseda().length();
    }
    return size;
}
private boolean areOnlySpaces(Beseda b)
{
    String str = b.getBeseda();
    int i = 0;
    while(i < str.length())
    {
        if(str.charAt(i) != ' ')
        {
            return false;
        }
        i++;
    }
    return true;
}
private void buildLine(int n, Beseda b2, Beseda b1)
{
    if(isSpecialChar(b2))
    {
        currentLine.add(b2);
    }
    else
    {
        if(currentLine.size() - 2 < 0)
        {
            currentLine.add(new Beseda(""));
            currentLine.add(b2);
        }
        else
        {
            if(isPreviousSpecial(currentLine.size()
                - n))
            {
                currentLine.add(b2);
            }
            else
            {
                if(isDoubleChar(b2))
                {
                    currentLine.add(b2);
                }
                else
                {
                    currentLine.add(new Beseda(""));
                    currentLine.add(b2);
                }
            }
        }
    }
}
private boolean isSpecialChar(Beseda b2)
{
    if(b2.getBeseda().equals(";")
        || b2.getBeseda().equals("(") || b2.getBeseda().equals(")") || b2.getBeseda().equals("[")
            || b2.getBeseda().equals("]") || b2.getBeseda().equals(".")||b2.getBeseda().equals("*")
                || b2.getBeseda().equals("<") || b2.getBeseda().equals(">") || b2.getBeseda().equals(","))
```

```
    {
        return true;
    }
    else
    {
        return false;
    }
}
private String setSpace(int distance)
{
    String str = "";
    int i = 0;
    while(i < distance)
    {
        str = str +"";
        i++;
    }
    return str;
}
private void printCurrentLine(List < Beseda > line)
{
    boolean isString = false;
    boolean isChar = false;
    boolean wasString = false;
    for(int i = 0; i < line.size(); i++)
    {
        Beseda b =(Beseda) line.get(i);
        if(i - 1 >= 0 && line.get(i - 1).getBeseda().equals("\n"))
        {
            int j = 0;
            while(j < line.get(i - 1).getDistance())
            {
                System.out.print("");
                j++;
            }
        }
        if(b.getBeseda().length() > 0)
        {
            if(b.getBeseda().charAt(0) == '"' || b.getBeseda().charAt(b.getBeseda().length() - 1) == '"')
            {
                if(!(b.getBeseda().charAt(0) == '"' && b.getBeseda().charAt(b.getBeseda().length() - 1) == '"'))
                {
                    {
                        if(isString)
                        {
                            isString = false;
                            wasString = true;
                        }
                        else
                        {
                            isString = true;
                            wasString = false;
                        }
                    }
                }
            }
            if(!b.getBeseda().equals("'\\''")&&!(b.getBeseda().charAt(0)
                == '\'') && !(b.getBeseda().charAt(b.getBeseda().length() - 1) == '\''))
            {
                if(b.getBeseda().charAt(0) == '\'' || b.getBeseda().equals(""))
                {
                    if(isChar)
                    {
                        isChar = false;
                    }
                    else
                    {
                        isChar = true;
                    }
                }
            }
        }
        if((b.getBeseda().equals(">") || b.getBeseda().equals("<")) && !isString && !isChar)
        {
            System.out.print("");
        }
        if(b.getBeseda().equals("") && !isChar)
        {
            if(i - 1 >= 0 && i + 1 < line.size())
            {
                Beseda temp1 = line.get(i - 1);
                Beseda temp2 = line.get(i + 1);
```

```
                if(!temp1.getBeseda().equals("! "))
                {
                    if(temp2.getBeseda().length() > 0 && temp2.getBeseda().charAt(0) != '.' && !isString)
                    {
                        if(temp1.getBeseda().length() > 0 && temp1.getBeseda().charAt(0) != '"' && temp2.getBeseda().length(
                             > 0 && temp2.getBeseda().charAt(temp2.getBeseda().length() - 1) != '"')
                        {
                            System.out.print(b.getBeseda());
                        }
                        else if(temp1.getBeseda().length() > 0 && temp1.getBeseda().charAt(0) == '"'
                            && temp1.getBeseda().length() > 1)
                        {
                            System.out.print(b.getBeseda());
                        }
                        else if(temp2.getBeseda().length() > 0 && temp2.getBeseda().charAt(temp2.getBeseda().length() - 1)
                            == '"' && temp2.getBeseda().length() > 1)
                        {
                            System.out.print(b.getBeseda());
                        }
                    }
                }
                else if(temp1.getBeseda().equals("! ")
                    && isString)
                {
                    System.out.print(b.getBeseda());
                }
            }
        }
        else
        {
            if(i - 1 >= 0 && line.get(i - 1).getBeseda().length() > 0)
            {
                if(line.get(i).getBeseda().equals("+") && !isString && wasString && !line.get(i - 1).getBeseda().equals(""))
                {
                    System.out.print("");
                }
            }
            if(!b.getBeseda().equals(""))
            {
                System.out.print(b.getBeseda());
            }
            else if(b.getBeseda().equals("") && line.get(i - 1).getBeseda().charAt(0)
                == '\'' && line.get(i + 1).getBeseda().charAt(0) == '\'')
            {
                System.out.print(b.getBeseda());
             }
        }
    }
    System.out.println();
}
private boolean isPreviousSpecial(int pos)
{
    Beseda b = currentLine.get(pos);
    if(b.getBeseda().equals("(") || b.getBeseda().equals("[") || b.getBeseda().equals(".")
        || b.getBeseda().equals("//") || currentLine.get(pos).getBeseda().equals("\n"))

    {
        return true;
    }
    return false;
}
private boolean isDoubleChar(Beseda b)
{
    if(b.getBeseda().equals("++")
        || b.getBeseda().equals("--"))
    {
        return true;
    }
    return false;
}
```

```
private void breakCurrentLine()
{
    getCharsFromList(currentLine);
    Collections.sort(priorityCharList);
    int i = 0;
    int lineLength = 0;
    int printedPosition = 0;
    int spaceDistance = 0;
    if(priorityCharList.size() == 1)
    {
        printCurrentLine(currentLine.subList(0, priorityCharList.get(0).getPosition()));
        currentLine.subList(0, priorityCharList.get(0).getPosition()).clear();
        currentLine.add(0, new Beseda(setSpace(currentLine.get(0).getDistance()
            + 1)));
        printCurrentLine(currentLine.subList(0, currentLine.size()));
        currentLine.clear();
    }
    else
    {
        while(i < priorityCharList.size() && !currentLine.isEmpty())
        {
            Znak z = priorityCharList.get(i);
            if(z.getPosition() - printedPosition >= breakDistance)
            {
                if(lineLength > 0)
                {
                    currentLine.add(0, new Beseda(setSpace(currentLine.get(0).getDistance()
                        + spaceDistance)));
                    int tralala = lineLength - printedPosition;
                    printCurrentLine(currentLine.subList(0, tralala));
                    currentLine.subList(0, lineLength + 1 - printedPosition).clear();
                    printedPosition = priorityCharList.get(i - 1).getPosition();
                    lineLength = 0;
                    spaceDistance++;
                }
                else
                {
                    lineLength = z.getPosition();
                }
            }
            else
            {
                if(currentLine.size() <= breakDistance)
                {
                    currentLine.add(0, new Beseda(setSpace(currentLine.get(0).getDistance()
                        + spaceDistance)));
                    printCurrentLine(currentLine.subList(0, currentLine.size()));
                    currentLine.clear();
                }
                else
                {
                    lineLength = z.getPosition();
                }
            }
            i++;
        }
        if(!currentLine.isEmpty())
        {
            currentLine.add(0, new Beseda(setSpace(currentLine.get(0).getDistance()
                + spaceDistance)));
            printCurrentLine(currentLine);
            currentLine.clear();
        }
    }
}
private void getCharsFromList(ArrayList < Beseda > list)
{
    int priority = 0;
    boolean isString = false;
    boolean isChar = false;
    priorityCharList = new ArrayList < Znak >();
    for(int i = 0; i < list.size(); i++)
    {
        Beseda b = list.get(i);
        char firstChar = 'c';
        char lastChar = 'c';
        if(b.getBeseda().length() > 0)
```

```
            {
                firstChar = b.getBeseda().charAt(0);
                lastChar = b.getBeseda().charAt(b.getBeseda().length() - 1);
            }
            if(!b.getBeseda().equals("'\\''")&&!(firstChar=='\'')&&!(lastChar=='\''))
            {
                if(firstChar == '"')
                {
                    if(!isString)
                    {
                        isString = true;
                    }
                    else
                    {
                        isString = false;
                    }
                }
                else if(firstChar == '\'')
                {
                    if(!isChar)
                    {
                        isChar = true;
                    }
                    else
                    {
                        isChar = false;
                    }
                }
                else if(b.getBeseda().equals("("))
                {
                    priority++;
                }
                else if(b.getBeseda().equals(")"))
                {
                    priority--;
                }
                else if((isArithmeticChar(b.getBeseda()) || b.getBeseda().equals("=")) && !isString && !isChar)
                {
                    priorityCharList.add(new Znak(getCharPriorityValue(b.getBeseda())
                        + priority, i, b.getBeseda()));
                }
                if(b.getBeseda().length() > 0)
                {
                    if(b.getBeseda().charAt(0) == '"' && b.getBeseda().charAt(b.getBeseda().length() - 1) == '"')
                    {
                        isString = false;
                    }
                }
            }
        }
    }
    private boolean isArithmeticChar(String znak)
    {
        if(znak.equals("*") || znak.equals("/") || znak.equals("-")
            || znak.equals("+") || znak.equals("||") || znak.equals("&&")
                || znak.equals("&") || znak.equals("|") || znak.equals("^")
                    || znak.equals("%") || znak.equals("<") || znak.equals("<=")
                        || znak.equals(">=") || znak.equals(">") || znak.equals("==") || znak.equals("!="))
        {
            return true;
        }
        return false;
    }
    private int getCharPriorityValue(String znak)
    {
        if(znak.equals("*") || znak.equals("/") || znak.equals("%"))
        {
            return 3;
        }
        else if(znak.equals("-") || znak.equals("+"))
        {
            return 2;
        }
        else if(znak.equals("||") || znak.equals("&&") || znak.equals("|")
            || znak.equals("&") || znak.equals("^")||znak.equals(" <")||znak.equals("<=")
                || znak.equals(">=") || znak.equals(">") || znak.equals("==")
                    || znak.equals("!=") || znak.equals("="))
        {
            return 1;
        }
        else
        {
            return 0;
        }
    }
}
```

6. TABELA SLIK

7. VIRI

[1] Whitesmiths style - Wikipedia, the free encyclopedia. Zadnji dostop: 8.1.2011. Dostopno na: http://en.wikipedia.org/wiki/Indent_style

[2] Horstmann style - Wikipedia, the free encyclopedia. Zadnji dostop: 8.1.2011. Dostopno na: http://en.wikipedia.org/wiki/Indent_style

[3] Banner style - Wikipedia, the free encyclopedia. Zadnji dostop: 8.1.2011. Dostopno na: http://en.wikipedia.org/wiki/Indent_style

[4] Code Conventions for the Java Programming Language - Zadnji dostop: 8.1.2011. Dostopno na: http://www.oracle.com/technetwork/java/codeconv-138413.html

[5] Java/C/C++ Source Code Formatter - Zadnji dostop: 13.12.2010. Dostopno na: http://www.jindent.com/