Универзитет „Св. Кирил и Методиј“ -Скопје
Факултет за електротехника и информациски технологии

ДИНАМИЧКА ВИЗУЕЛИЗАЦИЈА НА СОФТВЕР

-магистерски труд-

Ментор

Проф. Д-р Сузана Лошковска

Кандидат

Александра Стојанова

Скопје, 2014

СОДРЖИНА:

1. Вовед

Во современиот свет најчесто се јавува проблемот на преполнетост со информации и податоци. Со цел да се овозможи ефикасно филтрирање, поделба, пристап или разбирање на информациите и податоците, потребно е нивно соодветно претставување. Визуелизацијата на информации има потенцијал да им помогне не луѓето полесно да ги најдат и разберат податоците кои им се потребни, а тоа да се изведе на ефикасен начин [72][35].

Визуелизацијата претставува нова гранка во науката што помага за подобро претставување, разбирање и истражување на различни научни дисциплини.

Визуелизацијата игра значајна улога во претставувањето и разбирањето на различни појави и процеси, како во секојдневниот живот, така и во научните кругови, уште многу одамна. Потребата за визуелизација односно сликовито претставување за полесно разбирање на различни појави, била истакнувана од различни филозофи уште пред многу векови. Според Аристотел, 350 години пред нашата ера, дури и мислите се неможни без слика[13].

Така на пример, денес уште никој нема видено атом со голо око, но сите им е познато дека атомот изгледа како јадро околу кое кружат електроните по кружни орбити.

Денес, компјутерите претставуваат значајна алатка за создавање визуелизација и му помагаат на корисникот, подобро да ги разбере сложените феномени. Како резултат на тоа визуелизацијата станува посебна дисциплина во компјутерската наука.

Визуелизацијата не претставува само еден обичен начин на пресметување, туку преставува процес на трансформација на информациите и податоците во визуелна форма и на тој начин му се олеснува на корисникот лесно да ја набљудува и истражува информацијата.

Визуелизацијата им помага на научниците и инженерите да ги искористат визуелните карактеристики кои се скриени во податоците и на тој начин да ги истражат и анализираат скриените информации што не можат директно да се

забележат. Визуелизацијата ја игра главната улога во употребата на компјутерите во зголемувањето на човековото разбирање. Ова поле од компјутерската наука се нарекува засилување на интелегенцијата (intelligence amplification-IA), што е спротивно на вештачката интелегенција (artificial intelligence-AI), каде целта е зголемувањето на интелегенцијата на компјутерот [22].

Визуелизацијата, денес многу се користи во повеќе науки како: механиката, хемијата, физиката и медицината. За сите овие дисциплини, компјутерските научници имаат развиено различни софистицирани системи за генерирање на визуелизации за нивните потреби [14] [35] [41].

Но, од друга страна компјутерските програмери, многу малку ја користат визуелизацијата како алатка за искористување, дизајнирање и имплементирање и одржување на софтверот. Програмерите повеќе се обидуваат тие да се прилагодат на нивото што го нуди компјутерот, наместо да ги прилагодат компјутерските претстави на нивните способности и да бидат полесни за разбирање.

И покрај сложениот начин на претставување, терминологијата која се користи во компјутерската наука изобилува со метафори кои ги поврзуваат недопирливите софтверски и хардверски елементи со некои добро познати елементи од секојдневниот живот. Целта на таквите метафори е кај корисникот да се предизвикаат мисловни слики за полесно да се запаметат одредени концепти и да се искористи таквата аналогија за подобро разбирање на структурите или функциите. На пример, во компјутерската наука се користат терминте „автомат“ или „машини“ за означување на некои математички модели на пресметки. Исто така, се користат терминте „ленти“, „дрва“, „листови“, „редови“, „архиви“ и слични изрази со цел да се претстават некои компјутерски поими или податочни структури. Турнинговата машина претставува математички модел кој се состои од множества функции и релации. Аналогијата и поврзувањето на математичкиот модел со машина овозможува да се пренесуваат одредени поими од физичкиот свет во математичкиот свет и на тој начин се добива подобро разбирање на математичкиот модел [22] [72].

2. Визуелизација на софтвер

Софтверот, сам по себе претставува нешто нематеријално и невидливо, нешто што не може да се допре и да се почувствува, затоа многу тешко би било неговото истражување и креирање. Целта на визуелизацијата на софтверот не е да се произведат прецизни компјутерски слики, туку компјутерски слики кои можат да предизвикаат слики во мислите и ќе помогнат за подобро разбирање на софтверот. Затоа, наоѓањето на нови метафори за претставување на софтверот, не само што придонесува за подобра визуелизација, туку исто така, го подобрува начинот на зборување и објаснување на софтверските системи.

Постојат две поголеми дисциплини на визуелизацијата и тоа: научна визуелизација и визуелизација на информациите.

Во научната визуелизација се обработуваат физичките податоци, додека во визуелизацијата на информациите се обработуваат апстрактни податоци. Бидејќи и алгоритмите и програмите претставуваат еден вид на информација, визуелизацијата на софтвер претставува дел од визуелизацијата на информациите. Повеќе автори ја дефинираат визуелизацијата на софтвер како визуелизација на алгоритмите и програмите. Тоа претставува основна и најкратка дефиниција за визуелизација на софтвер. Меѓутоа со овој начин на дефинирање на оваа дисциплина се исклучуваат многу употреби на визуелизациските техники во компјутерската наука [53].

Визуелизацијата на софтвер може да се разбере како визуелизација на артефактите поврзани со софтверот и процесот на негов развој. Освен програмскиот код во овие артефакти припаѓаат и софтверските побарувања како и документацијата на дизајнот, промените во изворниот код, извештаите со грешки и слично. Истражувачите во софтверското инженерство ги испитуваат начините и употребата на компјутерската графика за претставување на различни елементи од софтверот, на пример, неговата статичка структура, неговото конкретно и апстрактно извршување и самиот

негов развој. Кога станува збор за визуелизација на софтвер, општо се мисли за визуелизација на неговата структура, однесување и развој [54].

Кога се зборува за визуелизација на структурата, се однесува на статистичките делови и релации од системот, односно оние делови кои можат да се пресметаат без стартување на програмата. Во овој дел припаѓаат програмскиот код, податочните структури, статичките графови на повици како и организацијата на програмата во модули.

Однесувањето на софтверот е поврзано со извршувањето на програмата со реални и апстрактни податоци. Извршувањето може да се изведе како секвенца од програмски состојби што ги содржат програмскиот код и податоците од програмата. Во зависност од програмскиот јазик, извршувањето може да се изведува на повисоко ниво на апстракција на пример, функција повикува друга функција или објект комуницира со друг објект [74].

Еволуцијата се однесува на процесот на развој на софтверскиот систем и се истакнува фактот дека програмскиот код се менува со текот на времето да се прошири неговата функционалност или да се отстранат можните грешки.

Визуелизацијата на софтверот претставува еден вид компјутерска уметност, а истовремено и наука за генерирање на визуелни претстави на различни аспекти на софтверот и неговиот процес на развој.

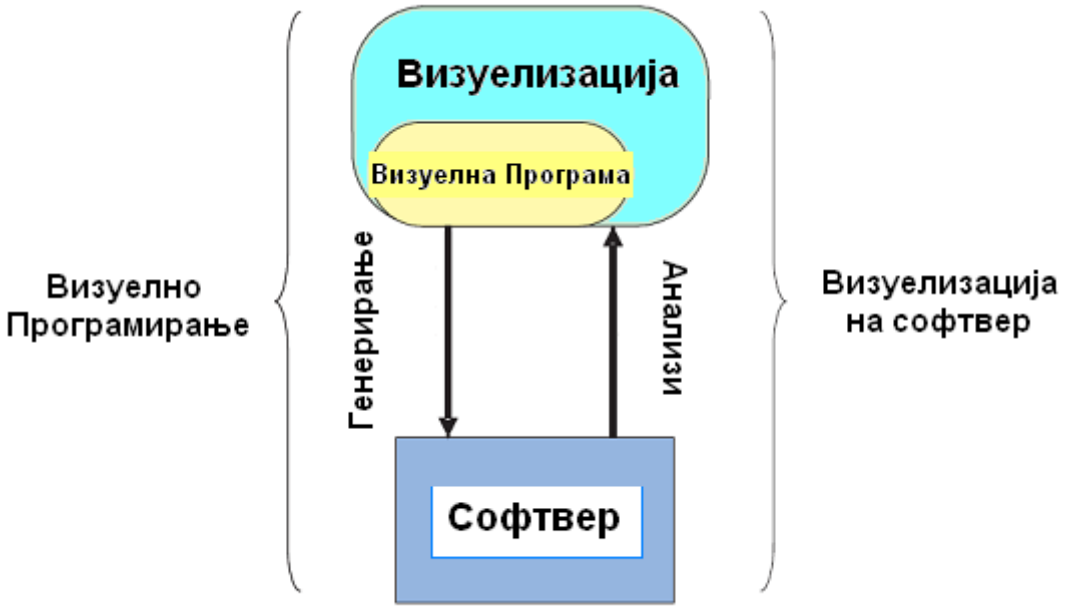
Целта на софтверската визуелизација е да помогне во разбирањето на софтверските системи и во подобрувањето на продуктивноста на процесот на развој на софтвер. Програмерите можат да ја користат визуелизацијата при развојот на програмите, при статистичката анализа на системот или при динамичкото однесување на системите. Исто така, ваквата дисциплина може да се користи и при дебагирање и полесно наоѓање на грешки и недостатоци во системите или при правењето на измени на веќе постоечки код. И секако, ваквата алатка особено може и да им помогне на почетниците во компјутерската техника и програмирањето, кои со помош на визуелизацијата можат полесно да ги разберат компјутерските програми и нивната работа, а со тоа оваа алатка особено може да има корист во едукациски цели кај студентите [72] [74].

2.1 Визуелизација на софтвер и визуелно програмирање

Во класичните програмски јазици, програмите се претставуваат како текст. Но, денес се почесто во програмските јазици е воведена визуелизацијата, односно се настојува со користење на помалку код а повеќе графички елементи да се добијат ефикасни програми. Значи, визуелните програми освен текстуални имаат и графички елементи. Значењето на програмите зависи од

просторниот распоред и поврзаноста меѓу визуелните елементи. Според различните програмски парадигми постојат различни програмски јазици, како на пример: јазици за контрола на тек, за тек на податоци, објектно-ориентирани јазици, јазици базирани на правила, јазици базирани на форми и хибридни програмски јазици. Многу визуелни програмски јазици само му дозволуваат на корисникот помалку или повеќе да го изгради визуелното синтаксичко дрво на текстуалната програма [71].

На слика 1 е прикажано дека визуелното програмирање и визуелизацијата на софтверот се комплементарни едно со друго. Визуелизацијата на софтвер генерира визуелизација од самата спецификација на софтверските системи, додека визуелното програмирање генерира софтверски системи од визуелните спецификации. Со комбинирање на двата пристапа се овозможува еден вид на повратна или кружна визуелизација. Тоа се постигнува на пример, со произведување на визуелна претстава од изворниот код на системот, со измена на визуелната претстава и со генерирање на нов систем.

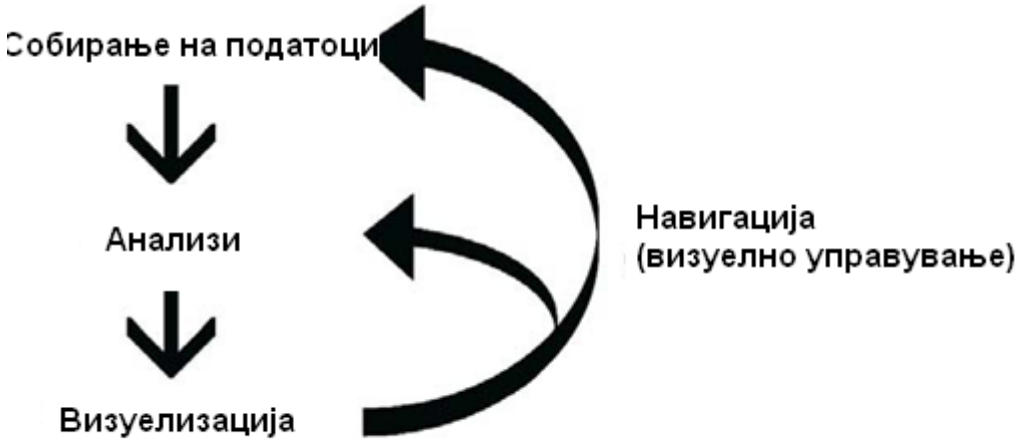


Слика 1. Визуелно програмирање наспроти визуелизација на софтвер [72].

2.2 Тек на визуелизација

Креирањето на компјутерските слики претставува последен чекор од визуелизацискиот тек. На слика 2 е претставен текот на визуелизацијата и на сликата се гледа дека во текот на визуелизацијата една фаза се користи како

влез за следната фаза. Фазите од текот на визуелизација се: собирање на податоци, анализа и визуелизација.



Слика 2. Визуелизациски тек.

Собирање на податоците: постојат неколку извори на информации за софтверскиот систем, вклучувајќи ги: изворниот код, софтверскиот дизајн, документацијата за корисникот, измените во состојбите за време на извршувањето и резултатите од тестовите. Методите што се користат за извлекување и собирање на соодветни податоци од дадените извори можат да бидат различни како и изворите.

Анализи: најчесто, количеството на информации е премногу големо за одеднаш да бидат претставени пред корисникот. Можат да се користат различни видови анализи со цел да се редуцира количеството на податоци и да се прикажат само најзначајните. Таквите анализи можат да бидат филтрирање на податоците, статички програмски анализи или статистички методи.

Визуелизација: резултантните податоци се мапираат во визуелен модел, кој информациите ги трансформира во геометриски или графички информации, а потоа ги рендерира на екранот или некој друг медиум како единечна слика или како секвенца од слики.

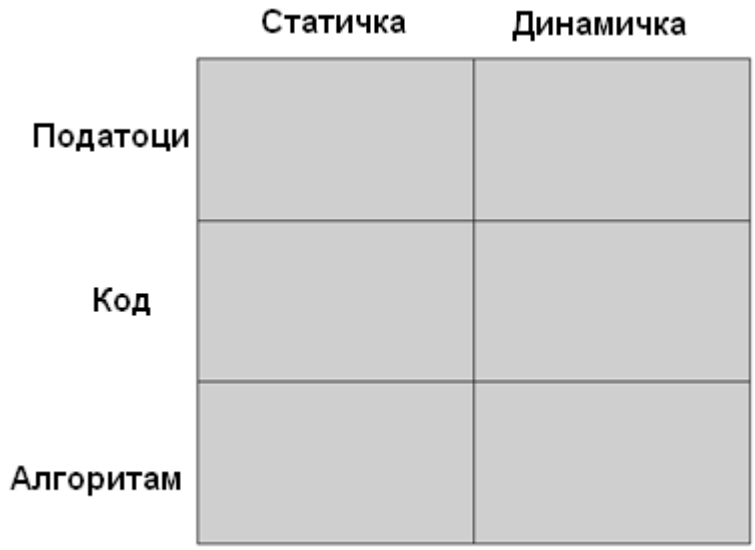
Кај интерактивната визуелизација, корисникот може да ги контролира претходните чекори од текот на визуелизацијата. Ваквиот метод на интеракција се нарекува пресметливо или визуелно управување.

2.3 Визуелизација на софтвер и реверзно инженерство

Реверзното инженерство претставува процес на анализирање на даден систем со цел да се определат компонентите на системот и нивните меѓусебни врски и да се креира претстава за системот во друга форма, односно во форма на повисоко ниво на апстаркција [23]. Истражувањата во реверзното инженерство се фокусираат на екстракција, подредување, претставување и пребарување на информациите, притоа се прави обид да се намали количеството на непотребни информации за некоја определена задача, да се анализираат добиените податоци и да добијат корисни апстракции за системот што се анализира. Бидејќи реверзното инженерство претставува интерактивен процес, кој постојано се надополнува и во кој резултатите од автоматските анализи треба да бидат претставени на корисникот и да можат да се валидираат и да се врати одговор за анализите, софтверското инженерство може да игра значајна улога во неговата реализација. Со помош на визуелизацијата на софтвер, корисникот може лесно да ги разбере податоците од анализите на реверзното инженерство.

2.4 Видови визуелизација на софтвер

Визуелизацијата на софтвер, а со тоа и алатките што се користат за добивање на визуелизацијата, може да се поделат според различни критериуми. Една поделба е направена од страна на Мајерс [11] [72]. Според него визуелизацијата на софтвер може да се подели на 6 различни области во матрица со димензии (2X3) (слика 3). Тој ја прави класификацијата користејќи две оски. На едната оска е нивото на апстракција со што, визуелизацијата на софтвер може да се подели на визуелизација на податоци, код и алгоритми. Визуелизацијата на алгоритми ги претставува алгоритмите на повисоко ниво на апстракција отколку што е претставен програмскиот код. На другата оска е нивото на анимација и според тоа, визуелизацијата на софтвер се дели на статичка и динамичка визуелизација. Статичката визуелизација вклучува само слика, но, не и анимација, додека динамичката е проследена со анимации.



Слика 3. Поделба на визуелизацијата на софтвер според Мајерс [72].

Stasko и Patterson [45] претствуваат друга поделба според која има четири оски на класификација и тоа: аспект, апстракција, анимација и автоматизација. Најсложената поделба ја претставуваат Price, Small и Baecker [12] и според нив постојат 47 различни правци на класификација, кои се групираат во шест супер-категории: опсег, содржина, форма, методи, интеракција и ефективност.

Постојат и други поделби на визуелизацијата на софтвер и на алатките што ја овозможуваат визуелизацијата, според различни атрибути [72].

- Според природата на артефактите што се визуелизраат. Најчесто тоа е кодот но, може да биде и документацијата, спецификацијата, архитектурниот дизајн, алгоритмите, податочните структури и сл.

- Според природата на визуелизацијата, таа се дели на статичка и динамичка. Една од најзначајните карактеристики на визуелизацијата и визуелизациските системи е природата на визуелизацијата.

- Според означувањата кои се користат, визуелизацијата може да биде текстуална или графичка (2D или 3D).

- Според гледната точка визуелизацијата може да се разгледува од гледна точка на функционалноста, на однесувањето, на структурата и на податоците. Од функционалната гледна точка се овозможува поглед на системот во целост или на одредени задачи. Од гледната точка на однесувањето се опишуваат врските меѓу настаните и одговорите на системот за време на извршувањето. Од гледна точка на податоците посебно внимание се посветува на објектите во системот и релациите меѓу

нив. И од гледна точка на структурата се опишуваат статичките врски меѓу објектите.

И покрај многуте класификации и поделби на визуелизацијата на софтвер, сепак основната поделба на визуелизацијата на софтвер е на статичка и динамичка визуелизација. Токму затоа, во овој труд се задржуваме на оваа поделба која може да биде според природата на визуелизација, според нивото на анимација и секако според видот на информациите што е потребно да се визуелизираат.

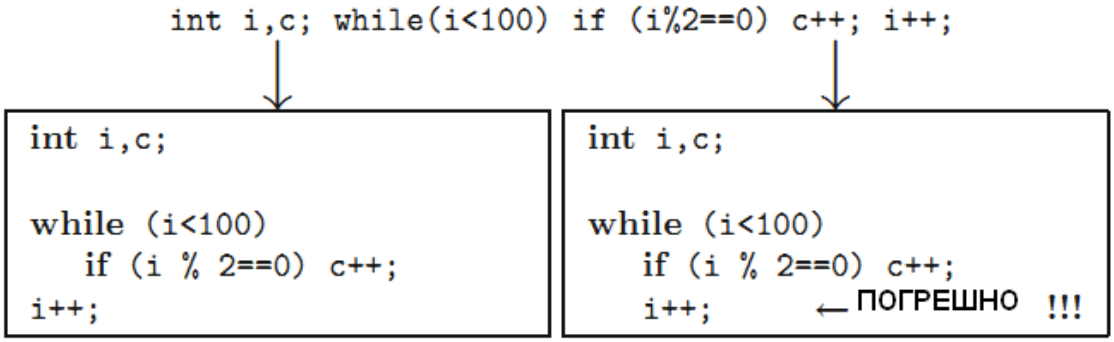
Посебен акцент е даден на динамичката визуелизација на софтвер и алатките кои овозможуваат ваква анимирана визуелизација.

3. Статичка визуелизација на софтвер

Статичката визуелизација е визуелизација во која софтверот се претставува во форма на слики без да се користи анимација. Ваквата визуелизација може да се користи на пример, за претставување на структурата на даден софтверски систем, за претставување на класната хиерархија на некој објектно-ориентиран систем, повикувачки граф за некој модул или граф за контрола на тек на некоја процедура.

3.1 Текстуална статичка визуелизација

И текстот, односно знаците со кои се пишува кодот, претставуваат еден вид статичка визуелизација. За полесно да може да се разбере кои команди на кој дел од програмата припаѓаат се користи таканаречено убаво пишување, односно „pretty printing“. Со овој начин на пишување внатрешните структури се пишуваат повнатре. Еден пример за точно и неточно преставување со pretty printing е прикажан на слика 4.



Слика 4. Пример за правилно и неправилно претставување на код со „pretty printing".

Во вториот случај инкрементирањето на i, изгледа како да е дел од if условот а не надвор од него и затоа многу лесно може лесно да се погреши.

Друг начин за статичка текстуална визуелизација е нагласувањето на даден дел од кодот. На пример, со потенцирање на некој дел од кодот можат да се издвојат клучните зборови (Слика 5).

```
private Node buildPostIncExpr(Node n) {
    String expr = treeIO.generateString(n);
    if (rho.containsNewVar(expr)) {
        System.out.println(expr);
        expr = rho.getNewVar(expr);
        System.out.println(expr);
    }
    String retStr = new String(expr + ".postInc()");

    return treeIO.generateSubTree("PrimaryExpression", retStr);
}
```

Слика 5. Пример за нагласување на дел од кодот како начин на статичка текстуална визуелизација

3.2 Статичка визуелизација со дијаграми

Друг пример за статичка визуелизација на софтвер претставуваат дијаграмите. Тие уште од почетоците на компјутерската наука се користат за прикажување на структурата на програмите. Кај дијаграмите релациите, односно врските меѓу одредени делови од програмата можат визуелно да се кодираат според позицијата, поврзувањето, соседството и содржината.

Според позицијата, на пример, почетниот јазел на еден автомат со конечна состојба се става налево, додека почетниот јазел кај дијаграмите за контрола на тек се става горе, а последниот јазел или десно или долу соодветно.

Според поврзувањето, врските во повикувачките графови покажуваат која функција повикува друга функција, а врските во графот за контрола на тек ги поврзуваат последователните акции.

Според соседството, кај дијграмите за контрола на тек, алтернативните акции се сместуваат една до друга.

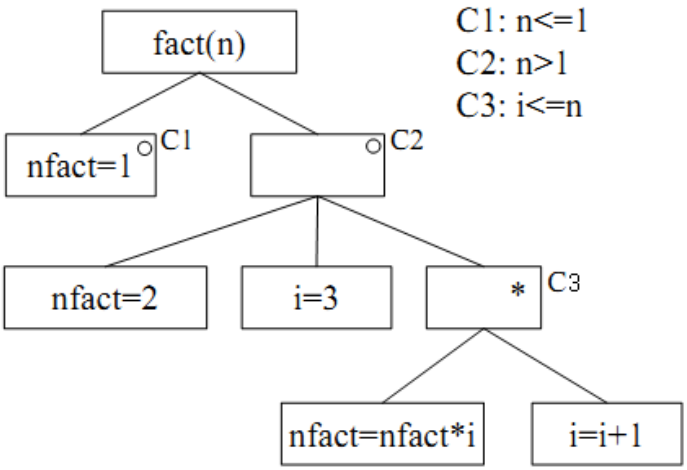
Според содржината, на пример, кај дијаграмите Nassi–Shneiderman, сложено дејство се претставува со правоаголник кој во него содржи нови правоаголници на неговите подредени дејствија.

Дијаграмите што се користат за преставување на структурата на програмата се: Џексоновите дијаграми [60], дијаграмите или графовите за контрола на тек , Наси-Шнајдерман дијграмите [30] и UML дијаграмите [81].

3.2.1 Дијаграми на Џексон

Според структурната програмска методологија на Џексон, податочните структури вклучени во програмата прво треба хиерархиски да се разложат со помош на ваквите дијаграми. Основните елементи на Џекосновите дијаграми се акциите односно дејствијата и тие можат понатаму да се разложат на подредени акции. Бидејќи и во понатамошноти излагање ќе биде спомната функцијата за наоѓање на факториел на даден број, како пример што може полесно да се престави со рекурзивна функција, и овде ќе биде претставена таа функција. Меѓутоа, во ова поглавје, функцијата ќе биде опишана нерекурзивно бидејќи ваквиот вид на статичката визуелизација не може многу да помогне во визуелизирањето и разбирањето на работата на рекурзивните повици. Во овој случај, функцијата за факториел е визуелизирана со помош на Џексоновите дијаграми (слика 6).

```
int fact(n) {
  if (n>1)
   { nfact=2;
     for(int i=3;i<=n;i++)
       nfact=nfact*i;
   }
  else
   { nfact=1;
   }
  return nfact;
}
```

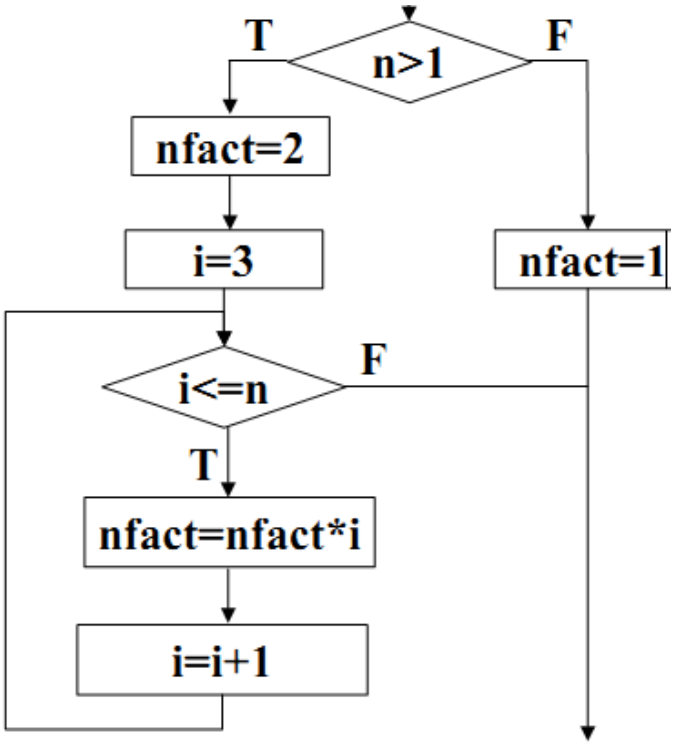


Слика .6 Пример за визуелизација со помош на Џекоснов Дијаграм.

3.2.2 Графови за контрола на тек

Графовите за контрола на тек претставуваат друг вид на дијаграмско претставување на структурата на програмата односно вид на статичка визуелизација. Кај ваквите дијаграми правоаголните јазли означуваат настани, активности, процеси, функции или изрази, а сликите во форма на ромб претставуваат услови односно разгранета структура и тие можат да имаат повеќе излези. Врските меѓу јазлите се во вид на стрелки и го следат текот на контролата. И овде, како пример за претставување на структурата на програмата со помош на овие дијаграми, повторно ќе биде искористена функцијата за наоѓање на факториел на даден број. (Слика 7)

```
int fact(n) { if (n>1)
   { nfact=2;
     int i=3;
     while(i<=n)
      { nfact=nfact*i;
        i=i+1;
      }
   }
  else
   { nfact=1;
   }
  return nfact;
}
```

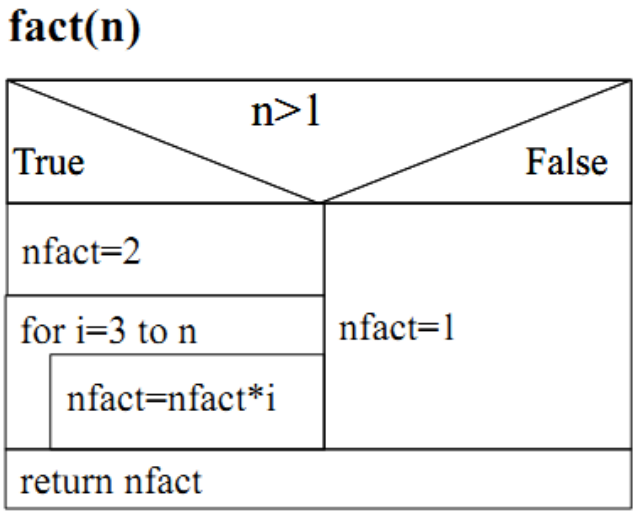


Слика 7. Пример за визуелизација со помош на граф за контрола на тек.

3.2.3 Наси-Шнајдерман дијграми

Следен вид на статичка визуелизација на софтверот со помош на дијаграми претставуваат Наси-Шнајдерман дијграмите кои уште се познати како структограми. Пример за визуелизација со помош на ваквите дијаграми е даден на Слика 8. И овде станува збор за визуелизирање на истата функција за наоѓање на факториел, но, изгледот на визуелизацијата е различен. Овде во една поголема структура која е сложена акција се вгнезуваат подредените акции во помали структури, односно правоаголници.

```
int fact(n) { if (n>1)
   { nfact=2;
     for(int i=3;i<=n;i++)
       nfact=nfact*i;
   }
  else
   { nfact=1;
   }
  return nfact;
}
```



Слика 8. Пример за статичка визуелизација со помош на структограми.

3.1.4. UML дијграми

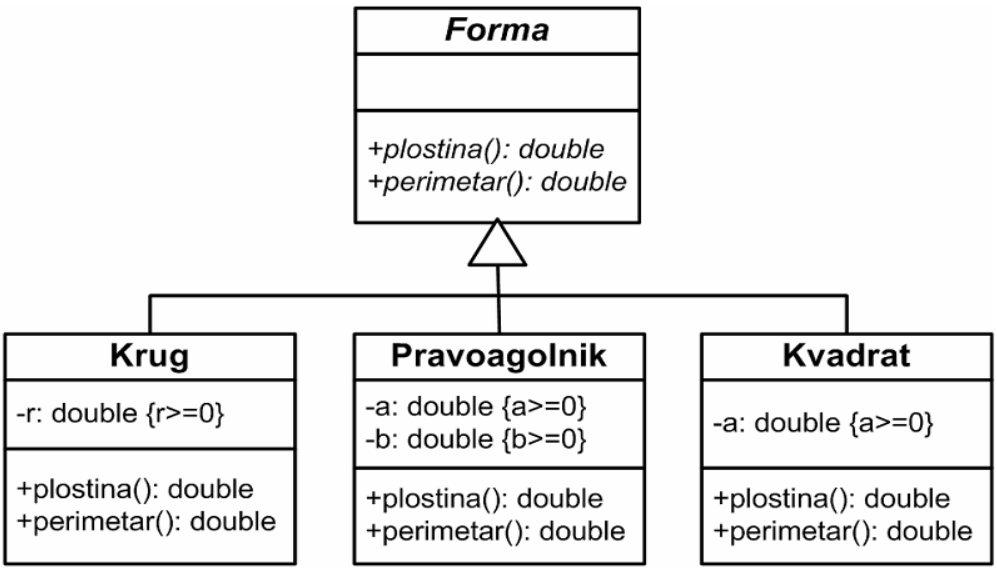
Последниот вид на дијаграми што ќе бидат опишани, а се користат за статичка визуелизација на програмите, се UML (Unified Modeling Language) дијаграмите. Тие се користат за објектно-ориентирано моделирање и нудат повисоко ниво на апстракција.

UML нотацијата вклучува повеќе дијаграми преку кои се гради моделот на системот што треба да се имплементира.

Дијаграмот кај UML претставува графичка презентација составена од множество на елементи, и претставува само еден поглед на еден дел од моделот. Дијаграмот не ја носи семантиката на системот, туку претставува приказ на дел од елементите од моделот.

UML овозможува поголем број на различни видови дијаграми, меѓу кои спаѓаат: use-case дијаграмите, класните и објектните дијаграми, дијаграмите на однесување (дијаграмите на состојби и дијаграмите на активности), интеракциските дијаграми (секвентните и колаборациските дијаграми), имплементациските дијаграми (компонентните и deployment дијаграмите) и дијаграмите за управување со модели (пакетните, дијграмите на подсистеми и дијаграмите на модели). Секој од овие видови дијаграми се користи за различни цели од моделирањето на системите и користи различни симболи со кое ги претставува состојбите и активностите. Со помош на сите овие UML дијаграми можат во целост да се моделираат сите аспекти од објектно-ориентираните системи.

На слика 9 е претставен пример за класен дијграм за претставување на хиерархиска структура.



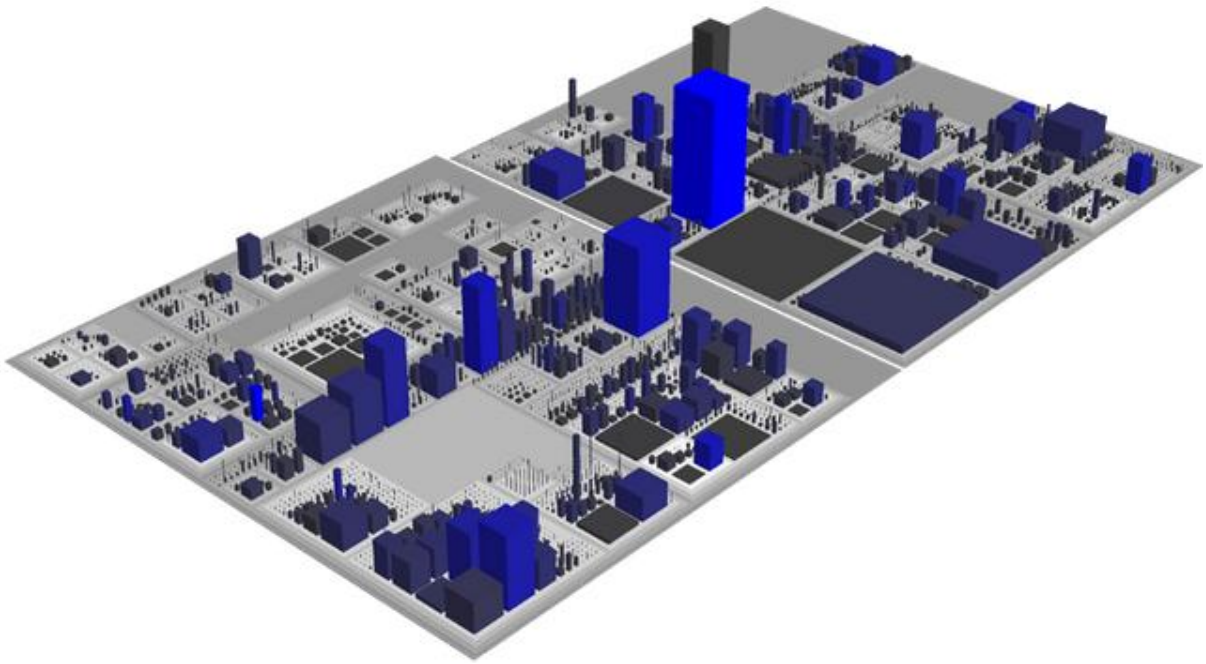
Слика 9. Пример за статичка визуелизација со UML класен дијаграм.

Многу програми денес овозможуваат автоматско генерирање на различни видови на UML дијаграми за дадена програма, или реверзното, од направените UML дијаграми можат да се добие кодот на програмата.

3.3 Статичка визуелизација на софтверска архитектура

Освен визуелизирање на кодот може да се визуелизира и архитектурата на програма со цел истата полесно да се разбере. При визуелизирањето особено е битно каква метафора ќе се избере при мапирањето. Па така, освен со текст и со дијаграми, статичката визуелизација може да се изведе и на други начини со користење на посебни метафори и мапирања на софтврски елементи со елементи од секојдневниот живот. Може да дојде до подобрување на визуелизацијата и со користење на 3 димензии наместо 2, кои му се поблиски за разбирање на човекот како корисник.

Така на пример, за визуелизирање на архитектурата на некоја Java програма се користи метафора на град [66]. Таква програма е Code City. На слика 10 е прикажана визуелизација на архитектура на Java Development Kit (JDK) v1.5 со помош на оваа програма.



Слика 10. Статичка 3D визуелизација на софтверска архитектура на Java Development Kit (JDK) v1.5 со помош на Code City [66].

3.4 Статичка програмска анализа

Статичката програмска анализа може да се користи и за динамичката визуелизација на софтвер, затоа овде одвојуваме и дел за неа.

Статичката програмска анализа има за цел да ги определи својствата на однесување на програмата пред таа да се изврши. Ваквата анализа особено се применува при апстрактното интерпретирање за докажување на точноста од анализите во согласност со семантиката на програмскиот јазик. Статичката анализа може да се подели во две фази:

1. Во првата фаза програмата се преведува во систем со равенства или ограничувања според делумен редослед на програмските својства. Резултатите од системот ја даваат точната информација за опредлено својство кое е цел на анализа. Делумниот редослед влијае на релативната прецизност на резултатите.

2. Во втората фаза се пресметува соодветното решение. Оваа фаза може да се потпира на множеството општи итеративни работни алгоритми или врз еден или повеќе симболични техники за разрешување кои се специфични на областа од интерес.

Првата фаза зависи од програмскиот јазик во кој е пишувана програмата што се анлизира и од видот на својството кое се анализира. Втората фаза не е многу зависна од анализата која се прави и може да зависи од некои општи решенија кои можат да послужат како индиректна помош во некои анализи.

Значи, статичката анализа пресметува својства на програмата кои се задржуваат при сите извршувања на програмата [24]. Меѓутоа не секое својство може да се пресмета, бидејќи не постои некоја општа програма која зема друга програма како влез и одлучува дали оваа програма ќе заврши по конечен број на чекори или би траела бесконечно. Тие својства кои не можат да се пресметаат пред извршувањето на програмата се нарекуваат „динамички“ [72]. Бројот на тоа колку пати ќе се изврши некоја точка од програмата претставува динамичко својство, додека фактот дека пограмската точка нема да се изврши за ниеден можен влез преставува статичко својство.

Анализата со контрола на тек го пресметува графот на контрола на тек на програмата. Едноставен програмски јазик би имал и едноставен граф на контрола на тек. Но, ако се земе вистински програмски јазик, тогаш кај нив секоја процедура има свој сопствен граф на контрола на тек и со тоа проблемот е посложен. Со помош на повиците кон процедурите кои можат да се јават во телото на дадена процедура, графовите меѓусебно се поврзуваат. Затоа можат да се разликуваат интрапроцедурални и интерпроцедурални графови на контрола на тек [72].

Многу современи програмски јазици содржат програмски покажувачи. Покажувачите се основната карактеристика на функционалните јазици од повисок ред, но, исто така постојат и во програмскиот јазик C. Проблемот е што вредностите на таквите функциски покажувачи се пресметуваат за време на извршувањето и можат да покажуваат кон сите функции од програмата или барем кон сите функции од исти вид. Како последница на тоа, при формирањето на графот на контрола на тек потребно е да се нацртаат врските од програмата која повикува функција преку функцискиот покажувач до сите функции кои се повикани. Сличен проблем настанува и кај објектно – ориентираните јазици како што е и Java. Во тој случај не постојат директни функциски покажувачи, туку референци кон објекти кои содржат функции, односно методи. Кој метод ќе се повика, зависи од извршувачкиот вид на објектот кон кој се рефренцира. Поради ваквото динамичко распределување на методите, интерпроцедуралниот граф на контрола на тек содржи врски кон сите такви методи кои можат да бидат повикани на основа на статичкиот вид на рефернцата. Со пресметувања на подобри приближувања кон извршувачкиот вид на референцата [21], може значително да се намали бројот на можни повици кон методите.

3.5 Анализа на тек на податоци

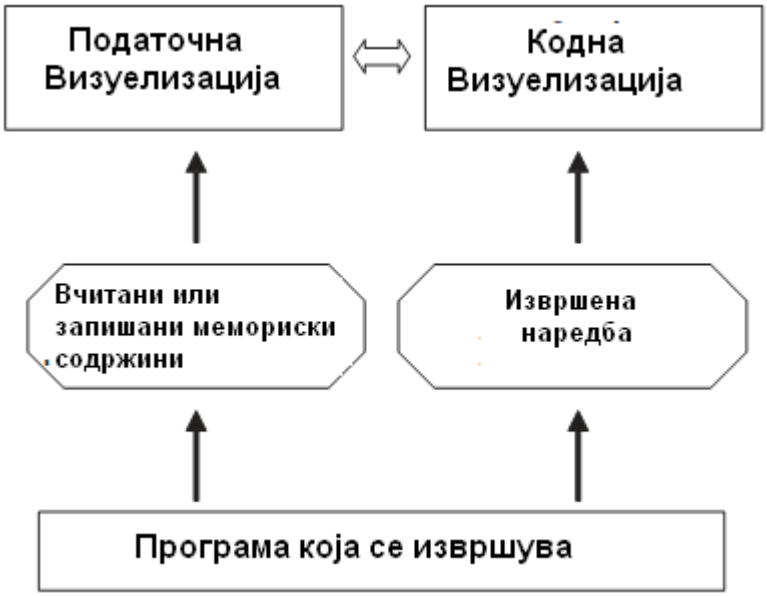
Анализата на тек на податоци пресметува информации за секој програмски покажувач, во врска со податоците до кој ќе достигне покажувачот за време на извршувањето. Анализата на текот на податоци овозможува значајни информации и повеќе се користи за оптимизација на компајлерите [67]. Во основа, анлизите за текот на податоци работат преку пропагирање низ локално достапните информации низ патеките во графот за контрола на тек. Можат да се разликуваат два вида проблеми на текот, врз основа на насоката во која информациите се пренесуваат низ врските [72], и тоа:

-проблеми на текот во насока напред, кои анализираат што може да се случи пред контролата да стигне до дадената програмска точка. На пример, доаѓање до некои дефинирања или некои достапни изрази.

-проблеми на текот во насока назад, кои анализираат што може да се случи откако контролата ја напушта дадената програмска точка.

4. Динамичка визуелизација на софтвер

Динамичката визуелизација на програмите го покажува однесувањето на програмата за даден влез, односно кои инструкции се извршуваат и како се менува програмската состојба (слика 11).



Слика 11. Динамичка програмска визуелизација

Во идеален случај податочната визуелизација и визуелизацијата на кодот можат да се комбинираат, така што корисникот гледа како извршувањето на дадена инструкција ја изменува меморијата.

Бидејќи динамичката визуелизација на софтверот нуди подобар преглед на работата на програмата поради вклученоста на анимации, а со тоа придонесува и до подобро разбирање на програмите, јас се одлучив поголемо внимание да посветам токму на ваквиот вид на визуелизација.

Динамичката визуелизација може да помогне особено при дебагирањето на програмите и во едукациски цели за изучување на некои алгоритми, и некои карактеристични програми, особено на пример, оние што содржат рекурзивни повици. Таквите програми ако се претставени само со код многу малку би

можеле да ги разберат почетниците, а со тоа и уште потешко би можеле да ги применуваат или пишуваат.

Динамичката визуелизација на програмите се состои од повеќе подобласти како:

- динамичката архитектурна визуелизација,

- анимација на алгоритми и

- визуелно дебагирање и тестирање.

Поделбата може да се направи главно според целите за кои се користи анимацијата. Визуелното дебагирање и тестирањето има за цел да им помогне на програмерите полесно да ги забележат грешките во програмите и да ги исправат. Анимацијата на алгоритми е земена како пример за програмската визуелизација која има главна цел да помогне во разбирањето и изучувањето на програмите или делови од програмите, како што се алгоритмите. Програмската визуелизација најмногу се користи во едукацијата. Динамичката архитектурна визуелизација има за цел подобрување на статичката визуелизација со воведување определен број на динамички елементи во неа со цел постигнување на подобра архитектурна слика која се менува со текот на времето.

4.1 Динамичко собирање на податоци

Постојат различни начини за собирање на податоци за време на извршувањето на програмата. Динамичкото собирање на податоци вообичаено го забавува извршувањето на програмите и дури може да го измени нејзиното извршување. Затоа, изборот на методот за собирање на податоци зависи од видот на податоците што се потребни и од нивната големина и опфатеност.

Најчесто, програмата се изменува, односно во програмскиот код се додава дополнителен мониторинг код. Таквата постапка се нарекува code instrumentation. Инструментализацијата може да се направи или на ниво на изворниот код или на ниво на машинскиот код. Вметнатиот код може да се додаде пред или по секоја инструкција во оригиналниот код, на крајот или на почетокот на секоја јамка или секоја интерација во јамката, при секој повик на методот, при влегувањето или излегувањето од некој метод или при специјално определени точки во програмата од страна на корисникот. Во едноставни случаи, вметнувањето на ваквиот код може да се прави рачно, но кај реални програми тоа треба да се прави автоматски.

Да се определи кога и како се пристапува до податоците, податочните структури можат да се заменат со нови кои ќе ги ловат промените или повиците на надворешни рутини (демони) кога ќе се пристапи до податочната структура.

Наместо менување на програмата може да се додаде паралелна нитка или процес која ги бара промените во програмската меморија. За дистрибуирани програми, пораките можат да се зачувуваат. Исто така, однесувањето може да се набљудува со негово извршување на виртуелна машина или интерпретер.

Серизоен проблем, особено за вградливите системи или дистрибуираните програми, е тоа што инструментизацијата како и некои од другите пристапи за собирање на податоци го менуваат времето на извршување.

За визуелизацијата на код, потребна е позицијата во програмата. Тоа може да биде точната линија на код, ако се користи интерпретер, или вредноста на програмскиот бројач или адресата на повиканиот метод, ако се собираат податоците на машинско ниво. За компајлираните програми, често претставува проблем, мапирањето на адресите од компајлираната програма во линии на изворен код. Некои компајлери дозволуваат во компајлираната програма, да се чуваат дополнителни информации за дебагирање, но ако компајлерот прави оптимизации како делење на код за расположливите изрази, односно една инструкција во компајлираниот код всушност може да одговара на неколку различни делови од изворниот код.

За податочната визуелизација, вообичаено се зачувуваат податоците за програмските променливи и параметрите од методите, или кај пониско ниво, вредностите на регистрите.

Ако е потребно, за определени цели, можно е делови па дури и целата содржина од програмата да се зачува за визуелизацијата. Кај дистрибуираните програми, не само пораките, туку исто така, потребно е да се зачувува локалното време на праќање и примање кај различни компјутери, со цел да може да се реконструира причинскиот редослед на пораките.

4.1.1 Динамичко собирање на податоци во Java

Бидејќи денешните понови алатки за динамичка визуелизација на софтвер се почесто пишувани во програмскиот јазик Java, на кратко се задржуваме на динамичко собирање на податоци во Java.

Постојат различни начини за собирање на информации при извршувањето на програми во Java.

Вклучување на повици кон методи. Ако е достапен изворниот код, можат да се додадат инструкции како дополнителен код за означување на објектот што се следи кога се извршува инструкцијата. На пример, повиците кон

методите можат да се изменат со цел да се зачува кога методот е повикан и кога завршува неговото извршување.

```
trace.beforeMethodCall("withdraw(int amount)");
account.withdraw(100);
trace.afterMethodCall("withdraw(int amount)");
```

Ваквиот пристап е многу флексибилен, бидејќи можат да се вклучат различни видови инструкции и не морат да се вклучи секое појавување на инструкцијата. Изменувањето и додавањето на делови (code instrumentation) во изворниот код може да се автоматизира со користење на аспект-ориентирани програмски алатки како AspectJ [70] или InjectJ [78].

Додавање на помошни делови код во телото на методот (Instrumenting Method Bodies). Со цел да се следат сите повици на некој даден метод, со претходниот пример ќе има потреба да се изменат сите повици на методот, со што би се добило огромно зголемување на кодот и тоа е можно само во случај ако на располагање е целиот изворен код од сите класи што содржат повици кон методот. Наместо, менување на повиците кон методите, исто така може да се менува телото на методот со додавање на код на почетокот на телото на методот и пред секој return израз во телото на методот:

```
void withdraw(int amount)
{   trace.startOfMethod("withdraw(int amount)");
    balance=balance-amount;
    trace.beforeReturnFromMethod("withdraw(int amount)");
    return; }
```

Method Stubs. Бидејќи телото на методот може да биде многу сложено или изворниот код можеби не е достапен за правење на трансформации, можат да се креираат method stubs кои го препраќаат повикот на методот кон оригиналниот метод, а сите повици кон оригиналниот метод во достапниот изворен код треба да се заменат со повици до method stub.

```
{ trace.startOfMethod("withdraw(int amount)");
    withdraw(amount)
    trace.beforeReturnFromMethod("withdraw(int amount)");
    return; }
```

Вметнување на Byte код. Наместо трансформирање на изворниот код, додавањето на дополнителен код можат да се прави на Byte код со користење на Java APIs, како JikesBT [76] или BCEL [52]. Ваквиот пристап е корисен во случаи кога изворниот код не е достапен, што особено е случај со користење на библиотеки од трети лица.

Проширени JVM. Постојат различни проширувања на Java виртуелната машина, која дозволува пристап до информации од текот на извршувањето,

без менување на апликацискатa програма. Кај Java 1.5, бил претстатвен Java Virtual Machine Tool Interface (JVMTI) за испитување на состојбата и контрола на извршувањето на апликацијата во текот на самото извршување. На пример, се дозволува клиентската програма да се регистрира со JVM да прима настани кога JVM се иницијализира или исклучува, објектите се алоцирани или ослободени, методите се повикани или завршуваат со повикувањето, нитките се стартуваат или завршуваат, класите се полнат или се празнат, собирањето на ѓубрето започнува или завршува, надгледувачите на синхронизираните методи или објекти се внесуваат или исклучуваат или нитката чека да влезе во монитор (надгледувачот).

4.2 Визуелизација на динамики

Главните пристапи за визуелизација на промените на информацијата со текот на времето се:

- акумулација,
- просторна проекција и
- анимација.

Овие пристапи можат да се применат и на динамичката податочна визуелизација и на динамичката кодна визуелизација.

Информациите можат да се собираат во текот на времето, на пример, податочната динамичка визуелизација може едноставно да ја прикаже средната вредност на променливите, или динамичката кодна визуелизација може да го прикаже бројот на извршување на една инструкција.

Намалувањето на промените на една вредност е лош избор, бидејќи се губат повеќе од информациите, но е многу корисно ако е потребно да се даде преглед на промените на многу елементи.

Ако вниманието се задржи на неколку елементи, нивните вредности можат да се претстават на временските оски, односно се користи просторна претстава на привремените информации. Во резултантниот дијаграм, можат да се гледаат тековите и фазите во даден временски период.

Со помош на анимирана компјутерска графика, наместо мапирање на времето по оските во 2D и 3D се овозможува претставување на слики во секој дел од времето. Анимацијата претставува секвенца од слики кои се претставуваат една по друга. Секоја слика ја претставува програмската состојба во дадена точка од извршувањето. Еден проблем со анимациите е тоа

што во секој момент се гледа една слика и треба да се користи меморијата на корисникот за претходната состојба. За подобро испитување и за користење во учењето, сликите од анимациите можат да се претставуваат една до друга за подобро да се воочат разликите при премин од една во друга состојба.

Пример:

Следната програма прави едноставна симулација што ја пресметува вредноста на секоја ќелија на матрицата со помош на просечната вредност од ќелијата и вредностите на нејзините соседни ќелии:

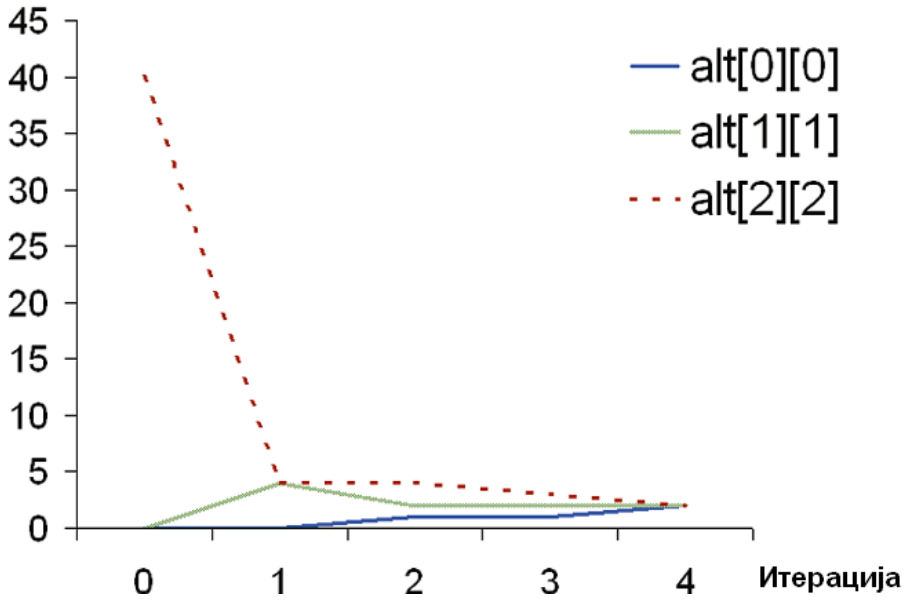
```
int m[][]= new int[5][5], m2[][] = new int[5][5];
int old[][]=m, new[][]=m2;
m[2][2]=40; // put a high value in the middle of the matrix
// and all other cells have the initial value 0
for (int t=0;t<4; t++)
    { for(int i=0; i<m.length; i++)
          { for(int j=0; j<m[i].length; j++)
                { int s=old[i][j];
                    int neighbors=1;
                              if (i>0) { s=s+old[i-1][j]; neighbors++;}
                              if (i<old.length-1) { s=s+old[i+1][j]; neighbors++;}
                              if (j>0) { s=s+old[i][j-1]; neighbors++;}
                              if (j<old[i].length-1)
                                 { s=s+old[i][j+1]; neighbors++;}
                              if ((i>0)&&(j>0)) { s=s+old[i-1][j-1]; neighbors++;}
                              if ((i>0)&&(j<old[i].length-1))
                                 { s=s+old[i-1][j+1]; neighbors++; }
                              if ((i<old.length-1)&&(j>0))
                                 { s=s+old[i+1][j-1]; neighbors++;}
                              if ((i<old.length-1)&&(j<old[i].length-1))
                                  { s=s+old[i+1][j+1]; neighbors++;}
                              new[i][j]=(int) Math.round(s/(double)neighbors);
                            }
           }
         }
```

Кодната визуелизација што користи акумулација може едноставно да укаже на тоа дека телата на првите 4 if изрази се извршуваат 80 пати, а тие на останатите 4 изрази се извршуваат само 64 пати. Во Табела 1 е прикажана податочната визуелизација: вредностите на матрицита се прикажуваат по секоја итерација. Последниот влез го прикажува резултатот од акумулирање на сите вредности од матрицата.

Табела 1. Вредности на матрицата за време на извршувањето

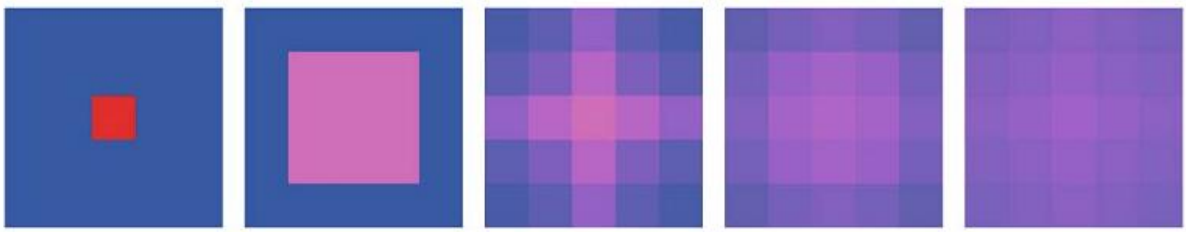
Почетна вредност	Прва итерација	Втора итерација	Трета итерација	Четврта итерација	Просек
0 0 0 0 0	0 0 0 0 0	1 1 2 1 1	1 2 2 2 1	2 2 2 2 2	1 1 1 1 1
0 0 0 0 0	0 4 4 4 0	1 2 3 2 1	2 2 2 2 2	2 2 2 2 2	1 2 2 2 1
0 0 40 0 0	0 4 4 4 0	2 3 4 3 2	2 2 3 2 2	2 2 2 2 2	1 2 11 2 1
0 0 0 0 0	0 4 4 4 0	1 2 3 2 1	2 2 2 2 2	2 2 2 2 2	1 2 2 2 1
0 0 0 0 0	0 0 0 0 0	1 1 2 1 1	1 2 2 2 1	2 2 2 2 2	1 1 1 1 1

На слика 12 вредностите на трите елементи од матрицата се претставени во временските оски. За средишниот елемент a[2][2], вредноста значително се намалува по првата итерација, а за најнадворешниот елемент a[0][0], вредноста за малку се зголемува. Вредностите на сите три елементи се спојуваат до 2.



Слика 12.Вредноста на три матрични вредности во временска оска [72]

Слика 13 ги прикажува секвенците слики во вид на анимација. Анимацијата е произведена со користење на броеви со подвижна запирка, наместо целобројни вредности со цел да се добие поголема прецизност и кодирање со бои за претставување на различните вредности на матричните елементи.



Слика 13. Секвентни слики во вид на анимација во текот на времето [72].

4.3 Динамичка архитектурна визуелизација

Вистинското однесување на работењето на софтверскиот систем може да се визуелизира на ниво на архитектура најмалку на два начина и тоа:

- со подобрување на статичките дијаграми и
- со добивање на дијаграми на однесување.

Архитектурните дијаграми што се користат за дизајн на софтверот, можат да се подобрат со помош на информации кои се добиваат за време на самото извршување на програмите, а исто така врз основа на овие информации можат да се добиваат дијаграмите на однесување. На овој начин со додавање на статичките анализи се подобрува архитектурната визуелизација.

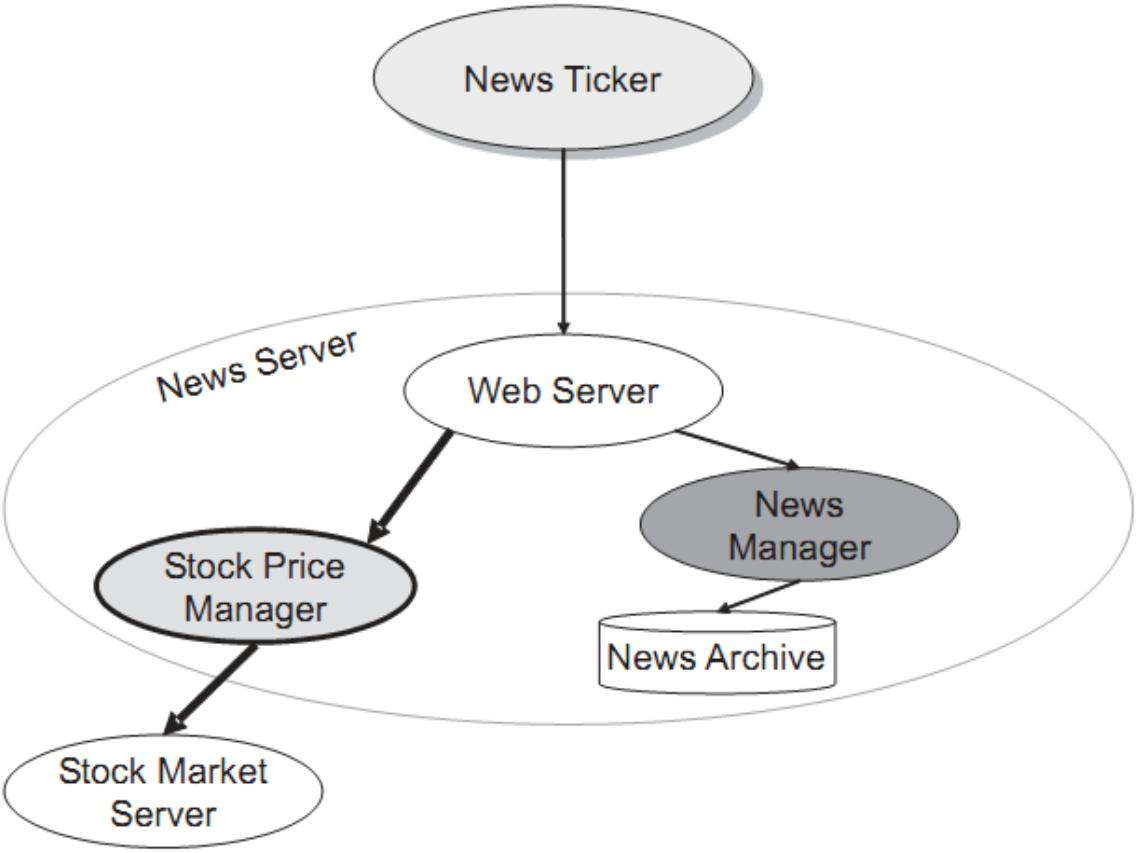
4.3.1 Подобрување на статичките дијаграми

Како пример за подобрување на статичките дијаграми може да се искористи алатката SoftArch [32]. Оваа алатка ги комбинира статичката и динамичката визуелизација, односно ја подобрува статичката визуелизација на архитектурата со информции добиени при самото извршување.

SoftArch претставува алатка за визуелен дизајн на софтверските архитектури. Од архитектурните дизајни направени од софтверскиот архитект, се добиваат Java класи и тоа може да се користи како почетна точка за имплементација. Класите кои се генерираат автоматски се инструментрализираат со што се овозможува ловење на одредени методски повици и настани за време на извршување. Ваквите информациите добиени за време на извршување можат да се користат за анализирање на точната имплементација на архитектурата.

SoftArch ги собира ваквите информации и ги претставува на ниво на елементи за дизајн на архитектурата. Додека се извршува имплементацијата, SoftArch продоложува да ги собира информациите и да ги прилагодува за визуелизација. На тој начина ја проширува статичката архитектурна визуелизација со динамички информации.

На Слика 14 е претставен софтверски динамичен архитектурен дизајн сличен на оние кои се добиваат со помош на SoftArch. Во овој пример, динамичката информација која се визуелизира е релативниот број од методски повици и настани. Дебелината на врските го покажува количеството на комуникација меѓу компонентите, дебелината на границите на компонентата го покажува количеството на дојдовна комуникација, а позадинската боја го покажува количеството на внатрешна комуникација [72].



Слика 14. Динамичка визуелизација на софтверска архитектура [72].

4.3.2 Добивање на дијаграми на однесување

Добивањето на дијаграми на однесување претставува вториот начин на добивање на динамичка архитектурна визуелизација. За подобро да се претстави ваквиот начин на визуелизација како пример може да се земе околината SHIMBA [79].

Околината SHIMBA за динамичка архитектурна визуелизација претставува комбинација и проширување на алатката RIGI [29] за статичка визуелизација и алатката SCED [47] за динамичка визуелизација. SHIMBA користи информации од статичката визуелизација за фокусирање и апстракција кај динамичката визуелизација.

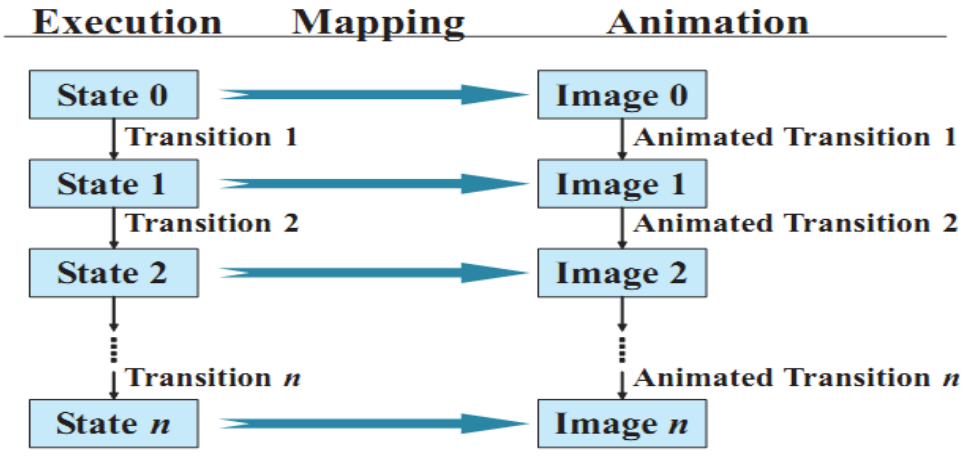
Кај статичката визуелизација, корисникот одбира софтверски елементи како класи и методи од соодветниот систем. Потоа системот се извршува и се собираат информации за соодветните елементи за време на извршувањето. На основа на информациите добиени при извршување на системот, SHIMBA произведува секвентни дијаграми. На основите на апстракцијата во статичката визуелизација (на пример, групирањето на софтверските елементи во

компоненти на повисоко ниво) SHIMBA исто така, може да прави апстракција на секвентните дијаграми и на тој начин да ја претставува интеракцијата меѓу компонентите од повисоко ниво.

4.4 Анимација на алгоритми

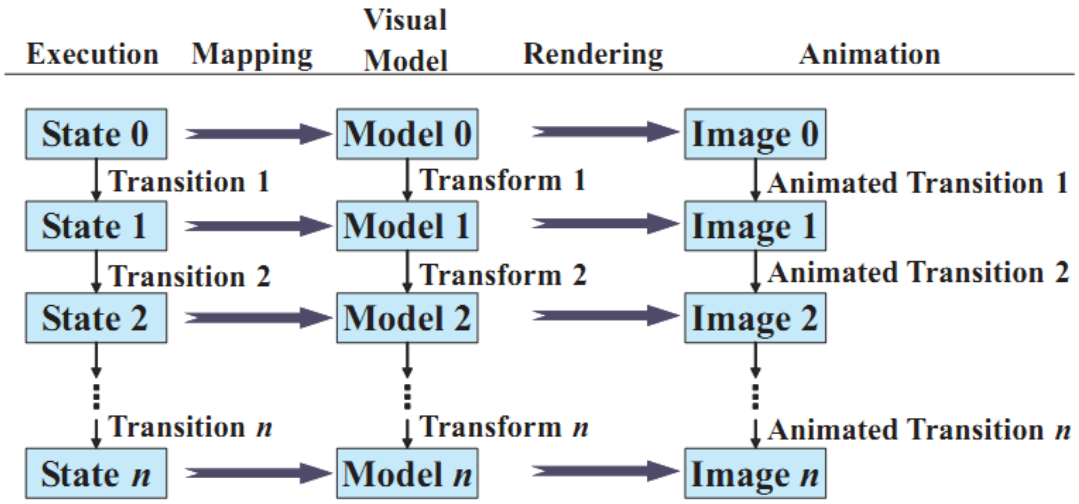
Кога станува збор за динамичка визуелизација на софтвер, прво се помислува на динамичката визуелизација на програмите, како најразвиен дел од динамичката визуелизација на софтвер. Еден дел или пример за динамичка визуелизација на програмите претставува анимацијата на алгоритмите. Анимацијата на алгоритмите се наоѓа на повисоко ниво на апстракција од визуелизацијата на програмите [1]. Повеќето од системите за анимација на алгоритмите се дизајнирани да визуелизираат еден алгоритам или група од алгоритми, наместо цели програми и најчесто се развивани да помогнат во изучувањето на истите, односно имаат едукациски цели [37] [64].

Анимацијата на алгоритмите е визуелизација на однесувањето на алгоритмите. Самиот термин „анимација“ доаѓа од глаголот “to animate”, што значи давање на живот, оживување. Тезата на Church-Turing [7] според која секој алгоритам може да се добие од Тјурингова машина претставува основа во компјутерската наука, но целта на анимацијата на алгоритмите секако не е да се визуелизира извршувањето на секој алгоритам со Тјуринговите машини, туку се земаат пресметковни модели кои се поблизу до проблемот што треба да се реши. Во сите случаи извршувањето на алгоритам од реална или математичка машина води кон секвенца од состојби. Секој чекор од извршувањето е премин од една состојба во друга. Анимацијата на алгоритмите ја мапира секоја состојба во визуелна претстава (слика) и преминот од една состоја во друга вообичаено го прикажува како анимација меѓу сликите (Слика 15).



Слика 15. Процесот на мапирање на состојбата во слика при анимацијата на алгоритмите [72].

Кај повеќето од системите за анимација на алгоритми постои и средишен слој. Состојбата се мапира во визуелни модели (графички објекти или геометриски податоци) кои понатаму се рендерираат и ја даваат сликата (Слика 16). Со користењето на ваквиот средишен слој, анимацијата може да се изведе на ниво на слика како и претходно, но предноста на ваквиот пристап е тоа што анимациите можат да се добиваат со последователни трансформации од еден модел во следен модел. Предизвикот при анимацијата на алгоритмите е да се најдат соодветните модели односно, соодветните графички апстракции за состојбите и преминот меѓу состојбите кои ќе бидат најприфатливи за корисникот.



Слика 16.Процесот на мапирање на состојбите во визуелни модели при анимацијата на алгоритми [72].

Различни луѓе имаат различни мотиви за анимирање на алгоритмите. Различните мотиви водат кон различни побарувања од анимациите и начинот на кој тие се прикажуваат. Некои од мотивите се следните:

Разбирање и изучување: наставниците ги визуелизираат алгоритмите со цел да им ги објаснат на нивните студенти.

Дизајн: софтверските развивачи ги визуелизираат алгоритмите за подобро да ги разменуваат идеите со другите експерти.

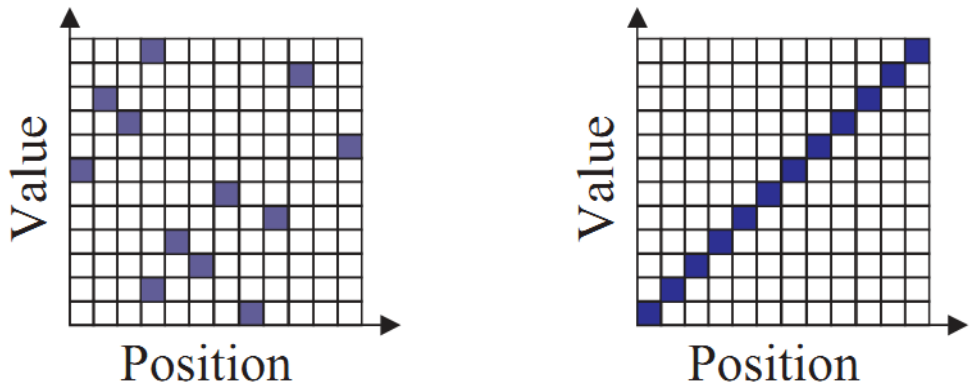
Оптимизација: софтверските развивачи ги визуелизираат алгоритмите за подобро да ги разберат како работат и да најдат начин да ги подобрат.

Дебагирање: програмерите користат визуелизација да ги најдат грешките во нивните програми и полесно да ги осознаат грешките и да ги корегираат.

4.4.1 Историја на системите за анимација на алгоритми

При опишувањето на развојот на системите за анимација на алгоритми поголемо внимание ќе посветиме на некои од системите кои имале поголем удел во развојот на денешните системи а другите ќе бидат само кусо опишани. Поголемо внимание за тоа што и како го визуелизираат ќе им биде дадено на системите Xtango, Polka и Samba [16] [28] [37] [38] кои претставуваат развојни точки во системите за анимација на алгоритми и многу од денешните системи ги имаат прифатено нивните карактеристики и начини на работа.

Првата анимација на алгоритам е филмот за обработка на листи со програмскиот јазик L6 [46]. Во следните анимации на алгоритми главен мотив било разбирањето и изучувањето на истите, односно тие имале образовни цели [8]. Вистинска движечка сила во полето на анимацијата на алгоритмите била направена со видеото *Sorting Out Sorting* [63] претставено на конференцијата ACM SIGGRAPH во 1981. Ова видео било креирано од страна на Ronald Baecker при Универзитетот во Торонто. На ова видео се прикажани 9 различни алгоритми за подредување и секоја вредност во листата е претставена со точка во матрицата (слика 17). Со ова видео е направен првиот посериозен чекор во областа на анимација на алгоритми и оттогаш наставниците ја користат ваквата анимација во своите предавања со цел да им помогнат на студентите за подобро разбирање и изучување на алгоритмите.

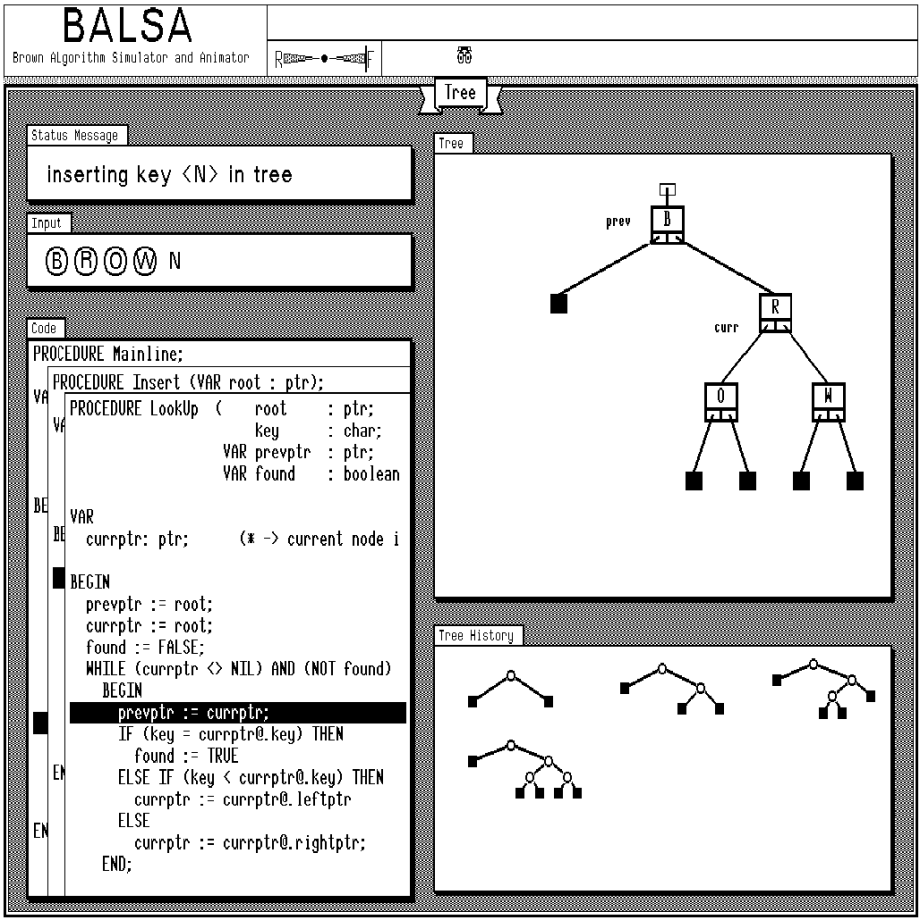


Слика 17. Анимација на подредување претставено во матрица, дел од Sorting Out Sorting [63].

Во основите на развојот на областа за анимација на алгоритми претставува и развојот на две значајни алатки за анимација на алгоритми развиени на Универзитетот Brown, кои имаат влијание врз развојот на останатите алатки со ваква намена. Овие системи се BALSA-I (Brown ALgorithm Simulator and Animator) [50] и TANGO (Transition-based Animation GeneratiOn) развиен во 1990 [39]. Во следните години Браун и Стаско развиле и други системи, водени од развојот и напредокот на технологијата во другите области, особено 3D компјутерската графика и мрежното работење.

BALSA-I бил првиот систем за анимација на алгоритми за широка употреба. BALSA-I претставува интерактивен систем за анимација на алгоритми кој подржува повеќе симулациски погледи од податочната структура на даден алгоритам и може да прикажува извршување на повеќе алгоритми истовремено. Развојот на ваквиот систем ги потикнал другите истражувачи да учествуваат во развојот на други системи за анимација на алгоритми, а исто така да работат и на поправање на недостатоците кои постојат кај овој систем. Појавата на TANGO го претставува шаблонот за дизан на анимациите и платформата на системите за анимација на алгоритми, кои подоцна се усвојуваат од другите системи за анимација.

Веднаш по развојот на овие два система, се развиваат и нивни наследници системи. BALSA-I има еден има еден систем следбеник и тоа BALSA-II. BALSA-II, кој исто така е развиен од Браун во 1988 година, е систем за анимација на алгоритми независен од доменот и користи повеќе слики и повеќе погледи, а има способност на скриптирање и кај него се додадени точки за менување на чекор по чекор и точки за стопирање. На Слика 18 е претставена анимацијата во BALSA, а истовремено се прикажани и различните погледи што ги нуди оваа програма.



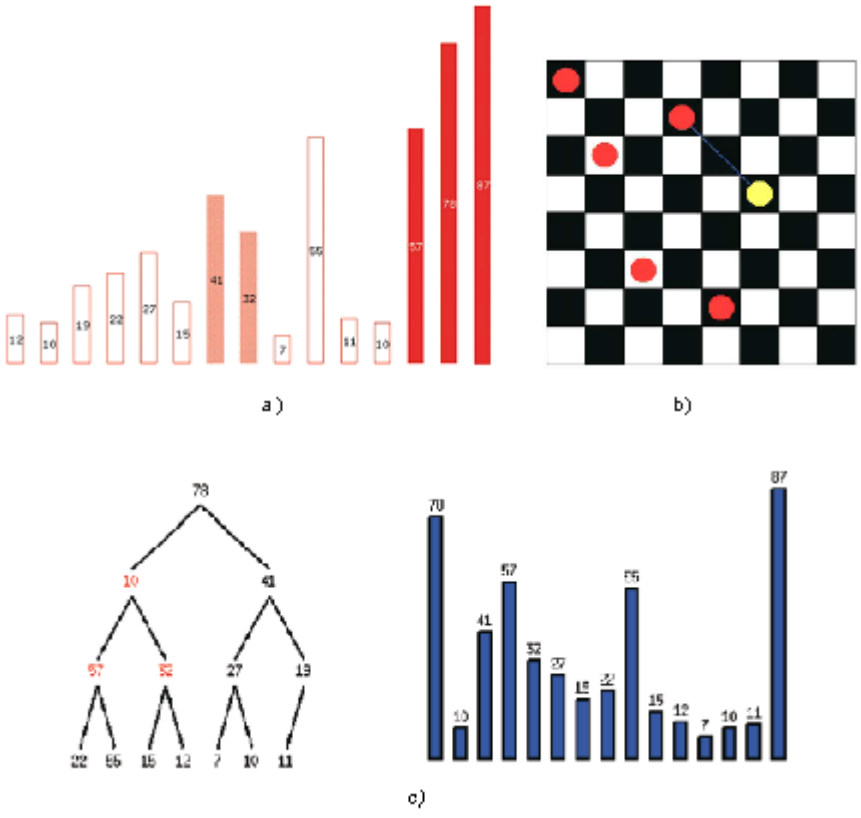
Слика 18. Приказ на анимација со BALSA [52] .

Од друга страна TANGO има повеќе следбеници. XTANGO е директен следбеник на системот TANGO. Тој претставува X window верзија на TANGO и користи парадигма на измена на пат (path-transition paradigm) со цел добивање на попрецизни анимации.

Со помош на XTANGO можат да се добијат анимации за бинарни дрва, поврзани листи, проблемот на бин-пакување, проблемот на n-кралици и некои алгоритми за подредување и тоа bubble sort, quicksort, heapsort и сл.

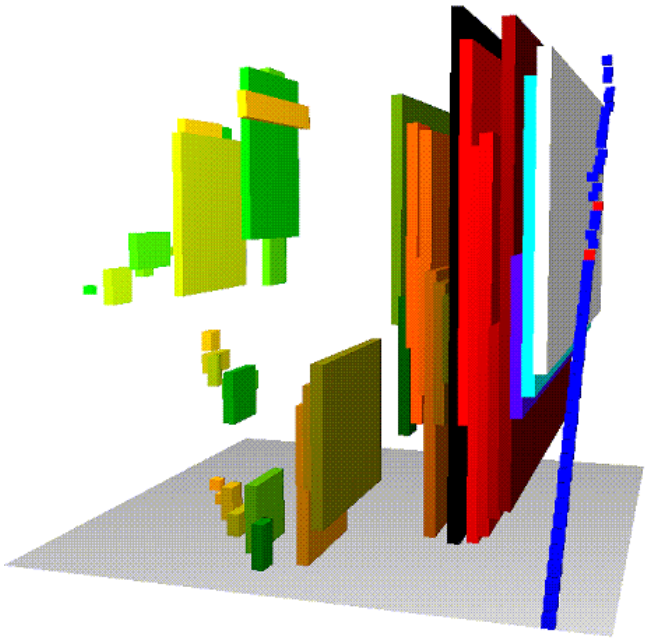
Кај анимацијата на алгоритамот за бинарно дрво, анимацијата овозможува бришење и внесување на јазлите во бинарно дрво. Анимацијата на поврзани листи го прикажува внесувањето и бришењето на елементите од листата. Анимацијата на алгоритамот за бин-пакување покажува различни стратегии за решавање на ваквиот проблем и тоа стратегија на следно сместување, стратегија на прво сместување и стратегија на најдобро сместување (next-fit, first-fit и best-fit). Анимацијата на алгоритамот што го решава проблемот на сместување на n кралици овозможува избирање на големината на шаховскиот простор и потоа ја анимира постапката за решавање на проблемот за поставување на n-те кралици во n редови. Анимацијата кај bubble sort покажува како поголемите елементи се придвижуваат кон крајот на листата. Во анимацијата на quicksort, елементите се прикажани како вертикални линии (решетки), деловите кои се добиваат при рекурзиите се прикажани со вгнездени рамки, а првиот пивот елемент (елементот каде се дели низата) е означен со различна боја. Анимацијата кај heapsort користи два погледа на истата податочна структура. Едниот поглед ја прикажува листата како дијаграм во вид на дрво, а другиот поглед како дијаграм со линии (решетки). Во првиот поглед може да се гледа како се врши применувањето на heap својството, додека вториот покажува како се подредува листата.

Овој систем особено бил користен за објаснување на споменатите алгоритми во компјутерските лаборатории за нивно подобро разбирање (Слика 19).



Слика 19. Приказ на некои од анимациите кои ги овозможува XTANGO a) анимација на bubble sort b) анимација на проблемот со n кралици c) двата погледи од анимацијата на heapsort [77].

XTANGO има други следбеници како системот POLKA и неговиот додаток Samba [1]. POLKA е дизајниран за добивање на истовремени анимации за паралелни програми. Тој претставува 2-D систем за анимација на алгоритми и има проширување POLKA 3-D. POLKA 3-D обезбедува 3-D погледи и 3-D елементи како конуси, сфери, цилиндри и сл. Корисниците не е неопходно да имаат претходни познавања за 3-D компјутерската графика да можат да го користат POLKA 3-D (Слика 20).

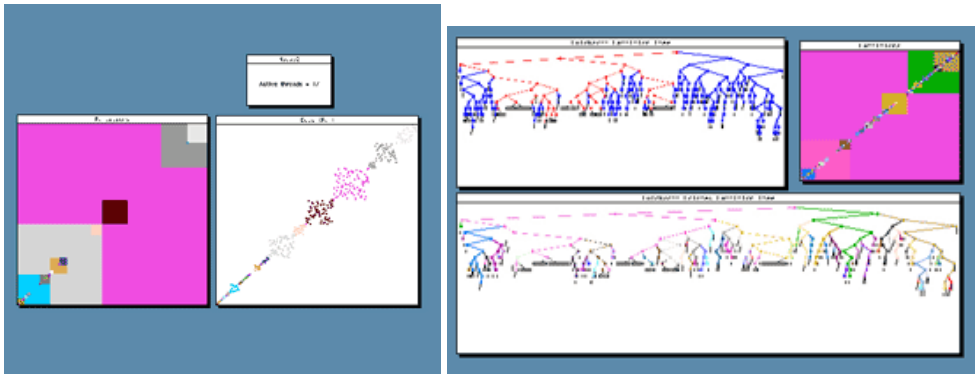


Слика 20. Приказ на анимација со POLKA 3D за алгоритамот quick sort (преземено од [88)].

Samba претставува интерактивен интерпретер за анимации што чита ASCII команди и ги прави соодветните анимации. Постои и Java верзија на Samba наречена JSamba.

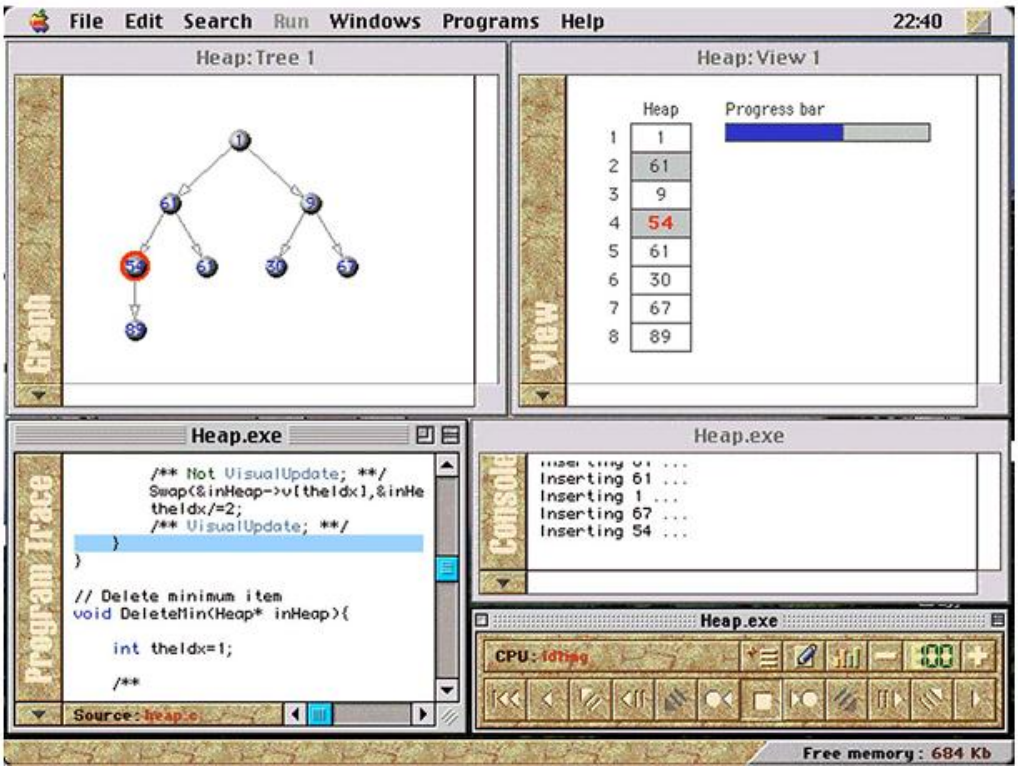
Други користени системи за анимација на алгоритми се Zeus [56] [57], Leonardo [62] [89], CATAI [25], Mocha [42].

Zeus е развиен на Универзитетот Браун во 1991 и заедно со BALSA и BALSA-II се смета за еден од првите поголеми интерактивни системи за визуелизација на софтвер. Ваквиот систем подржува повеќе синхронизирани погледи и им дозволува на корисниците да ги менуваат погледите и да ги менуваат претставите на податочните елементи (Слика 21). Zeus е имплементиран кај повеќе-нитки и повеќе-процесорска околина и на тој начин овој систем може да анимира паралелни програми.



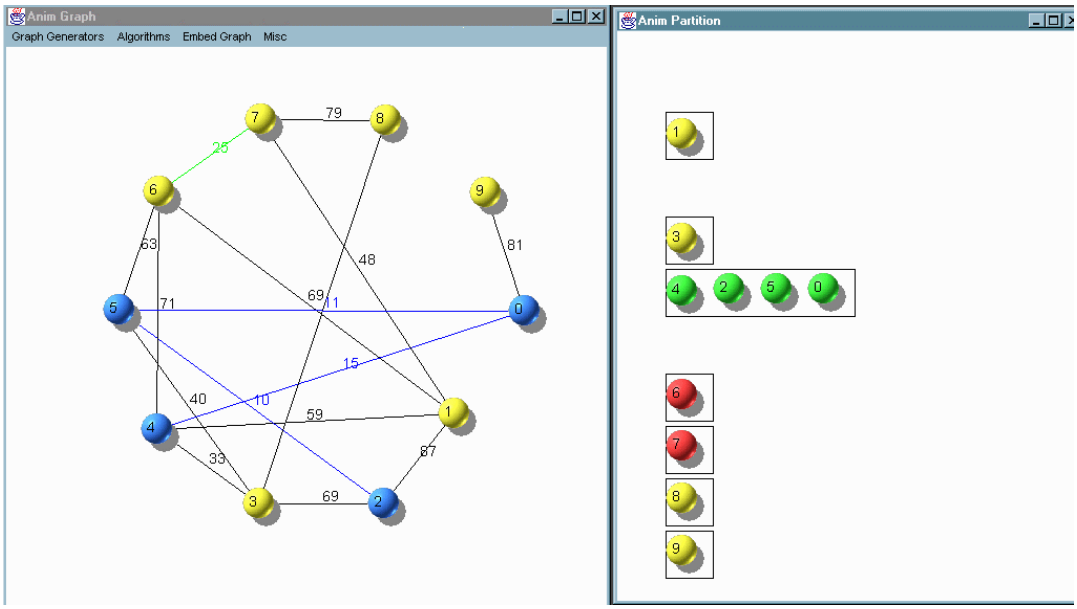
Слика 21. Приказ на околината на Zeus со анимација на алгоритамот quick sort [87]. .

Leonardo е вградлива околина за развој и анимација на C програми. Ваквиот систем ги вградува развојот и анимацијата на C програмите заедно со зголемена контрола на корисникот врз анимацијата, заедно во една иста околина. На слика 22 е претставен приказ на околината на Leonardo при анимацијата на heapsort алгоритамот.



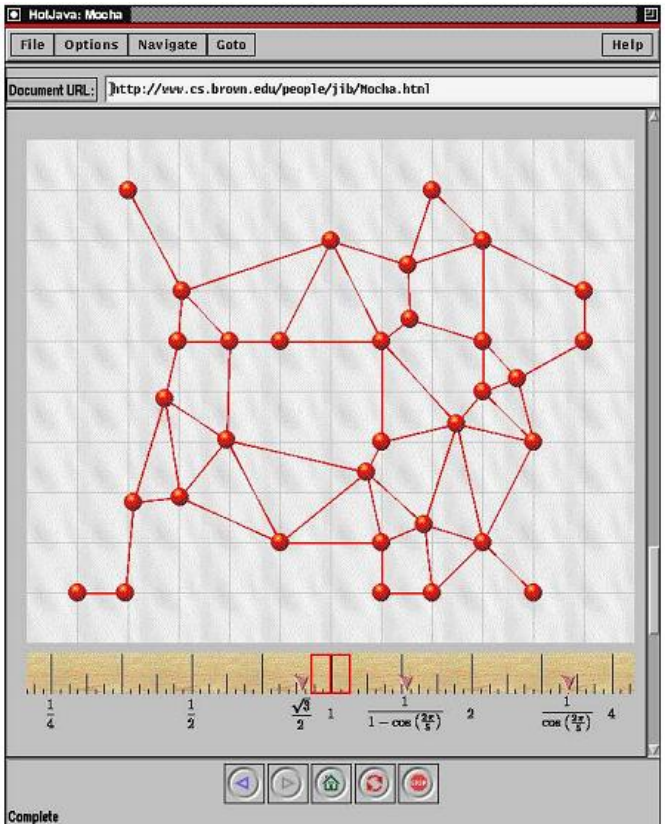
Слика 22. Приказ на околината на Leonardo со анимација на алгоритамот heap sort (преземено од [89]).

CATAI претставува систем за анимација кој анимира C++ програми [25]. Тој се темели на дистрибуирани објектни технологии и дозволува неколку корисници да делат иста анимација преку виртуелна апстракција. Овој систем најмногу има образовни цели. Комуникацијата и синхронизацијата меѓу анимациските клиенти и анимираниот алгоритам се обезбедува преку Java анимациски сервер кој користи CORBA технологија. На слика 23 е прикажана дел од анимацијата на алгоритамот на Крускал со помош на CATAI.



Слика 23. Приказ на околината на CATAI со анимација на алгоритамот на Крускал [90].

Mocha претставува дистрибуиран модел со клиент сервер архитектура која оптимално ги дели софтверските компоненти на еден вообичаен систем за анимација на алгоритми. Овој систем ги надминува проблемите наследени од X window и Java моделите. На слика 24 е прикажана дел од околината на Mocha.



Слика 24. Приказ на околината на Mocha [42].

Кај моделот Mocha, само интерфејсниот код е сместен на корисничката машина, додека алгоритамот се извршува на сервер кој работи на провајдерската машина.

Првите системи за анимација на алгоритми биле зависни од платформата. Со помош на развојот на новите технологии, употребата на World Wide Web и развојот на програмирањето во програмскиот јазик Java, развивачите се насочуваат кон градење на системи за анимација на алгоритми кои ќе бидат независни од платформата, со отворен и слободен пристап и ќе бидат достапни преку интернет. Некои развивачи исто така, вклучуваат и мултимедијални елементи во нивните системи. Употребата на системи за анимација на алгоритми се проширува не само кај работните училници и лаборатории туку се применуваат и на далечинско учење.

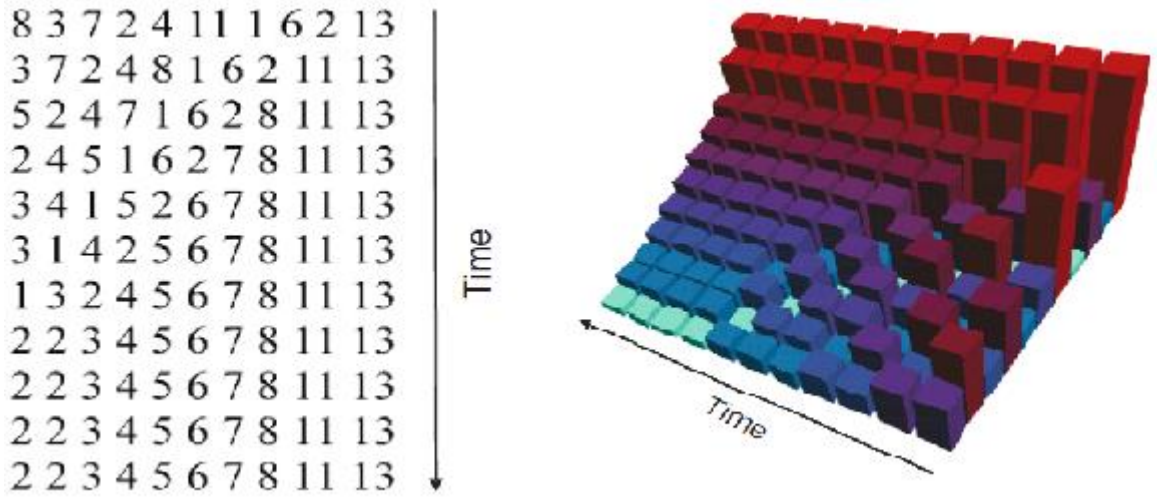
Денес постојат и многу други системи за анимација на алгоритми и програми како на пример: Animal, Daphne, GANIMAL, Gasp, GeoWin, Jawaa, Jeliot и многу други. Некои од денешните системи што постојат имаат сложена архитектура и имаат посебни побарувања во однос на технологијата да можат да работат. Но, поголемо внимание им се посветува на системите кои имаат не толку сложена архитектура, не се многу големи и се полесни за имплементација и за употреба.

4.4.2 Анимација на алгоритми со користење на 3D

За употребата на трета димензија во прикажувањето на анимациите има повеќе причини. Една од причините е естетиката. Три димензионалните графички елементи кои се рендерираат со фото-реалистична техника на рендерирање се поинтересни за многумина. Друга причина е самата природа на човековото око кое е навикнато на гледање во три димензии. Следната причина е димензионалноста. Додавањето на нова димензија може да овозможи додавање на дополнителни информации на основниот дводимензионален приказ. Овие дополнителни информации можат да се однесуваат на додавање на дополнителни погледи, односно истиот објект може да биде прикажан на различни начини и од различни агли. Но, кога станува збор за додавање на дополнителни информации тие можат да се однесуваат и на додавање на временска оска во приказот. На тој начин можат да се претстават состојбите на објектот во различни временски точки. Друга причина за користење на трета димензија може да биде тоа што во некои области податочните структури и алгоритмите се наследно три димензионални.

На слика 16 е прикажана имплементација на алгоритамот за подредување bubble sort со VRML и JavaScript . Третата димнезија во тој случај се користи за прикажување на историјата на подредувањето, односно на *Z* оската се

зачувуваат сите претходни состојби низ кои минува низата за време на подредувањето.



Слика 16. Приказ на подобрување на анимацијата за bubble sort со користење на 3D [72].

4.4.3 Архитектура на алатките за анимација на програми

Алатките за анимација на софтвер се дизајнирани за различни цели. Најчесто алатките не можат да бидат и лесни за разбирање и лесни за употреба а истовремено и доволно моќни за работа. Што може да прави алатката и како тоа го прави најмногу зависи, пред сè од целта на употреба. За имплементација на повеќето алатки за анимација на програмите ги следат некои од следните архитектури.

Ad hoc архитектура: Кај оваа архитектура анимациите на еден алгоритам се имплементираат без воопшто да се користи каква било алатка, туку се се имплементира почнувајќи од почеток.

Архитектура со библиотеки: Да се имплементира анимација на еден алгоритам се користат библиотеки што содржат графички делови, контролни елементи и сл. Таквите библиотеки можат да вклучуваат и графички кориснички интерфејси, на пример, со копчиња за почеток, крај или пауза.

Архитектура на специјални податочни видови: При програмирањето на алгоритамот се користат податочни видови, кои имаат вградени визуелизации и на тој начин анимацијата претставува само нус-производ при стартување на апликацијата.

Архитектура *Postmortem*: Алгоритамот и алатката за анимација се две различни апликации. Кога се извршува алгоритамот се зачувуваат некои

делови, односно се прави анимациски план, кој понатаму се визуелизира со помош на различни компоненти.

Архитектура на интерсни настани: Алгоритамот при одредени точки од програмата се бележи со интересни настани. При извршувањето, овие настани се праќаат кон еден или повеќе погледи. Ваквиот пристап најчесто го користи MVC8 (Model View Controller) шаблонот за дизајн на софтвер.

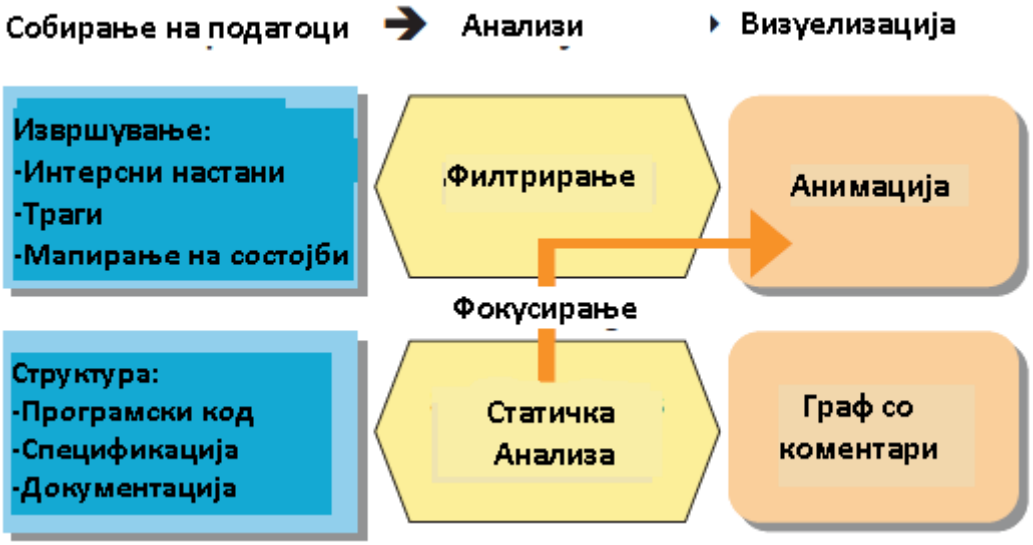
Декларативна архитектура: Ознаките и алгоритамот се одвоени. Постојат два пристапа, првиот е мапирање на состојбата а вториот користи системи кои се засноваат на ограничувања. Кај првиот пристап демонот ги следи промените на состојбата и соодветно на тоа ја ажурира состојбата на визуелизацијата. Кај вториот пристап програмата која се визуелизира самата по себе е напишана во јазик кој се базира на ограничувања.

Семантички насочена архитектура: Алгоритамот се извршува со користење на визуелен преведувач или дебагер кој автоматски ги произведува визуелизацииите, но најчесто на некое ниско ниво на апстракција.

Декларативната и семантички насочената архитектура се неинвазивни, односно да се добие визуелизацијата на програмата нема потреба да се менува програмскиот код.

4.4.4 Апстрактна анимација на алгоритми

Еден од најголемите проблеми при анимацијата на програмите претставува фокусирањето, односно кои делови од податочната структура треба да се прикажат на екранот. Не секогаш е потребно да бидат истовремено прикажани сите податочни структури кои учествуваат во визуелизацијата, бидејќи количеството на податоци го прави неможно истовремено прикажувањето на сите податоци. Затоа, потребно е внимателно да се гради сликата која ќе се покаже на екранот. На слика 25 е дадено како може да се комбинира визуелизацискиот тек од статичката и динамичката програмска визуелизација. Иако постои голем потенцијал во статичкото анализирање на програмите за фокусирање при нивните анимации, сепак во голема мера ваквиот потенцијал не е доволно искористен.

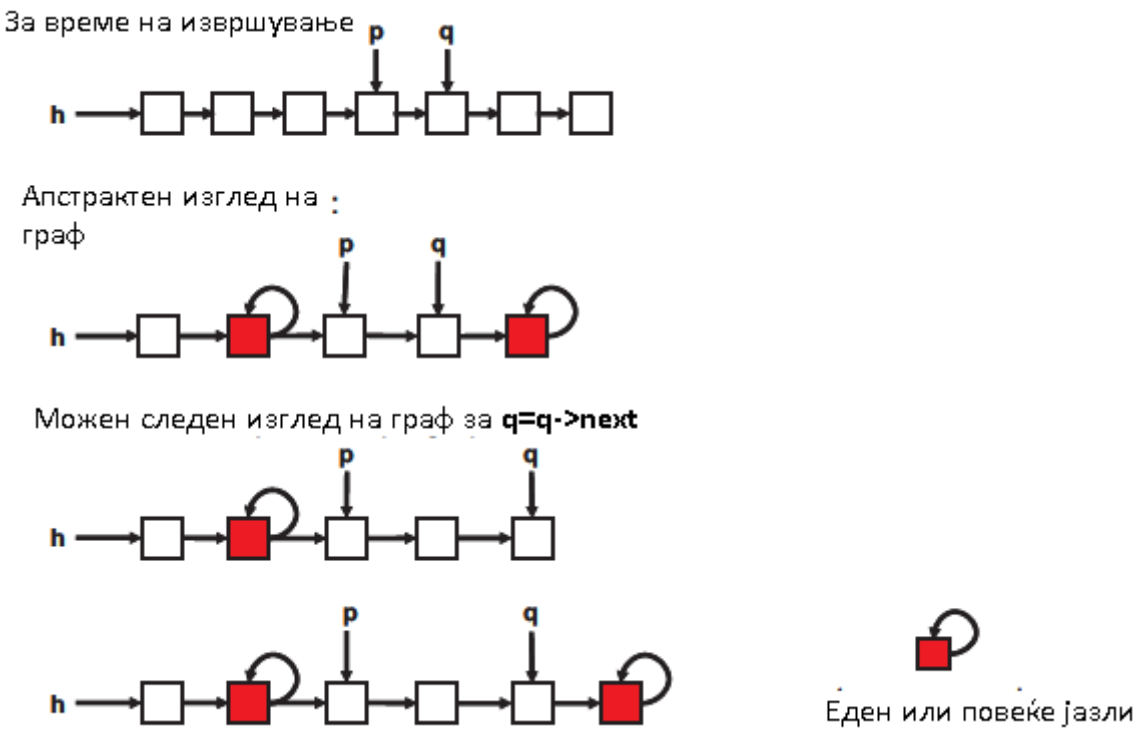


Слика 25. Користење на програмска анализа за фокусирање во анимацијата

Во секоја програмска точка може да се пристапи само до мал дел од податоците. Едно можно решение е ваквото делче на податоци да се определи со статичка анлиза, со која може да се пресмета информацијата за податочните структури до кои може да се пристапи во секоја точка од програмата.

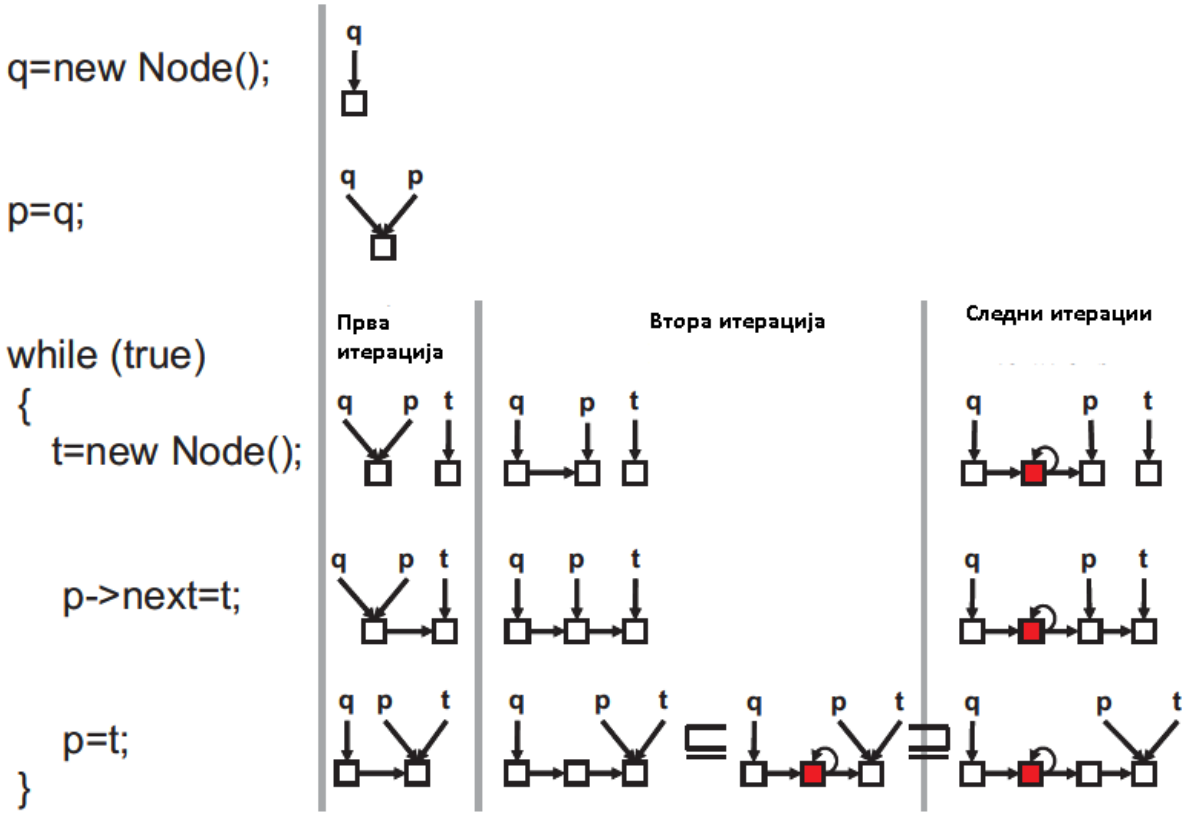
Ваквите анализи се нарекуваат анализи на форми и се развиени од Sagiv, Reps и Wilhelm [55]. Анализата пресметува апстрактна претстава од поврзани податочни структури и се фокусира на активните делови од овие структури. За секоја точка од програмата се даваат конечно множество на графички фигури. Braune и Wilhelm предложиле дека апстрактното извршување треба да се анимира со помош на графичките фигури, како основа. Резултатот на анимацијата го нарекле „алгоритамски објаснувања“, со цел да нагласат дека во секоја програмска точка се прикажани константи на податочните структури. Ваквиот пристап понатаму се проширува со прикажување на неструктурирани константи.

На слика 26, се прикажува апстрактното извршување во дадена програмска точка (q=q->next). Според графот на апстарктни форми пред извршувањето покажувачот q покажува кон елемент веднаш до елементот кој е покажуван од покажувачот p и има барем уште еден елемент по елементот. Ако се изврши задачата q:=q->next, постојат две можни апстрактни форми на графот. Првата форма го покажува случајот, кога во листата има точно еден елемент повеќе, а вториот случај е кога има најмалку два елемента повеќе.



Слика 26. Апстрактно програмско извршување на q=q->next [72].

На тој начин апстрактната форма на графот претставува множество на конкретни форми на граф, односно покажувачки структури кои можат да се случат за време на извршувањето на програмата. Нека *γ* е функцијата што се добива, за дадена апстрактна форма на граф, од множеството на преставени конкретни форми. Преминот од апстрактната состојба *as*1 до состојбата *as*2 е легална, само ако постои премин од конкретна состојба *cs*1 во конкретна состојба *cs*2, каде *cs*1 се претставува со *as*1, а *cs*2 со *as*2, односно $cs1 \in \gamma(as1)$ и $cs2 \in \gamma(as2)$. Визуелното апстрактно извршување ги покажува само легалните премини. На слика 27 е претставено апстрактното визуелно извршување на едноставна јамка што креира една бескрајна листа. За секоја програмска точка, на десната страна, се претставуваат форми на графови кои ги опишуваат различните можни состојби на програмата по извршувањето на инструкцијата во дадената програмска точка.



Слика 27. Пример за апстрактно извршување на едноставен програмски сегмент [72].

За секоја програмска точка во јамката, постојат три такви случаи: еден случај за првата итерација, еден за втората и еден случај за сите следни итерации.

Така, за секоја точка во јамката, сите можни состојби се опишуваат со множество на графови во три форми, кои ги прикажуваат структурните константи кои се добиваат по извршувањето на секоја програмска точка. Затоа, апстрактното визуелно извршување исто така, може да се гледа и како проверка на точноста на оделни програмски делови. Апстрактните форми на графовите за втората и следните итерации во последниот ред се присоединуваат во една прикажана меѓу нив. Апстрактната форма на граф $as2$ ја присоединува апстрактната форма на граф $as1$, односно $as1 \subseteq as2$, ако сите конкретни форми на графови претставени со $as1$ се исто така, претставени и со $as2$, однсоно $\gamma(as1) \subseteq \gamma(as2)$. Апстрактната анимација на програмите уште е на самиот почеток. Дури и за мали програми, множеството на форми на графови за секоја програмска точка е многу големо. Како едно решение може да биде разделувањето на ваквите множества, така што секој дел содржи форма на граф која претставува слични случаи и затоа нема потреба од посебна визуелизација.

4.5 Визуелно дебагирање и тестирање

Друга цел при динамичката визуелизација на програмите може да преставува поголемата прегледност што ја нуди при дебагирањето на програмите и поправањето на грешките.

Целта на дебагирањето е да се отстранат можните постоечки грешки во програмата, да се најде нивната локација, причината поради која се појавуваат и секако, на крај да се поправат. Кога станува збор за користењето на визуелизацијата при дебагирање, постојат два вида на визуелизации:

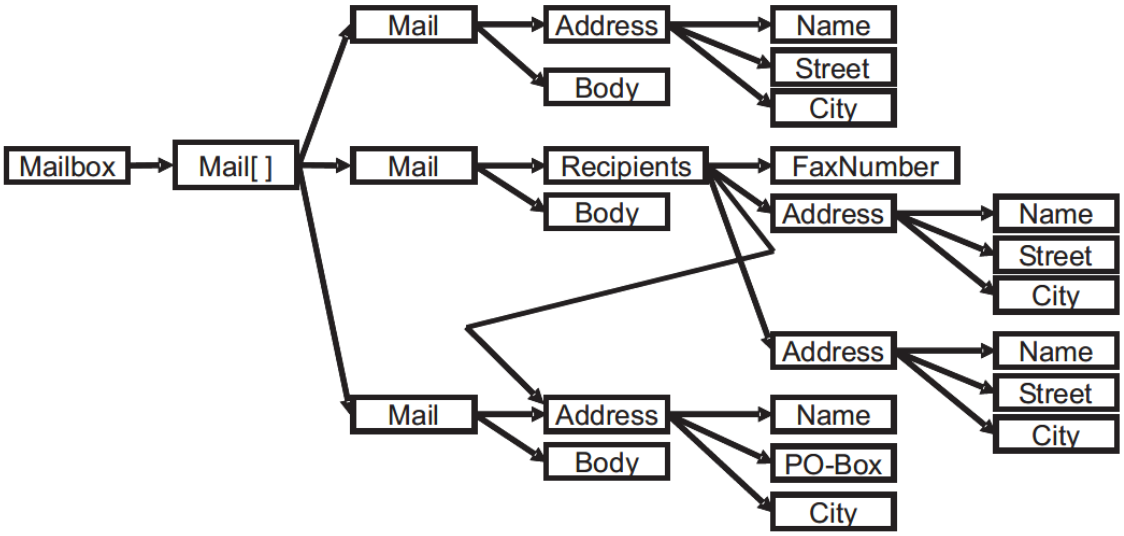
-визуелизации кои ги покажуваат состојбите или меморијата

-визуелизации кои ги покажуваат програмскиот код и резултатите од тестирањето.

Дури и наједноставната програма во C или во Java, која само печати нешто на екран, се состои од неколку илјади бајти, поради многуте податочни структури кои се барани од страна на извршувачкиот систем. За некои мултимедијални апликации ваквото количество податоци е многу поголемо. Прикажувањето на сите податоци е неможно, затоа на програмерот му се потребни техники за избирање само на одредени делови од програмската состојба или наоѓање начин да се соберат податоците да се прилагодат на екранот. При дебагирањето прво, се определуваат изразите кои се вклучени, а потоа се намалува проблемот со избирање на изрази кои се можни носители на грешки.

Визуелизацијата на меморијата може да се прикаже со помош на мемориски графови. Јазлите ја прикажуваат содржината на меморијата, а врските покажуваат на можни пристапни патеки. Мемориските графови се создаваат со одмотување на сите пристапни податочни структури во програмата.

На слика 28 е прикажан делот од меморикиот граф за дадена Java апликација.

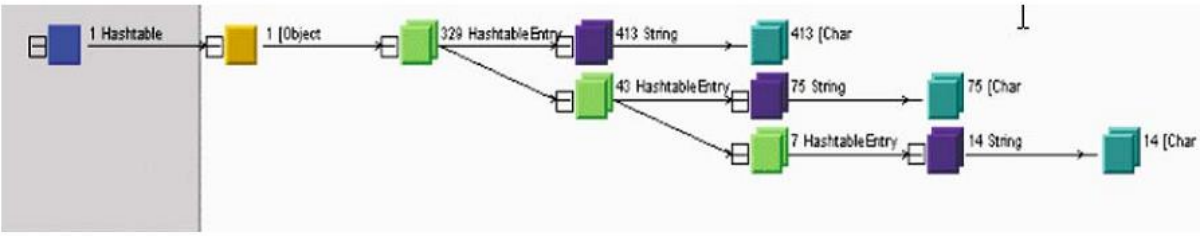


Слика 28. Пример за мемориски граф на Java апликација [72].

На пример, ако се утврди изразот или методот во кој програмата паѓа, со гледање на разликата во меморискиот граф пред и по извршувањето на изразот или методот, можат да е забележат некои несакани ефекти кои не можат јасно да се видат при читање на кодот.

Ако се занемарат вредностите на јазлите во мемориските графови и се задржи само структурата, тогаш над нив можат да се применат стандардните операции кои важат за графови. Како резултат на тоа, можно е да се пресметат разликите во мемориските графови кои се резултат на различни извршувања на една иста програма. Со забележувањето на разликите во мемориските графови во иста програмска точка на успешно извршена и на програма во која се јавува грешка, можат да се определат оние места во програмата кај кои се јавува грешката.

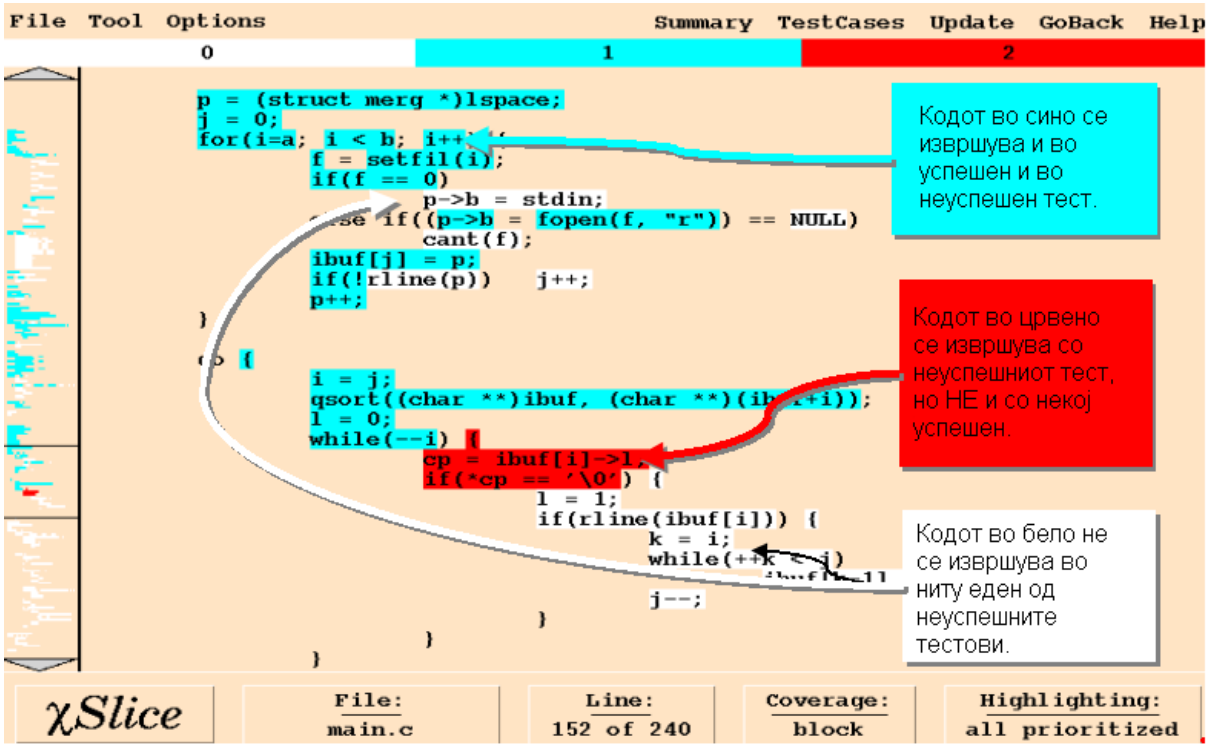
Наместо користење на мемориски графови, визуелниот Java дебагер Jinsight [84] ги развил шаблоните на референци (Reference patterns), кои имаат структурата на дрва. Кај шаблоните со референци кои покажуваат напред (forward reference pattern), секој јазол ги прикажува сите објекти од дадена класа кои се рефернцирани од страна на барем еден објект од јазелот родител. Кај шаблоните со референци кои покажуваат назад (backward reference pattern), секој јазел ги претставува сите објекти на една класа која рефернцира барем до еден од објектите петставници на јазелот родител. Шаблоните со референци, почетно биле развиени за наоѓање на мемориските преполнувања кај Java програмите [83]. Слична таква постапка може да се искористи за издвојување на извршувачките шаблони од динамичкиот гарф на повици . На слика 29 е прикажан пример за шаблон со референци за хаш табела кој покажува напред.



Слика 29. Пример за шаблон со референци кој покажува напред [72].

Наоѓањето на оние делови од програмата што имаат неточни вредности, не значи дека програмерот знае кој дел од програмскиот код води до тие грешки и кој дел треба да се поправи. Визуелизацијата што служи за полесно наоѓање на грешките ги истакнува оние делови од програмата кои се најверојатни места за појава на грешки. Техниките што го овозможуваат тоа главно се водат со идејата дека оние делови од програмата кои најмногу се извршуваат кога програмата предизвикува грешка, се поверојатни места за појава на грешка.

Статичкото отсекување (slice) претставува множество од програмски покажувачи кои можат да влијаат на вредноста на определен излез или на инстанца од променлива во одредена програмска точка. Статичките отсекувања се пресметуваат со помош на статичката програмска анализа. Извршувачкото отсекување (slice) претставува множество од сите програмски покажувачи кои се извршуваат за даден влез. А, динамичко отсекување е множество на сите програмски покажувачи за даден влез, што навистина влијаат врз програмскиот покажувач или на истанца од променлива во одредена програмска точка. Односно, динамичкото отсекување претставува подмножество на извршувачкото отсекување. Значајна операција на динамичките отсекувања (slices) е dicing. Dice претставува разликата меѓу две отсекувања A и B (разликата A-B). Пример за таква алатка која нуди slicing и dicing кај C програмите е X-Slice [27] [86]. Оваа алатка овозможува динамичко визуелно дебагирање. На слика 30 е прикажан пример од визуелизација со помош на X-Slice.



Слика 30. Пример за визуелно тестирање со X-Slice (преземено од [86]).

Тестирањето претставува процес на извршување на дадена програма со намера да се откријат потенцијални грешки. Тестирањето се прави со стартување на програмата со некои влезни податоци и се проверува дали се добива очекуваниот излез или однесување. Користењето на визуелизацијата при извршувањето на програмата може да помогне при полесно откривање на локацијата на грешката, а со тоа да помогне побргу таа да биде поправена.

5. Алатки за динамичка визуелизација на софтвер

Денес постојат најразлични алатки што служат за визуелизација на софтвер.

Дури и денешните развојни околини за програмските јазици сè почесто вклучуваат визуелни елементи во програмирањето со цел да се приближат до корисникот а го напуштаат традиционалното програмирање со користење само на чист код.

Во овој труд ќе се задржиме на динамичката визуелизација на софтвер, особено на алатките што денес се користат за образовни цели [18][19], односно за изучување на начинот на кои работат програмите и одредени програмски сегменти.

Бидејќи постојат различни алатки кои се развиени од цели тимови на соработници на повеќе универзитети во светот и тие се добро разработени и корисни во однос на целите за кои се применуваат, правењето на некоја релативно едноставна програма што ќе успее да визуелизира само мал дел од тоа што веќе постоечките алатки го имаат направено би било бескорисно. Затоа целта на ова истражување е добро да се разработат алатките што веќе постојат, да се нагласат нивните добри и лоши страни, во кои дел го подобруваат изучувањето и каде би можеле да се искористат.

Најголемо внимание би посветиле на некои делови од програмирањето и програмските сегменти што се релативно тешки за разбирање особено од страна на почетниците во програмирањето. Некои такви делови се податочните структури и рекурзијата. Податочните структури особено претставени во програмските јазици C и C++ изобилуваат со многу единечни и двојни покажувачи кои за почетниците го претставуваат најтешкиот дел од програмирањето. Ако ваквите елементи кои тешко можат да се разберат само со користење на код, се претстават во вид на слика или анимација, корисникот добива појасна претстава за изгледот и начинот на нивното функционирање.

Некои од програмите кои детално ќе ги разработиме се DDD [2] [9] [10] [91], JGrasp [43] [44] [92], SRec [6] [31] [93] и Jeliot 3 [5] [59] [69] [85].

Поголем дел од денешните алатки и програми што се достапни и можат слободно да се користат се темелени на програмскиот јазик Java, а многу помалку некои од другите програмски јазици.

Од алатките издвоени за обработка само кај DDD се користи C или C++ програмскиот јазик, а сите други како основен програмски јазик го користат Java.

Причината поради која се одлучивме токму за овие алатки е тоа што во нив се интегрирани повеќе од својствата што се користат кај повеќето постоечки алатки. Тие точно и прецизно ги претставуваат, односно визуелизираат програмските делови за кои се наменети. Друга причина за која ги одбрав овие алатки е тоа што на корисникот му даваат можност да користи чист Java или C код, без да мора тој самиот кодот да го преведува во некој друг програмски јазик или да додава елементи во кодот со цел да се добие посакуваната визуелизацијата. Исто така, нивната предност во однос на другите постоечки програми за таа намена е тоа што тие не работат само на определен код кој е веќе предефиниран во самата програма и само него можат да ги визуелизираат, туку нудат можност корисникот да пишува свој код и да го гледа резултатот од неговата визуелизација.

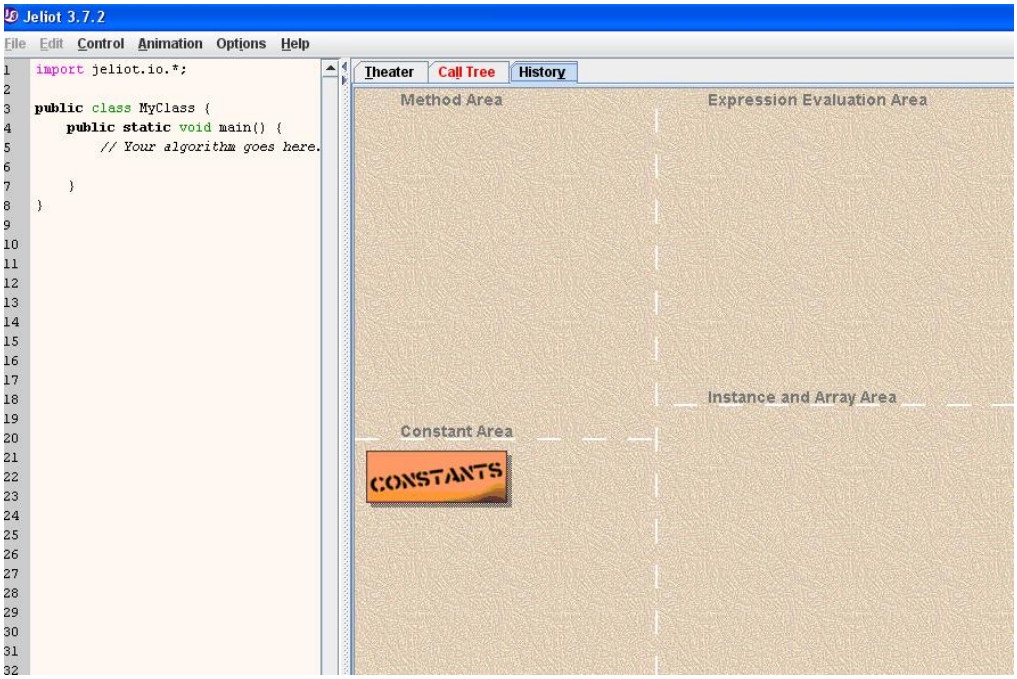
DDD го земавме како претставник за визуелизирање на податочните структури во програсмкиот јазик C, во кои се користат покажувачи. На тој начин полесно би се разбрало значењето на покажувачите и потребата од нивното користење. Значи оваа алатка, или поточно дебагер, најмногу може да се искористи во тој дел. Ако истиот код се изврши во алатката Jgrasp во тој случај нема да се добие толку голема прецизност при покажувањето на мемориските локации и начинот на кој работат покажувачите. SRec го избравме како алатка што специјално е наменета за прикажување на рекурзиите. А останатите алатки ги избравме да направиме споредба во однос на начинот и визуелизацијата што нудат на истите елементи.

5.1 Jeliot 3

Jeliot 3 претставува алатка за визуелизација на програми. Ваквата апликација визуелизира како се интерпретира програмата. Како што тече анимацијата, на екранот се прикажуваат повиците кон методите, променливите и операциите. На тој начин може да се следи текот на програмата чекор по чекор. Програмите можат да се креираат од самиот почеток или можат да се модифицираат претходно постоечките примери на код. Java програмата што се анимира, нема потреба од никакви дополнителни повици, сите визуелизации автоматски се генерираат. Jeliot 3 ги разбира повеќето Java програмски елементи и нуди можност за нивна визуелизација и анимација. Посебно тешко

за визуелизација претставува анимирањето на објектно-ориентираните карактеристики, како што е на пример, наследувањето. Но, оваа алатка успешно го изведува прикажувањето на објектно-ориентираните концепти, визуелизирајќи ги објектите и наследувањето.

Основната карактеристика на Jeliot 3 е целосната или полу автоматската визуелизација на податоците и контролните текови. Значи нема потреба од додавање на дополнителен код или користење на посебни елементи за добивање на визуелизацијата. Корисникот користи исклучително Java код а автоматски се добива визулизацијата на напишаната програма. На слика 31 е даден изгледот на работната околина на Jeliot3.



Слика 31.Изглед на работната околина на Jeliot 3.

Анимација на програма кај Jeliot 3 се случува во посебен дел кој се нарекува театар. Тоа всушност е поголемиот панел надесно од кодот. Театарот се дели на четири различни области: област за методи каде се сместуваат деловите од методите кои содржат променливи; област на константи, каде се прикажуваат константите и статичките променливи; област на објекти, каде се алоцираат објектите или се референцираат од променливите од делот на методите; област за прикажување на пресметките.

Главните предности на Jeliot 3 се следните:

- системот е лесен за употреба
- визуелизацијата што се добива е иста во сите случаи на визуелизрање, а исто така е целосна и без прекини.
- визуелизацијата подржува големо множество на едноставни програми напишани во програмскиот јазик Java.

Добра страна на оваа алатка е тоа што нуди можност за проширување и надградување.

Меѓутоа, постојат две главни несогласувања меѓу Jeliot и Java. Сите класи потребно е да бидат во една датотека и за работа со влез излез кај Jeliot потребно е да биде вклучен пакетот jeliot.io.*. Тој ги обезбедува методите: `void Output.println(), int Input.readInt(), double Input.readDouble(),char Input.readChar(), String Input.readString()`, а исто така е поддржан и стандардниот излез.

Jeliot користи Dynamic Java и затоа подржува голем дел од карактеристикте на Java, особено оние поедноставните кои се користат за почетничко изучување и разбирање на програмскиот јазик Java.

Но, сепак постојат одредени елементи што уште не се вклучени.

Со Jeliot 3 се прифатливи и можат да се анимираат следните компоненти:

- Вредности од видот на String, сите примитивни податочни видови и едно-димензионални вектори, статички променливи, изрази што ги вклучуваат сите бинарни и унарни операции освен instanceof, сите контролни структури и циклуси (if, while, for), условни изрази од видот (израз?израз1:израз2), повикување на методи, а вклучено е и рекурзивното повикување на методи, конструктури, резервирање на објекти.

А, некои од елементите што не ги подржува Jeliot 3 се следните:

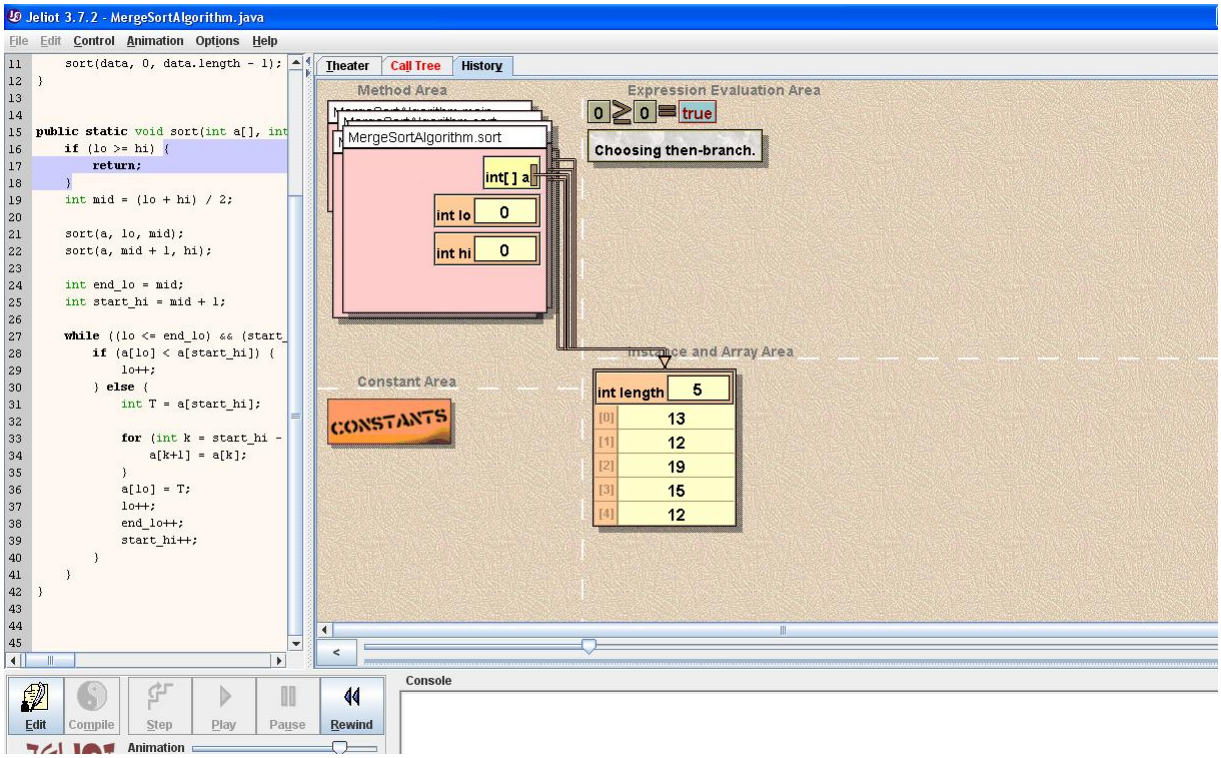
- пристапи до super поле, низи со референци (освен String), две или повеќе димензионални вектори, методи кои враќаат инстанца од класа.

Дизајнот на корисничкиот интерфејс е еден од главните аспекти кај алатката, и бидејќи е наменета за почетници, интеракцијата со корисникот е олеснета.

Делот односно рамката што се наоѓа на левата страна од екранот служи за пишување или правење измени на веќе постоечки код. Кога се отвора или се креира нова датотека, таа се отвора веднаш во делот за уредување и изменување на кодот. Кога се компајлира програмата се затвора делот за уредување и во тој момент кодот не може да се менува, а се отвора делот со слики односно започнуваат анимациите. Откако анимацијата е завршена може да се врати програмата во делот за уредување на кодот со кликнување на копчето Edit. При текот на анимацијата делот кој се визуелизира во кодот потемнува, за корисникот да има појасна слика кој чекор од кодот моментално се извршува и се прикажува на екранот во делот со анимација.

Во делот со кодот има и нумерирање на секој ред, со цел да се добие поубава прегледност на програмата што се визуелизира. И во случај на грешка

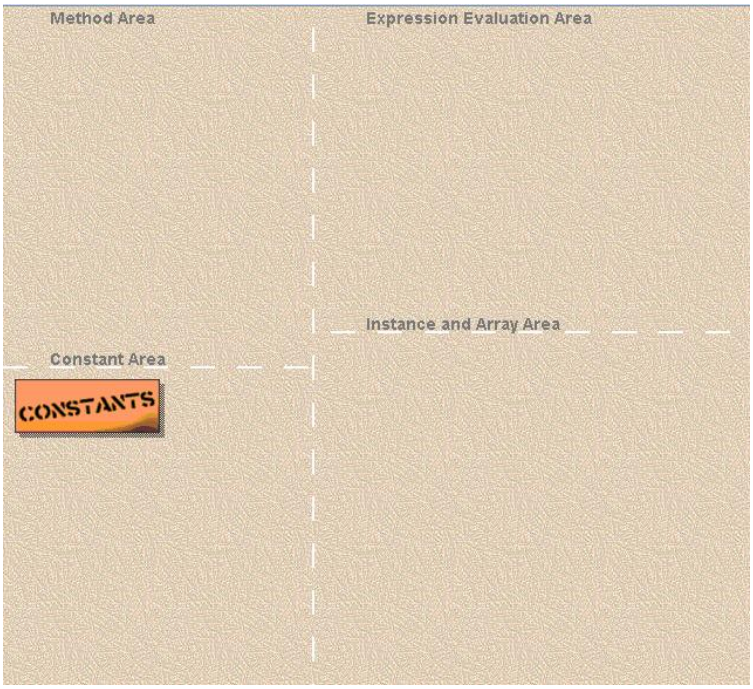
полесно да може до се дојде до грешката и таа да биде побргу исправена. Корисничкиот интерфејс на Jeliot 3 е прикажан на Слика 32.



Слика 32. Кориснички интерфејс на Jeliot 3

Делот со менијата се состои од повеќе менија и копчиња што можат да се користат при уредувањето или визуелизацијата на програмата. За време на визуелизацијата копчињата се оневозможуваат да се отстапи повеќе простор на визуелизацијата на кодот. Делот за контрола се состои од повеќе копчиња. Ако настане грешка за време на извршувањето на програмата, во погледот за грешки се прикажува грешката, а во делот со код се потенцира делот во кој најверојатно настанала грешката. Ако програмата има потреба од некакво внесување на податоци тој се прикажува во делот за визуелизација, а ако дава некаков излез тој се печати на излезната конзола на дното од прозорецот.

Кај Jeliot 3 визуелизацијата е едноставна и блиска до корисникот и секој корисник може лесно да ја користи без некои поголеми проблеми. Сите визуелизирачки компоненти имаат своја сопствена област на екранот, како што е покажано на слика 33.

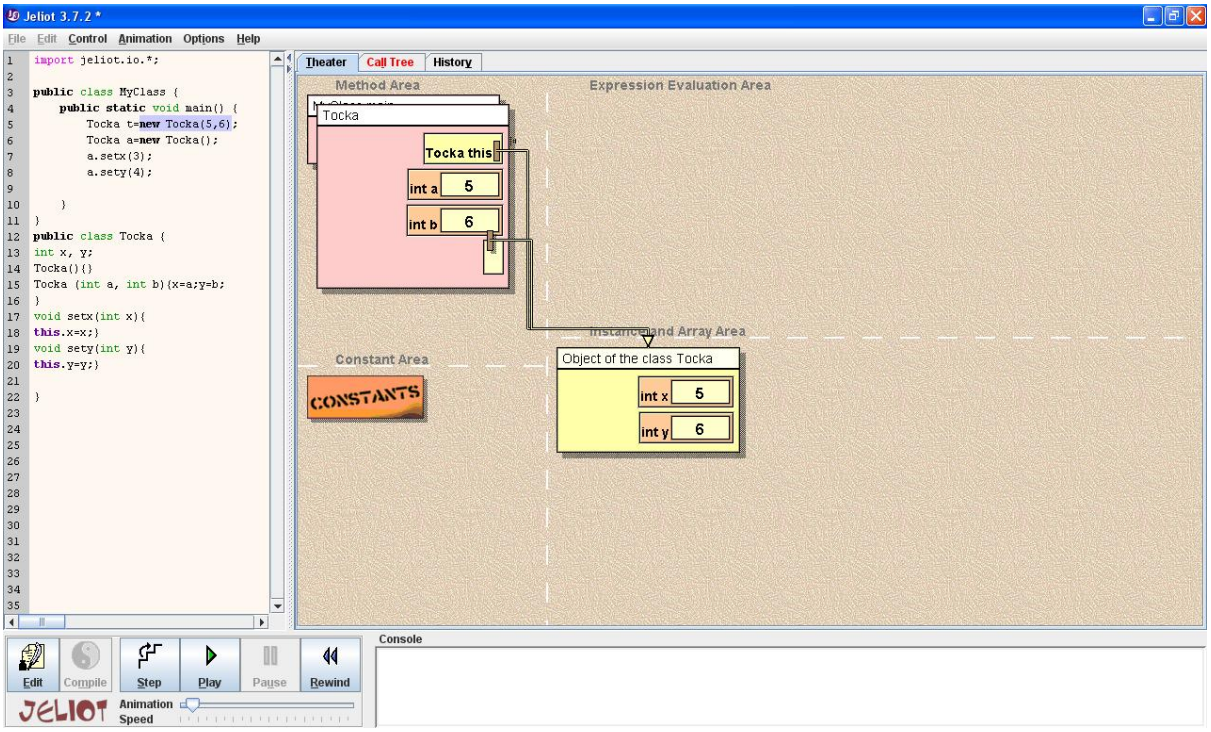


Слика 33. Визуелизациски области кај Jeliot 3.

Во делот со методите се претставени функциите од кодот кои тековно се извршуваат. Тие се претставуваат во вид на правоаголници. Ако програмата има потреба од некакви пресметки и пресметување на изрази тој дел се прави во делот со пресметки. Ако се користат константи тие на екранот се прикажуваат во долниот лев агол, односно областа која е резервирана за константите. И ако се користат низи или инстанци на класи и објекти тие се преставени во долниот десен дел од областа за анимации.

Визуелизациите се формирани така да бидат што поблиски до спецификацијата за јазикот Java. Целиот визуелизациски материјал е поврзан и целосен, така што секој визуелизациски елемент има свое сопствено место од каде се појавува. Дополнително, се пресметуваат сите изрази, така што корисникот нема потреба да претпоставува од каде доаѓа секоја вредност. Исто така, визуелизацијата и програмскиот код се поврзани со нагласување на делот од кодот кој се визуелизира и на тој начин можат да се најдат причините и последиците на секоја визуелизација. Сите изрази и објаснувањата за нив се прикажани блиску една до друга, со цел да се поврзе вредноста на изразот со резултатното објаснување.

За објектно-ориентираното програмско визуелизирање, се користи обележување слично на UML дијаграмите. Објектите се претставуваат како правоаголници кои содржат атрибути со нивните вредности (слика 34).



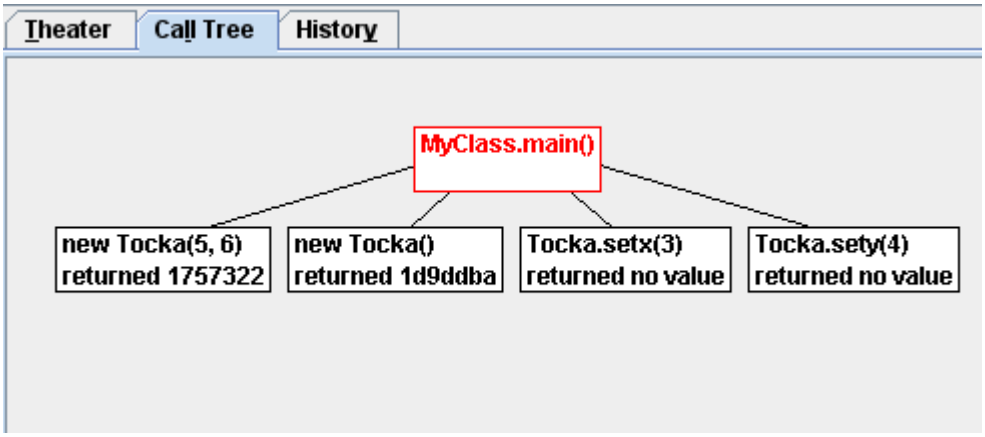
Слика 34. Приказ на објектно оринетирана едноставна програма со Jeliot 3

Референците се претставени како линии што го содржат објектот со соодветната променлива, на тој начин му се овозможува на објектот, во секој момент да има неколку референци.

На слика 35 е претставен пример со едноставна класа за точка со два конструктора. На сликата конкретно е претставен еден објект од класата и јасно се гледа неговата поврзаност со класата Tocka. Објектот од класата како и самата класа се претставени како правоаголници во кои се наоѓаат атрибутите, а референците односно врската меѓу класата и објектот се преставени со линии со стрелки, на чиј крај се наоѓа објектот.

Во делот со визуелизација може многу јасно да се претстават сите концепти на објектно-ориентираното програмирање, како наследување, енкапсулација, полиморфизам и слично.

Во делот на дрвото со повици се прикажуваат функциските повици што тековно се извршуваат и вредностите кои тие ги враќаат во форма на дрво. Ваквото дрво со повици може да се види како посебен прозорец во делот со анимации (Call Tree). А делот во кои течат анимациите се во посебен прозорец –театар. Корисникот може да се префрла од еден во друг поглед во анимацијата според неговата желби и потреби. Пример за дрво со повици на истата програма со класата Tocka е даден на слика 35.



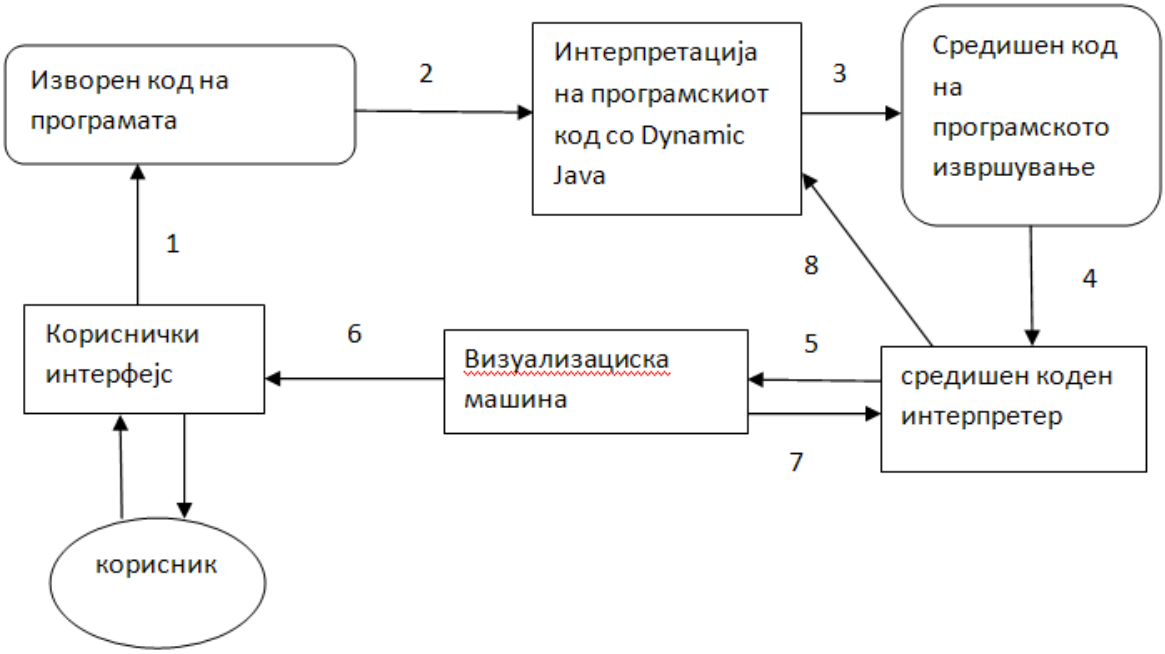
Слика 35. Визуелизација преку дрво на повици кај Jeliot 3.

Jeliot подобро работи со едноставни и кратки програми или програмски сегменти што можат детално да се разработат и полесно да се сфати нивниот начин на работа. Односно, оваа програма може да помогне во разбирање на основните програмски концепти и особено може да биде корисна за почетници во програмирањето. За поголеми програми извршувањето на програмата трае многу долго и корисникот може да го загуби интересот поради чекање за извршување на програмата во целост. Исто така, недостаток кај оваа алатка е тоа што некои својства кои важат за Java уште не се имплементирани кај неа.

Функционалната структура на Jeliot 3 е прикажана на слика 36. Корисникот стапува во интеракција со корисничкиот интерфејс и го прави изворниот код на програмата (чекор 1). Изворниот код се испраќа на Java интерпретерот и се изделува средишниот код (чекори 2 и 3). Се интерпретира средишниот код и се даваат упатства за визуелизациската машина (чекори 4 и 5).

Корисникот може да ја контролира анимацијата со пуштање, паузирање, премотување и следење на анимацијата чекор по чекор (чекор 6).

Исто така, корисникот може да внесе податоци, на пример, цел број или стринг, во програмата која се извршува од страна на интерперетерот (чекори 6, 7 и 8).



Слика 36. Функционална структура кај Jeliot 3.

Jeliot 3 е алатка што лесно се наоѓа, може да се преземе и нејзината инсталација е многу едноставна, што ја прави алатката лесна за користење, посебно во почетните курсеви за програмирање. Интеракцијата на корисникот кај Jeliot 3 е во можноста корисникот да може да го менува кодот и во секој момент да може да забележи каква промена настанува во анимацијата.

5.2 SRec

SRec претставува систем за анимација, кој е специјално наменет за визуелизирање на рекурзии кај алгоритми имплементирани во Java. Претходните верзии на SRec овозможуваат три погледи и тоа: trace view, поглед на контролниот стек, и поглед на рекурзивните дрва. Кај поновите верзии значително е подобрена и олеснета интеракцијата на корисникот со програмата, а во последната верзија се додадени и три доплнителни визуелизации, специјално наменети за раздели и владеј алгоритмите. Денешниот систем е во можност истовремено да прикажува два погледа, со цел подобро испитување на делот што се визуелизира.

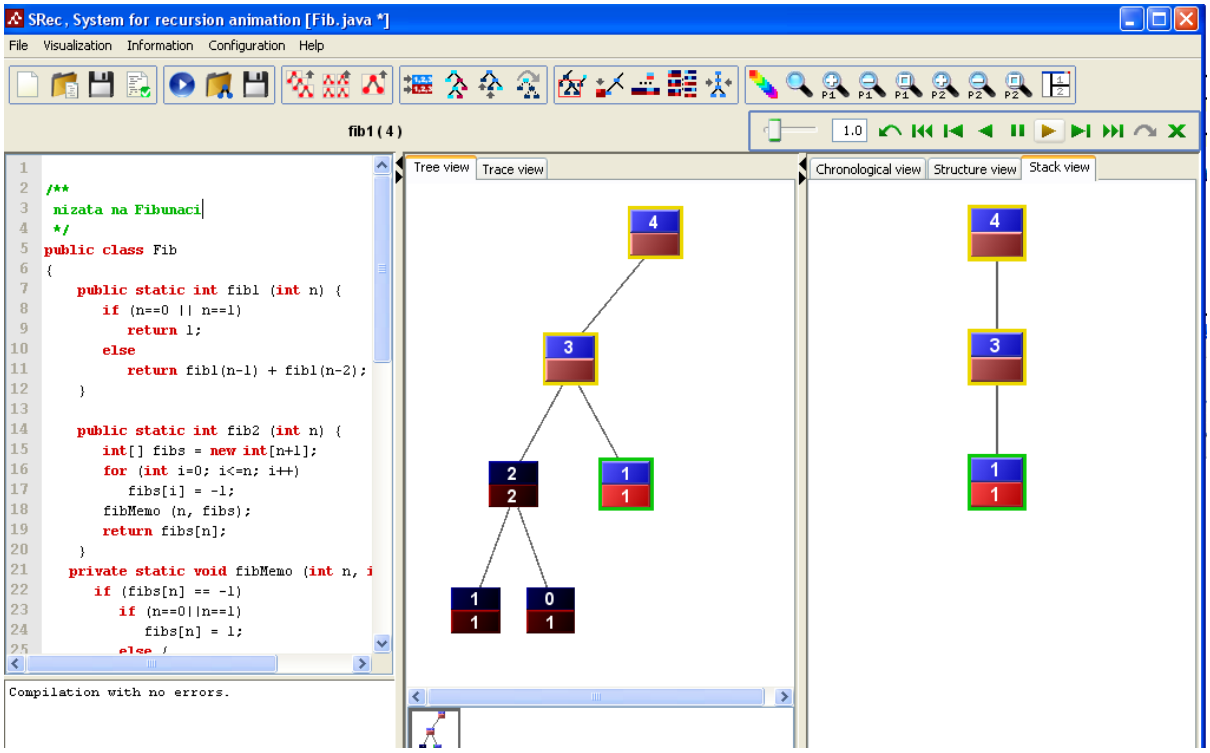
SRec е алатка за анимација на програми што ги генерира анимациите полуавтоматски [93]. Ако датотеката содржи алгоритми од видот раздели и владеј, корисникот нив мора да ги прецизира во дијалог прозорец. Понатаму, тој може да стартува некое повикување на метод и за време на неговото извршување се генерираат траги од соодветните настани.

По завршувањето, корисникот може слободно да стапи во интеракција со неговата визуелизација и анимација, која автоматски се генерира од трагите на настаните. Откако ќе се визуелизира даден настан се оди на следниот и на тој начин се создава динамичка визуелизација, односно анимација.

Оваа алатка е специјално наменета за визуелно прикажување на рекурзијата, како програмска техника што најтешко може да се разбере од страна на почетниците во програмирањето. За програмите и програмските делови кои во себе немаат рекурзивни повици ваквата алатка не е многу корисна. Различните погледи што ги нуди, им дава на корисниците детална претстава во која тие можат чекор по чекор да го следат извршувањето на рекурзивниот повик.

Програмскиот код што се визуелизира може лесно да се менува и уредува што значи дека има добра интеракција корисник-програма. Можат да се користат сите програми кои во себе имаат рекурзивни повици. Дури може да се користат и програмски сегменти или само функции без да се пишува комплетна програма во која е вклучен и main методот. Како пример за илустрација на SRec околината е користен кодот за наоѓање на n-ти член од низата на Фибоначи.

Изгледот на околината на SRec е дадена на слика 37.



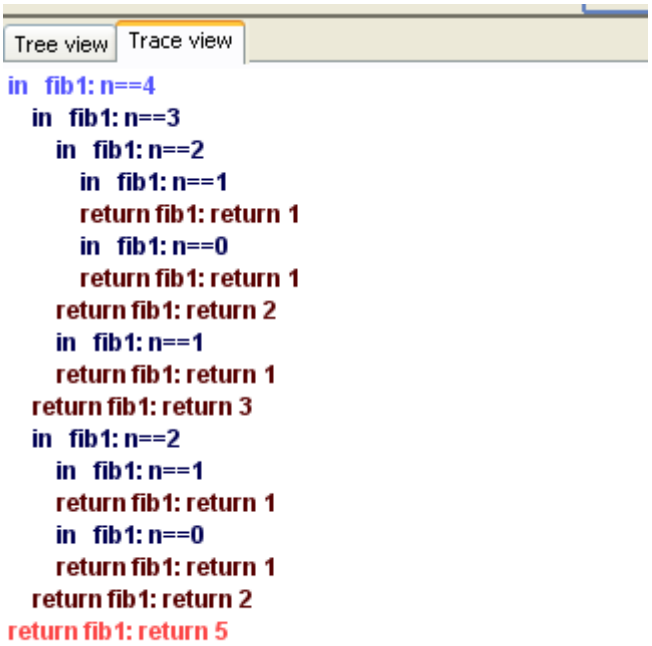
Слика 37. Изглед на околината SRec.

На левата страна се наоѓа делот со кодот. Нумериран е секој ред во кодот и во случај ако се појави грешка, корисникот веднаш знае каде точно настанала таа. Информацијата и објаснувањето за грешките се јавуваат под делот за уредување на кодот.

Програмите што се користат се програми напишани во чист Java код, без потреба од менување на кодот или учење на нови синтаксички правила. Единственото нешто, што може да претставува и олеснување во пишувањето на програмите е тоа што нема потреба од пишување на main методот, а програмата сепак може да се компајлира без грешки.

Делот кој се наоѓа на десно од делот за уредување на кодот е дел од анимацијата што се нуди. Тоа е погледот на рекурзивното дрво. Во овој дел се креира дрво што на едноставен начин го отсликува текот на рекурзијата. Овде чекор по чекор се креира дрвото, а јазелот кој е тековно активен е со зелена боја. Оние делови (јазли), кај кои рекурзивниот повик е завршен се затемнети, а оние делови кои ќе бидат опфатени со рекурзивниот повик се претставени со жолта боја. Листовите од дрвото се основните случаи на рекурзијата.

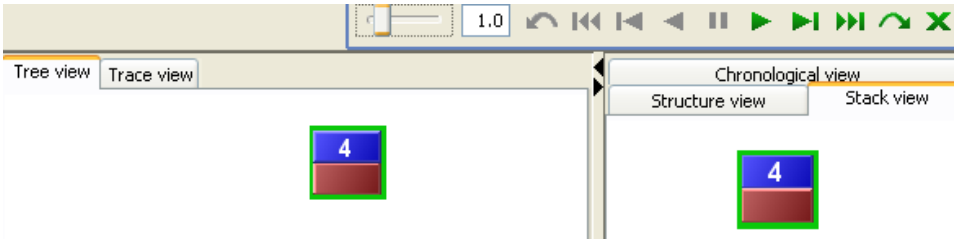
Во овој дел постои и можност на менување на погледот од поглед на рекурзивно дрво во Trece view, кои помагаат да се формира рекурзивното дрво. Trace view е прикажан на слика 38.



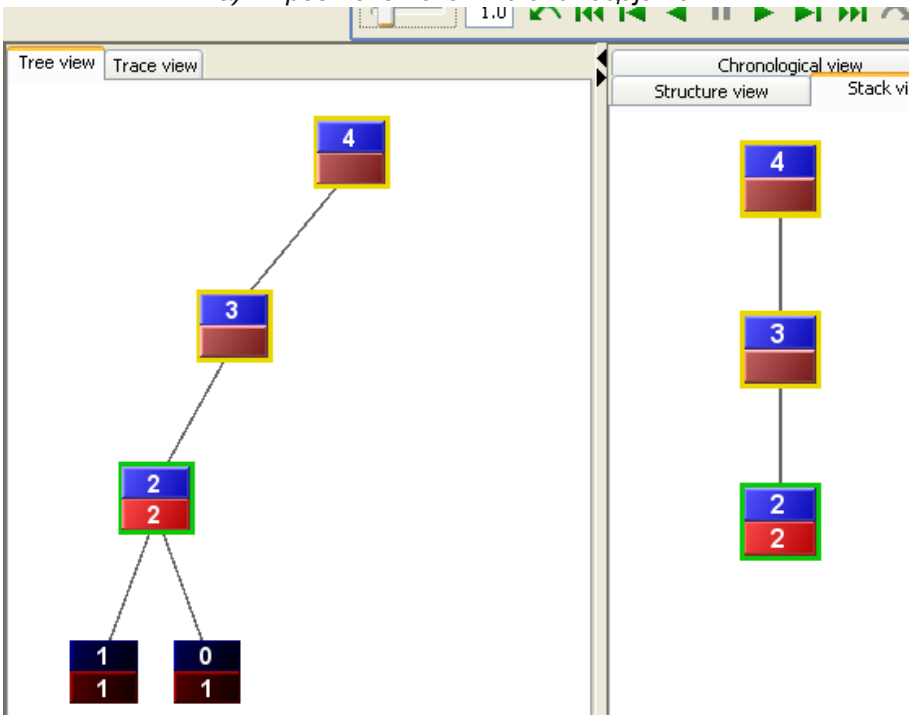
Слика 38. Визуелизација со приказ на Trace view кај SRec.

Другиот поглед кој се нуди со ваквата алатка е поглед на контролниот стек. Во контролниот стек се сместуваат елементите од рекурзивното дрво што треба да се разработат, и јазелот кој е тековно на врвот на стек е јазелот кој

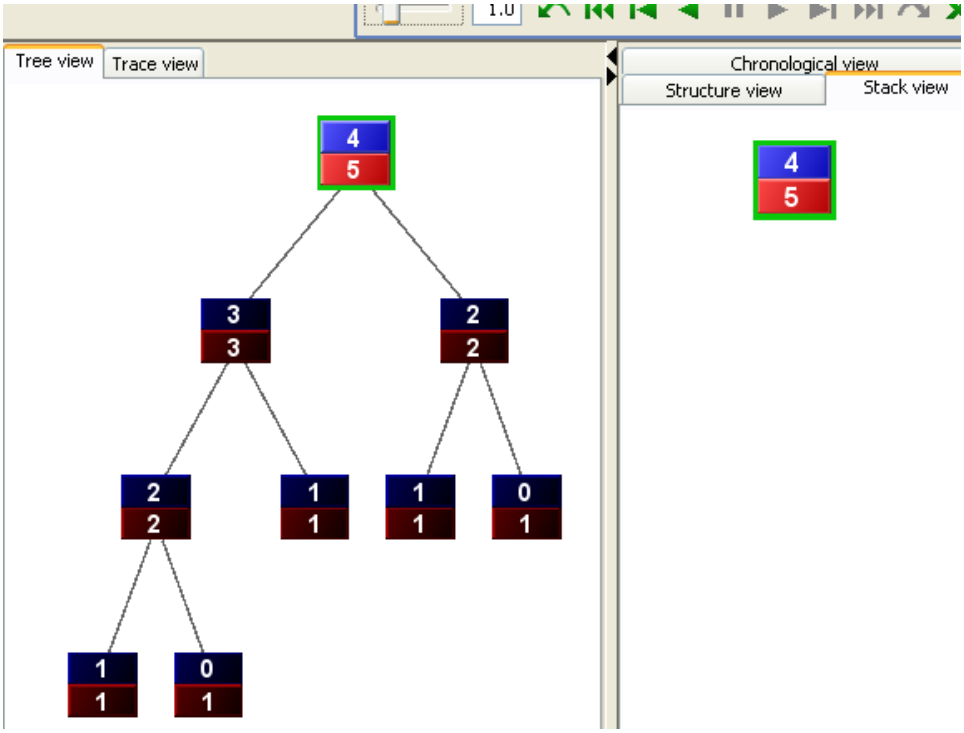
моментално е опфатен со рекурзијата. По завршувањето анимацијата во погледот на контролниот стек останува само еден јазел во кој се наоѓаат влезните аргументи на функцијата и резултатот што го враќа функцијата (Слика 39).



a) Пред почетокот на анимацијата



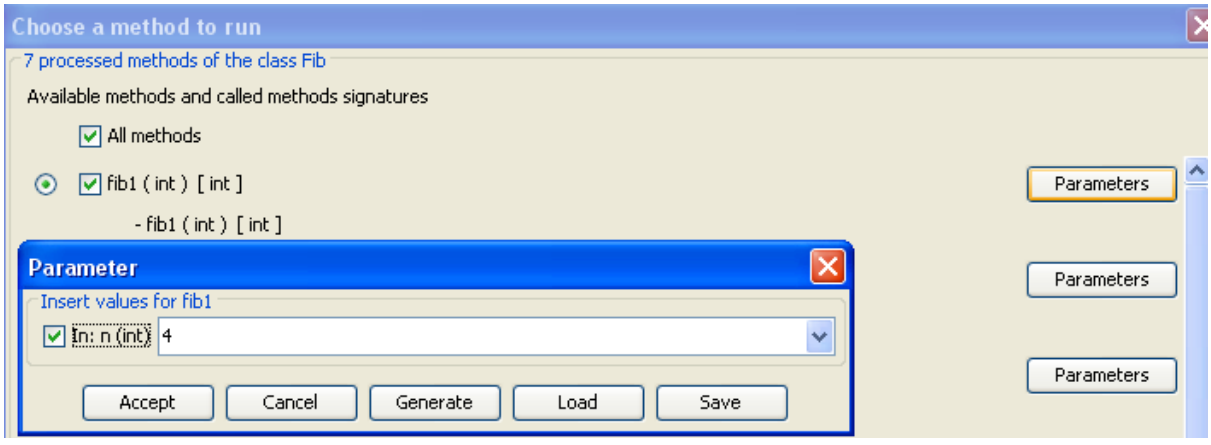
b) Во текот на анимацијата



в) По завршување на анимацијата

Слика 39. Поглед на рекурзивното дрво и контролниот стек при анимација кај SRec

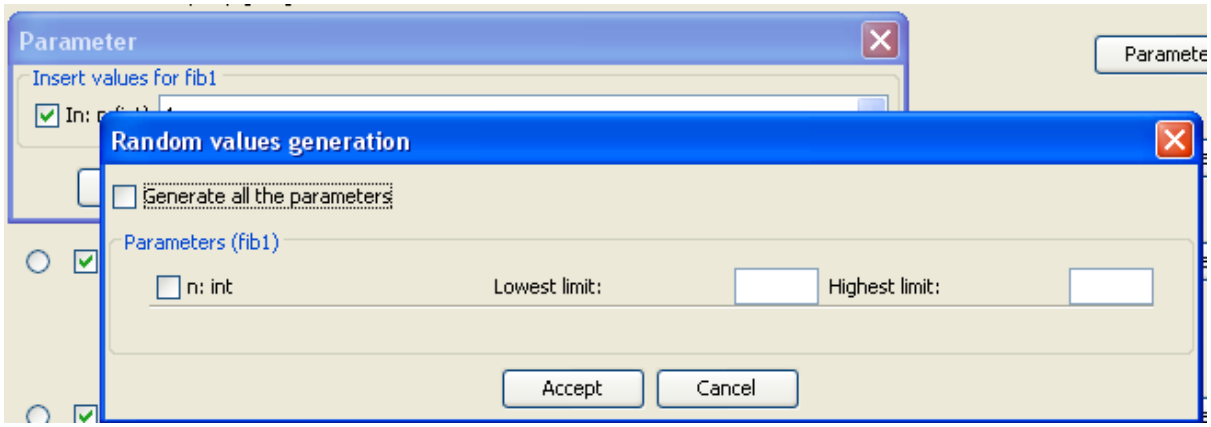
Бидејќи овде нема потреба од пишување на главна функција во која ќе се повикуваат рекурзивните функции со поставување на влезни аргументи, постои посебен дел во кои се внесуваат влезните аргументи. Тоа се прави пред започнување на анимацијата каде пред корисникот се отвора дијалог прозорец во кој се внесуваат влезните аргументи. Или постои можност тие да бидат внесени по случаен избор (Слика 40).



Слика 40. Дијалог прозорец кај SRec за внесување на почетни влезни аргументи на рекурзивната функција.

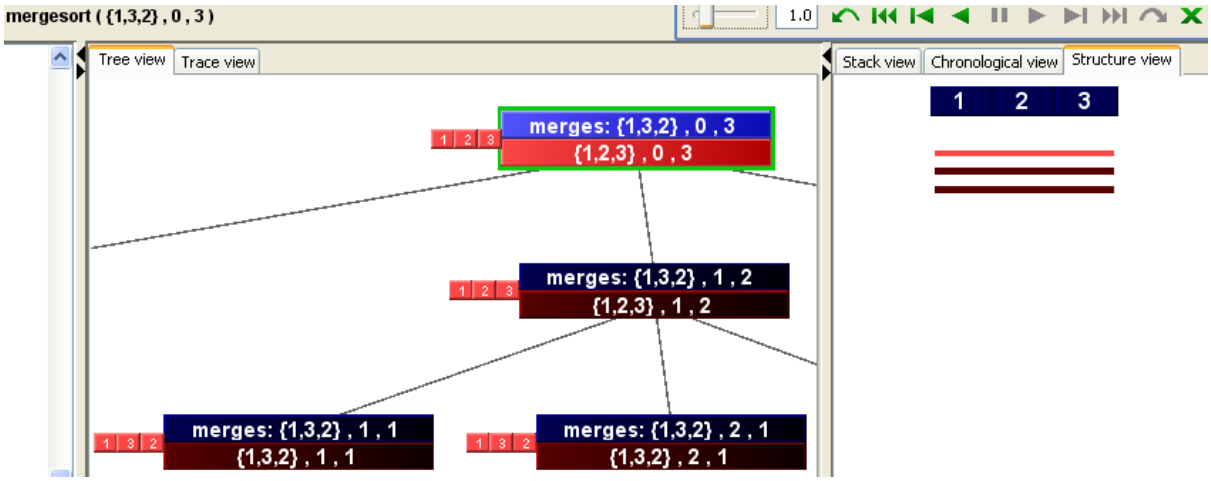
Ако корисникот сака влезните аргументи да бидат внесени по случаен избор се оди на делот Generate. Каде корисникот треба да внесе опсег на

вредности во кои можат да се движат вредностите на влезните аргументи (слика 41).

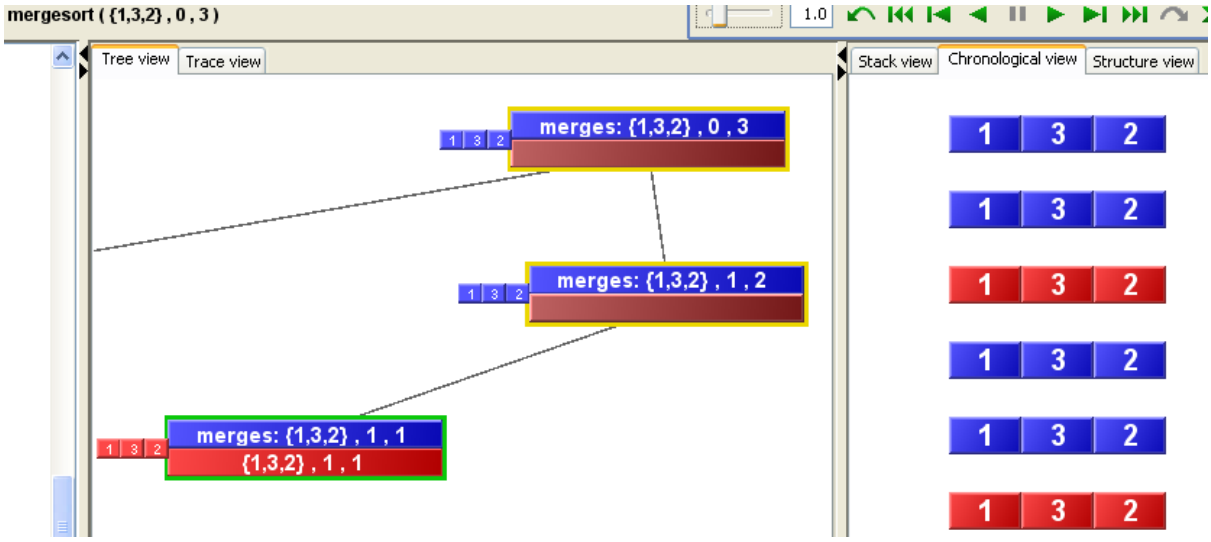


Слика 41. Дијалог прозорец кај SRec за внесување на почетните влезни аргументи на рекурзивната функција по случаен избор.

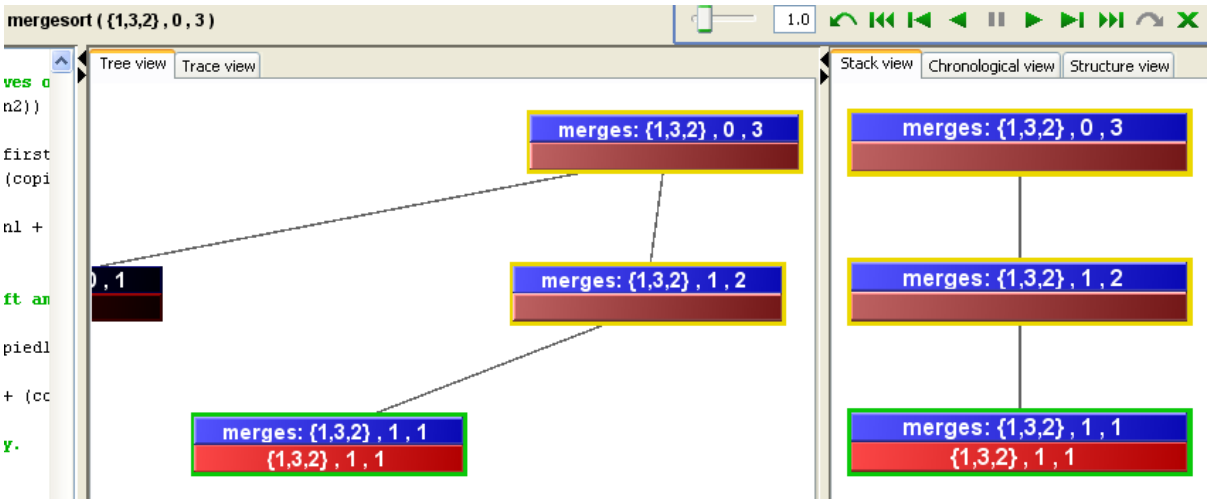
Рекурзијата е застапена и кај алгоритмите од видот „Раздели и владеј“. Ваквиот вид алгоритми SRec веднаш ги препознава, и за нив нуди дополнителни погледи. Како пример за прикажување на ваквиот вид на алгоритми го земавме mergesort (подредување со слевање). На слика 42 е прикажан погледот со рекурзивно дрво и структурниот поглед, а дополнително се нуди и хронолошки поглед (Слика 43) и останува уште погледот на контролниот магацин слика 44.



Слика 42. Поглед со рекурзивно дрво и структурен поглед кај SRec.



Слика 43. Поглед со рекурзивно дрво и хронолошки поглед кај SRec.



Слика 44. Поглед со рекурзивно дрво и поглед на контролниот магацин кај SRec.

Оваа алатка лесно се инсталира. Инсталациска е верзијата 1.1, додека верзијата 1.2 директно се стартува како .jar датотека. Тоа ја прави оваа алатка едноставна за користење. Верзијата 1.1 е и лесно достапна на интернет може лесно да се преземе, а многу да помогне во разбирањето на рекурзивните функции.

Недостаток кај оваа алатка е тоа што работи само со прости видови на податоци, односно не нуди можност влезни или излезни аргументи на рекурзивните функции да бидат инстанци од класи. Доколку се внесе некој голем број за влезен аргумент во функцијата тогаш рекурзивното дрво станува огромно и тешко за следење. Во таков случај прегледноста при визуелизацијата на рекурзијата се губи.

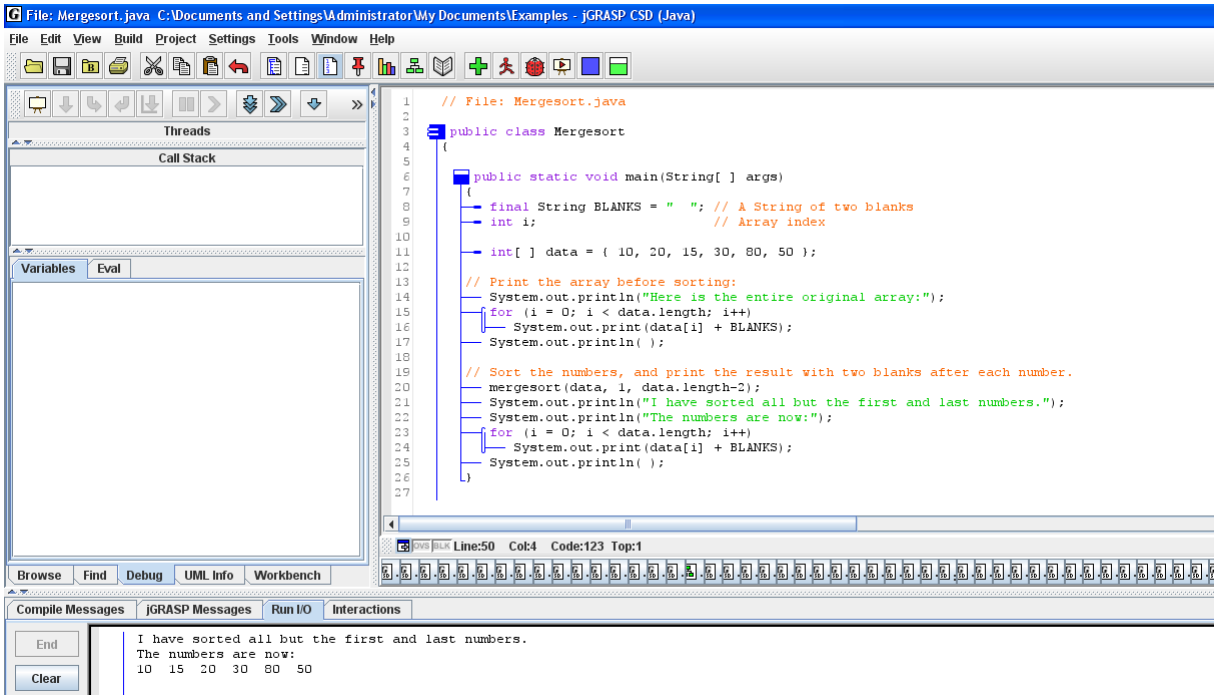
5.3 JGrasp

JGrasp е алатка односно развојна околина што овозможува автоматски да се создаваат софтверските визуелизации а со тоа придонесува за зголемување на разбирливоста на софтверот. Оваа околина е имплементирана во Java и нејзина предност е тоа што може да работи на сите платформи со помош на Java виртуелната машина. jGRASP може да компајлира и извршува програми напишани во Java, C, C++, jGRASP Objective-C, Python, Ada, и VHDL. За сите овие програмски јазици создава контролно структурни дијаграми (Control Structure Diagrams - CSD). А исто така, оваа околина овозможува создавања на Complexity Profile Graphs (CPGs) за Java и Ada и UML класни дијаграми за Java. Само за програмите напишани во Java, оваа околина овозможува создавање на динамички погледи на сложените структурни елементи кои можат да се видат при процесот на дебагирање.

Дебагерот работи само за Java програми. Погледите кои се менуваат со извршување на секој чекор од програмата, можат да ги распознаваат објектите на некои од податочните структури, како на пример: магацини, редови, поврзани листи, двојно поврзани листи, бинарни дрва или хаш табели и да ги претставаат на различен начин во текот на процесот на дебагирање. За Java програмите постои и посебен поглед (Canvas view) во кој може да се следи анимацијата на податочните структури.

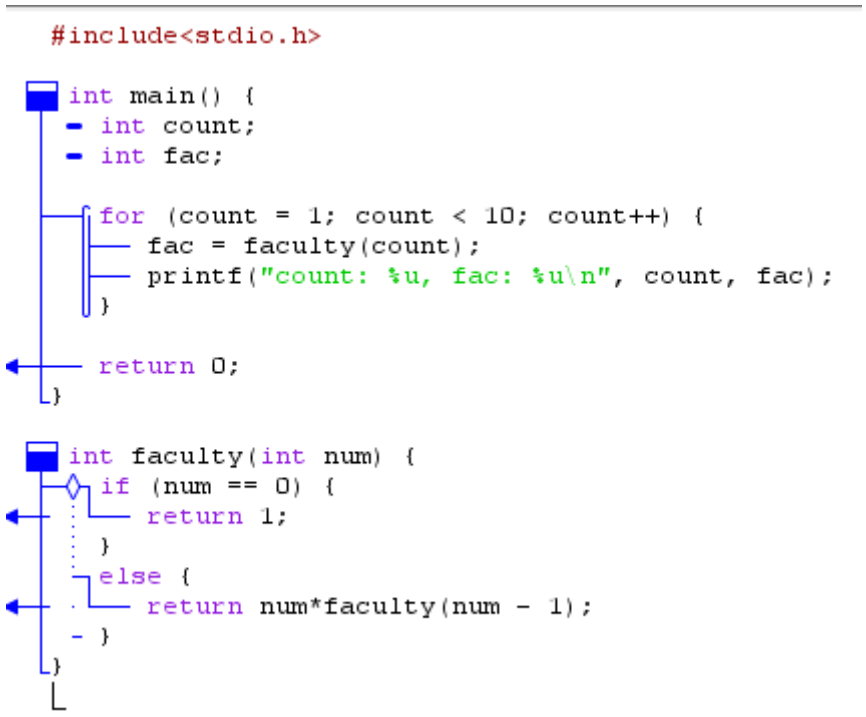
Програмите кои се визуелизираат треба да бидат напишани во чист програмски јазик, на пример, чист Java или C/C++, без потреба од додавање на дополнителни елементи. Кај jGrasp, како и при стандардно компајлирање и извршување на Java или C/C++ програмите тие мора да имаат main() метод.

На слика 45 е прикажан изгледот на околината jGrasp.



Слика 45. Изглед на околината jGrasp.

Околината има дел за уредување на кодот. Кодот кој се пишува автоматски се структурира и уште веднаш нуди голема прегледност. Тековниот пример е за алгоритамот mergesort (подредување со слевање) во Java. Но, ваквиот изглед програмата го нуди и за програми напишани во C/C++. Тоа што овој уредувач на код го прави подобар од другите е неговата визуелна прегледност. Сите програмски елементи се претставени на различен начин, со помош на што може лесно да се гледа каде почнува и каде завршува класата, кои се локални а кои се глобални променливи, во кој дел има циклус, што враќа дадена функција како излезен аргумент, и каде има разгранувања во програмата. На слика 46 е претставен код во C прикажан во околината на jGrasp.



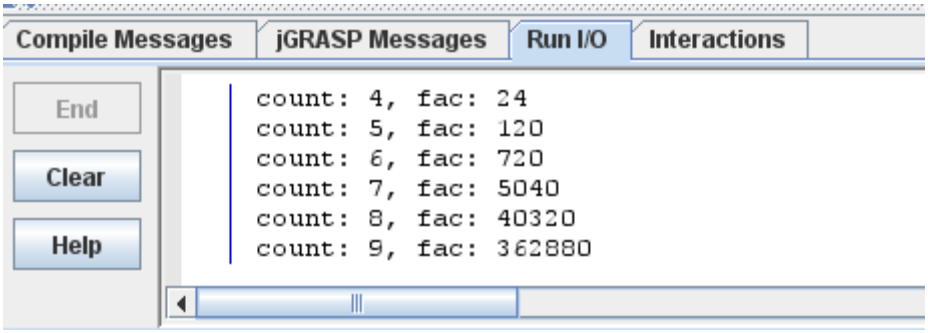
Слика 46. Визуелизација на код напишан во програмски јазик C со помош на jGrasp.

Класите се претставуваат со помош на ознаката , функциите односно методите со помош на , а променливите со знакот . Ако на дадено место во програмата има структура на разгранување таа се прикажува со , а циклусите се прикажуваат со знакот . Кога функција враќа вредност тогаш тој дел од кодот е прикажан со . На ваков начин лесно можат да се забележат вгнездените циклуси, јасно може да се види која е областа на важење на дадена променлива или каде е телото на дадена функција. На ваков начин ќе бидат избегнати можните синтаксички грешки или ако постојат полесно можат да бидат отстранети. Но, овој дел ги визуелизира програмите само статички односно ваквата визуелизација на кодот не се менува во времето на извршување на програмата.

Интеракцијата на програмата со корисникот е на високо ниво. На корисникот му се нуди можност во секој момент да го менува кодот и да одлучува за изгледот на анимациите. Исто така, ако програмата има потреба од корисничко внесување на податоци, тоа корисникот може лесно да го направи во текот на извршувањето на програмата.

Околината има посебен дел односно рамка (дел со пораки – конзола), во која се внесуваат вредности преку тастатура од страна на корисникот, ако програмата има потреба од нив, и во истиот дел се дава крајниот резултат при

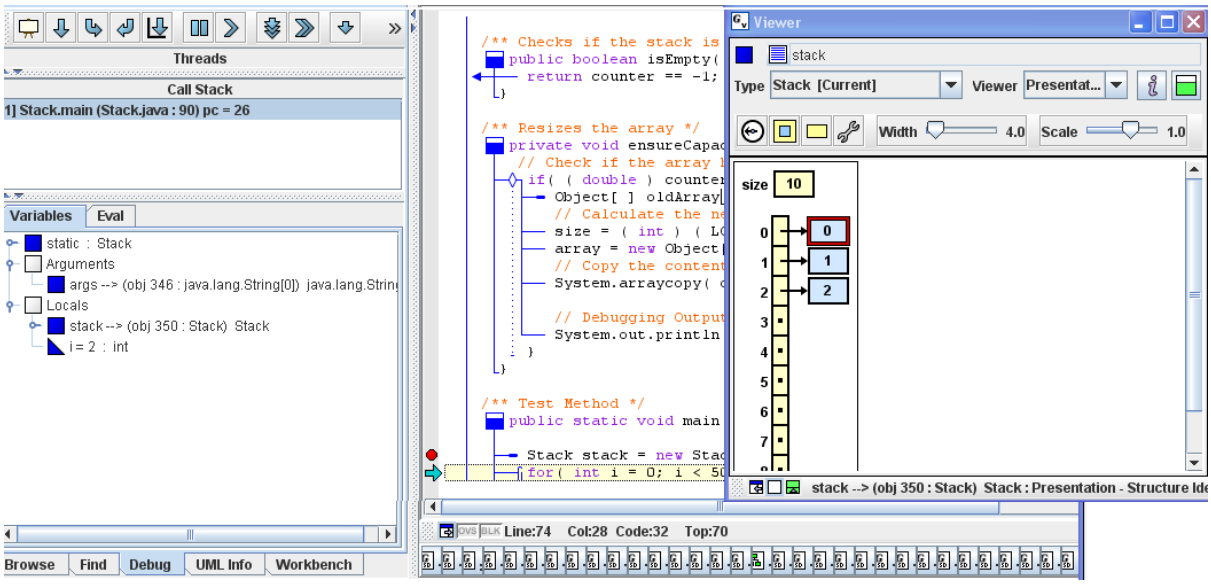
извршувањето на програмата. Таа рамка најчесто се наоѓа во долниот дел од програмата. Ако во програмата се јави некаква грешка или исклучок, објаснувањето на грешката се дава исто така, во овој дел со пораки (слика 46).



Слика 47. Конзола –дел со пораки кај jGrasp.

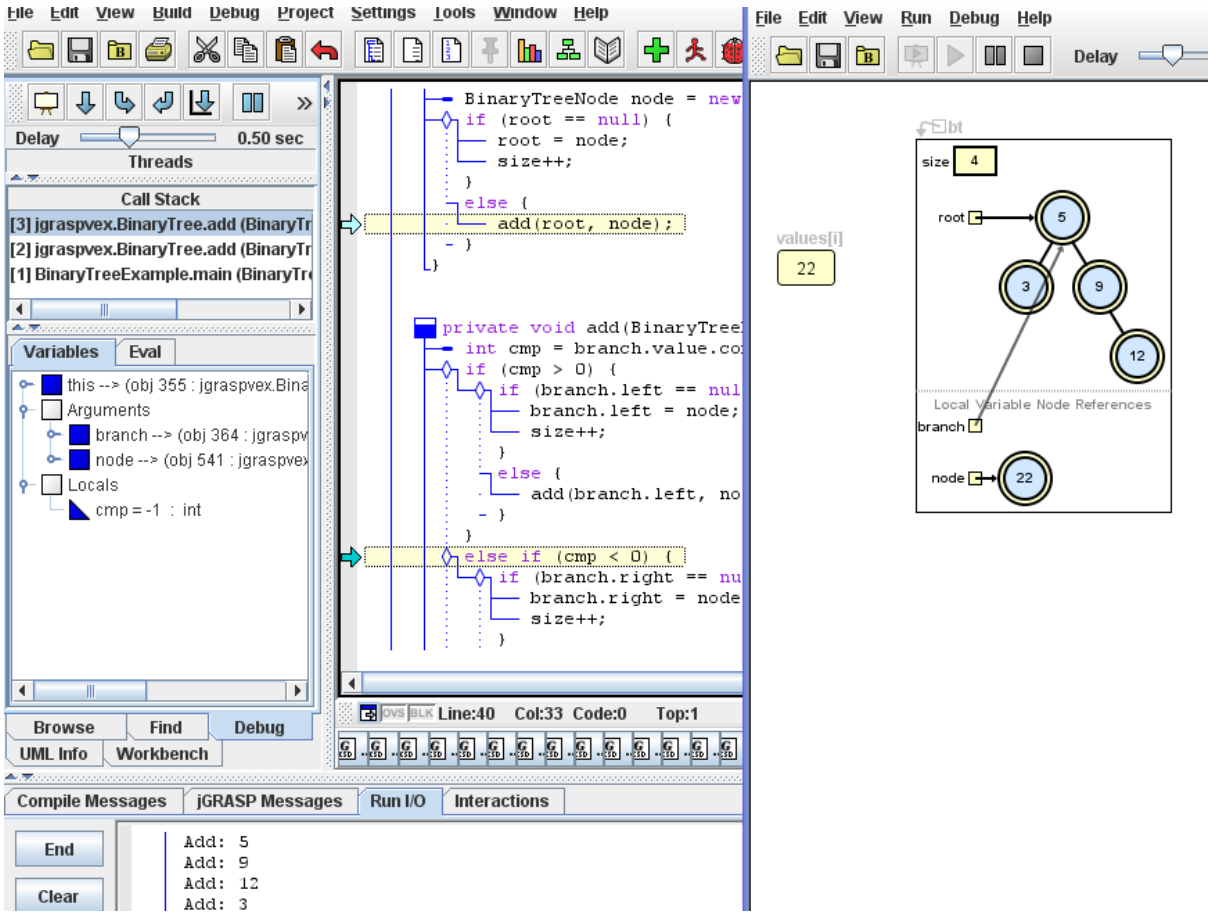
jGrasp нуди динамичка визуелизација само на програмите напишани во Java. Визуелицацијата се добива со дебагирање, каде во секој чекор се гледаат измените кои настануваат во структурата во текот на извршувањето на програмата. Динамичката визуелизација посебно ги опфаќа податочните структури, го дава начинот на градење на структурата и нејзиното менување во текот на извршувањето на програмата.

Податочните структури подобро можат да се научат ако на некој начин бидат претставени со помош на слики. На таков начин можат јасно да се разберат основните елементи на секоја податочна структура како и операциите кои се извршуваат врз нив. На пример, креирање на структурата, додавање на елемент, бришење на елемент, пребарување и сл. Бидејќи во оваа околина има вградено механизам кој ги разликува основните податочни структури, ако во програмата се користат некои од нив, тие лесно можат да се распознаат и да бидат преставени во форма на слики. На слика 48 е претставен поглед за програма што реализира податочна структура стек (магацин).



Слика 48. Поглед на визуелизација во jGrasp на програма која реализира податочна структура магацин.

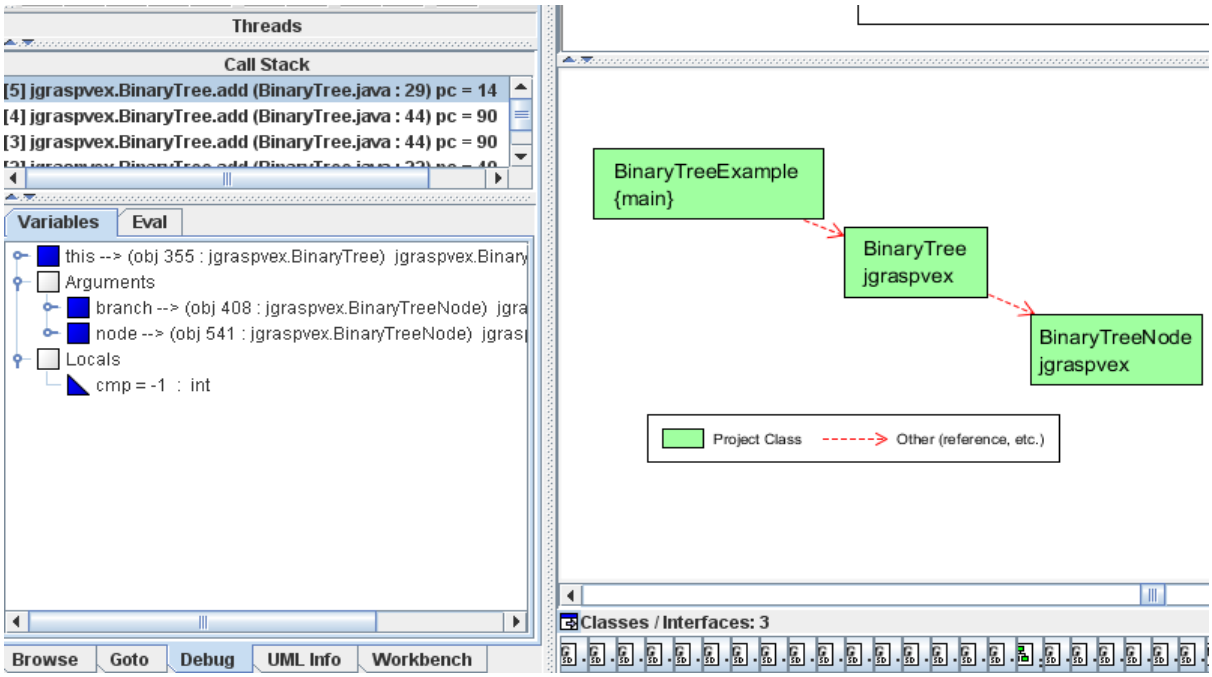
Изгледот на сликите корисникот може да го менува, а му се нуди избор од неколку опции. Корисникот може да ја дебагира програмата чекор по чекор и да гледа во секој момент кој дел од кодот на каков начин влијае врз измената на изгледот на сликата што му се претставува. На слика 49 е дадена програма за градење на бинарни дрва и нејзината анимација во jGrasp.



Слика 49. Поглед од анимација во jGrasp на програма која реализира податочна структура бинарни дрва.

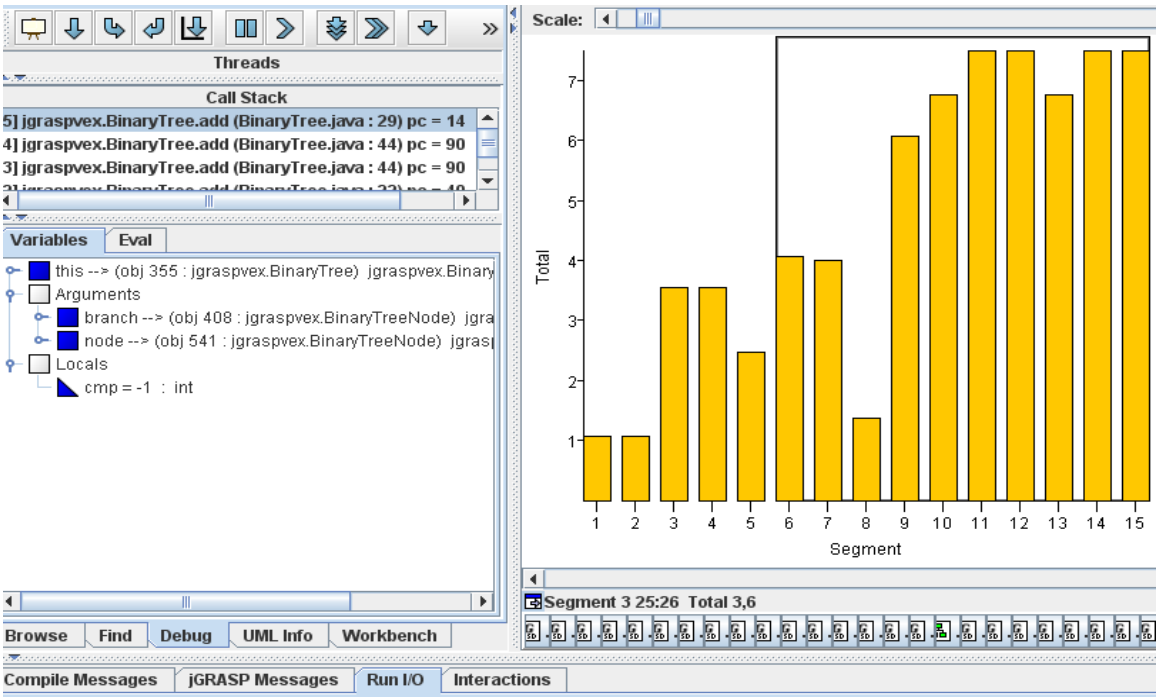
Во секоја програма корисникот може да избере кои елементи да бидат прикажани во делот со анимации и чии промени на вредности ќе ги следи. Значи, корисникот сам одлучува кој дел од програмата сака да го следи со анимации.

jGrasp нуди и друг вид на статичка визуелизација на објектно-ориентираните програми каде може автоматски да генерира UML класни дијаграми. UML класниот дијаграм за претходната задача со бинарно дрво е прикажан на слика 50.



Слика 50. Визуелизација со UML класен дијаграм на програма со бинарни дрва во jGrasp.

Друг вид на визуелизација на кодот што го нуди оваа околина се графиконите на сложеност (Complexity Profile Graph-CPG). Изгледот на графиконот на сложеност изгенериран за истиот код е даден на слика 51 . Но и овој начин на визуелизација е статичен и не се менува во текот на извршување на кодот.



Слика 51. Визуелизација со приказ на графикон на сложеност во jGrasp на програма со бинарни дрва

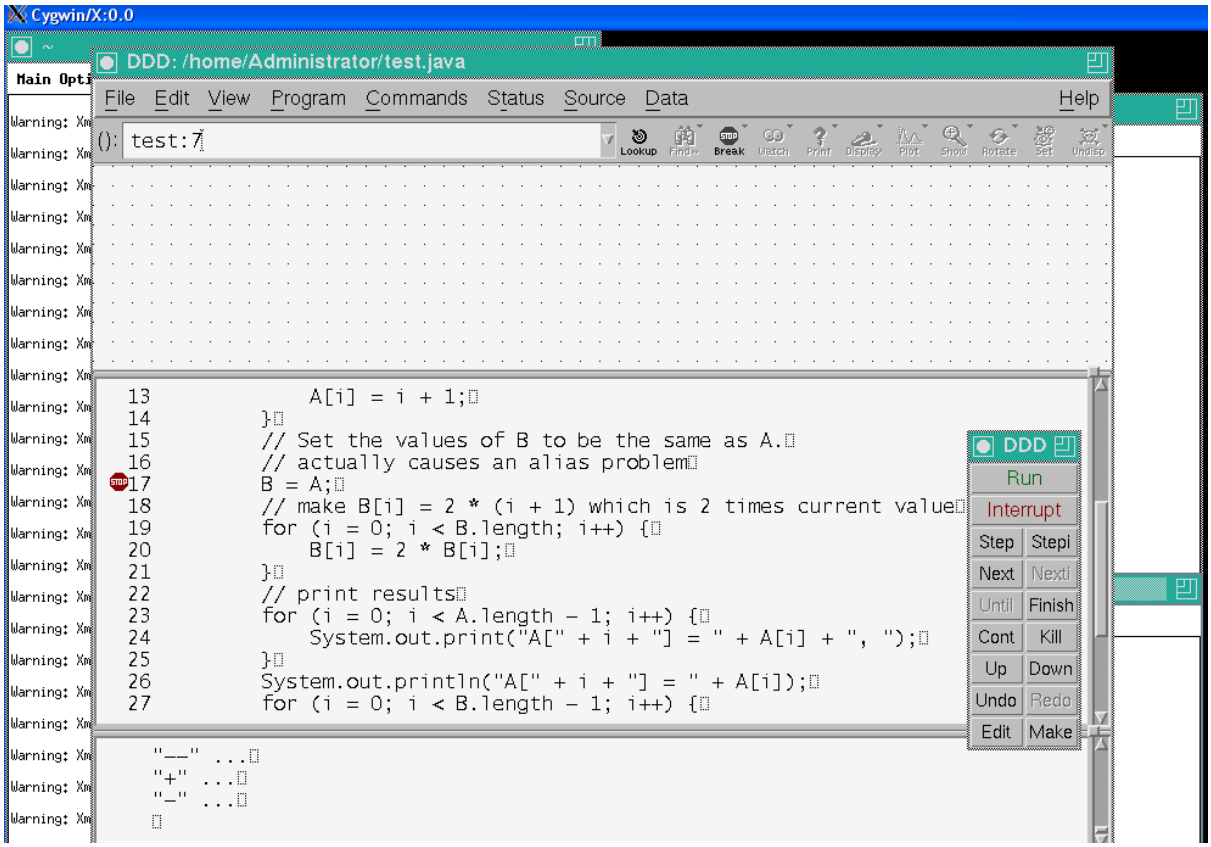
Развојната околина jGrasp може лесно да се најде на интернет и да се преземе. Бидејќи може да работи на секоја платформа, корисникот без разлика на кој оперативен систем работи може на лесен начин да ја инсталира и да ја користи. Статичките визуелизации лесно се генерираат, особено контролните структурни дијаграми кои автоматски се генерираат уште кога корисникот почнува да го пишува кодот во делот за уредување на код. Останатите статички визуелизации се генерираат со користење на делот со алатки, без некои тешкотии. Во алатникот лесно можат да се најдат алатките за генерирање на UML дијаграми или графиконите на комплексност.

Извршувањето и компајлирањето на програмите е исто така едноставно, особено ако програмите се пишувани во Java. Но, ако програмите се пишувани во C/C++, потребен е соодветен компајлер кој треба да се инсталира и да се додаде во PATH системската променлива, а потоа во програмата да се избере соодветниот компајлер. Во спротивен случај C/C++ програмите нема да можат да се компајлираат и извршуваат.

Динамичките визуелизации се добиваат преку дебагирањето на програмата, а за тоа корисникот интерфејс не е доволно усовршен за корисникот да може без читање на упатството, да ги генерира.

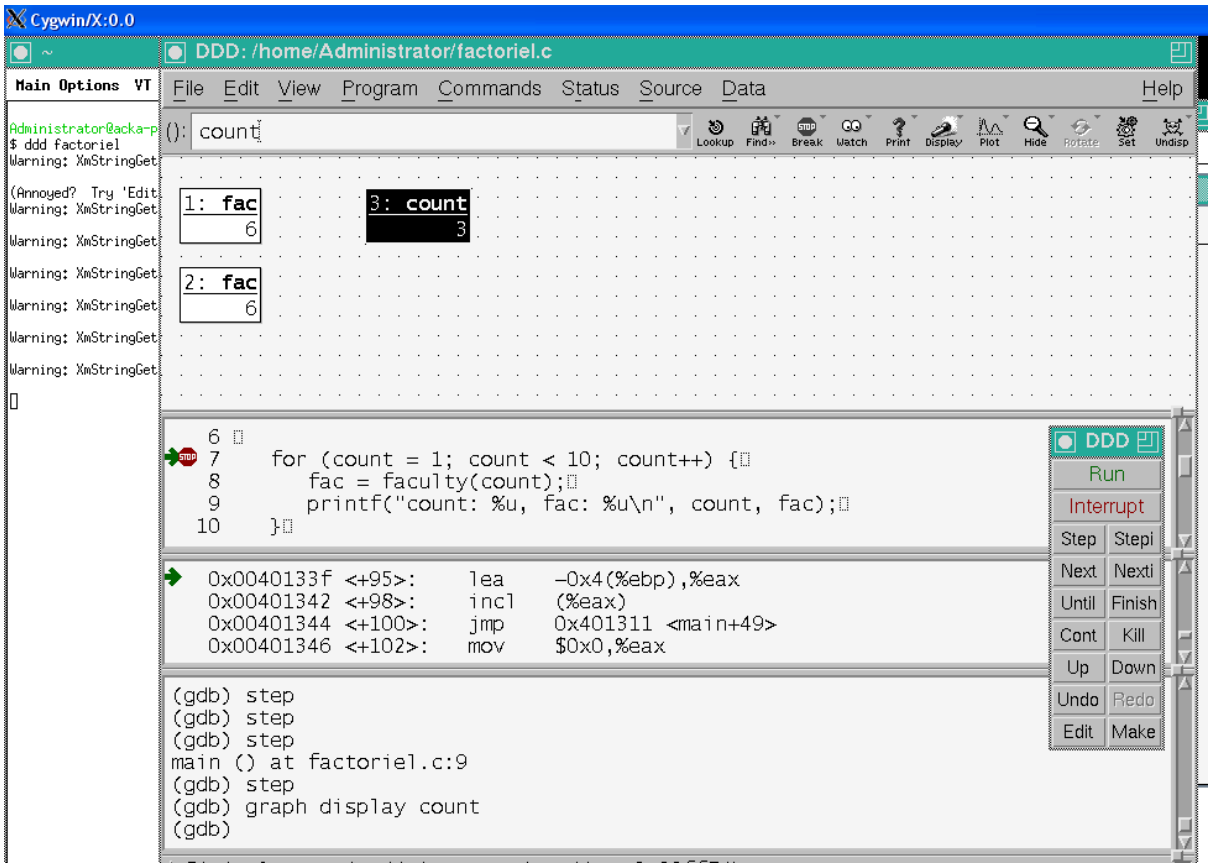
5.4 DDD

DDD (Data Display Debugger) претставува алатка за дебагирање на програмите напишани во C/C++ Java или Perl. Поточно DDD претставува графички интерфејс за дебагерите на командна линија како што се GDB, DBX, WDB, Ladebug, JDB, XDB, Perl, bash и Python. DDD ги претставува податочните структури во форма на графови. Тоа што го прави различен овој дебагер од останатите слични на него основни дебагери како gdb, е тоа што освен основните операции на дебагер овозможува и графичко прикажување на податочните структури. Особено е погоден за преставување на покажувачите кои се карактеристични за C/C++ програмите. Покажувачите, особено двојните покажувачи се дел кој тешко може да се разбере и мисловно да се прикаже. Затоа визуелното претставување на покажувачите може да помогне за нивно подобро разбирање. На слика 52 е прикажан графичкиот интерфејс на DDD за едноставна програма напишана во Java, а на слика 53 е даден графичкиот интерфејс на DDD за едноставна програма напишана во C.



Слика 52. Графички интерфејс на DDD за едноставна програма напишана во Java.

DDD нуди повеќе погледи: дел во кој се наоѓа кодот, дел со пораки на кој се прикажува излезот и се опишуваат настанатите грешки, дел за прикажување на податоците и дополнително кај програмите напишани во C/C++ има дел со машински код. Делот во кој е прикажан кодот не овозможува негово директно уредување. Ако е потребно кодот да се менува тоа се прави во некој обичен уредувач на текст.



Слика 53. Графички интерфејс на DDD за едноставна програма напишана во C.

Во делот каде е прикажан кодот можат да се додаваат точки на прекин , текот на програмата се следи ред по ред а тековниот ред е означен со . Секоја од инструкциите е прикажана во машински код.

Графичкиот интерфејс на оваа алатка потсетува на DOS околина, можни се интеракции со корисникот, но е оневозможено менување на кодот во самата програма. Корисникот може да го користи делот со алатки да го стартува извршувањето на програмата, да постави точки на прекин, и да го следи извршувањето чекор по чекор. Корисникот може на едноставен начин да прикаже елемент од програмата во делот за приказ на податоци, со означување на елементот и избирање на display.

Ако настане грешка, особено ако има пристап до недозволена меморија, грешката се јавува во делот за прикажување на пораки, во тој дел се јавува и крајниот излез на програмата.

Програмите кои можат да се визуелизираат на ваков начин се едноставни конзолни апликации напишани во чист програмски јазик (на пример C/C++ или Java). Кај сите програми, да можат да се извршуваат задолжителна е главната функција.

DDD е лесно достапен на интернет и може лесно да се преземе. Но, неговата инсталација претставува проблем посебно ако не се инсталира на Linux системи. Ако се инсталира на Windows систем потребно е да се користи Cygwin (колекција од алатки кои овозможуваат изглед и чувство на Linux околина под Windows). Бидејќи DDD не е самостоен софтвер за негова инсталација и работа потребни се повеќе пакети, на пример: gcc, X11, make, java, jdb и др. И покрај сите инсталирани пакети може да јави проблем при стартување кој се јавува поради проблем во X11 системите. Инсталацијата и стартувањето на DDD во целост бара доста време, што претставува недостаток за оваа програма.

6. Визуелизација на програми и анализа на алатките

6.1 Визуелизација на покажувачи

Податочните структури листи, и единично поврзаните и двојно поврзаните, кога се реализираат во програмски јазик C или C++ мора да се користат покажувачи. Покажувачите се еден од најтешките елементи од програмирањето, особено за почетниците. Па затоа изучувањето на овие податочни структури, особено операциите со двојно поврзаните листи претставуваат најголем проблем. Нивното изучување би било полесно ако постои начин секој чекор од програмата што користи податочна структура листа, да се претстави визуелно. Со визуелното претставување потребно е да се прикаже и делот кон кој даден јазел од листата покажува, односно да се визуелизираат и претходникот и следбеникот на даден јазел.

DDD овозможува визуелно претставување на податочните структури - листи на начин со кој јасно се гледа како јазлите од листата се поврзани во меморискиот простор, односно како се претставени во меморијата.

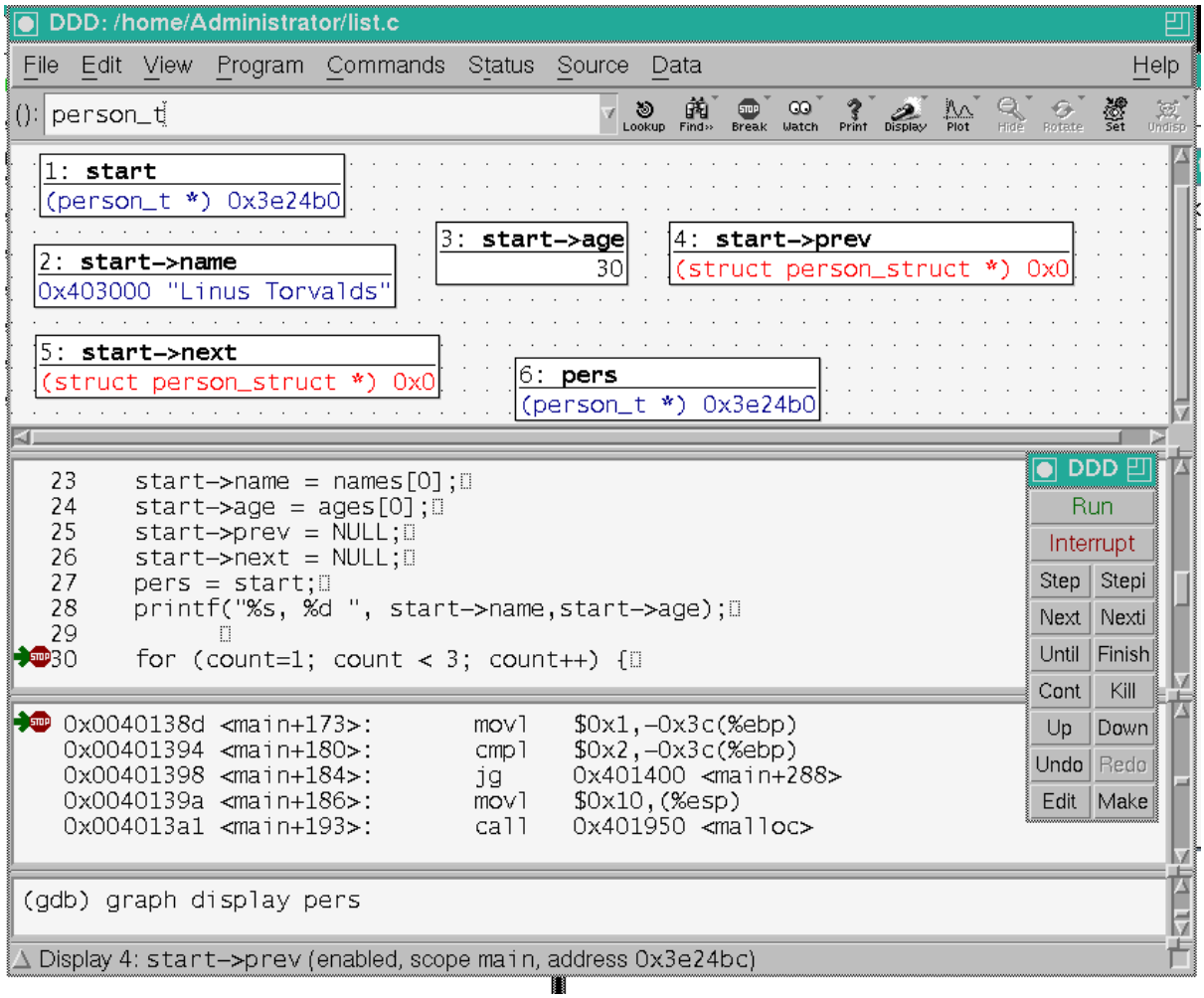
Со цел да се види визуелизацијата за структурата двојно поврзани листи, што ја нуди програмата DDD, одбравме едноставен код пуштен преку DDD за илустрација.

Кодот е следен:

```
#include<stdio.h>

int main() {
   typedef struct person_struct {
      /* Data elements *7
      char* name;
      int age;

      /* Link elements */
      struct person_struct *next;
      struct person_struct *prev;
   } person_t;

   person_t *start;
   person_t *pers;
   person_t *temp;

   char *names[] = {"Linus Torvalds", "Alan Cox", "Rik van Riel"};
```

```
    int ages[] = {30, 31, 32};
    int count;  /* Temporary counter */

    start = (person_t*)malloc(sizeof(person_t));
    start->name = names[0];
    start->age = ages[0];
    start->prev = NULL;
    start->next = NULL;
    pers = start;
    printf("%s, %d ", start->name,start->age);

    for (count=1; count < 3; count++) {
       temp = (person_t*)malloc(sizeof(person_t));
       temp->name = names[count];
       temp->age = ages[count];
       pers->next = temp;
       temp->prev = pers;
       pers = temp;
       printf("%s, %d ", temp->name,temp->age);
    }
    temp->next = NULL;

    printf("Data structure created\n");
    return 0;
}
```

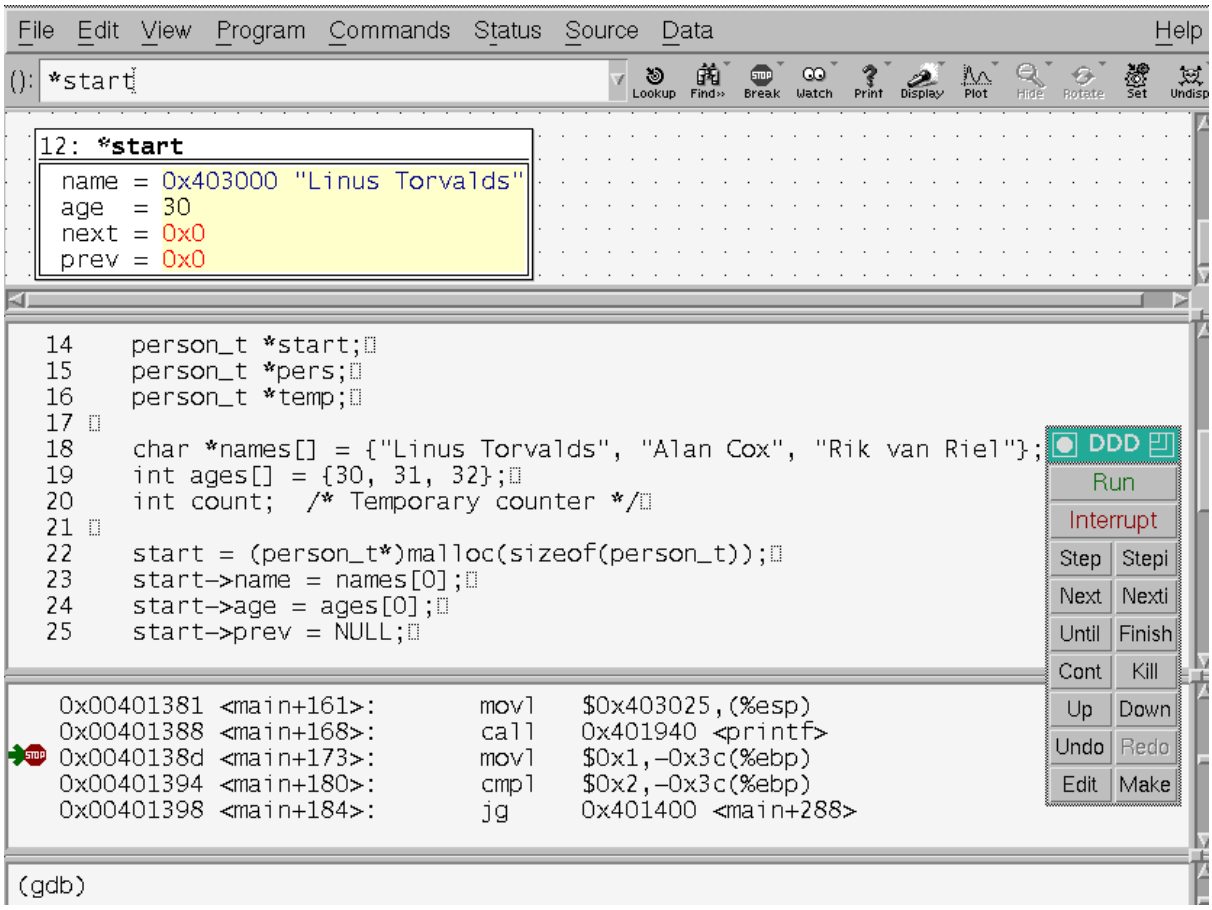
Овој код прво мора да се компајлира, па да се отвори преку DDD. За таа цел користевме терминал на Cygwin. Таму прво ја компајлираме програмата со наредбата `gcc -c -o list list.c` . Откако програмата е преведена треба да се стартува DDD. Тоа се прави со наредбата `startx`. Тогаш се отвара нов терминал, и во тој терминал ја пишуваме наредбата `ddd list` да ја отвориме програмата во DDD. Приказот на отворената програма може да се види на слика 54.



Слика 54. Приказ од визуелизација со DDD на програма напишана во C за листи.

При отворањето на програмата може да се гледа кодот, но тој не може директно да се менува. Линиите од кодот се нумерирани со цел полесно да може да се најде некоја евентуална грешка. Ставаме точка на прекин на 30-та линија и го стартуваме извршувањето на програмата преку Run.

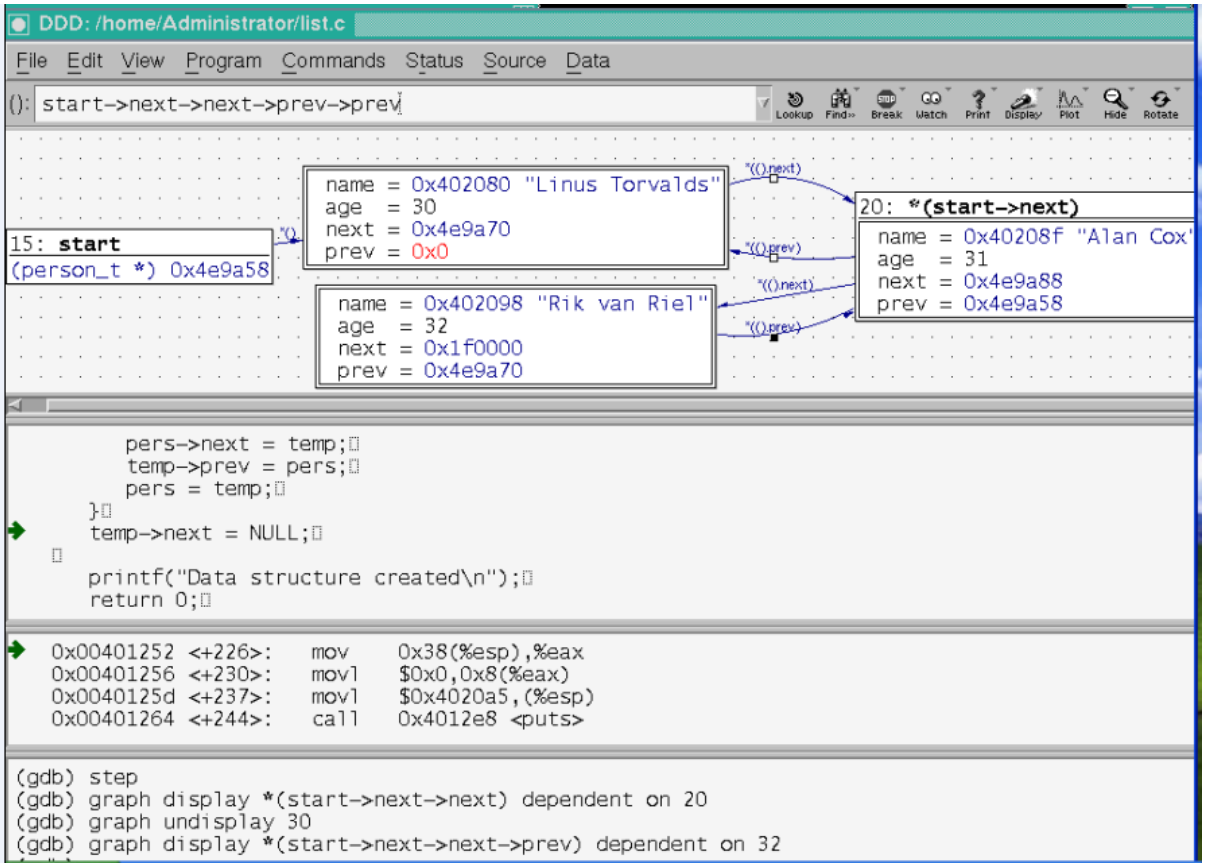
Во делот за приказ на податоците можеме да ги претставиме потребните елементи. На почетокот тоа се елементите на јазелот start (името, возраста и неговиот претходник и следбеник, како и адресата на која се наоѓа). Елементите можеме да ги прикажеме поединечно или споени во еден јазел ако го прикажеме *start (слика 55).



Слика 55. Приказ од визуелизација на јазел со DDD.

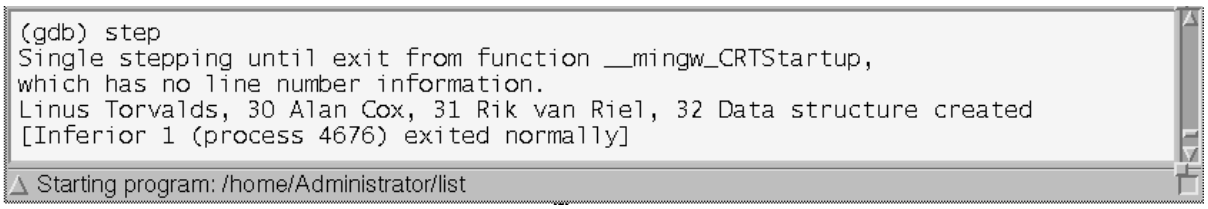
Во најдолниот дел од прозорецот можеме да ги следиме тековните активности што ги правиме при дебагирањето (слика 56). Додека опционално ни се нуди можност да го гледаме и претставувањето на машинскиот код каде секоја наредба е претставена со машинска инструкција како и инструкција во асемблерски јазик со адреси прикажани во хексадецимален формат.

По неколку последователни чекори во дебагирањето се добива податочната структура - двојно поврзана листа во делот на податочен приказ. Приказот е детален и секој јазел е претставен со неговата мемориска локација и со врски е поврзан со неговите соседни јазли (Слика 56).



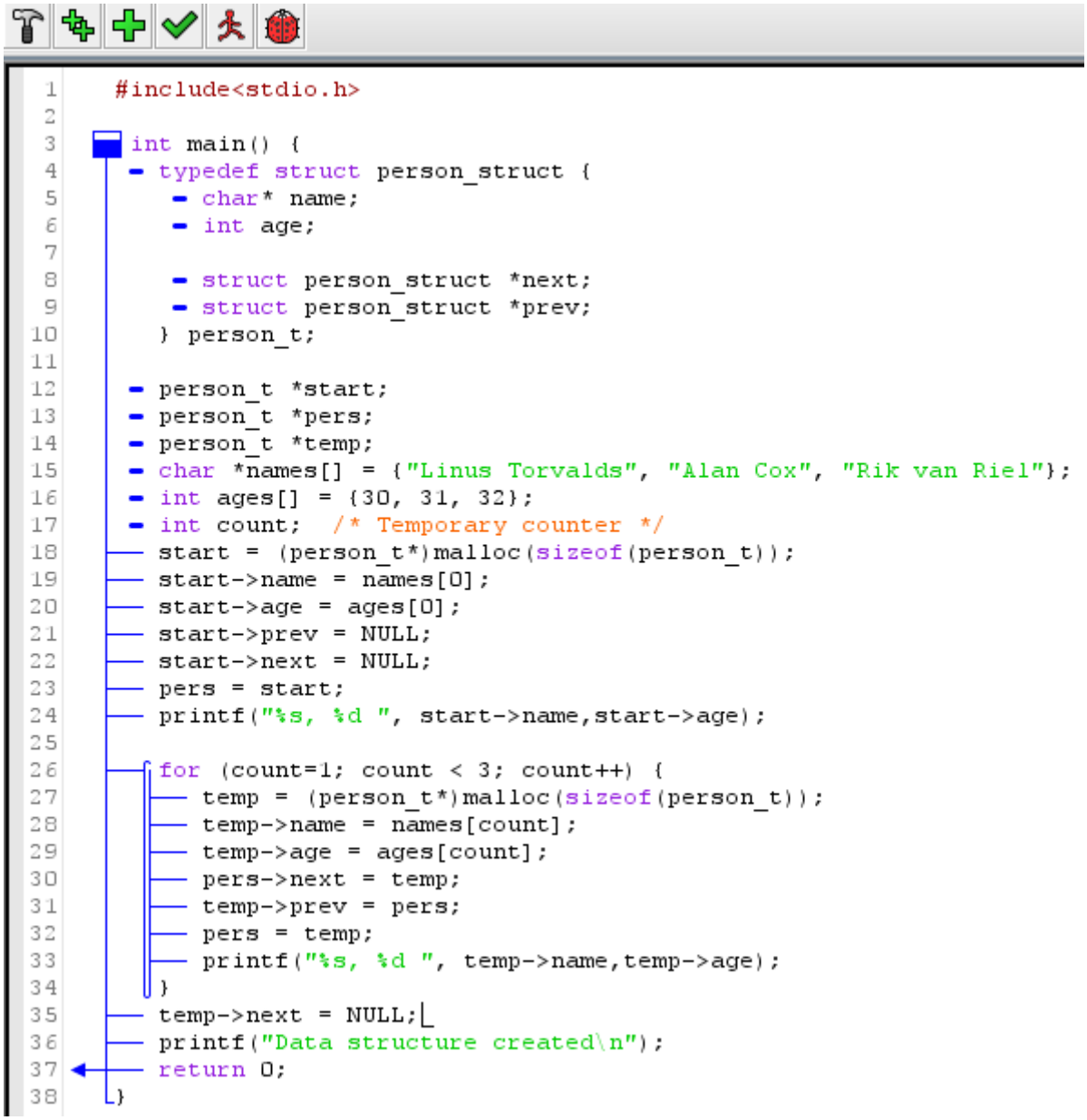
Слика 56. Визуелизација на двојно поврзана листа со приказ на претходник и следбеник на секој јазел

Кога се завршува со дебагирањето во делот со пораки се прикажува резултатот, односно излезот од програмата (слика 57).



Слика 57. Приказ на конзола кај DDD со испечатениот резултат од програмата со листи.

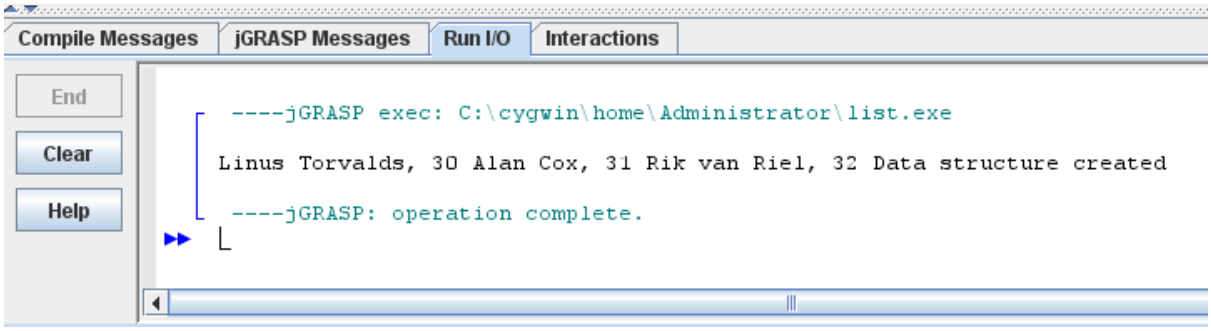
Истиот код напишан во програмски јазик C го отвораме со jGrasp. И таму го компајлираме и го пуштаме да се изврши. Бидејќи кај jGrasp, дебагерот е развиен само за програми напишани во Java, а не и за C, делот со дебагирање и претставување на структурите овде не може да се види. Со помош на jGrasp овој код за двојно поврзани листи може само да се визуелизира статички со помош на контролно структурен дијаграм (CSD) (слика 58). Овде кодот може лесно да се менува и има многу подобро изразена структурираност. Лесно можат да се забележат сите програмски елементи: структурата, променливите покажувачите, for циклусот, низите.



```c
#include<stdio.h>

int main() {
  typedef struct person_struct {
     char* name;
     int age;

     struct person_struct *next;
     struct person_struct *prev;
    } person_t;

  person_t *start;
  person_t *pers;
  person_t *temp;
  char *names[] = {"Linus Torvalds", "Alan Cox", "Rik van Riel"};
  int ages[] = {30, 31, 32};
  int count;  /* Temporary counter */
  start = (person_t*)malloc(sizeof(person_t));
  start->name = names[0];
  start->age = ages[0];
  start->prev = NULL;
  start->next = NULL;
  pers = start;
  printf("%s, %d ", start->name,start->age);

  for (count=1; count < 3; count++) {
     temp = (person_t*)malloc(sizeof(person_t));
     temp->name = names[count];
     temp->age = ages[count];
     pers->next = temp;
     temp->prev = pers;
     pers = temp;
     printf("%s, %d ", temp->name,temp->age);
  }
  temp->next = NULL;
  printf("Data structure created\n");
  return 0;
}
```

Слика 58. Визуелизација со jGrasp на програма со листи напишана во C.

По извршувањето на програмата во делот за пораки исто како и кај DDD и овде се печати резултатот (слика 59).



Слика 59. Приказ на конзола кај jGrasp со испечатениот резултат од програмата со листи.

Другите алатки, Jeliot и SRec, воопшто не подржуваат C и C++ програми затоа споредбата на визуелизациите за оваа програма останува меѓу овие две алатки.

6.2 Визуелизација на рекурзии

Рекурзијата е програмска техника што овозможува дадена функција сама себе да се повикува, т.е. во функцијата има повик кон истата функција. Рекурзијата може да се користи и за замена на некој циклус, во случај кога тоа ќе помогне проблемот да се поедностави.

Секоја рекурзија треба да ги има следните карактеристики:

– Да постои едноставен основен случај за кој постои решение и враќа вредност како резултат. Понекогаш постои повеќе од еден основен случај.

– Да постои начин со кој проблемот ќе се сведе поблизу до основниот случај. Односно, да се издвои дел од проблемот кој ќе даде поедноставен проблем.

– Да постои рекурзивен повик кој едноставниот проблем го проследува повторно кон функцијата.

Прво, потребно е да се препознаат основните случаи. Основен случај е наједноставниот можен проблем на функцијата. Треба да се внимава преку основниот случај да се врати точна вредност како резултат на функцијата. Во тој случај рекурзивната функција се состои од `if - else` изрази, каде основниот случај враќа една вредност, а не основните случаи повторно ја повикуваат истата функција но, со помало множество на податоци како влезни параметри.

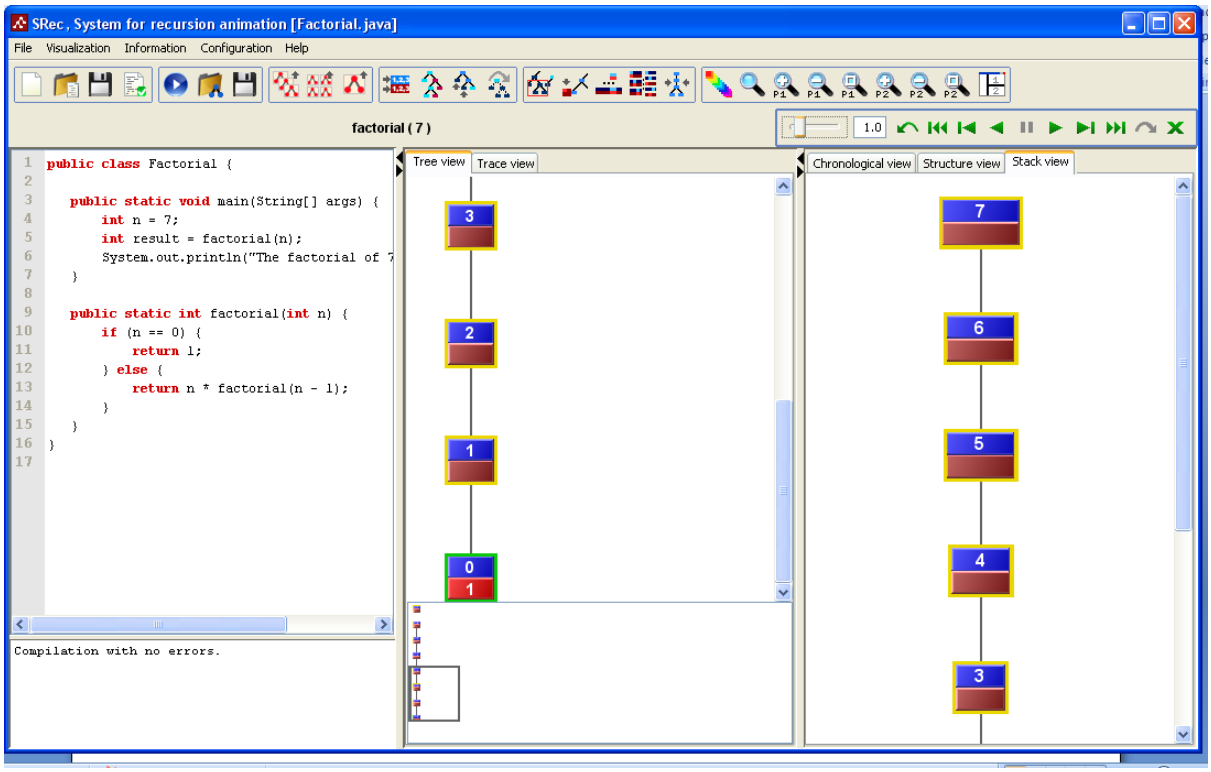
Начинот на работа на рекурзијата е тежок за сфаќање, и затоа добра визуелна претстава може да биде клучна помош во правење чекор до нејзино разбирање.

Работата на рекурзијата најдобро може да се разбере со визуелизирање на едноставен пример на рекурзивна функција. Како пример за рекурзија ќе ја претставиме функцијата за наоѓање на факториел на даден број.

Кодот на програмата за факториел, реализиран во Java, е следен:

```
public class Factorial {

 public static void main(String[] args) {
     int n = 7;
     int result = factorial(n);
     System.out.println("The factorial of 7 is " + result);
 }

 public static int factorial(int n) {
     if (n == 0) {
         return 1;
     } else {
         return n * factorial(n - 1);
     }
 }
}
```

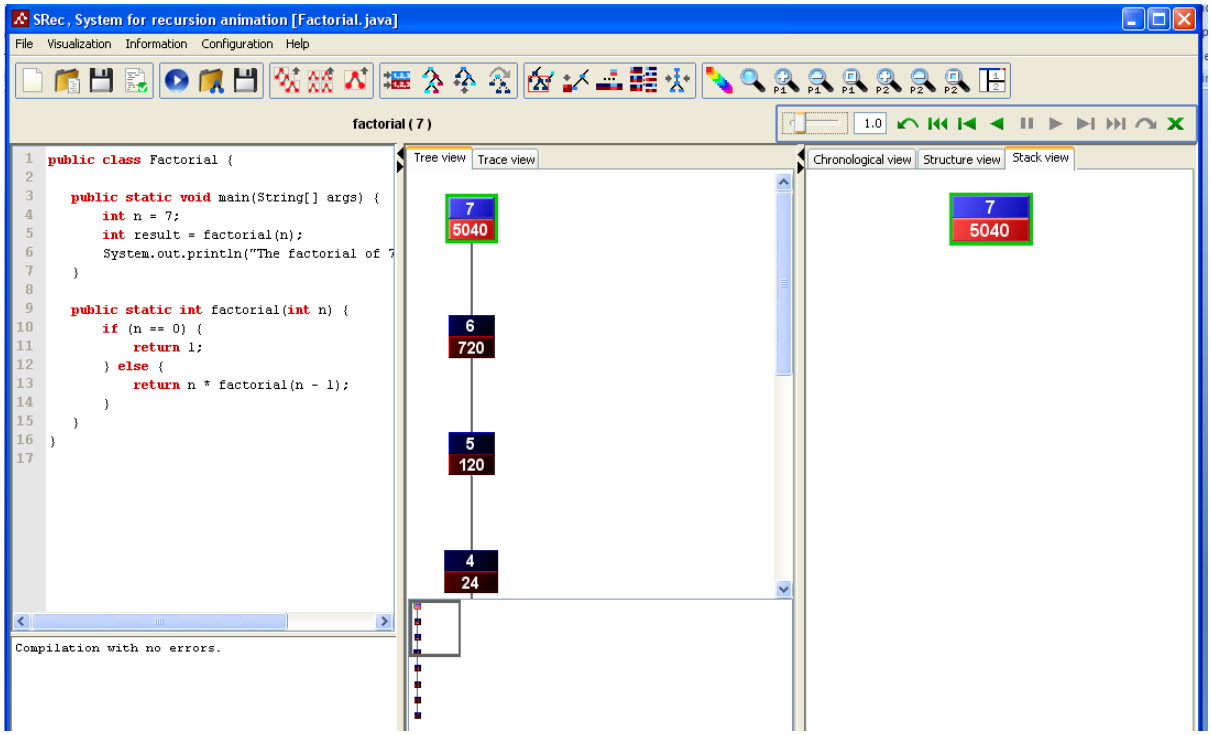
Прво, кодот за наоѓање на факториел на даден број ќе го визуелизираме со помош на програмата SRec, која е специјално наменета за претставување на рекурзии. На слика 60 е прикажан дел од анимацијата на оваа програма.



Слика 60. Визуелизација на рекурзија со SRec на програма за наоѓање на факториел на даден број – доаѓање до основниот случај од рекурзијата.(Приказ на поглед на дрво и контролен магацин).

Со оваа програма се претставува дрвото на градење на рекурзијата. Последниот јазол од дрвото е основниот случај односно factorial(0). Секој јазол е претставен во форма на четириаголник, горниот дел од четириаголникот е влезот а долниот излезот на функцијата. Дрвото се разгранува надолу, чекор по чекор, додека не се стигне до основниот случај (слика 59).

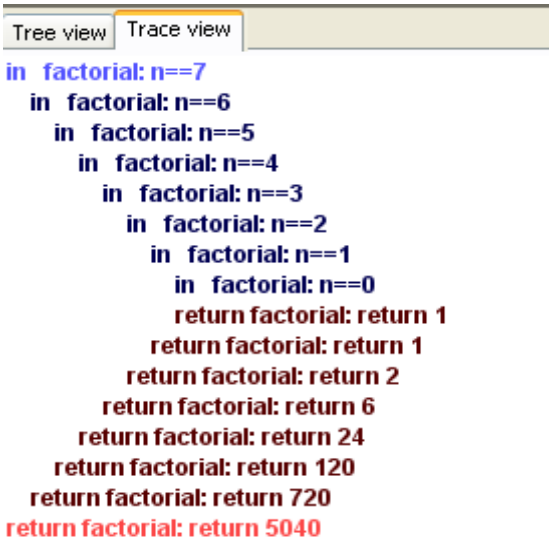
Откако е достигнат основниот случај се пополнува излезот на секој јазел во дрвото, одејќи во спротивна насока, односно нагоре. Анимацијата завршува кога ќе се потполни излезот на почетниот јазол (слика 61).



Слика 61. Визуелизација на рекурзија со SRec на програма за наоѓање на факториел на даден број – завршување на рекурзијата.(Приказ на поглед на дрво и контролен магацин).

Со помош на ваквата визуелизација може да се гледа секој чекор од извршувањето на рекурзивната функција, а со тоа да се разбере и начинот на нејзината работа.

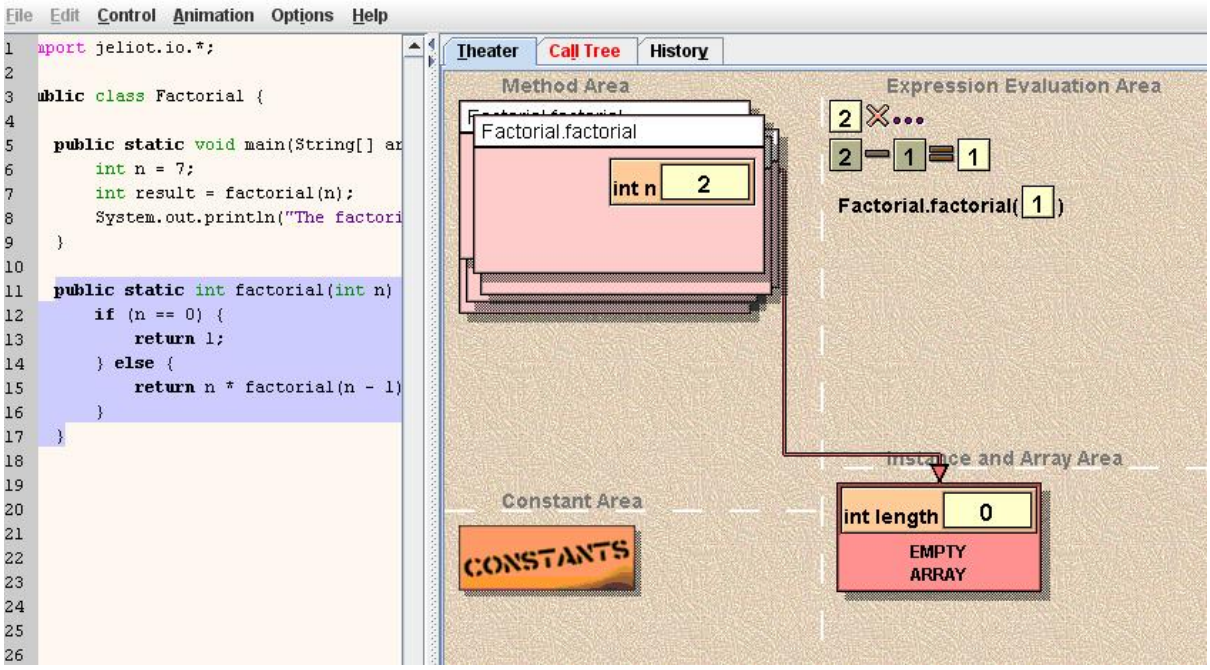
Начинот на работа на рекурзијата може да се види и во Trace view. За конкретниот пример, Trace view е прикажана на слика 62.



Слика 62. Визуелизација на рекурзија со SRec на програма за наоѓање на факториел на даден број – завршување на рекурзијата (Приказ на Trace View).

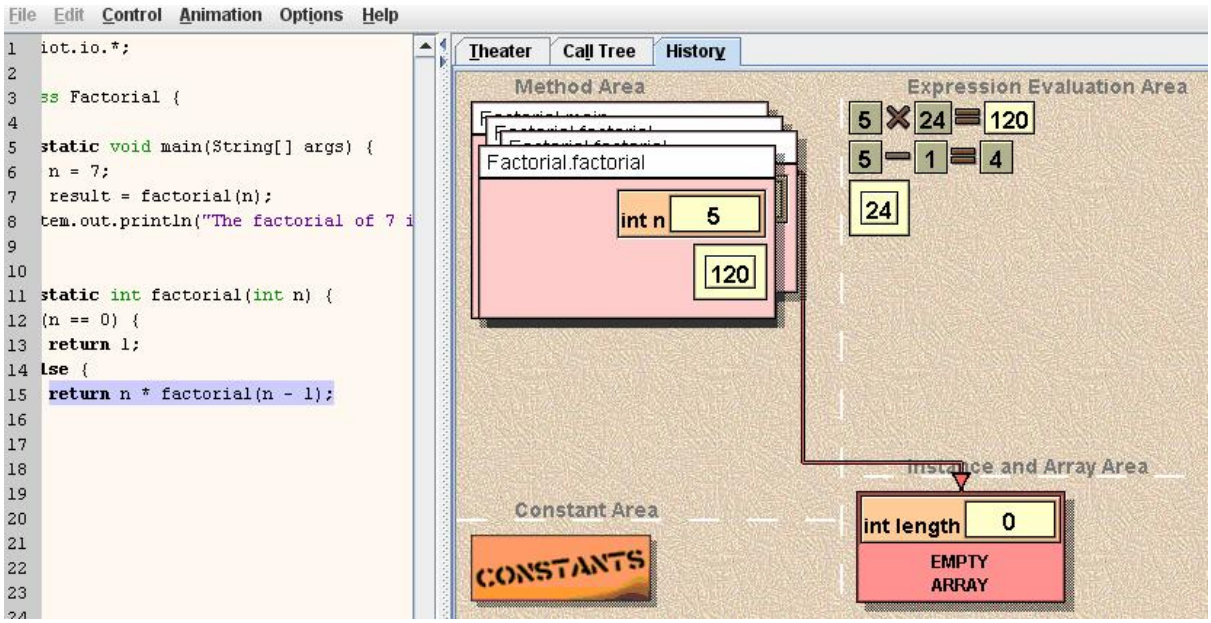
И во погледот на трагови е прикажан начинот на работа на рекурзијата. Прво, е претставено доаѓањето до основниот случај, па враќањето наназад да се добие резултатот на функцијата според соодветниот влезен аргумент.

Ако истата програма за наоѓање на факториел на даден број се пушти во Jeliot 3, се добива сосема поинаква визуелузација. На слика 63 е прикажан дел од оваа визуелизација.



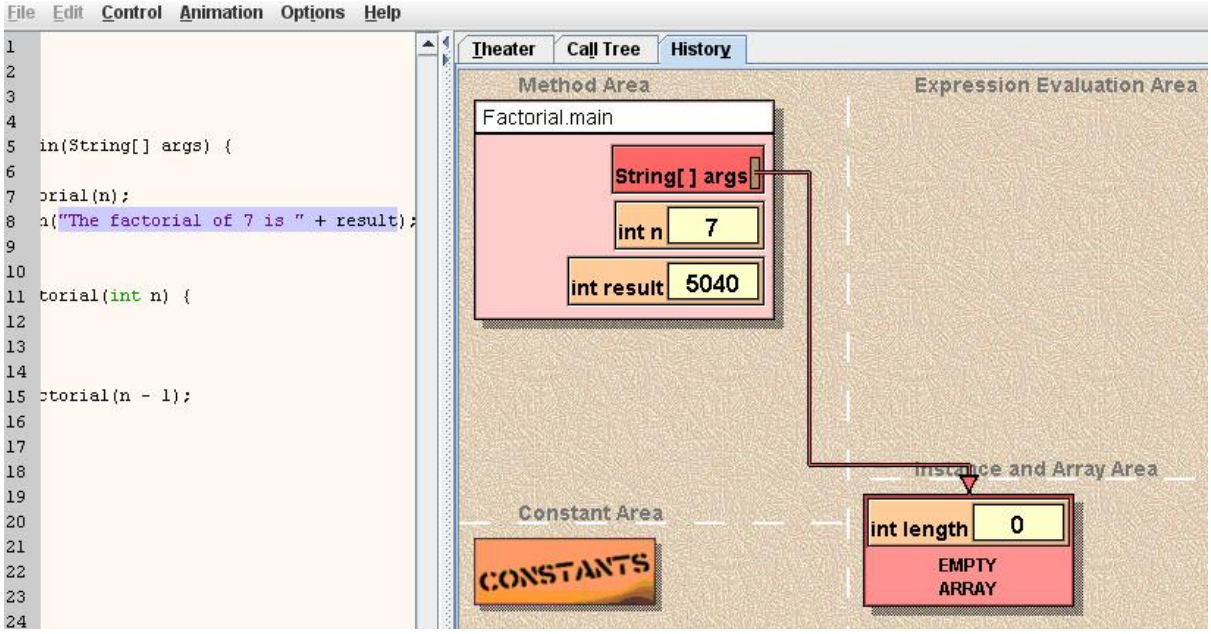
Слика 63. Визуелизација на рекурзија со Jeliot 3 на програма за наоѓање на факториел на даден број – пред доаѓањето до основниот случај од рекурзијата.(Приказ на поглед со театар).

Овде, секој нов повик на функцијата со својот влезен аргумент е претставен со нов правоаголник во делот за методи. Во делот за пресметки се прави тековната пресметка. На слика 63, текот на програмата е стигнат до пресметка на факториел од бројот два а пресметката е 2* factorial(2-1). Во делот за кодот на левата страна потемнетиот дел го прикажува делот кој моментално се визуелизира.



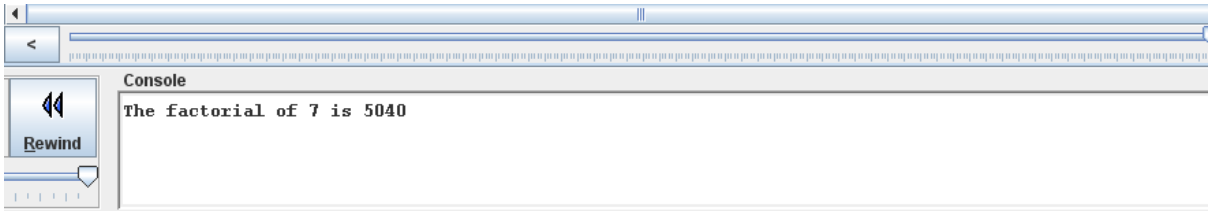
Слика 64. Визуелизација на рекурзија со Jeliot 3 на програма за наоѓање на факториел на даден број – при враќање наназад во рекурзијата (Приказ на поглед со театар).

Откако текот на програмата стигнува до основниот случај, постепено во делот за методи се јавува резултатот за соодветниот влезен аргумент, а кога ќе се добие резултатот тој повик кон метод се отстранува од делот за методи. Се тргнува од основниот случај, односно 0 и постепено се оди наназад се додека не се дојде до почетниот случај (слика 64). На крај, останува еден правоаголник за главната функција каде е претставен бројот чиј факториел се бара и добиениот резултат со рекурзијата (слика 65).



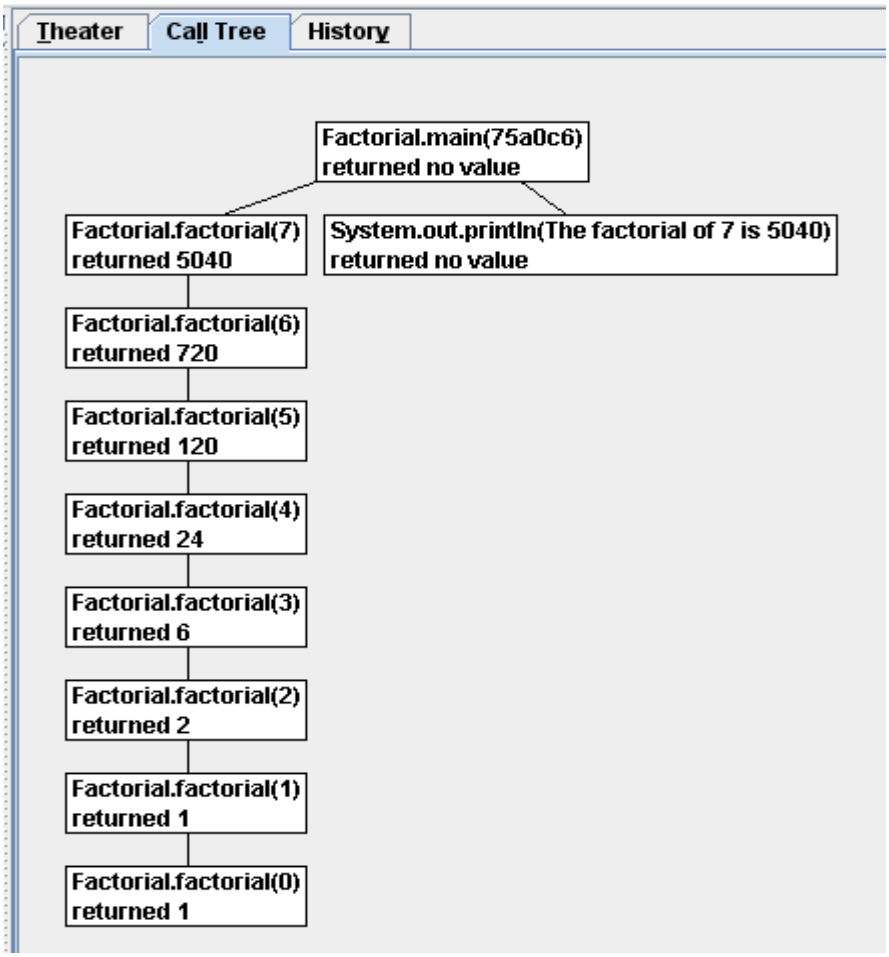
Слика 65. Визуелизација на рекурзија со Jeliot 3 на програма за наоѓање на факториел на даден број – завршување на рекурзијата (Приказ на поглед со театар).

Откако завршува анимацијата во конзолата е прикажан излезот на програмата, односно се печати резултатот (слика 66).



Слика 66. Приказ на конзола кај Jeliot 3 со резултатот од програмата за наоѓање на факториел на даден број со рекурзија.

Освен овој дел на анимацијата, Jeliot 3 нуди и визуелизација на рекурзијата со помош на градење на дрво на повици, од каде јасно може да се види редоследот на повици кон функциите. Кога во функцијата има повик кон истата функција, станува збор за рекурзија. На тој начин може да се следи текот на рекурзијата (слика 67).



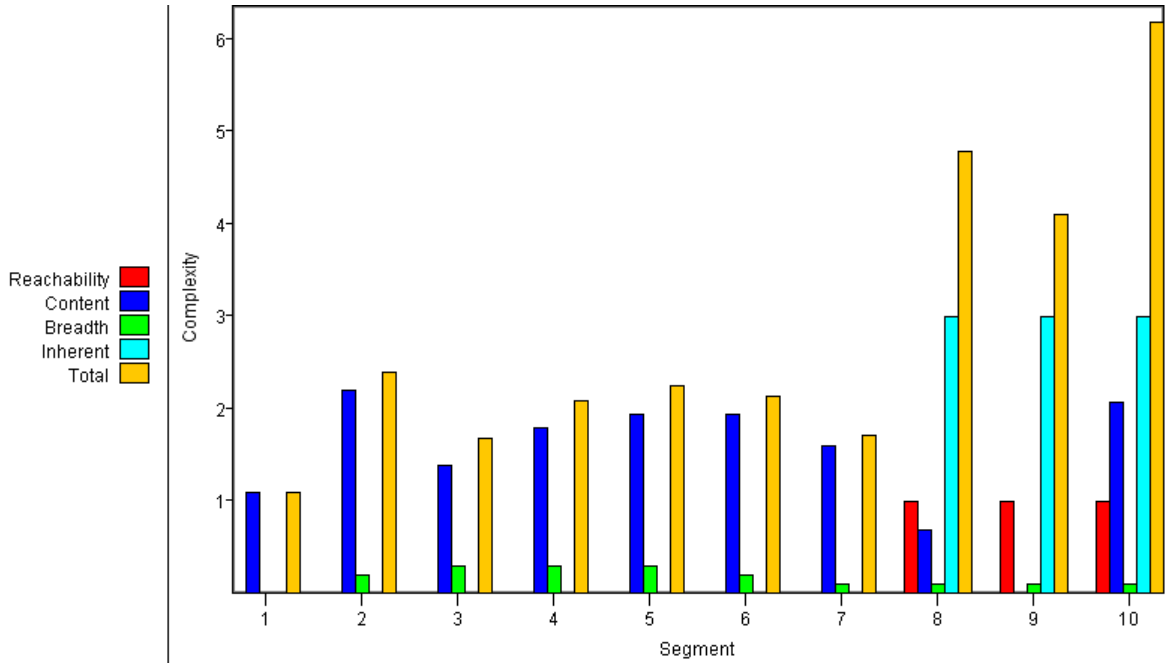
Слика 67. Визуелизација на рекурзија со Jeliot 3 на програма за наоѓање на факториел на даден број – завршување на рекурзијата (Приказ на дрво на повици).

Ако кодот за факториел го прикажеме со jGrasp, прво нешто што веднаш се добива е статичка визуелизација на кодот (слика 68).

```
1  public class Factorial {
2
3      public static void main(String[] args) {
4          int n = 7;
5          int result = factorial(n);
6          System.out.println("The factorial of 7 is " + result);
7      }
8
9      public static int factorial(int n) {
10         if (n == 0) {
11             return 1;
12         }
13         else {
14             int result=n * factorial(n - 1);
15             return result;
16         }
17     }
18 }
```

Слика 68. Статичка визуелизација со jGrasp на кодот од програма за рекурзивно пресметување на факториел на даден број.

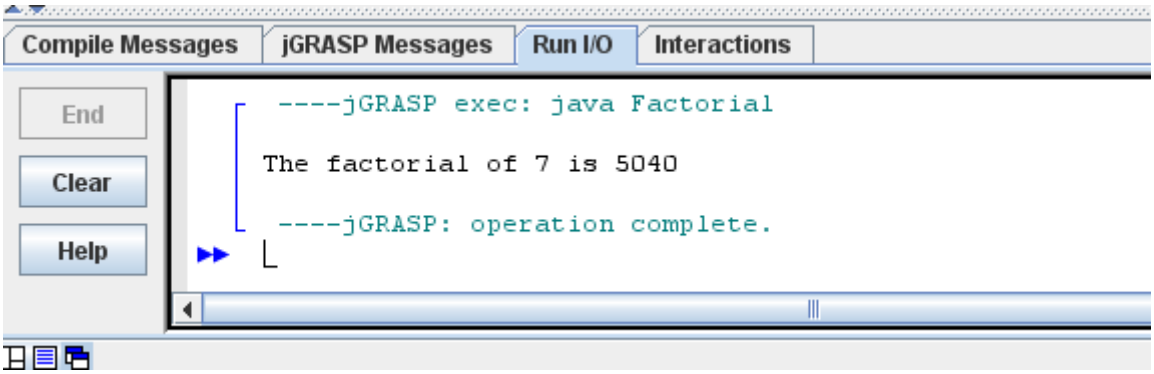
Со помош на ваквата визуелизација се гледа структурираноста на кодот, се гледа која класа се користи, кои се функциите, кои се променливите и на кои места во програмата има разгранување. jGrasp нуди и друг начин на статичка визуелизација со CPG (Complexity Profile Graph) (слика 69).



Слика 69. Статичка визуелизација со jGrasp со генерирање на CPG -Complexity Profile Graph на кодот од програмата за наоѓање на факториел на даден број

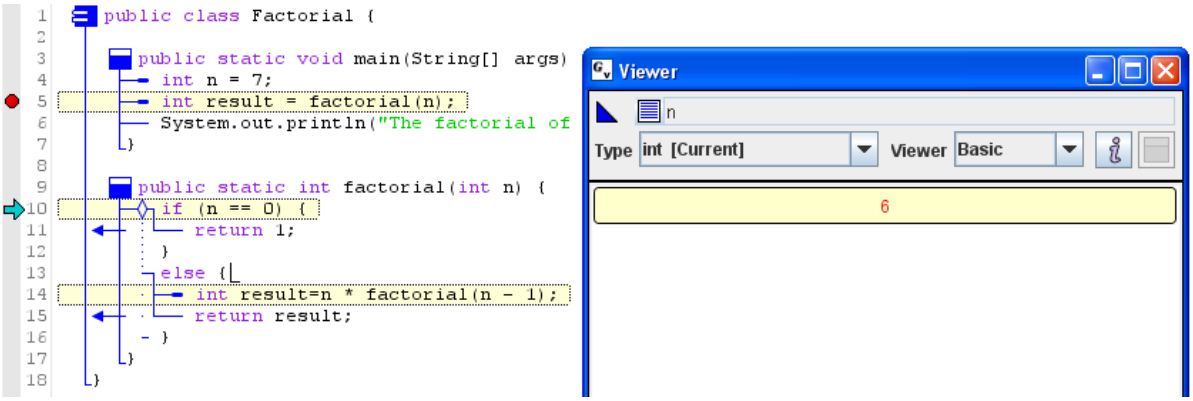
Со овој начин на визуелизирање се следи комплексноста на кодот, од различни аспекти, сложеност на пристап, содржина, ширина на код и вкупна сложеност на кодот. Според овој приказ може да се забележи дека станува збор за мала програма со код од 10 редови со наредби.

Меѓутоа, и претходниот и овој начин на претставување воопшто не помагаат во разбирањето на рекурзивниот повик на функцијата. Ако се изврши програмата во конзола се печати резултатот (слика 70).



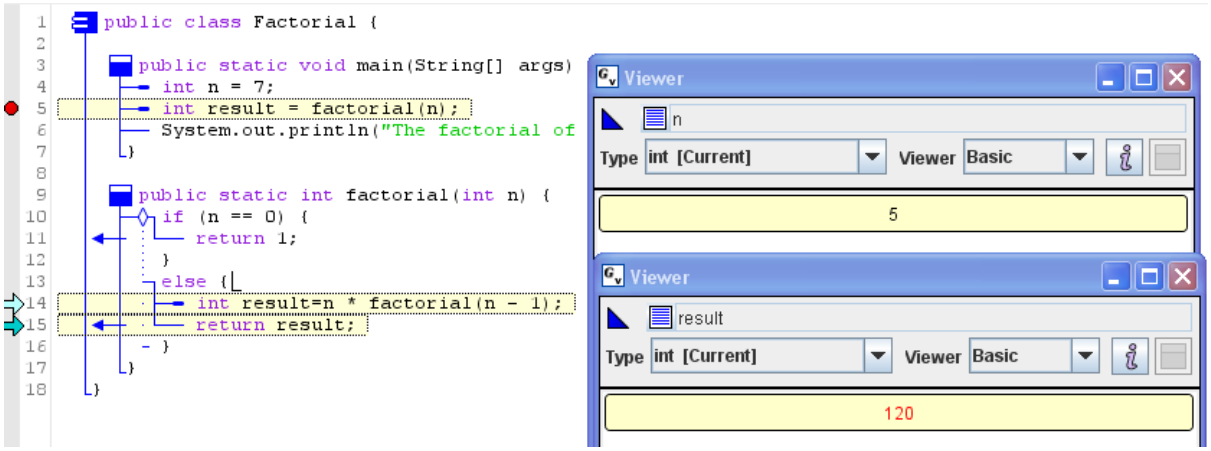
Слика 70. Приказ на конзола кај jGrasp со резултатот од програмата за рекурзивно пресметување на факториел на даден број.

Ако се дебагира програмата полесно ќе може да се следи нејзиниот тек. Во посебен прозорец можат да се забележат измените кои настануваат во променливите во текот на извршувањето (слика 71). Со промената на вредностите на променливите може да се сфати работата на рекурзијата.



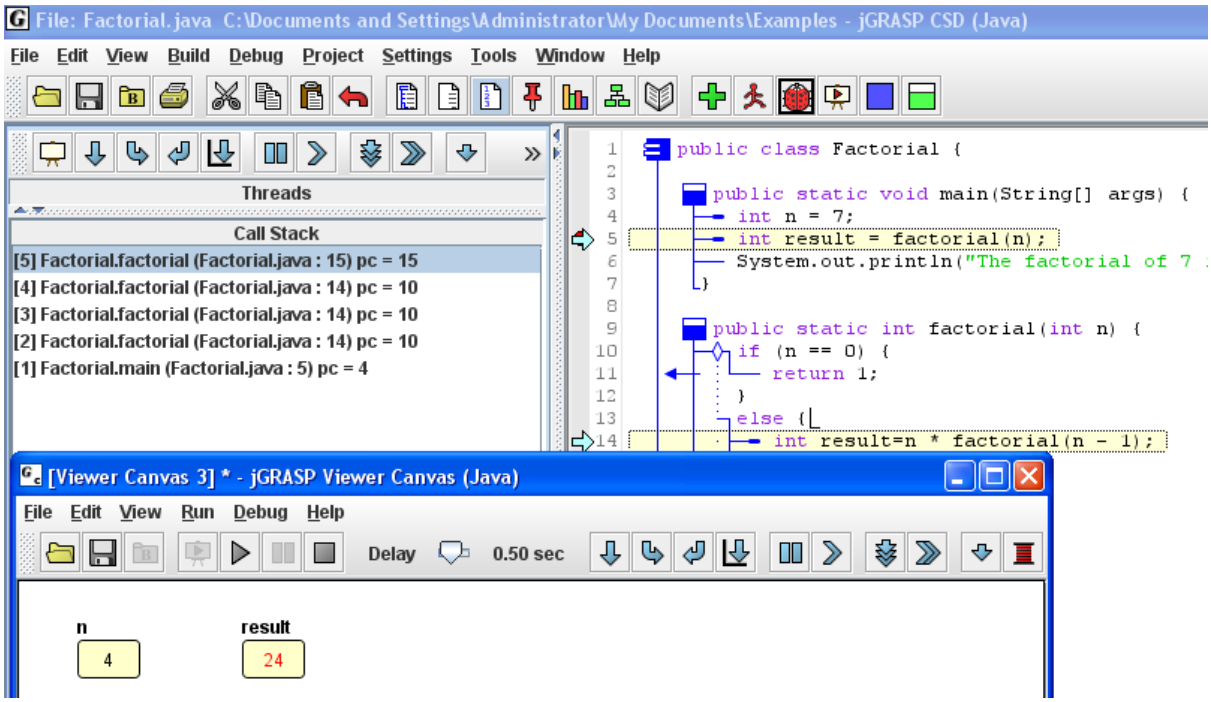
Слика 71. Приказ од дебагирање во jGrasp на програмата за пресметување на факториел на даден број (поглед со вредност на променливата n пред доаѓањето до основниот случај на рекурзијата)

Прво, се гледа како аргументот n на функцијата се намалува со секој повик, кога ќе се дојде основниот случај се појавува првиот резултат, односно резултатот за основниот случај и понатаму се менуваат вредностите на променливите n и result соодветно, почнувајќи од основниот случај и враќајќи се наназад код почетниот аргумент (слика 72).



Слика 72. Приказ од дебагирање во jGrasp на програмата за пресметување на факториел на даден број (поглед со вредност на променливите n и result при враќањето наназад во рекурзијата).

Со jGrasp може да се формира и анимација на промена на вредностите на променливите во посебен поглед (Viewer Canvas). Истовремено, при дебагирањето може да се гледа и промената на магацинот на повици (Call Stack). На слика 73 е прикажана дел од визуелизацијата на промената на вредностите на променливите n и result во текот на рекурзијата.

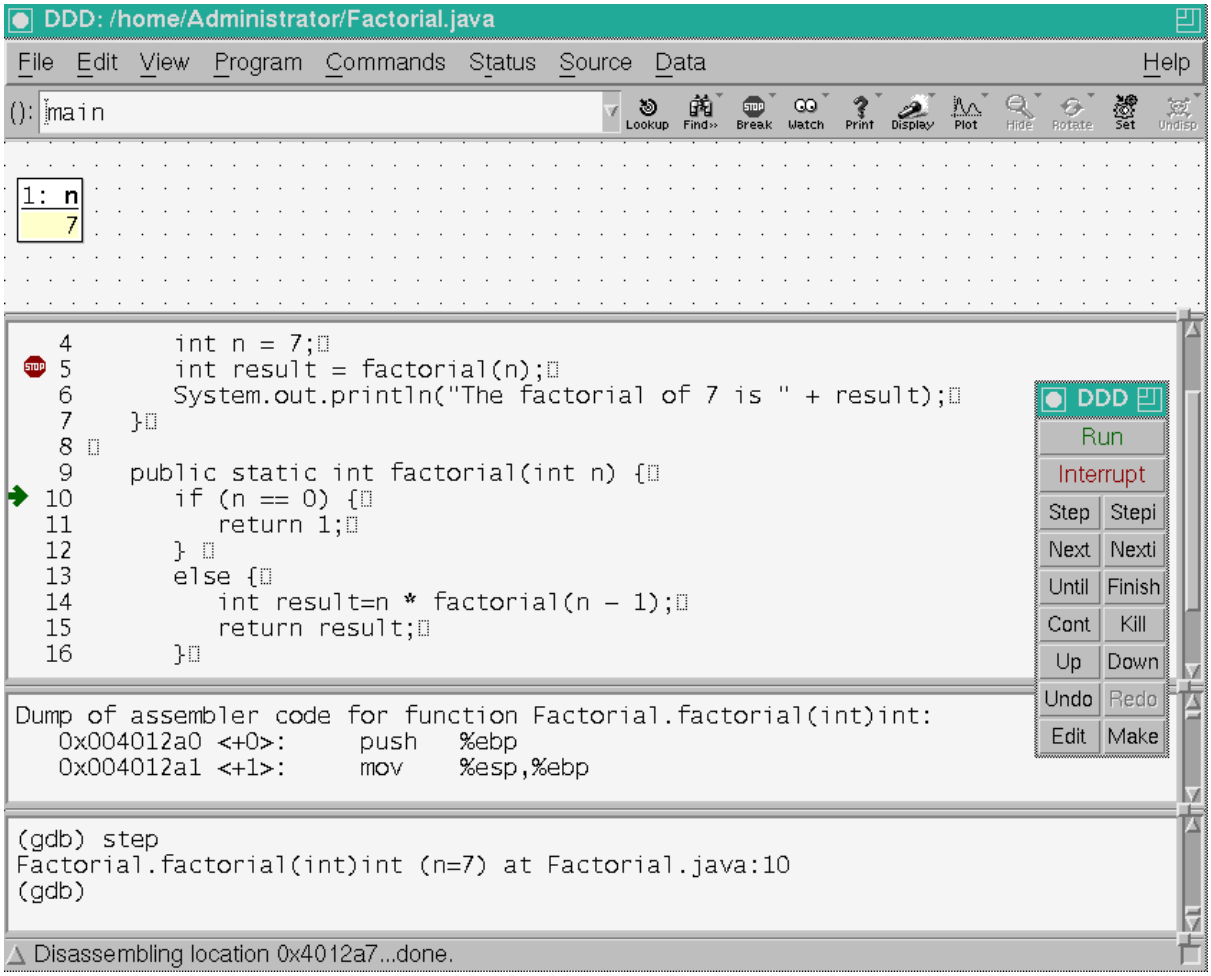


Слика 73. Приказ од дебагирање во jGrasp на програмата за пресметување на факториел на даден број со анимација (Viewer Canvas со промена на вредностите на променливите n и result при враќањето наназад во рекурзијата).

Во магацинот на повици може да се види колку повици кон функциите има и штом се добие некој резултат, како тој повик се отстранува од магацинот.

Останува уште оваа програма, за пресметување на факториел на даден број, да ја извршиме со дебагерот DDD. DDD пред се е наменет за програми напишани во C или C++ и јавува проблеми кај програми напишани во Java. Но, тоа најчесто е пример кај посложени програми. Бидејќи ова е едноставна програма и ако е напишана во Java може да се дебагира со помош на DDD.

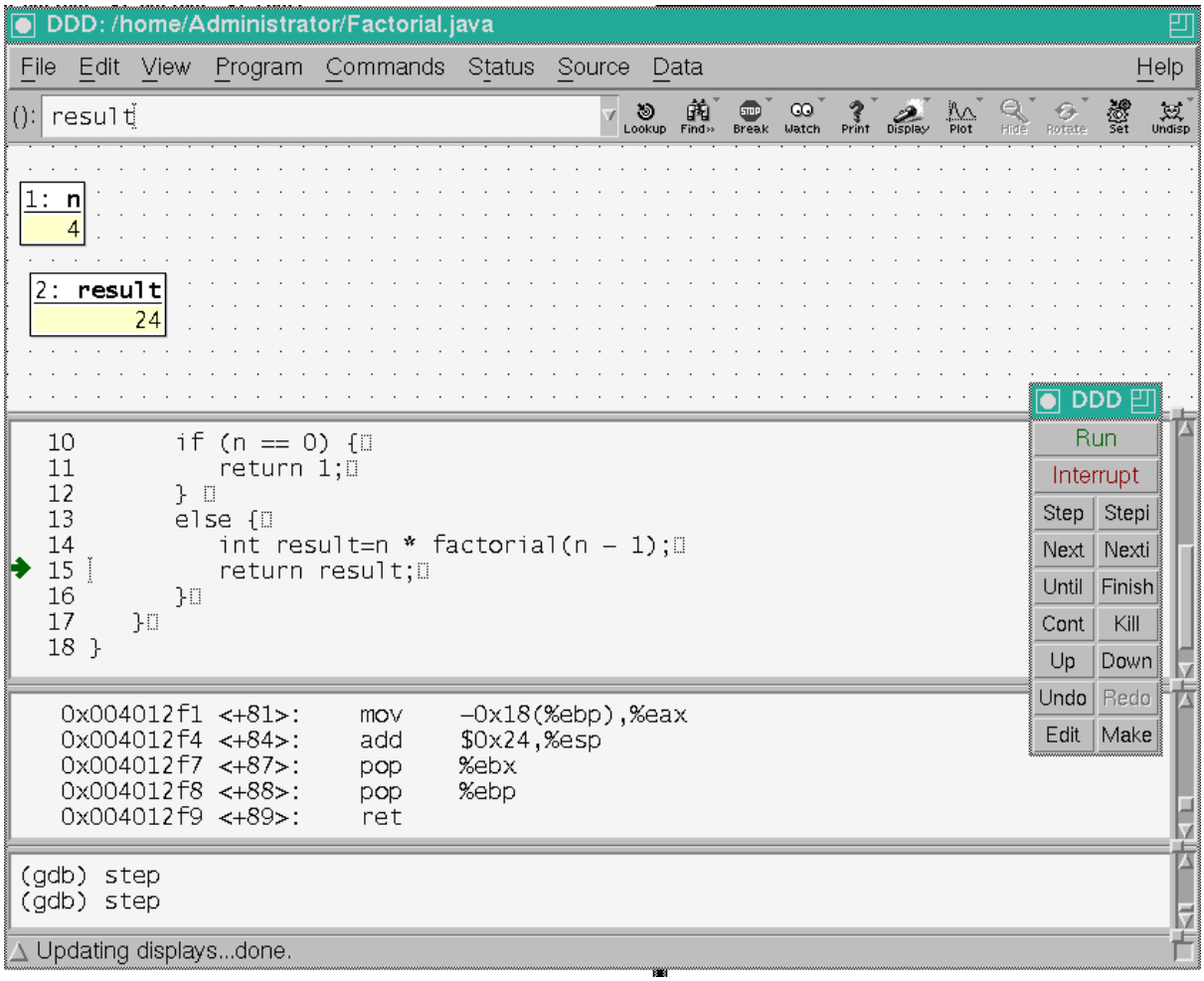
Прво, точката на прекин ја поставуваме на линијата број 5 и почнуваме со дебагирањето одејќи чекор по чекор (слика 74).



Слика 74. Приказ од дебагирање во DDD на програмата за пресметување на факториел на даден број (поглед со вредност на променливата n пред доаѓањето до основниот случај на рекурзијата).

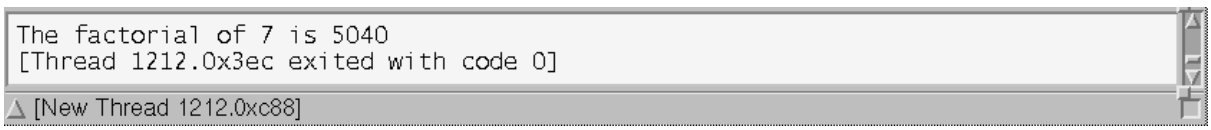
Со помош на DDD може да се следат машинските инструкции кои одговараат на секој извршен ред од програмата. Во делот за приказ на граф може да се претстават вредностите на променливите кои учествуваат во програмата (слика 74).

Во тековната програма, на почетокот, во делот за граф во форма на четириаголник се претставува променливата n, со вредност 7, која е почетен влезен аргумент на функцијата factorial. Во делот за кодот може да се следи кој дел тековно се извршува. Вредноста на n со секој повик на функцијата се намалува за 1, се додека не дојде до 0, што е и основниот случај. Понатаму, од основниот случај се формира резултатот, односно вредноста на променливата result се прикажува во делот за приказ (слика 75).



Слика 75. Приказ од дебагирање во DDD на програмата за пресметување на факториел на даден број (поглед со вредност на променливите n и result при враќањето наназад во рекурзијата)

По целосното извршување на програмата во конзолата се печати резултатот (слика 76).



Слика 76. Приказ на конзола кај DDD со резултатот од програмата за пресметување на факториел на даден број со рекурзија.

Слично како и при дебагирањето со jGrasp и овде кај DDD, визуелизацијата на рекурзијата се сведува на приказ на променливите, и менување на нивните вредности со секој нов повик на рекурзивната функција. На тој начин и со истовремено следење на кодот може да се разбере текот на рекурзијата, особено нејзините основни карактеристики: одење кон основниот случај и враќање наназад со резултатот.

Сите овие четири програми на различен начин ја прикажуваат рекурзијата на истата програма. Кај Jeliot 3 и SRec се користи анимација што автоматски се генерира. Овие две програми користат повеќе визуелни ефекти за прикажување на рекурзијата, а истовремено нудат и повеќе различни погледи, со цел нејзино подобро разбирање и изучување. Затоа, кога се работи за едноставни рекурзии подобри за изучување би биле овие две програми. Додека, jGrasp и DDD ја визуелизираат рекурзијата при дебгирање и нудат помалку визуелни ефекти за нејзино прикажување. Иако не се многу добри за изучување на рекурзијата, сепак можат многу да помогнат во нејзиното разбирање.

6.3 Визуелизација на поголеми програми

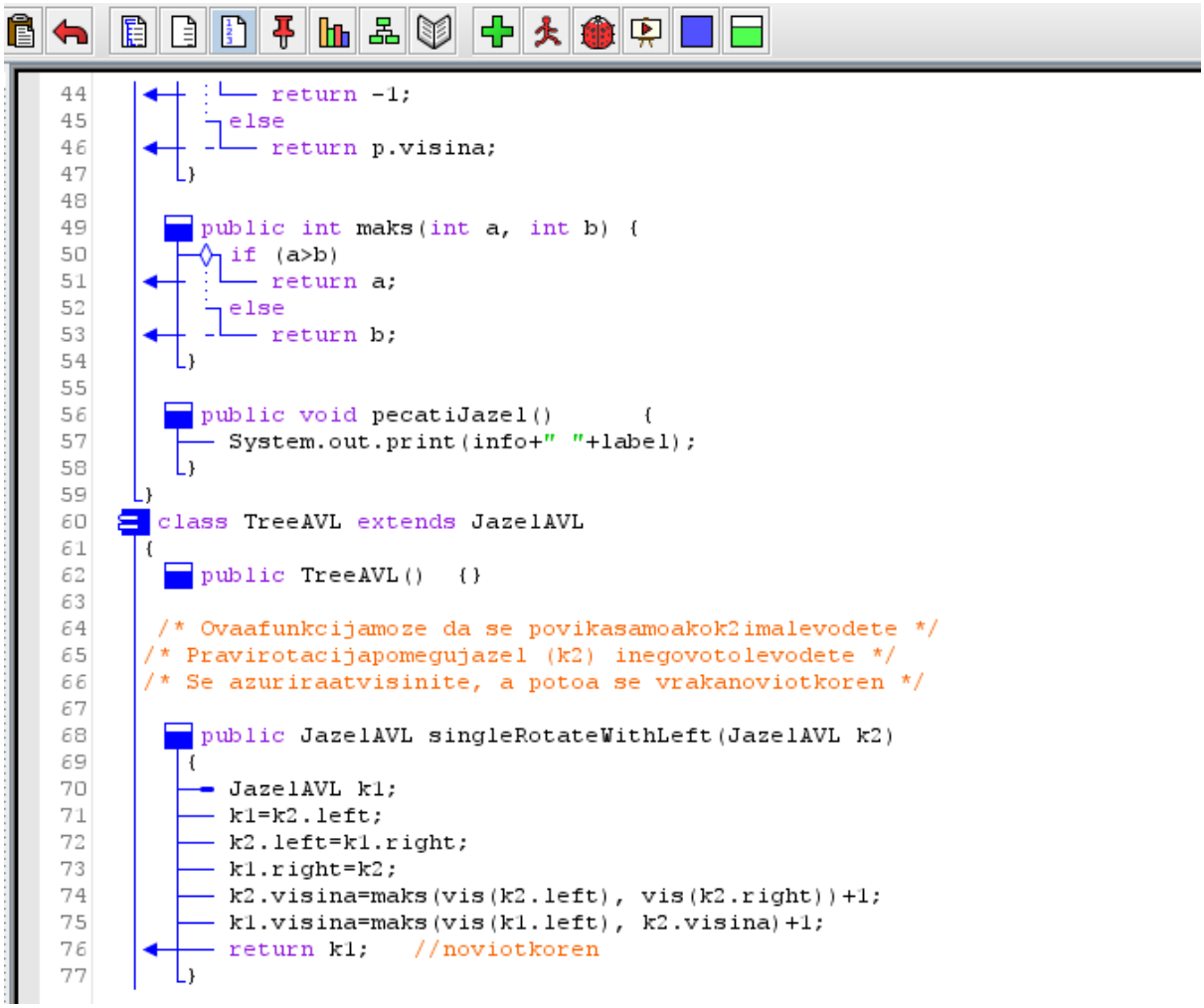
До сега разгледаните визуелизации се однесуваа на програми кои се релативно едноставни и содржат малку код. Каква визуелизација би пружиле овие системи доколку се користат посложени и поголеми програми? За тоа да го испитаме ќе искористиме програма која содржи рекурзии и посложени податочни структури, како и повеќе класи и методи. Како таква програма може да се земе програма која додава и брише елементи во бинарно пребарувачко балансирано дрво. Ќе искористиме две варијации на ваква програма. Првата програма е имплементација на AVL дрво во кое се вметнуваат елемети а тоа истовремено треба да ја задржи својата балансираност. А втората програма е различна имплементација на AVL дрво каде се користат помалку рекурзии, но освен додавање е имплементирана и можноста за бришење на јазли, а подоцна повторно реконструирање на дрвото со цел да биде балансирано.

6.3.1 Визуелизација на програмата `AvlTest1.java` со jGrasp и Jeliot3

Кодот на првата програма `AvlTest1.java` е даден во Додаток 1.

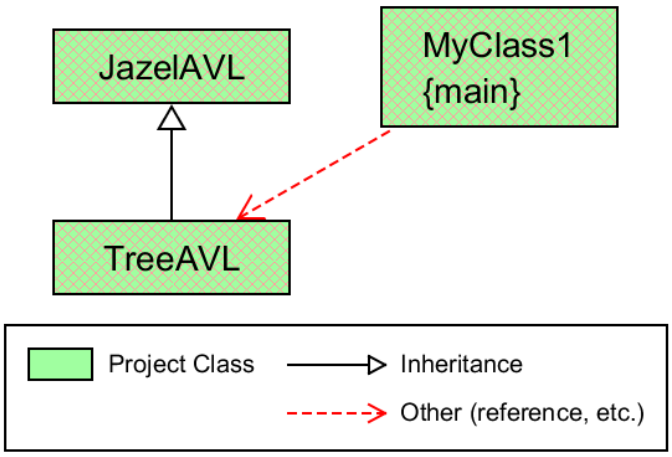
Прво, кодот од програмата го вметнуваме во ситемот jGrasp. На овој начин, веднаш се добива еден вид статичка визуелизација на кодот. Јасно се гледаат сите класи, нивните атрибути, постоечките методи и разгранувачки структури. Исто така лесно може да се забележи каде започнува и каде завршува секоја класа, секој метод и секоја од структурите на разгранување. Сите предефинирани видови на податоци се претставени со различна боја, така

може лесно да се воочи ако се направи некоја случајна грешка при пишувањето или ако се пропушти некој дел, на пример, не се затвори некоја класа или некој метод. Дел од ваквата статичка визуелизација на програмата што ја нуди jGrasp е дадена на слика 77.



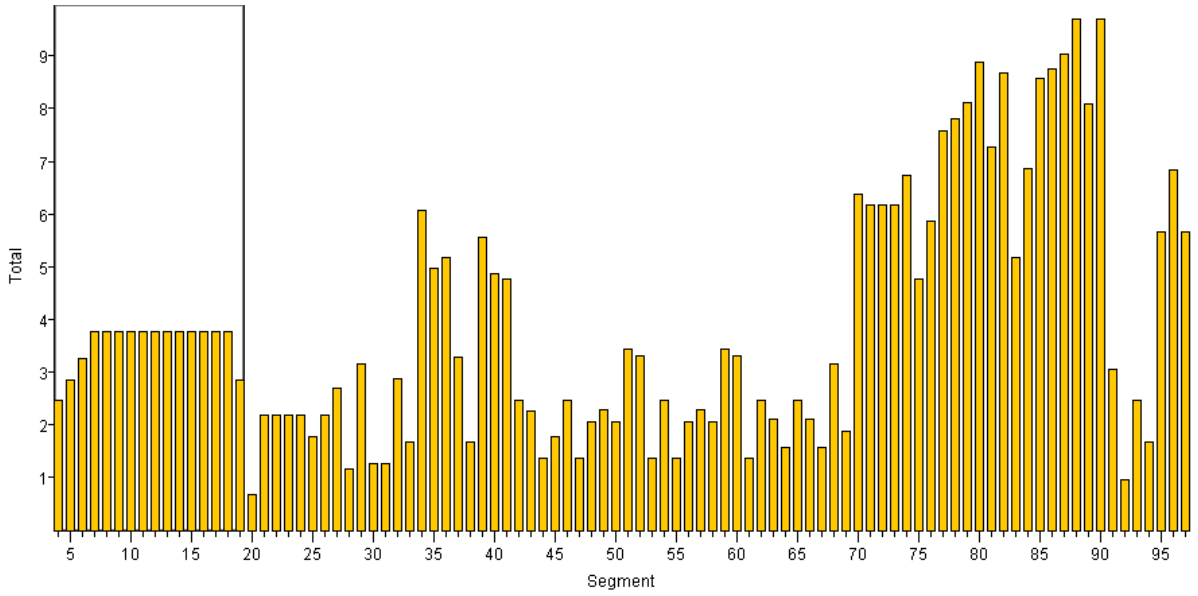
Слика 77.Статичка визуелизација со jGrasp на програмата AvlTest1java, како пример за релативно поголема програма

Освен ваквиот вид на статичка визуелизација jGrasp за оваа програма нуди и UML статичка визуелизација на класите кои учествуваат во програмата. UML статичката визуелизација на класите за програмата е прикажана на слика 78.



Слика 78. UML класен дијаграм како дел од статичка визуелизација на програмата AvlTest1.java со помош на jGarsp

jGrasp како и за другите програми напишани во Java така и за оваа нуди и визуелизација со која се прикажува комплексноста на кодот (Complexity Profile Graph-CPG). Со помош на ваквата визуелизација се прикажува вкупната комплексност на кодот, што опфаќа, комплексност на содржината, големината на наредбата, сложеност при пристапот, вгнездувањето, разгранувањето и сл. Ваквиот тип на визуелизација за оваа програма е даден на слика 79.



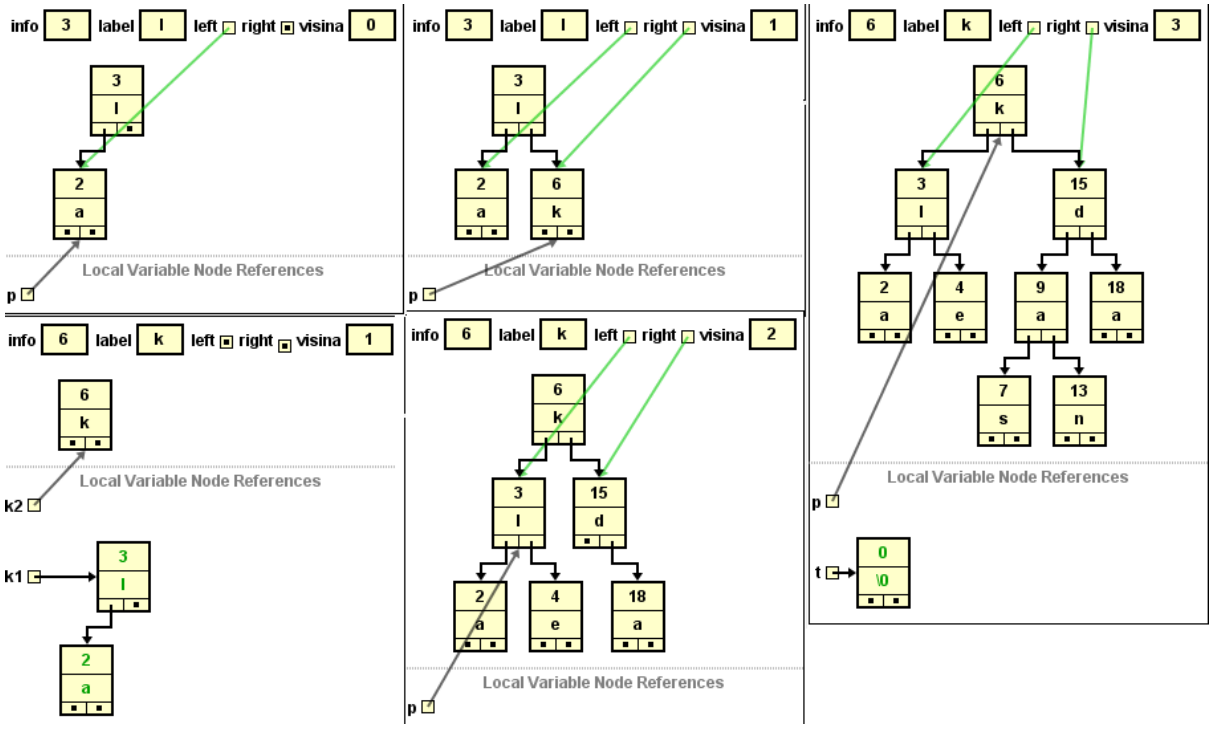
Слика 79. CPG график како дел од статичката визуелизација на програмата AvlTest1.java со помош на jGrasp

Со помош на оваа визуелизација се забележува дека програмата се состои од 97 линии на код кој содржи наредби, а најголема вкупна комплексност има кај 90-тиот ред.

Освен статичките визуелизации jGrasp нуди и динамичка визуелизација преку која јасно се гледа градењето на AVL дрвото.

Бидејќи визуелизацијата на дрва како податочна структура е вградена во системот, оваа програма може лесно да се визуелизира, преку претставување на вредностите на јазлите и врските што даден јазел ги има со останатите јазли. Освен податочната структура за дрва, овој систем нуди поддршка за визуелизација и за сите останати основни податочни структури како низи, магацини, редови, единечно и двојно поврзани листи и бинарни дрва. Секоја програма која вклучува вакви или малку модифицирани основни податочни структури може на едноставен начин да се визуелизира со помош на овој систем, без разлика на големината на програмата. На таков начин лесно може да се сфати и разбере структурата и начинот на градење и поврзување на ваквите податочни структури.

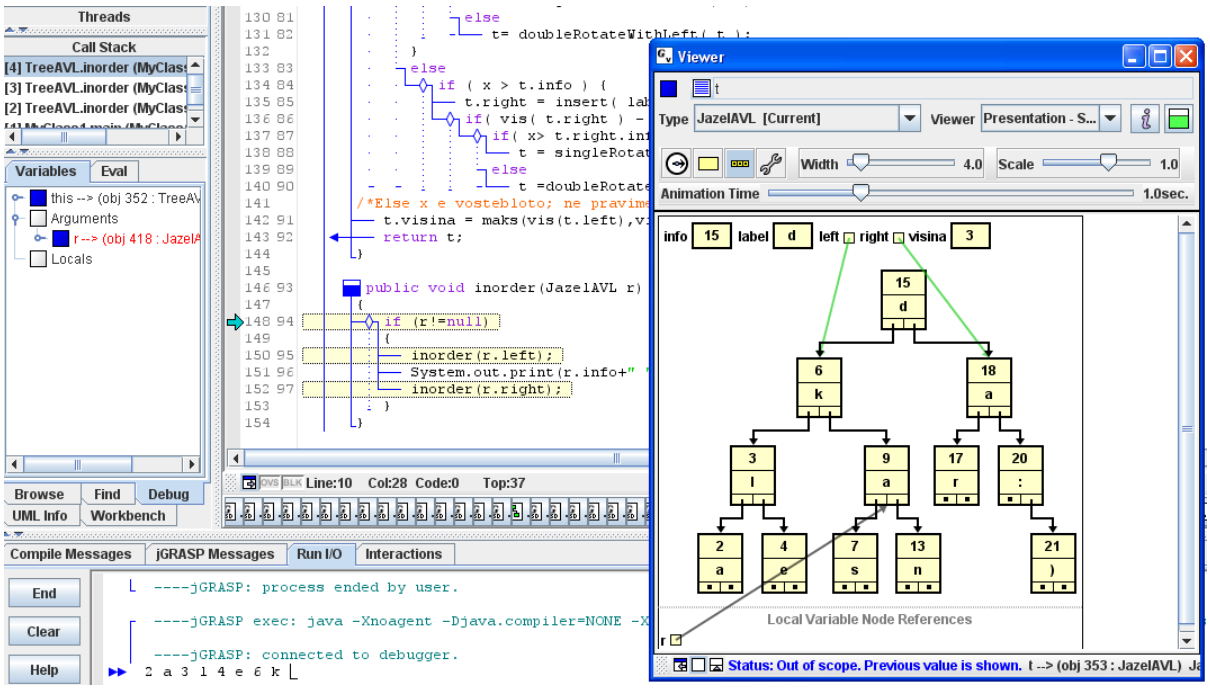
Пресеци како дел од динамичката визуелизација на оваа програма со помош на jGrasp се дадени на слика 80.



Слика 80. Пресеци од динамичката визуелизација на програмата AvlTest1.java со помош на jGrasp

Од сликата јасно се гледа кон што покажува тековниот јазел во секој момент од извршувањето, кој јазел е корен и кон што покажуваат левото дете и десното дете за дадениот јазел. Тековниот корен е прикажан со неговите полиња за информација (info), ознака (label), лево дете (left), десно дете (right) и неговата тековна виснина (visina). Доколку некој јазел нема доделено

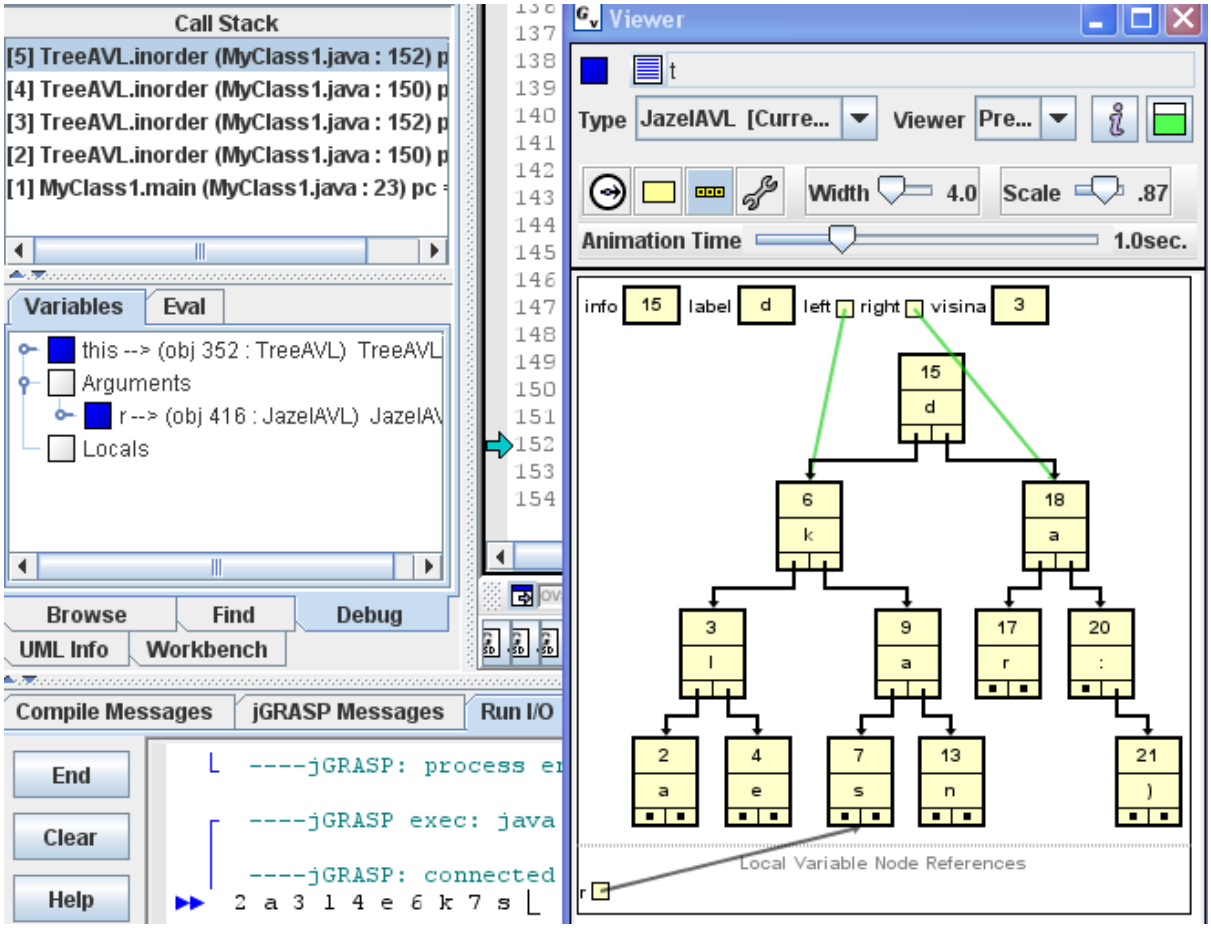
вредности во полињата за информација и ознака тие се прикажани соодветно како 0,\0. Ако јазелот нема деца тогаш е прикажан со нулеви покажувачи .



Слика 81.Пресек од динамичката визуелизација на програмата AvlTest1.java со помош на jGrasp.

Истовремено со приказот на визуелизацијата, во текот на дебагирањето на програмата во делот со кодот, може да се гледа кој дел од визуелизацијата на која линија код од програмата одговара. Односно може да се следи секој чекор од промената на дрвото. Кодот, кој во моментот се визуелизира, е означен со испрекинат правоаголник. Тоа е прикажано на слика 81. На тој начин може да се забележи како влијае секоја наредба на промената на изгледот на дрвото.

Крајниот изглед на програмата, односно деловите кои се печатат се прикажани на конзолната линија, но истовремено со печатењето визуелно е претставен и начинот на кој се читаат елементите од дрвото пред да се испечатат (Слика 82). Козолната линија го дава излезот на програмата, а во левиот горен агол е прикажан изгледот на повикувачкиот магацин, преку кој се гледаат повиците кон методите. Бидејќи тековната функција inoredr е рекурзива, од повикувачкиот магацин може да се забележи бројот на рекурзивни повици кој во моментот е искористен. Од сликата се гледа дека тековниот рекурзивен повик е четвртти, односно претставува петти елемент во магацинот бидејќи првиот елемент е главната функција (main).



Слика 82. Приказ на дел од резултатот на конзолната линија и приказ на повикувачкиот магацин како дел од динамичката визуелизација на програмата MyClass.java со jGrasp.

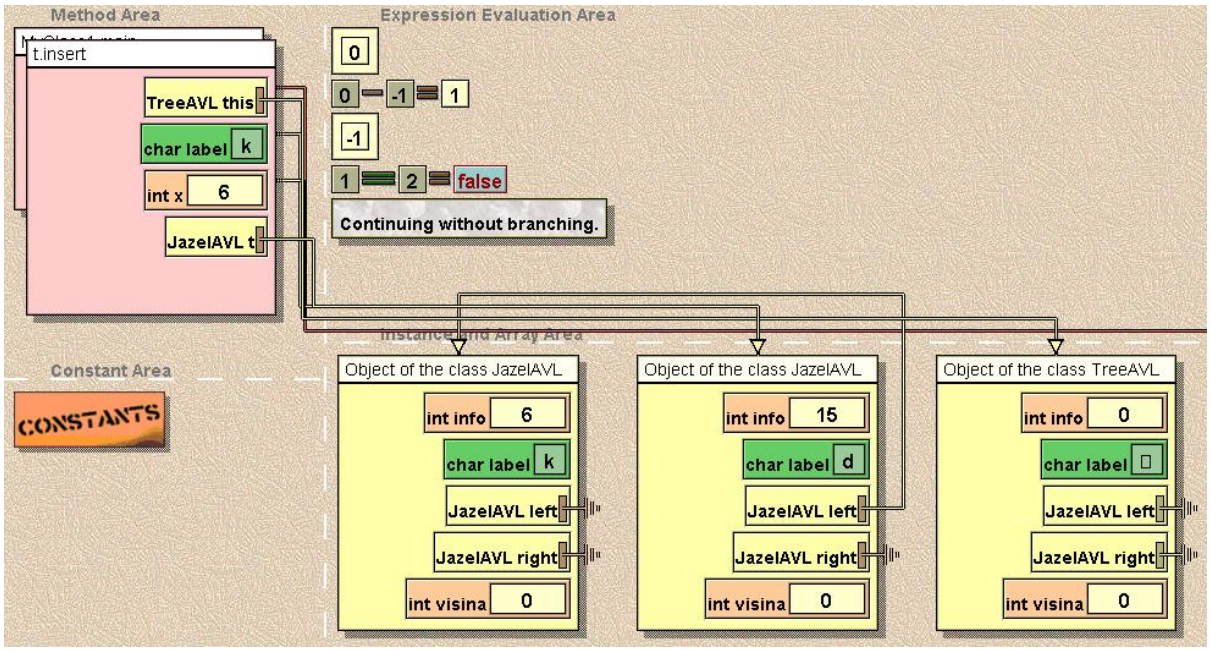
Времето потребно за целосно визуелизирање на ваква програма е значително поголемо од времето потребно за визуелизации на малите програми, но сепак не е премногу долго и е доволно да се разбере начинот на работа на програмата.

Ако истата програма, со мали измени на почетокот, се пушти да се извршува во системот Jeliot3, визуелизацијата што се добива е сосема поинаква. Во однос на статичката визуелизација, Jeliot3 нуди добра статичка визуелизација со нумерирање на секој ред од кодот и означување на основните типови на податоци и резервираните зборови со различна боја (слика 83).

```
29  }
30  class JazelAVL
31  {
32     public int info;
33     public char label;
34     public JazelAVL left;
35     public JazelAVL right;
36     int visina;
37
38     public JazelAVL()     {}
39
40     public JazelAVL(int i)   {   info=i;     }
41     public JazelAVL(int i, char l)   {   info=i;  label=l;   }
42
43     public int vis(JazelAVL p)  {
44        if (p==null)
45           return -1;
46        else
47           return p.visina;
48     }
49
50     public int maks(int a, int b) {
51        if (a>b)
52           return a;
53        else
54           return b;
55     }
56
57     public void pecatiJazel()      {
58        System.out.print(info+" "+label);
59     }
60  }
61  class TreeAVL extends JazelAVL
```

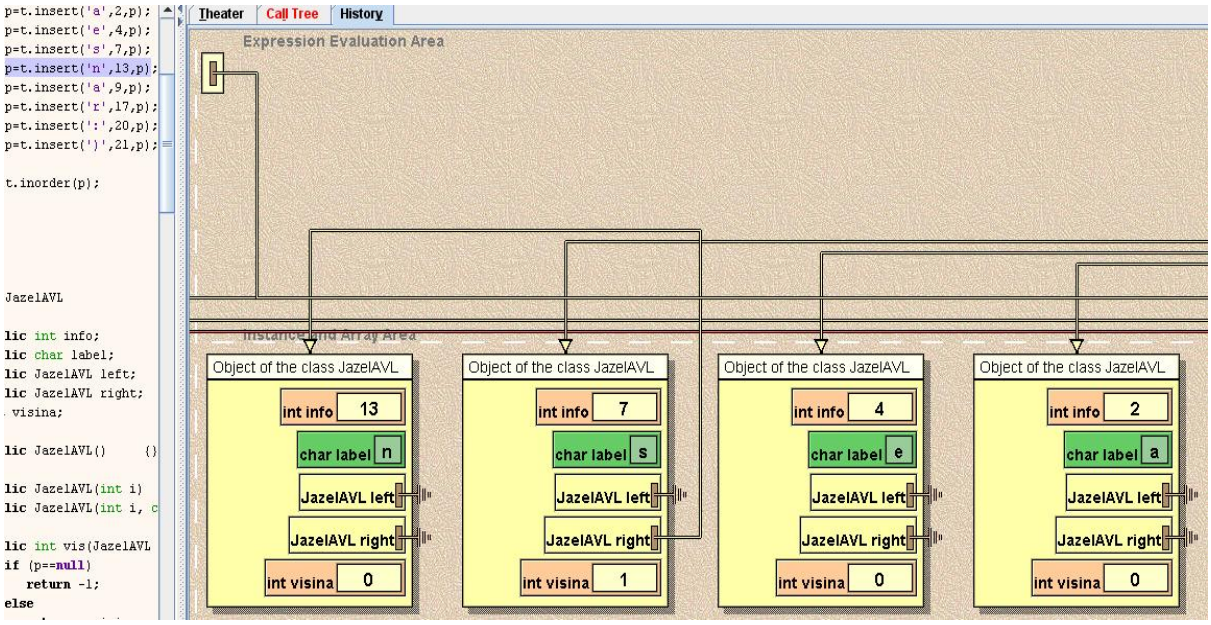
Слика 83. Дел од статичка визуелизација на програмата AvlTest1.java со Jeliot3

Што се однесува до динамичката визуелизација, Jeliot3 навлегува во деталите на програмата. Детално го обработува и го прикажува секој повик на методите, детално ги објаснува сите пресметки и споредби, навлегува во секое програмско разгранување или циклус, прикажувајќи ги сите неопходни пресметки. На ваков начин можат јасно да се забележат составните елементи на даден објект од класа (во оваа програма јазел) како и референцирањата од повиците на методите кон објектите. Ова особено јасно е покажано на почетокот од визуелизацијата кога бројот на внесени јазли односно создадени објекти од класата е мал и лесно можат да се следат референцирањата од повикот на методите и врските кои постојат меѓу јазлите (слика 84). Исто така во првиот дел од визуелизацијата може многу лесно да се забележи како секоја пресметка влијае на промената на изгледот на објектите.



Слика 84. Дел од динамичката визуелизација на програмата AvlTest1.java со помош на Jeliot3. (при почетокот на анимацијата)

Со текот на анимацијата, како што се формираат поголем број на објкети, се испреплетува поврзаноста меѓу јазлите, се намалува видливоста и потешко може да се следи промената на објектите, како и влијанието на пресметките врз нив (слика 85).



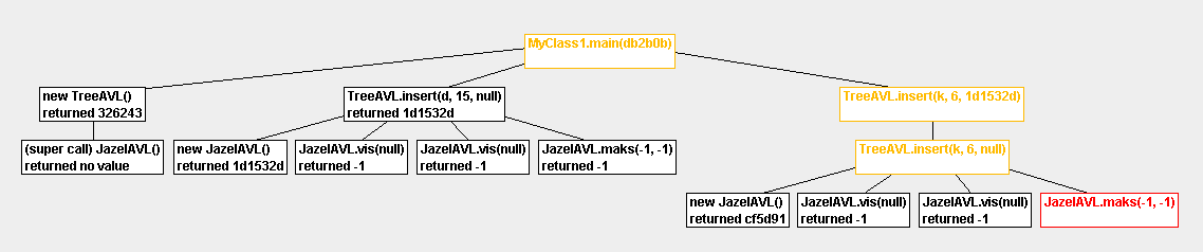
Слика 85 Дел од динамичката визуелизација на програмата AvlTest1.java со помош на Jeliot3. (средината на анимацијата)

Иако е намелна видливоста сепак е можно следењето на промените во поврзаноста меѓу јазлите. Од друга страна може лесно да се забележува и

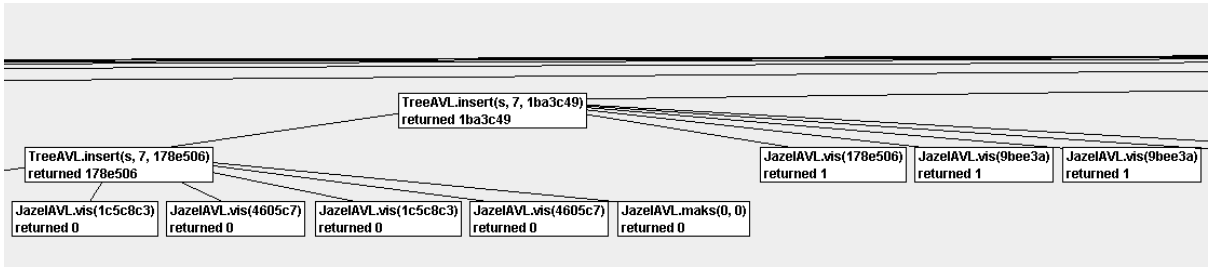
кодот кој предизвикува некоја промена што настанува во делот за визуелизација (Слика 85).

Самиот факт што во случајот се претставува програма со дрво, а визуелизацијата го преставува во вид на листа се нарушува визуелната претстава која вообичаено ја имаме за дрва и следењето на промените може да станува малку потешко. Исто така, некои од визуелизациите и пресметките се повторуваат и на тој начин постепено може да се изгуби вниманието на корисникот.

Другиот дел од динамичка визуелизација која ја нуди Jeliot 3 е постепеното градење на дрвото на повици. На почетокот од анимацијата ова дрво е мало и лесно му се следат промените, но, постепено станува пообемно и поразгрането а со тоа и потешко за следење, сепак неговото следење не е невозможно (слика 86).



а)



б)

Слика 86. Дрвото со повици (Call Tree) од првиот дел на анимацијата на програмата AvlTest1.java а), на крајот на анимацијата б)

Од визуелизацијата на оваа програма со Jeliot3 се забележува дека овој систем ги прикажува и најситните детали од пресметките, токму поради тоа, времето потребно за визуелизација на оваа програма значително се зголемува и тоа станува долго и трае околу 1 час. Исто така и поставеноста на елементите постепено станува прилично непрегледна и веќе потешко можат да се следат промените.

6.3.2 Визуелизација на програмата `TestAVL22.java` со Jeliot3 и jGrasp

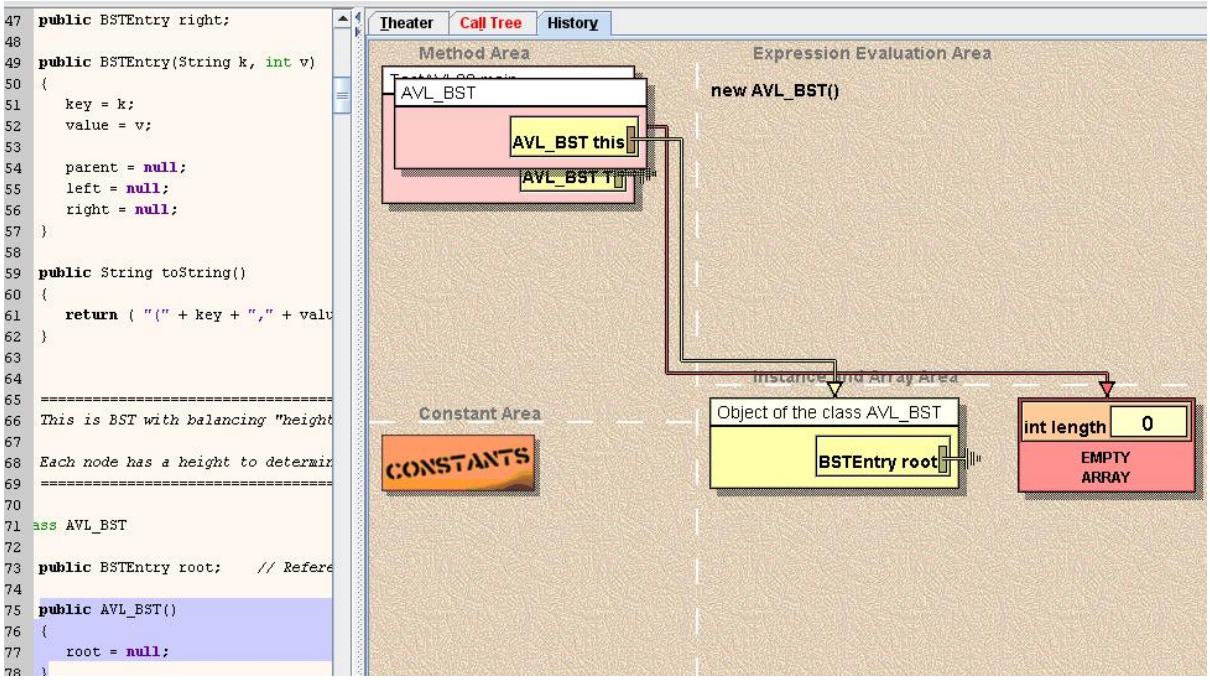
Дополнително, искористуваме нова прилично поголема програма која е слична на претходната, но, нуди поинаква имплементација на AVL дрва. Таа

опфаќа повеќе код и има вметнато доплнителни елемнти за бришење на јазел и дополнителни елементи за воведување поголема прегледност во печатењето на крајниот резултат. Во таков случај се добиваат и значителни разлики во начинот и времето на визуелизирање. Кодот на втората поголема програма `TestAVL22.java` што ќе пробаме да ја извршиме со овие два система е даден во Додаток 2.

Кодот на оваа програма е прилично обемен. Во вакви случаи можни се повеќе грешки и ваквите програми се потешки за разбирање. Затоа добра визуелизација на програми од ваков тип е многу значајна за намалување на можните грешки и подобрување на разбирливоста.

Овој код ако го пуштиме да се извршува во Jeliot3, тогаш слично како и кај претходната програма, Jeliot3 ќе навлезе во деталите на кодот при што ќе се добие детална слика за секој чекор.

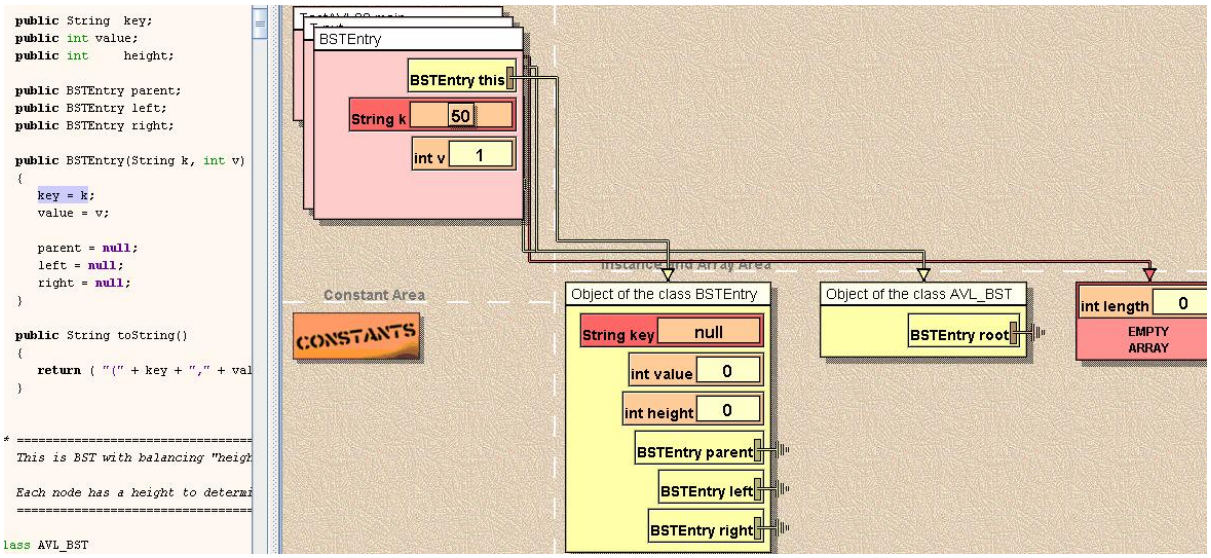
На слика 87 е прикажан дел од визуелизацијата на почетокот од извршувањето на програмата.



Слика 87. Дел од визуелизацијата на програмата TestAVL22.java со помош на Jeliot3 (почеток на анимацијата).

Со ваквиот начин на визуелизација јасно се гледа секој повик на методите и креирање објект од дадената класа. Исто така се забележува начинот на поврзување и референцирање кон објектите.

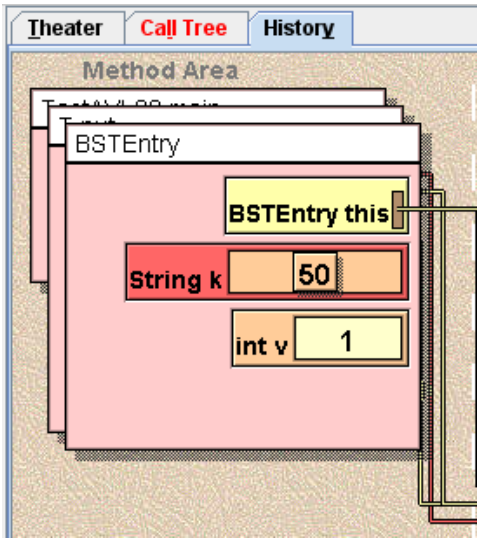
На слика 88 е прикажан дел од визуелизацијата на програмата со Jeliot3 пред вметнување на првиот јазел во дрвото. Јазелот е креиран но сè уште не се поставени вредностите на неговите полиња.



Слика 87. Дел од визуелизацијата на програмата TestAVL22.java со помош на Jeliot3 (дел при креирање на првиот јазел).

Овде убаво можат да се забележат составните елементи на објектот од класата BSTEntry, како и начинот на искористување на конструкторот на класата при формирање на објекти од истата. Може јасно да се види постапката како секое поле од објектот ја добива соодветната вредност која постои во повикот на даден метод во главната функција.

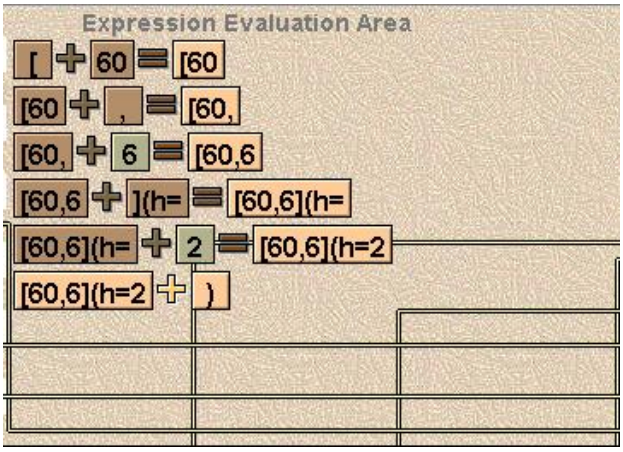
Секој нов повик на метод се сместува во делот за приказ на методи. Во овој дел се таложат сите повици кон методи како во магацин (Слика 88).



Слика 88. Дел од визуелизацијата на програмата TestAVL22.java со помош на Jeliot3 (делот со повици на методи).

Во делот за правење на пресметки во целост се прават сите пресметки кои се дадени во кодот и се неопходни за извршување на следниот чекор во

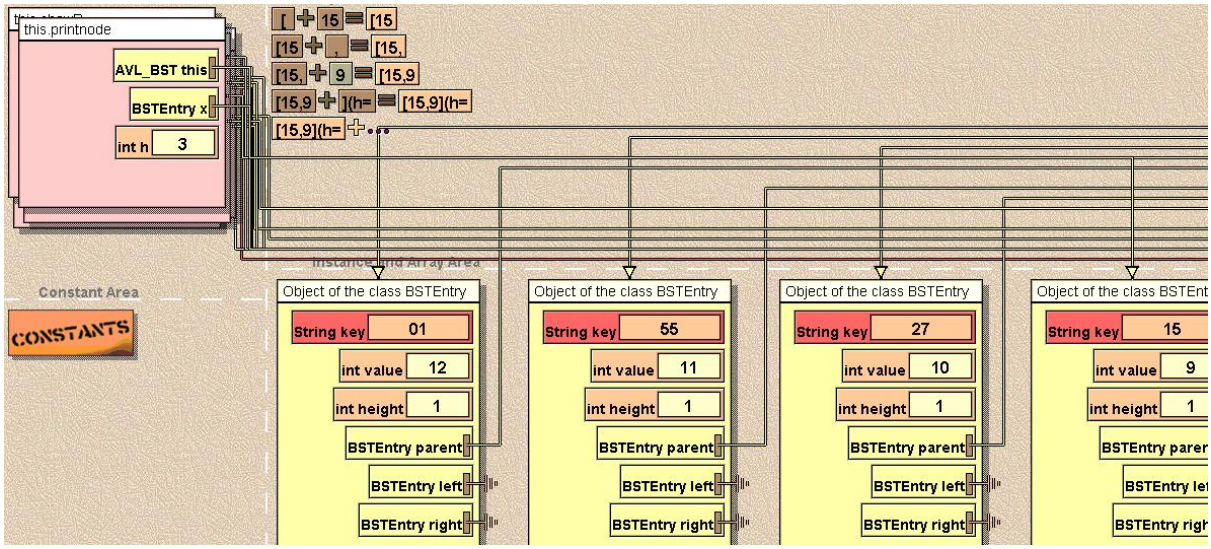
визуелизацијата. На слика 89 е претставена дел од визуелизацијата на програмата во делот за правење на пресметки.



Слика 89. Дел од визуелизацијата на програмата TestAVL22.java со помош на Jeliot3 (делот со пресметки).

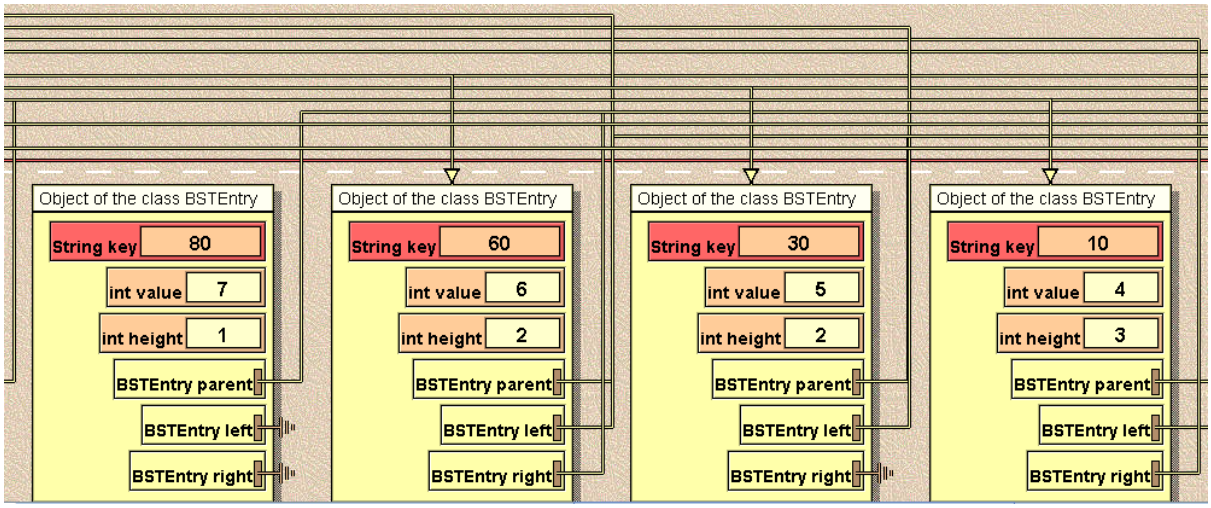
Jeliot3 прави опсежна и детална визуелизација, но тоа од друга страна бара многу време за следење на визуелизацијата во целост.

На почетокот од визуелизацијата кога има малку елементи кои се вклучени во програмата и мал е бројот на елементите што треба да се визуелизираат, ваквата визуелизација е добра и јасна за разбирање. Но, подоцна во конкретниот случај, кога веќе има додадено повеќе елементи за приказ во таеатерот ваквиот начин на визуелизација станува непогоден. Тешко можат да се забележат врските меѓу јазлите и не се опишани податочните структури, во овој случај дрва, на начин на кој полесно можат да се разберат односно на начин на кој тие функционираат. Врските меѓу јазлите постојат, но, откако ќе се вклучат повеќе јазли прегледноста на поврзаноста веќе се намалува. Од друга страна, јазлите како објекти на класата BSTEntry ја задржуваат својата видливост, но, се прикажани како елементи на некоја низа и редоследот на нивно сместување е според редоследот на кој се вметнати. Тоа е прикажано на слика 90.



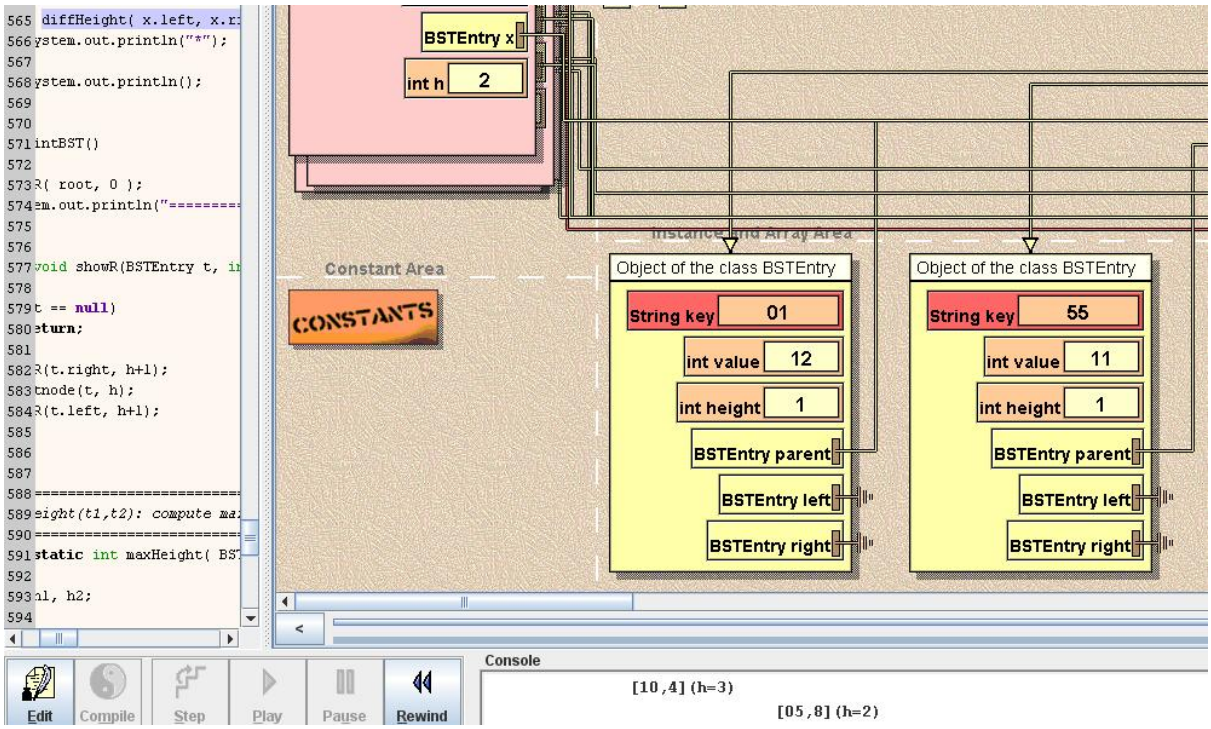
Слика 90. Дел од визуелизацијата на програмата TestAVL22.java со помош на Jeliot3 (ситуација со помала прегледност).

Исто така, и при бришењето на јазелот со вредност 80, тој останува да постои во делот со објекти само врските кон останатите јазли му се прекинати но, тоа тешко може да се забележи (Слика 91).



Слика 91. Дел од визуелизацијата на програмата TestAVL22.java со помош на Jeliot3 (ситуација при бришење на јазелот со вредност 80).

Слично како и кај претходната програма, во текот на анимацијата делот кој треба да се испечати се прикажува во козолниот дел. Конзолата е прикажана на Слика 92 во долниот десен агол.



Слика 92. Дел од визуелизацијата на програмата TestAVL22.java со помош на Jeliot3 (ситуација при печатење на резултатот на конзолата).

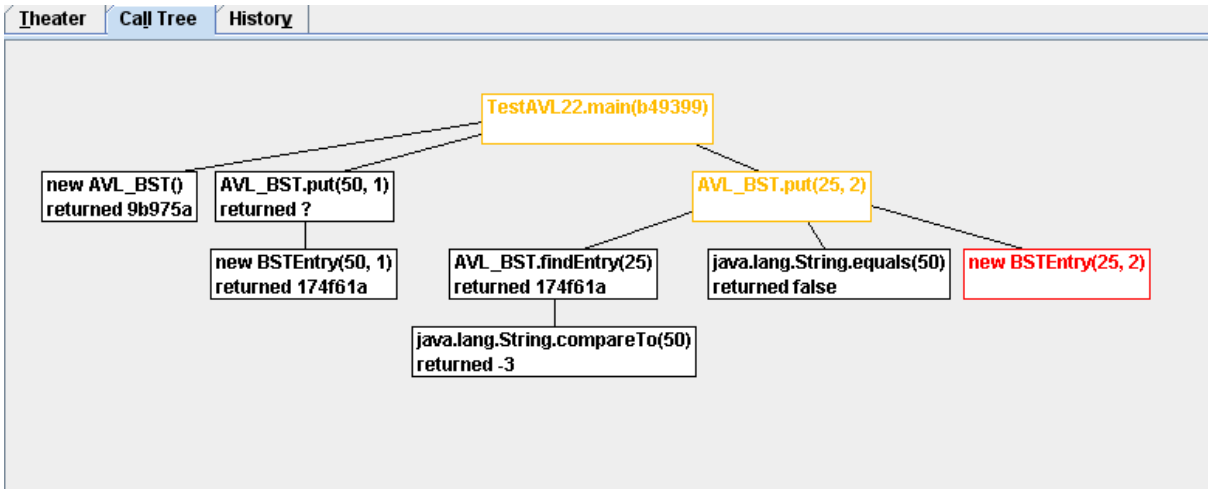
Излезот по завршувањето на анимацијата во конзолата е следен:

```
                                [80,7] (h=1)
                [75,3] (h=3)
                                [60,6] (h=2)
                                                [55,11] (h=1)
[50,1] (h=5)
                                [30,5] (h=2)
                                                [27,10] (h=1)
                [25,2] (h=4)
                                                [15,9] (h=1)
                                [10,4] (h=3)
                                                [05,8] (h=2)
                                                                [01,12] (h=1)
================================
=========================================
Testiranje na remove()
=========================================
After tri_node_restructure: (75,3) (60,6) (55,11)
                                [75,3] (h=1)
                [60,6] (h=2)
                                [55,11] (h=1)
[50,1] (h=5) *
                                [30,5] (h=2)
                                                [27,10] (h=1)
                [25,2] (h=4)
                                                [15,9] (h=1)
                                [10,4] (h=3)
                                                [05,8] (h=2)
                                                                [01,12] (h=1)
================================
=========================================
After tri_node_restructure: (50,1) (25,2) (10,4)
                                                [75,3] (h=1)
                                [60,6] (h=2)
```

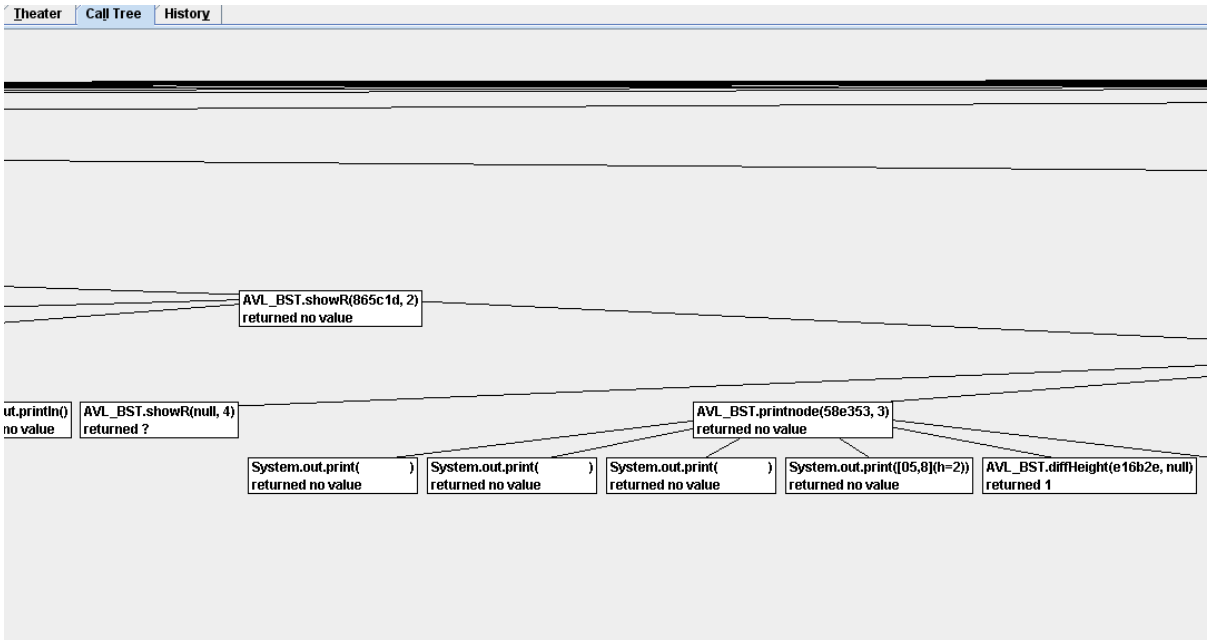
```
                                                    [55,11] (h=1)
                    [50,1] (h=3)
                                        [30,5] (h=2)
                                                    [27,10] (h=1)
[25,2] (h=4)
                                        [15,9] (h=1)
                    [10,4] (h=3)
                                        [05,8] (h=2)
                                                    [01,12] (h=1)
============================
=====================================
                                                    [75,3] (h=1)
                                        [60,6] (h=2)
                                                    [55,11] (h=1)
                    [50,1] (h=3)
                                        [30,5] (h=2)
                                                    [27,10] (h=1)
[25,2] (h=4)
                                        [15,9] (h=1)
                    [10,4] (h=3)
                                        [05,8] (h=2)
                                                    [01,12] (h=1)
============================
```

Негативна страна при визуелизирањето на оваа програма со помош на Jeliot3, е тоа што анимацијата се извршува многу долго време (повеќе од 5 часа), така што лесно може да се загуби вниманието од корисникот. Исто така, негативност претставува сè помалата прегледност на поврзаноста меѓу програмските елементи.

Со зголемувањето на програмскиот код исто така, се губи прегледноста и кај погледот на дрвото со повици кон методите. Во почетокот на анимацијата кога дрвото се состои од малку повици лесно може да се следи секој нов повик и начинот на кој тој се додава во дрвото, но, на крајот од анимацијата оваа слика значително се влошува (Слика 93).



а)



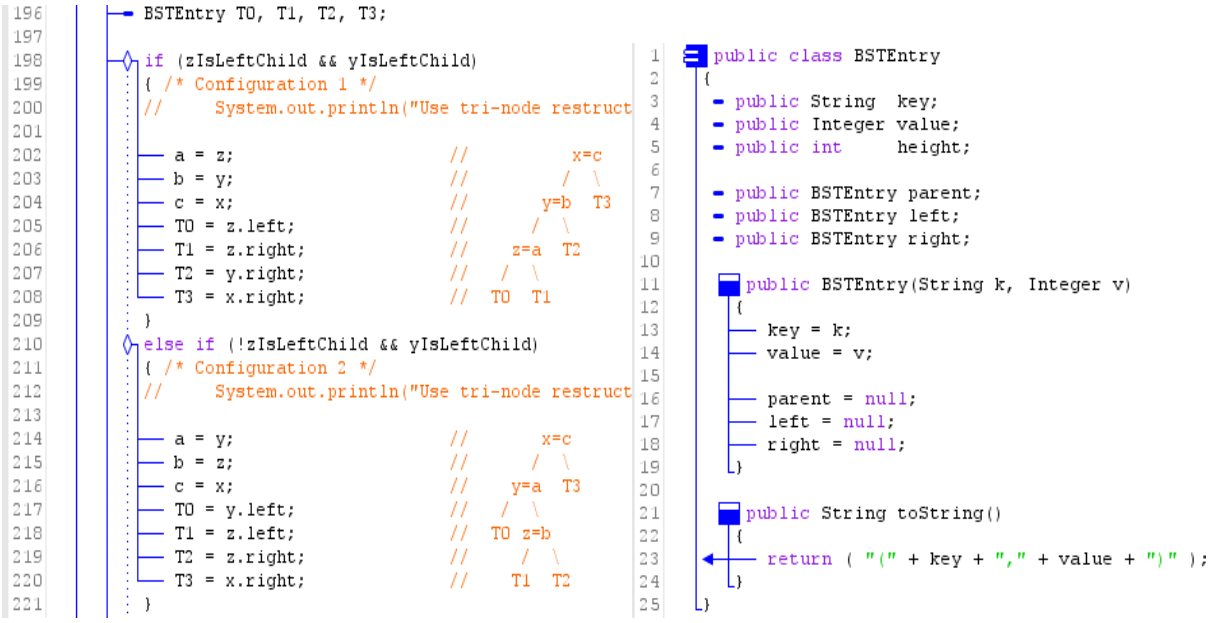
б)

Слика 93. Дел од визуелизацијата на програмата TestAVL22.java со помош на Jeliot3 (поглед на Call Tree).
а) ситуација на првиот дел од анимација (поголема прегледност);
б) ситуација на крајот од анимацијата (многу мала прегледност);

Ако сега истата програма ја пуштиме да се визуелизира со помош на дебагирање во jGrasp, сликата која се добива е поинаква.

Исто како и за претходниот пример и кај оваа програма уште веднаш при отварањето со jGrasp, веднаш се добива статичка визуелизација на кодот, преку која лесно можат да се разликуваат програмските елементи и да се откријат и поправат евентуалните синтаксички грешки. Преку дебагирањето од друга страна можат да се откријат можните грешки во однесувањето на програмата. Наоѓањето на грешките како и разликувањето на програмските

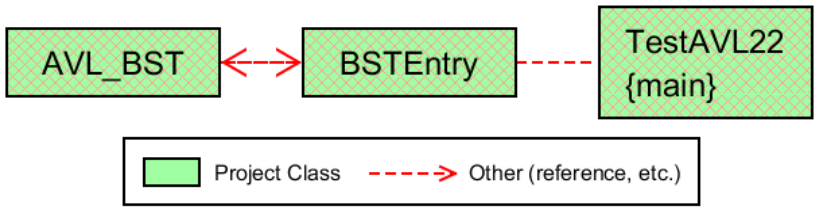
елементи кај вакви поголеми програми е доста значајно, бидејќи лесно можат да настанат грешки. Лесно може да се забележи кои сè типовите на податоци се користат, кои се променливите, кои се класите и кои се нивните атрибути, каде во кодот се наоѓаат функциите или кој дел од кодот претставува коментар. Исто така и нумерирањето на редовите помага полесно да бидат откриени можните грешки. Дел од таквата статичка визуелизација на кодот за оваа програма е прикажан на слика 94.



а) *б)*

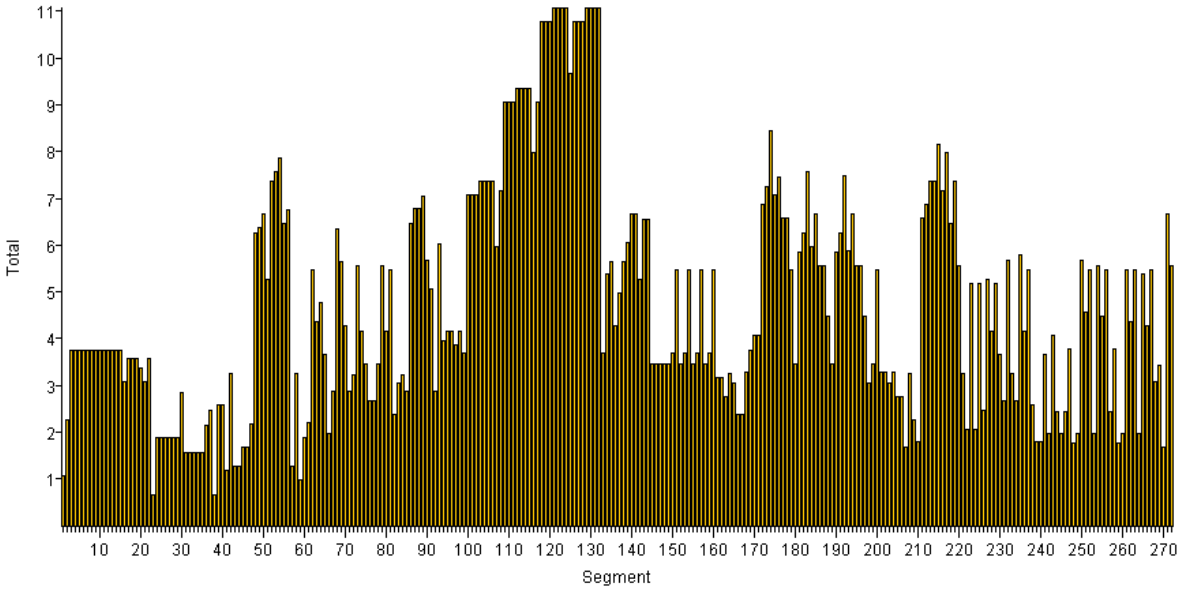
Слика 94. Дел од статичката визуелизацијата на програмата TestAVL22.java со помош на jGrasp.

Исто како и за претходната програма и овде jGrasp нуди генерирање на UML класен дијаграм, што претставува уште еден вид на статичка визуелизација на кодот. Генерираниот UML класен дијаграм за оваа програма е прикажан на слика 95.



Слика 95. UML класен дијграм за програмата TestAVL22 генериран со помош на jGrasp како дел од статичката визуелизација.

Друг вид на статична визуелизација кој ја нуди овој систем е преставувањето на графикот на комплексност на кодот (Comlexity Profile Graph-CPG). CPG за оваа програма е прикажан на слика 96.



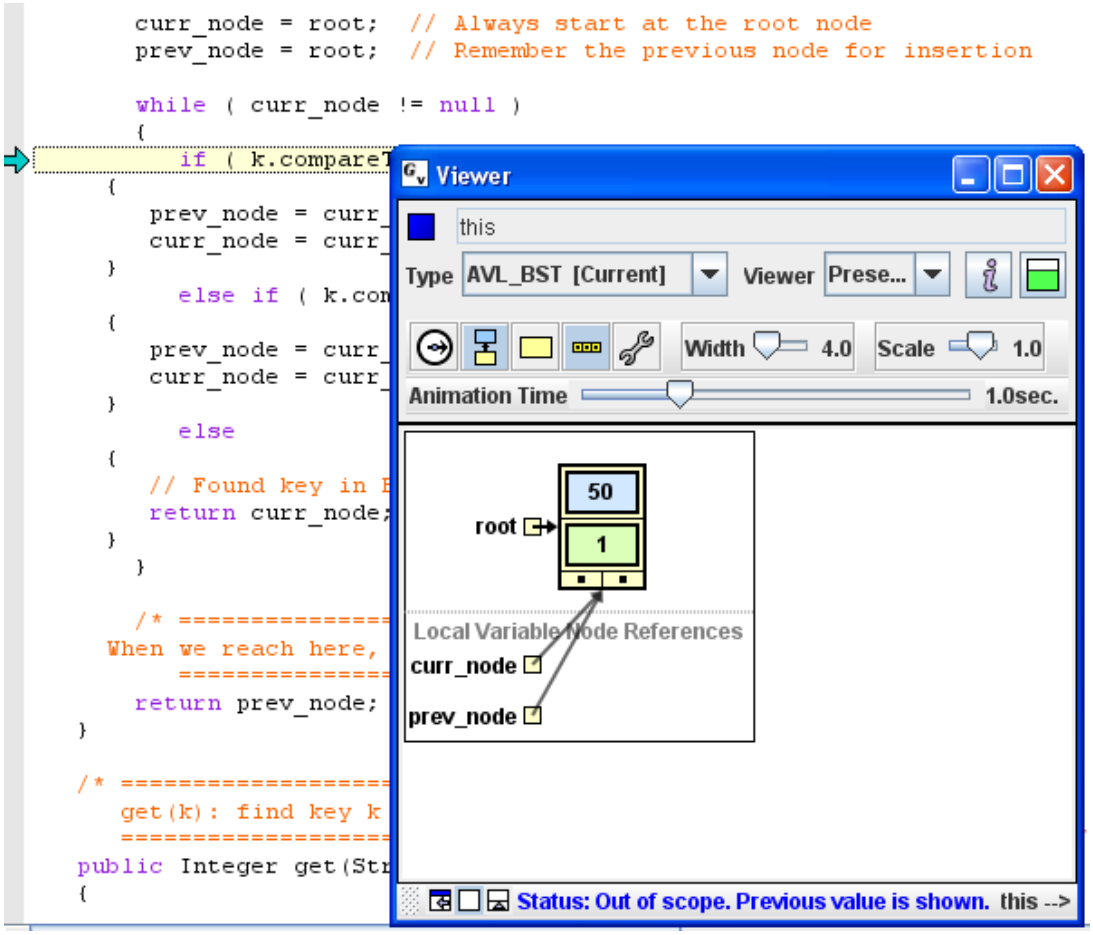
Слика 96. Comlexity Profile Graph-CPG за програмата TestAVL22 генериран со помош на jGrasp како дел од статичката визуелизација.

Од визуелизацијата се гледа дека програмата има вкупно 274 редови код со наредби, кои ја формираат големината на програмата, а најголема вкупна сложенсот на кодот има во 115-130-тиот ред.

Освен статичката, jGrasp и за оваа програма нуди динамичка визуелизација при дебагирање на програмата. Со помош на ваквото дебагирање лесно можат да се откријат некои лоши референцирања или дел од семантичките грешки.

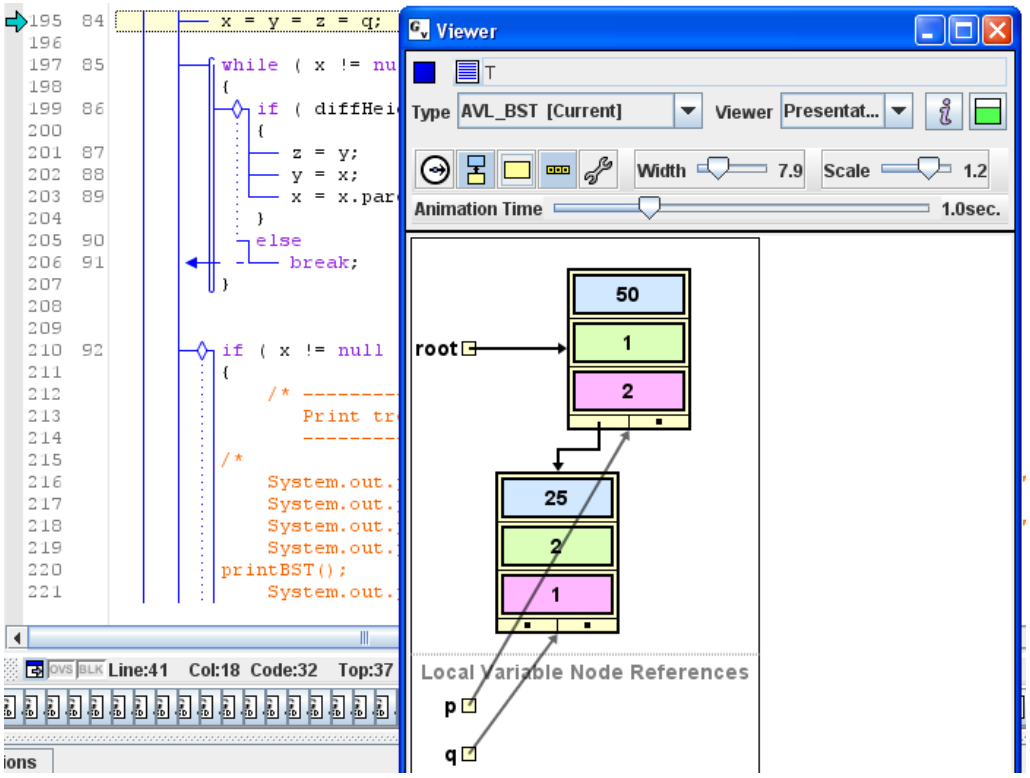
При динамичката визуелизација, jGrasp се задржува на суштинските податочни структури во програмата и не навлегува во визуелизирање на детали како што тоа го прави Jeliot3. Со помош на ваквата визуелизација може брзо и лесно да се разбере секој чекор од програмата и менувањето на податочната структура, во овој случај бинарно дрво, како и различните референцирања кои постојат во програмата.

Изгледот на јазелот од програмата визуелизиран со jGrasp заедно со променливите кои референцираат кон него е прикажан на слика 97.



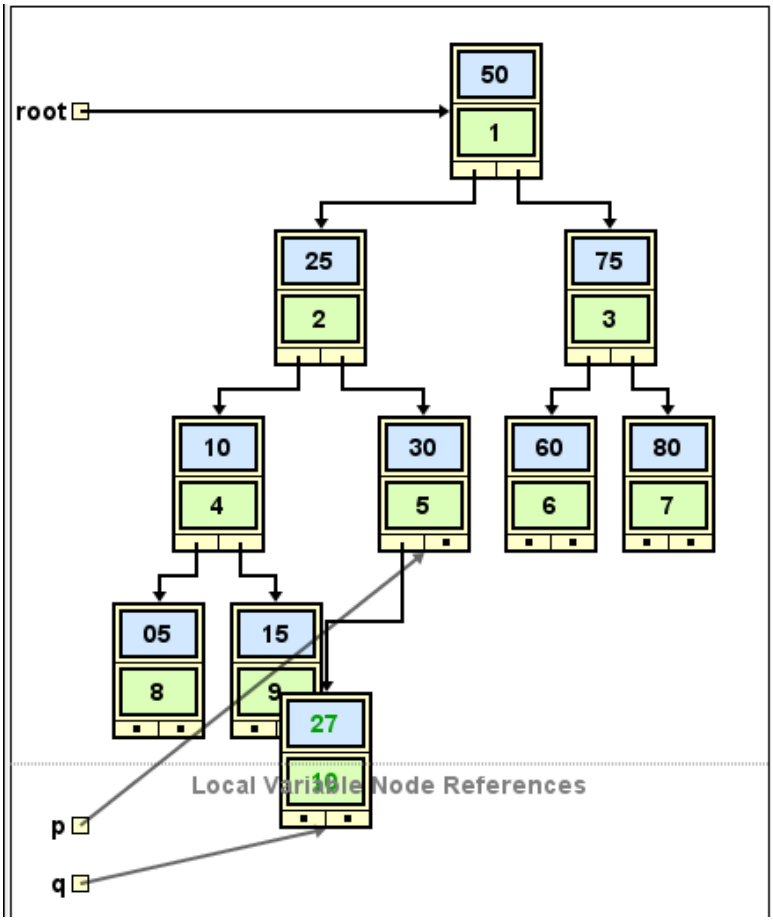
Слика 97. Дел од динамичката визуелизација за програмата TestAVL22 генериран со помош на jGrasp (приказ на јазелот зод AVL дрвото, заедно со неговите вредности за key и value).

При визуелирањето на јазелот се прикажани неговите вредности за key и value. Во секој момент се гледа кон кој јазел покажува коренот односно променливата root. Доплнително со правење на одредени измени во поставките може да се гледаат и другите атрибути како на пример, тековната висина на јазелот (Слика 98).



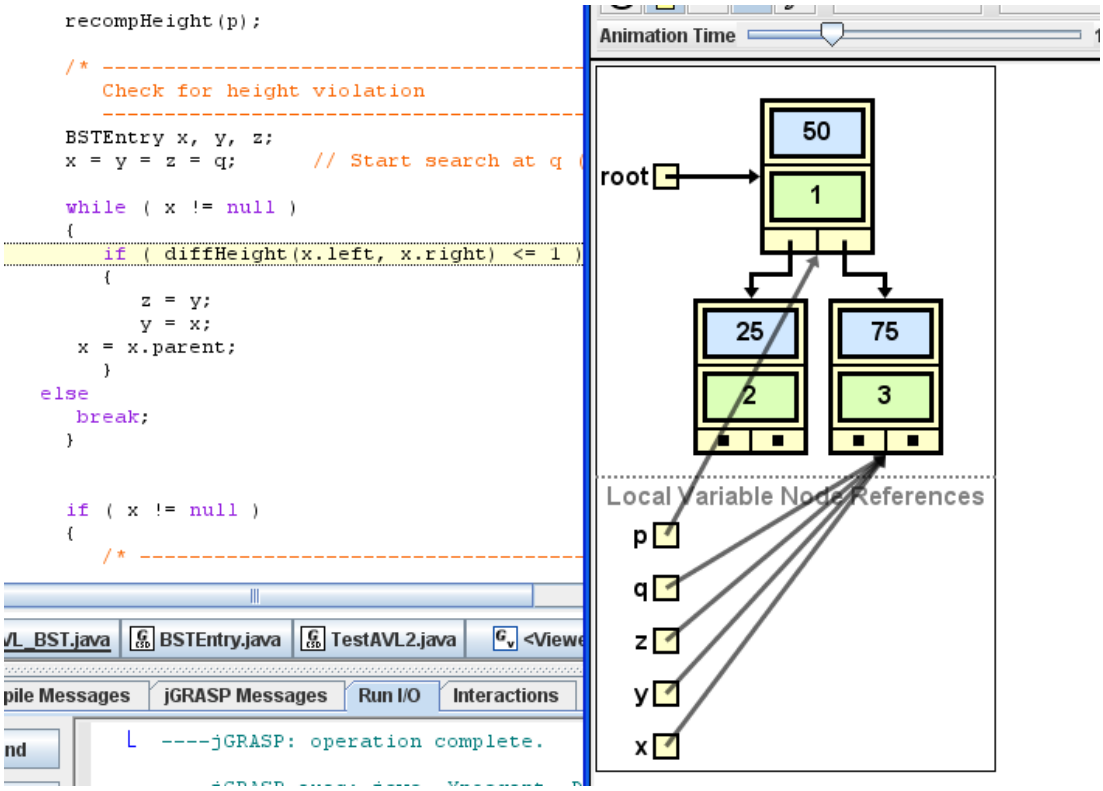
Слика 98. Дел од динамичката визуелизација за програмата TestAVL22 генериран со помош на jGrasp (приказ на јазелот од AVL дрво, заедно со неговите вредности за key и value и height).

Со помош на визуелизацијата преку jGrasp може да се забележи начинот на градење на бинарното дрво (Слика 99). Системот автоматски ја препознава структурата дека станува збор за бинарно дрво и ги претставува операциите кои се понудени во кодот (пребарување, додавање или бришење на јазли).



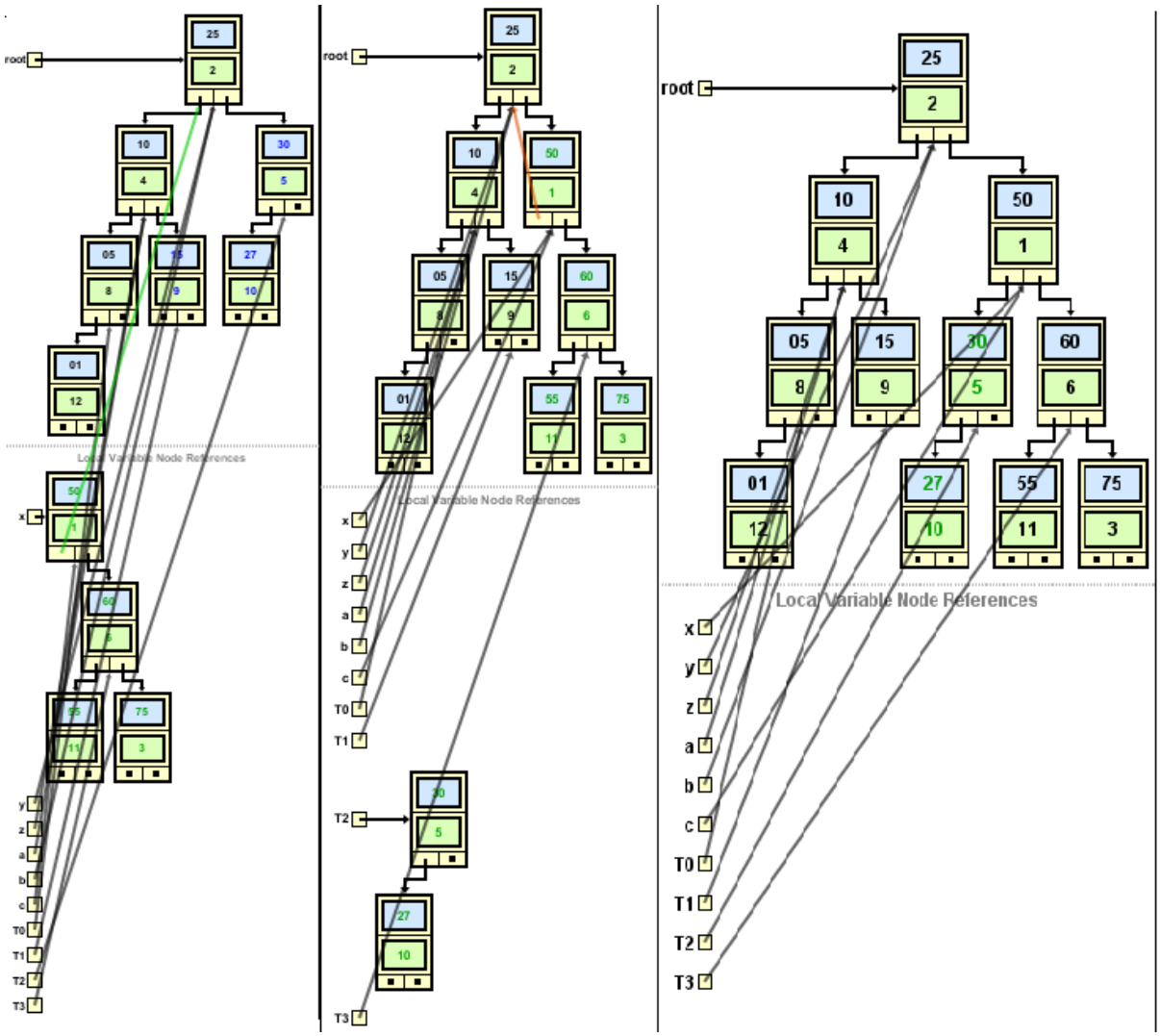
Слика 99. Дел од динамичката визуелизација за програмата TestAVL22 генериран со помош на jGrasp (приказ надел од постапката на градење на AVL дрвото).

Може да се разбере и начинот на референцирање на секоја променлива во даден чекор од дебагирањето (Слика 100). Истовремено може да се гледа и кодот, со цел да се забележи кој дел од кодот како влијае на измената на дрвото што се визуелизира (Слика 100).



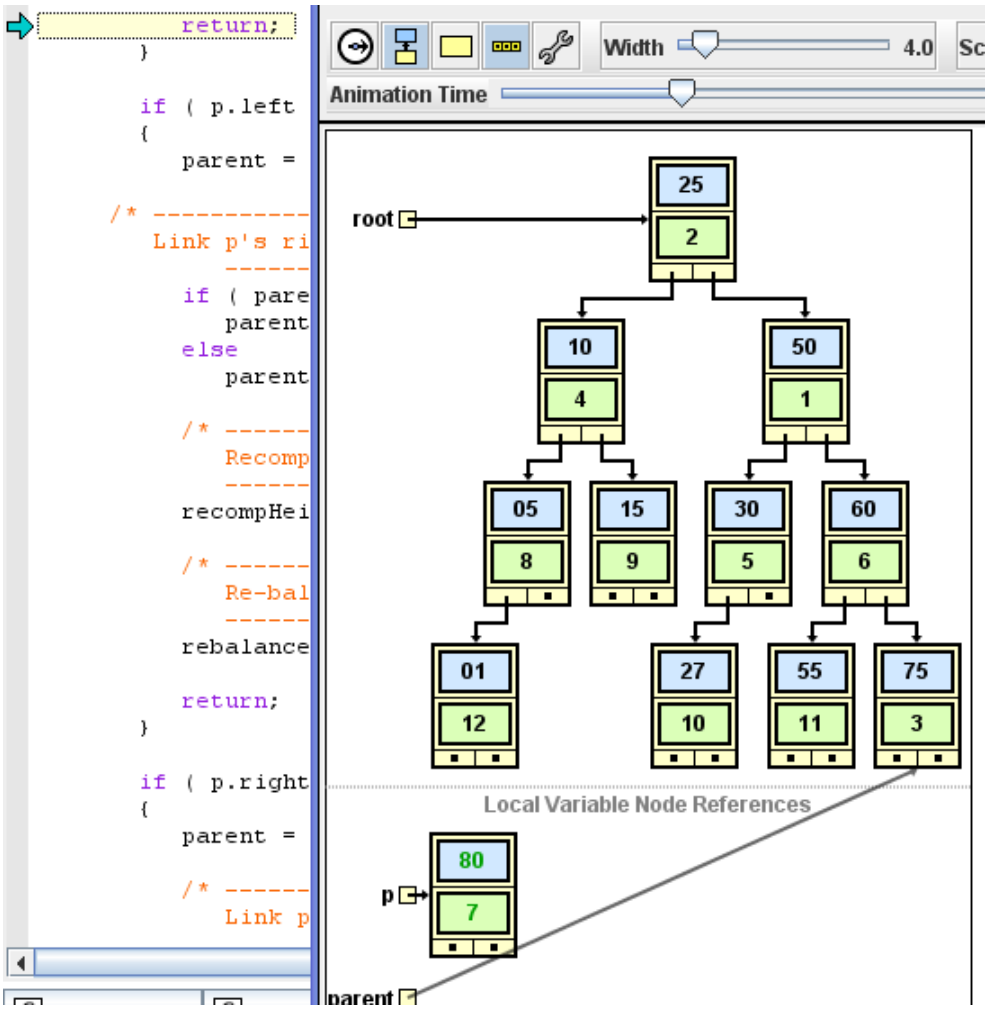
Слика 100. Дел од динамичката визуелизација за програмата TestAVL22 генериран со помош на jGrasp (приказ на тоа кон што покажуваат секоја од помошните променливи).

Откако се вметнати сите јазли наведени во главната функција, се оди на балансирање на дрвото, а со помош на визуелизацијата може да се забележи и начинот на кој се прави балансирањето (Слика 101).



Слика 101. Дел од динамичката визуелизација за програмата TestAVL22 генериран со помош на jGrasp (приказ на постапката за балансирање на бинарното дрво).

За разлика од Jeliot3, JGarsp јасно го визуелизира и бришењето на даден јазел и по неговото бришење тој веќе не постои во дрвото (Слика 102).



Слика 102. Дел од динамичката визуелизација за програмата TestAVL22 генериран со помош на jGrasp (приказ на бришење на јазел од AVL дрвото).

Со визуелизирање на поголема програма (програма која има повеќе код и вклучува повеќе пресметки) во jGrasp, се зголемува потребното време за дебагирање. Времето е значително поголемо од визуелизирање на помали програми (значително е зголемено времето за дебагирање во однос на претходната програма), но, сепак е не е премногу долго и е доволно да се разберат начините на функционирање на основните податочни структури. Доколку постојат некои модификации на некои од основните податочни структури кои ги поддржува системот, тој нема да понуди прецизна визуелизација на истите. Освен динамичка овој систем нуди и одлична статичка визуелизација посебно значајна кај поголемите програми.

За разлика од овие два система, SRec не овозможува визуелизации на поголеми програми. Иако оваа програма содржи рекурзии, сепак не може да се визуелизира со помош на SRec бидејќи SRec не поддржува рекурзии во кои има сложени структури или кориснички дефинирани типови на податоци. И да се работи за сложена програма во која има повеќе прости рекурзии SRec ќе ги

издвои рекурзиите и секоја ќе ја анимира посебно, тоа нема да влијае на прегледноста на визуелизацијата.

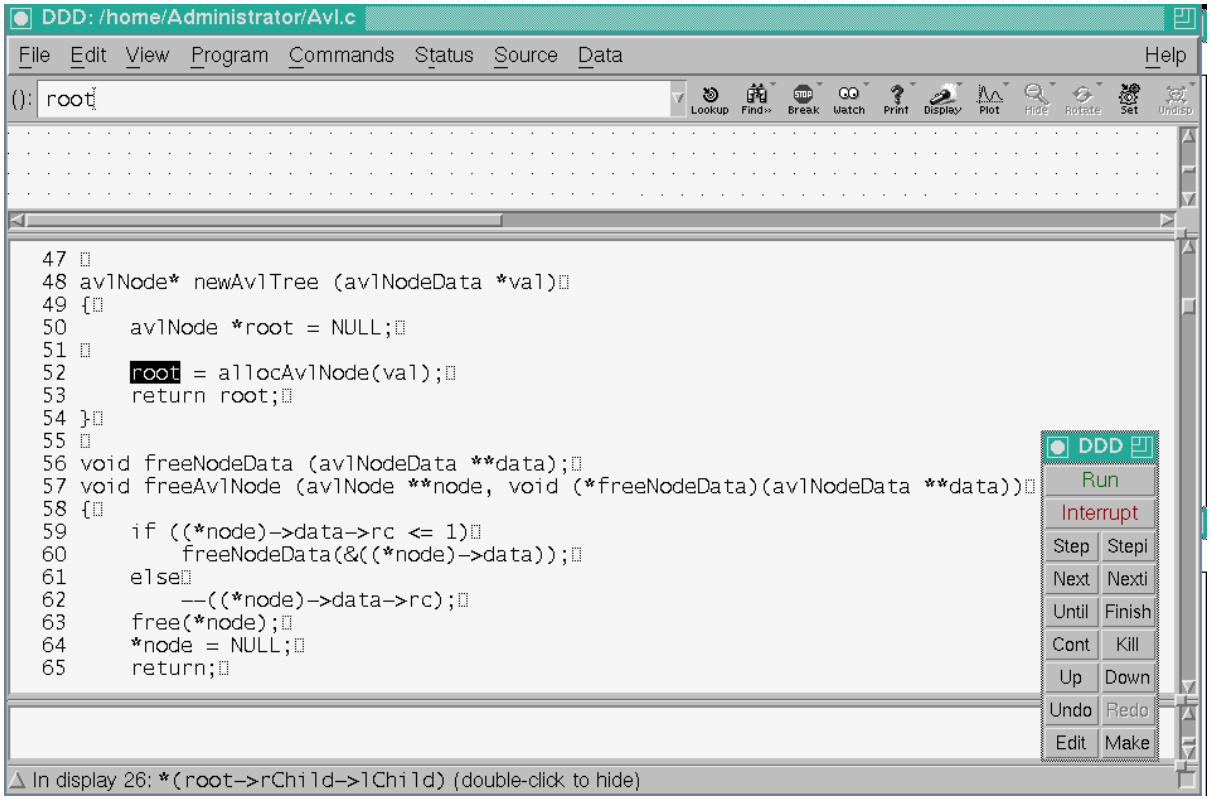
Влијание на посложена програма за SRec може да биде едноставна рекурзија која не содржи сложени структури но, има ставено голем број како почетна вредност. На тој начин се зголемува времето на извршување на анимацијата, а истовремено се намалува прегледноста при следењето на визуелизацијата. Значи SRec ги јавува истите карактеристики, при работа со рекурзии со големи почетни вредности, како останатите системи при работа со поголеми програми.

DDD системот не нуди визуелизација на програми напишани во Java, која би била значјна за разбирање или за следење на текот на програмата. Единствено нешто што може да се следи кај DDD за ваквите програми е истото како и за малите програми, односно менувањето на вредноста на одредени променливи во текот на дебагирањето (како што беше прикажано на слика 74 и слика 75) но, не и приказ на постоечките сложени структури и начинот на нивна работа.

6.3.3 Визуелизација на програмата `Avl.c` со DDD

Со цел да видиме како ќе се однесува DDD при визуелизација на поголеми програми напишани во C ќе искористеме програма напишана во програмскиот јазик C, што е слична со претходните примери за поголеми програми напишани во Java, односно програма што работи и имплементира AVL дрва. Во оваа програма има користење на референци како и единечни и двојни покажувачи. Кодот на оваа програма е даден во `Avl.c` Додаток 3.

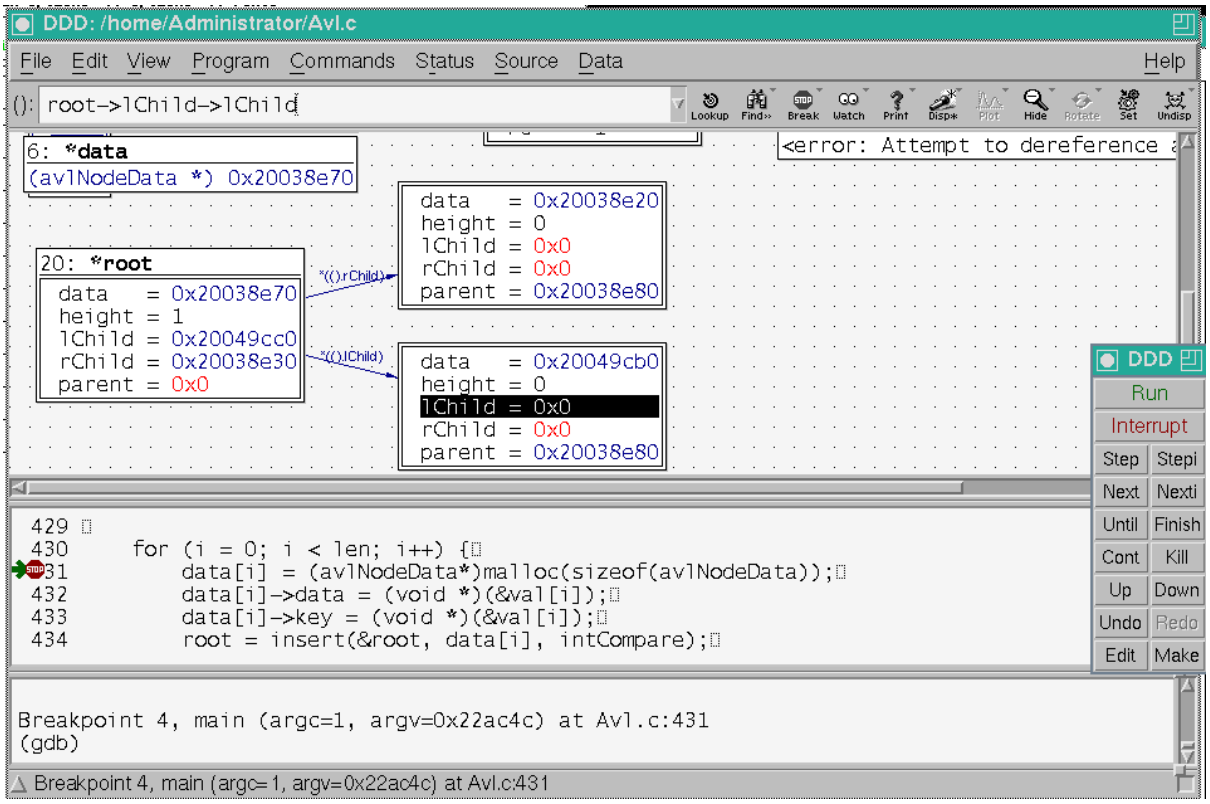
Ако оваа програма ја отвориме во DDD се отвора делот во кој се гледа кодот и се прикажува мала статичка визуелизација со нумерирање на секој ред (слика 103).



Слика 103. Дел од статичката визуелизација на програмата Avl.c со помош на DDD.

Ако ја дебагираме програмата во DDD, се отвора можност да ги избереме елементите кои сакаме да бидат прикажани во делот за визуелизација. Во овој случај е значајна интеракцијата на корисникот со системот, бидејќи во делот за визуелизација можат да се прикажат само елемнтите кои се во интерес на разгледување, а не целокупното однесување на програмата. Доколку корисникот не избере елементи што треба да бидат прикажани во делот за приказ, тогаш ништо нема да биде визуелизирано. DDD не дозволува голема интерактивност на корисникот во случај на едитирање на кодот, но интеракцијата е неопходна во текот на визуелизацијата односно дебагирањето.

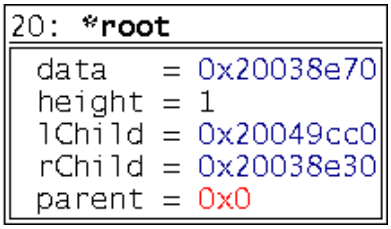
Бидејќи конкретната програма е за работа со AVL дрво, добро би било да биде прикажана поврзаноста на јазлите, а на тој начин полесно да се сфатат покажувачите, посебно двојните покажувачи (Слика 104).



Слика 104. Дел од динамичката визуелизација на програмата Avl.c со помош на DDD (приказ на вредност на покажувач).

При одењето на следни чекори во програмата, доколку во тој програмски дел не е програмскиот елемент (променливата, структурата или некој покажувач) што е избран за приказ, може делот за визуелизација да се исчисти, а потоа кога програмскиот тек ќе се врати во делот на интерес повторно да бидат прикажани побараните податоци.

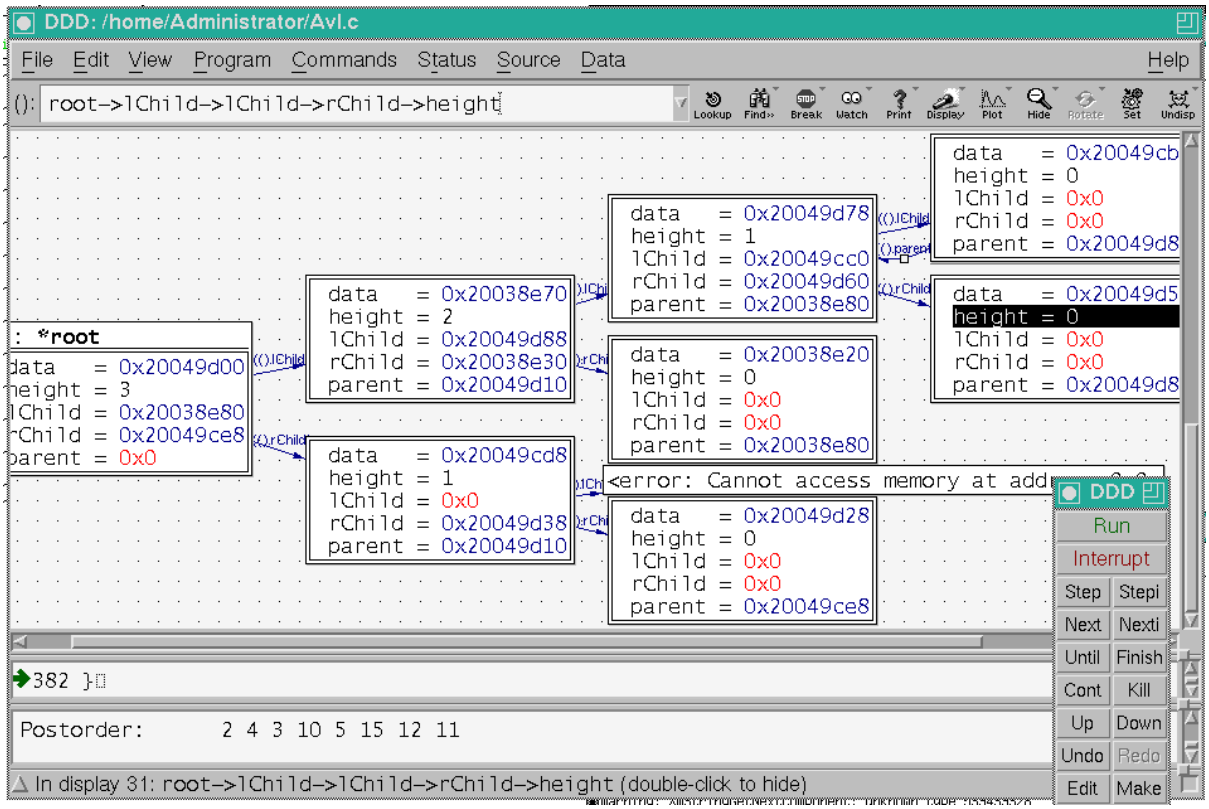
На визуелизацијата може јасно да се забележат врските меѓу јазлите и вредностите на атрибитите на структурите, меѓутоа некои од вредностите се претставени во машински код. На пример: вредноста на атрибутот data на јазелот наместо вредност 5 во приказот на визуелизацијата стои 0x20038e70. Исто така, и за вредностите на неговите деца стои адреса изразена во машински код (слика 105).



Слика 105. Приказ на јазел од програмата Avl.c што е дел од нејзината динамичка визуелизација со помош на DDD.

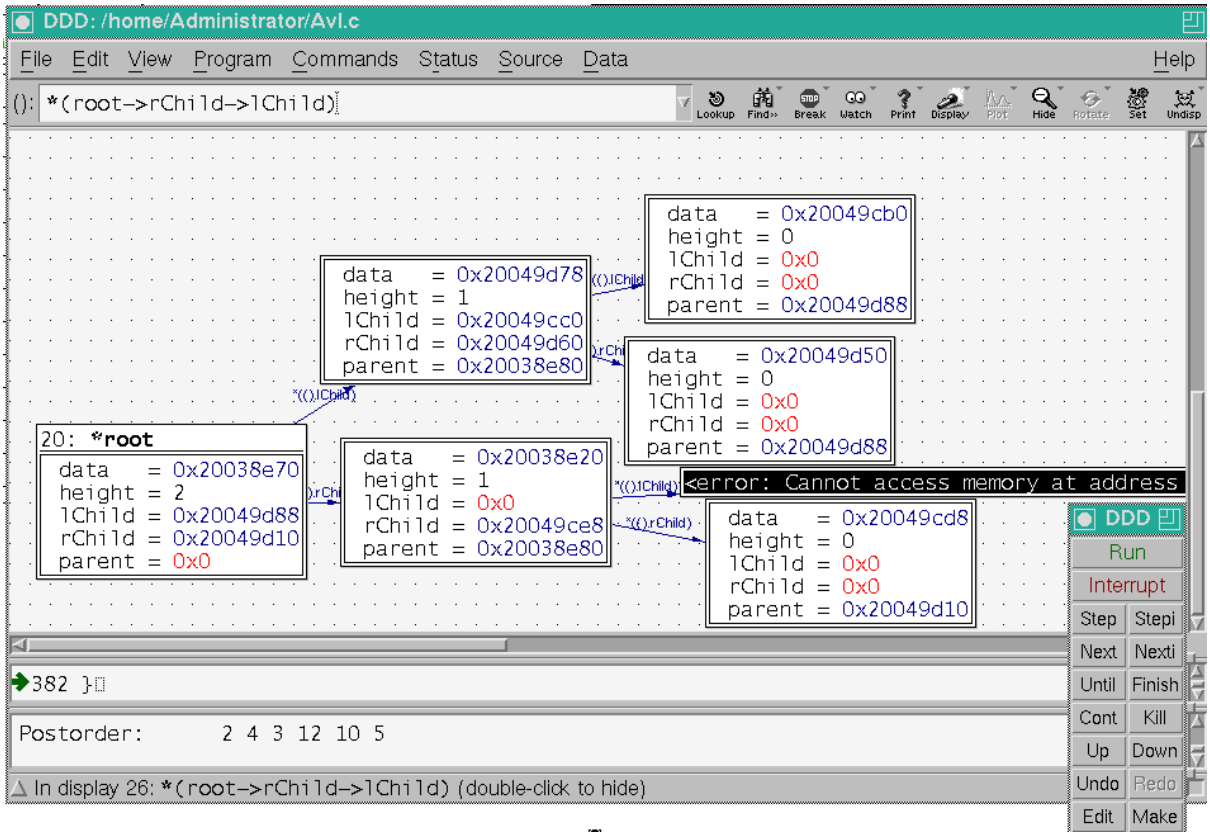
Ваквиот начин на претставување може да биде збунувачки за корисникот, но од друга страна дава јасна слика точно на која слободна мемориска локација покажува дадена променлива.

Во текот на дебагирањето со зголемување на бројот на јазли за приказ може да се намали видливоста на врските и поврзаноста на јазлите, затоа е значајно да се внимава и да не се прикажуваат елементи што не се од интерес, а истовремено да се внимава и на распоредот на прикажаните елементи. Во нашиот конкретен случај доколку се додадат повеќе јазли при дебагирањето, но, ако се внимава да не се додадат премногу други елементи и ако се внимава на нивниот распоред (да бидат така распределени да ја следат формата на дрвото), може да се добие јасна слика за поврзаноста (слика 106).



Слика 106. Дел од динамичката визуелизација на програмата Avl.c со помош на DDD (приказ на повеќе јазли од дрвото. Со добар распоред на јазлите прегледноста е добра).

Делот што го печати програмата како излез може да се забележи најдолу во конзолната линија (Слика 106). При бришењето на јазли тие автоматски се отсрануваат од дрвото и повеќе не се прикажуваат во делот за приказ. При бришењето може да настане промена во повразаноста на јазлите и затоа е потребна повторна интеракција на корисникот, да биде прикажана јасна слика без испреплетување на врските и преклопување на елементите (Слика 107)



Слика 107. Дел од динамичката визуелизација на програмата Avl.c со помош на DDD (приказ на дрвото, по бришење на 2 јазла).

Ако во некој јазел постои покажувач, но тој не покажува кон ништо, а е избран за приказ, тогаш како вредност на тој покажувач се јавува порака за грешка дека не може да се пристапи до таа адреса (слика 106, слика 107). Во горниот десен агол се гледа на кој начин од коренот се покажува до означениот елемент. На тој начин стануваат многу појасни повеќекратните покажувачи.

При визуелизацијата на поголеми програми со DDD се забележува дека е потребно повеќе време за визуелизација а тоа се должи на поголемиот број на чекори што треба да се изминат да се дебагира програмата. Може да настане потешкотија во приказот особено ако се јави потребата од приказ на повеќе елементи, но, сепак постои можност за претставување на јасна слика. Овде визуелизацијата и начинот на претставување на елемнтите во делот за приказ многу зависи од корисникот и со правилна распределба нема да настане непрегледност. Од друга страна, ваквиот вид на визуелизација не навлегува во деталите за начинот на работа на функциите туку се задржува само кон претставување на врските меѓу елементите и менување на нивните вредности. Разбирањето на начинот на работа на функциите останува само на корисникот без помош од визуелизацијата. Затоа оваа алатка може да се искористи за дебагирање кај поголеми програми со цел да се најдат лошите референцирања.

6.4 Генерална анализа на алатките за визуелизација

Од погоре направените анализи на алатките можеше да се забележи начинот на кој секоја од тие алатки се справува со визуелизацијата на различни програмски делови и во различни ситуации. Од досега кажаното, може да се направи една генерална заедничка групна анализа на сите четири алатки што ги користевме како претставници на алатките за визуелизација на софтвер. Причината за која ги избравме токму тие алатки за анализа е тоа што тие претставуваат цели системи, овозможуваат визуелизација на код кој корисникот ќе го внесе а не веќе предефиниран код. Исто така, не бараат измена на кодот или внесување на дополнителни елементи во кодот за генерирање на потребната визуелизација. Секоја од овие алатки ги визуелизира програмите од различни аспекти и сите тие можат да бидат искористени во различни ситуации. Сепак, нивна заедничка карактеристика е тоа што сите тие можат да се користат за подобро разбирање на кодот и полесно наоѓање на можните грешки. Сите овие програми најмногу се наменети за едукативни цели и би можеле да се користат за разбирање на различни аспекти од програмирањето. Генералната анализа на овие алатки е претставена во Табела 2.

Табела 2. Генерална анализа на алатките Jeliot3, jGrasp, SRec, DDD.

	Jeliot3	jGrasp	SRec	DDD
Можност за визуелизација на C/C++ програми	Не нуди можност	Нуди можност за статичка визуелизација но не и за динамичка	Не нуди можност	Најголема можност
Можност за визуелизација на Java програми	Нуди можност и за статичка и за динамичка визуелизација. Има мали ограничувања поради неиплементириност за некои понапредни карактеристики кои инаку постојат во Java. (пристап до super поле, повеќедимензионални полиња, методи кои враќаат инстанци од поле)	Нуди можност и за статичка и за динамичка визуелизација. Ограничувања при динамичката виузелизација на основни податочни структури.	Нуди можност, ограничување на програми кои во себе содржат рекурзивни функции.	Нуди многу мала можност
Визуелизација на рекурзивни функции	Нуди можност. Со претставување на секој повик кон методите и детално анализирање на пресметките може да	Може да се следи бројот на тоа колку пати ќе биде повикана рекурзиваната функција и како	Специјално наменета за рекурзивни функции. Може детално да се разбере начинот	Може само да се следи промената на вредностите на променливите кои учествуваат во функцијата

	се разбере работата на рекурзиите. За подобро претставување и разбирање на рекурзиите може да се искористи и делот на Call Tree	се менуваат вредностите на променливите при секоја итерација.	на функционирање на рекурзиите. Нуди повеќе погледи за приказ. Визуелизацијата ја врши со претставување на рекурзивни дрва.	при секоја итерација.
Визуелизација на покажувачи	Не нуди можност	Нуди мала можност	Не нуди можност	Најдобро можат да се следат покажувачите и референцирањата
Визуелизација на поголеми програми	Нуди можност. И кај релативно мали програми времето за завршување на анимацијата е долго, а веќе кај поголеми програми анимацијата трае и по неколку часови. Не се задржува на начинот на работење на функциите туку на визуелизација во детали на основните програмски елементи. Кон крајот на анимацијата може да настане непрегледност.	Нуди можност и за статичка (CPG, UML и CSD) и за динамичка визуелизација. Динамичката визуелизација ја прави преку дебагирање. Има ограничена можност за претстава на податочни структури. Колку е поголем кодот на програмата толку подолго време трае динамичката визуелизација.	Секоја од рекурзивните функции ги претставува одделно. Но, има ограничувања за типовите на податоци кои се користат. Не нуди можност за користење на сложени типови податоци.	Нуди можност, но потребно е корисникот да се фокусира на приказ на одредени програмски елемнти, со цел да не настане непрегледност. Главно се концентрира за приказ на врските (покажувачи и референцирања)
Време на траење на динамичката визуелизацијата	И по неколку часови кај големи програми.	Времето потребно за дебагирање, зависи од тоа каде е поставена точката на прекин. Колку повеќе чекори за дебагирање подолго време за визуелизација. Кај поголеми програми тоа е неопходно да се сфати начинот на работа на програмата.	Доколку влезниот аргумент на рекурзивната функција е поголем број во тој случај, визуелизацијата трае долго додека не се стигне до основниот случај. Но, за разбирање на рекурзијата доволни се и мали вредности како влезни аргументи.	Време колку што е потребно за дебгирање, зависи од тоа каде е поставена точката на прекин. Колку повеќе чекори за дебагирање подолго време за визуелизација.
Јасност и прегледност на визуелизацијата	Голема прегледност кај мали програми, колку се зголемува	Јасна прегледност за визуелизациите	Јасна слика при анимација на рекурзии со мали	Прегледноста повеќе зависи од корисникот. Тој

	сложеноста на програмата и бројот на креирани објекти, прегледноста се намалува. Кај поголеми програми, голема прегледност на почетокот од анимацијата, кон крајот таа се губи.	што ги нуди како статичка така и динамичка.	вредности за влезни аргументи. Колку е поголем бројот на влезни аргументи толку е помала прегледноста кон крајот на анимацијата.	одлучува кои елементи ќе бидат претставени и на кои начин тие ќе бидат распределени. Програмата автоматски ги поставува само врските.
Начин на визуелизација	Визуелизацијата се генерира автоматски	Динамичката визуелизација се генерира при дебагирање со мала интервенција на корисникот. Понатаму промените во визуелизацијата со секој чекор се генерираат автоматски.	Визуелизацијата се генерира автоматски	Динамичката визуелизација може да се генерира при дебагирање, но што ќе биде прикажано зависи од корисникот.
Едноставност за користење од страна на корисникот	Најмногу пријателски настроен кон корисникот и многу лесен за употреба и инсталирање.	Пријателски настроен кон корисникот, но не во толкава мера. Нуди различни можности но можни се мали потешкотии при генерирање на анимација. Лесен за инсталација.	Лесен за инсталација и прилично лесен за употреба, мали потешкотии при генерирање на анимации за рекурзии од типот раздели и владеј.	Потежок за инсталација и не многу пријателски настроен.
Степен на интеракција со корисникот	Интеракцијата на корисникот е во можноста на едитирање на кодот и менување на различни подгледи. Во делот на анимацијата корисникот нема некоја особена интеракција освен регулирање на брзината анимацијата паузирање и стопирање.	Корисникот може да прави измени во кодот, може да менува различни погледи. Во текот на анимацијата корисникот ја опредлува брзината и обликот и бојата прикажаните елементи. На друг начин корисникот не може да влијае на визуелизацијата	Корисникот не може да го менува кодот откако е внесен во програмата. Не може да се влијае ниту на анимацијата. Тој може само да ги менува погледите на пристап и во мал степен да го менува изгледот на корисничкиот интефејс.	Корисникот не може да го менува кодот кога програмата е отворена со DDD, но од друга страна значително многу може да влијае во поставеноста и прикажувањето на визуелните елементи.
Најголема можна употреба	Визуелизирање на мали и едноставни програми со цел	Визуелизирање на програми кои во себе ги	Визуелизирање на едноставни рекурзивни	Визуелизирање на покажувачи и референци,

	запознавање, изучување и разбирање на функционирањето на основните програмски елементи. Посебно корисна за разбирање на објектно-ориентирани концепти.	користат некои од основните структури на податоци како низи, листи хеш табели или бинарни дрва. На тој начин многу полесно би можеле да се разберат начините на кои тие функционираат.	функции кои во себе не вклучуваат сложени податочни типови. Со цел разбирање на начинот на функционирање и работа на рекурзиите.	особено повеќекратни покажувачи, со цел да се разбере нивното функционирање.

7. Заклучок

Визуелизацијата на софтвер претставува значаен дел од изучувањето и користењето на софтверот. Неговата значајност е особено голема посебно поради фактот што софтверот и неговите компоненти се всушност природно не видливи и неопипливи. Единствениот видлив дел од софтверот е кодот, кој сам по себе претставува само множество од наредби кои работат како една целина. Начинот на работа на кодот може многу тешко да се разбере ако тој е претставен само со линии текст. Претставувањето на кодот со помош на слика, што не мора да е прецизна, туку доволна да предизвика асоцијации во главата на корисникот за тој да може полесно да го разбере кодот, игра значајна улога во изучувањето на програмите, нивниот развој и наоѓање на можните грешки. Ако дополнително се воведе анимацијата односно динамичката визуелизација за претставување на начинот на функционирање на програмите тогаш ефектот на разбирање би бил уште поголем.

Постојат голем број на системи и алатки за визуелизација на софтвер, на кои работат цели тимови и кои се доста усовршени и нудат различни можности. Поради тоа наша цел не беше да изградиме нов систем кој секако ни од далеку нема да може да ги постигне истите функционалности како некој што некаде веќе постои, туку да ги истражиме веќе изградените системи. Од истражувањето требаше да издвоиме неколку системи кои вклучуваат функционалности на повеќе други и нудат повеќе различни можности. Бидејќи основното нешто при работа со софтвер е неговото добро разбирање, повеќе се насочивме кон изнаоѓање на алатки кои во голема мера го помагаат тоа. Особено ни беше значаен фактот програмите кои тие можат да ги визуелизираат да не бидат веќе предефинирани туку корисникот сам да одлучува за програмата што сака да му биде визуелно претставена. Друг параметар кој требаше да ни биде заеднички е да можат да визуелизираат програми напишани во некои од постоечките програмски јазици без да се додадваат делови или да се прават измени во кодот. Во спротивно би се добило само вртење во круг.

Од истражувањето на постоечките алатки се одлучивме за анализа на четири од нив, Jeliot3, jGrasp, SRec и DDD. Откако детално ги разработивме,

секоја од овие алатки ја тестиравме за различни софтверски делови кои се потешки за сфќање (рекурзија, покажувачи, различни податочни структури, некои видови на алгоритми). Секоја од алатките ја тестиравме и за работа со поголеми програми. Потоа направивме и групна анализа и споредба на алатките. Од резултатите кои ги добивме и направената анализа забележавме дека иако сите нудат некаков вид на визуелизација на исти програми, секоја од нив нуди различно нивно визуелно претставување. Сите алатки ги претставуваат програмските елементи различно и за некои програмски елементи нудат подобра визуелизација а за други полоша. Така на пример, Jeliot3 овозможува најдобра визуелизација на основните програмски елементи и објектно/ориентираните концепти, SRec е специјално наменет за разбирање на рекурзиите, DDD најдобро ги визуелизира покажувачите, а jGrasp нуди најмногу можности, но, сепак најдобро се справува со визуелизација на програмите што вклучуваат основни податочните структури. Затоа секоја од овие алатки може да се користи за различи програмски делови или визуелизирање на иста програма од различни аспекти според потребите на корисниците.

Додаток 1

```
//AvlTest1.java

import java.io.*;
import java.util.*;


public class AvlTest1 {
   public static void main(String[] args) throws IOException {
      JazelAVL p=null;
      TreeAVL t= new TreeAVL();

      p=t.insert('d',15,p);
      p=t.insert('k',6,p);
      p=t.insert('a',18,p);
      p=t.insert('l',3,p);
      p=t.insert('a',2,p);
      p=t.insert('e',4,p);
      p=t.insert('s',7,p);
      p=t.insert('n',13,p);
      p=t.insert('a',9,p);
      p=t.insert('r',17,p);
      p=t.insert(':',20,p);
      p=t.insert(')',21,p);

      t.inorder(p);

   }

}
class JazelAVL
{
   public int info;
   public char label;
   public JazelAVL left;
   public JazelAVL right;
   int visina;

   public JazelAVL()     {}

   public JazelAVL(int i)   {   info=i;     }
   public JazelAVL(int i, char l)   {   info=i;  label=l;   }

   public int vis(JazelAVL p)  {
      if (p==null)
         return -1;
      else
         return p.visina;
   }

   public int maks(int a, int b) {
      if (a>b)
         return a;
      else
         return b;
   }

   public void pecatiJazel()      {
      System.out.print(info+" "+label);
   }
}
class TreeAVL extends JazelAVL
{
   public TreeAVL()  {}

 /* Ovaa funkcija moze da se povika samo ako k2 ima levo dete */
/* Pravi rotacija pomegu jazel (k2) i negovoto levo dete */
```

```
/* Se azuriraat visinite, a potoa se vraka noviot koren */

   public JazelAVL singleRotateWithLeft(JazelAVL k2)
   {
      JazelAVL k1;
      k1=k2.left;
      k2.left=k1.right;
      k1.right=k2;
      k2.visina=maks(vis(k2.left), vis(k2.right))+1;
      k1.visina=maks(vis(k1.left), k2.visina)+1;
      return k1;   //noviotkoren
   }

/* Ovaa funkcija moze da se povika samo ako k1 ima desno dete */
/* Pravi rotacija pomegu jazel (k1) i negovoto desno dete */
/* Se azuriraat visinite, a potoa se vraka noviot koren */

   public JazelAVL singleRotateWithRight(JazelAVL k1)
   {
      JazelAVL k2;
      k2=k1.right;
      k1.right=k2.left;
      k2.left=k1;
      k1.visina=maks(vis(k1.left), vis(k1.right))+1;
      k2.visina=maks(vis(k2.right), k1.visina)+1;
      return k2;   //noviotkoren
   }
   public JazelAVL doubleRotateWithLeft(JazelAVL k3)
   {
            /* Rotacijapomegu K1 i K2 */
      k3.left = singleRotateWithRight( k3.left );
            /* Rotacijapomegu K3 i K2 */
      return singleRotateWithLeft( k3 );
   }

 /* Ovaa funkcija moze da se povika samo ako k3 ima desno dete */
/* i desnoto dete na K3 ima levo dete */
/* Se pravi desna -leva rotacija */
/* se vraka noviot koren */

   public JazelAVL doubleRotateWithRight(JazelAVL k3 )
   {
            /* Rotate between k1 and k2 */
      k3.right = singleRotateWithLeft( k3.right );
            /* Rotate between k3 and k2 */
      return singleRotateWithRight( k3 );
   }
   public JazelAVL insert(char label,int x, JazelAVL t)
   {
      if( t == null )
      {
             /* Kreiraj I vratisteblo so edenjazel */
         t = new JazelAVL() ;
         t.info = x; t.label=label; t.visina = 0;
         t.left = t.right = null;
      }
      else
         if( x < t.info )
         {
            t.left = insert( label,x, t.left );
            if (( vis( t.left ) - vis( t.right )) == 2 )
               if( x < t.left.info )
                  t = singleRotateWithLeft( t );
               else
                  t= doubleRotateWithLeft( t );
         }
         else
            if ( x > t.info ) {
               t.right = insert( label,x,  t.right );
               if( vis( t.right ) - vis( t.left ) == 2 )
                  if( x> t.right.info )
```

```
                    t = singleRotateWithRight( t );
                else
                    t =doubleRotateWithRight( t ); }
    /*Else x e vo stebloto; ne pravime nisto */
       t.visina = maks(vis(t.left),vis(t.right))+1;
       return t;
    }

    public void inorder(JazelAVL r)
    {
       if (r!=null)
       {
          inorder(r.left);
          System.out.print(r.info+" "+ r.label+" ");
          inorder(r.right);
       }
    }

} // kraj za klasata
```

Додаток 2

```
//TestAVL22.java
import java.io.*;
import java.util.*;

public class TestAVL22
{
   public static void main(String[] args)
   {
      AVL_BST T = new AVL_BST();

      T.put("50", 1);
      T.put("25", 2);
      T.put("75", 3);
      T.put("10", 4);
      T.put("30", 5);
      T.put("60", 6);
      T.put("80", 7);
      T.put("05", 8);
      T.put("15", 9);
      T.put("27", 10);
      T.put("55", 11);
      T.put("01", 12);
      T.printBST();

      System.out.println("=============================================");
      System.out.println("Testiranje na remove();
      System.out.println("=============================================");

      T.remove( "80" );
      T.printBST();
      System.out.println("=============================================");
   }
}

class BSTEntry
{
   public String  key;
   public int value;
   public int     height;

   public BSTEntry parent;
   public BSTEntry left;
   public BSTEntry right;

   public BSTEntry(String k, int v)
   {
      key = k;
      value = v;

      parent = null;
      left = null;
      right = null;
   }

   public String toString()
   {
      return ( "(" + key + "," + value + ")" );
   }
}

/* ============================================================
   Pretstavuvanje na BST so balansirana visina (= AVL)
   Sekoj jazel ima visina so cel da se odluci dali BST e balanisrano.
   ============================================================ */

class AVL_BST
```

```
{
   public BSTEntry root;    // Referencira na korenot od BST

   public AVL_BST()
   {
      root = null;
   }

   /* ================================================================
      findEntry(k): naoga jazel so kluc k
      Vraka kako rezultat:  referenca do (k,v) ako k e vo BST
               Referenca do roditelot na(k,v) ako k ne e vo BST
      ================================================================ */
   public BSTEntry findEntry(String k)
   {
      BSTEntry curr_node;   // pomosna promenliva
      BSTEntry prev_node;   // pomosna promenliva

       /* ----------------------------------------
     Naoganje na jazelot so kluc key == "k" vo BST
         ---------------------------------------- */
      curr_node = root;  // Sekogas se zapocnuva so jazelot koren
      prev_node = root;  // Potrebno e da se zapomni prethodniot jazel

      while ( curr_node != null )
      {
         if ( k.compareTo( curr_node.key ) < 0 )
         {
            prev_node = curr_node;       // Se zacuvuva prev. node
            curr_node = curr_node.left;  // Prodolzuva prebaruvanjeto vo levoto poddrvo
         }
         else if ( k.compareTo( curr_node.key ) > 0 )
         {
            prev_node = curr_node;       // Se zacuvuva prev. node
            curr_node = curr_node.right; // Prodolzuva prebaruvanjeto vo desnoto poddrvo
         }
         else
         {
         // Najden e jazelot vo BST
            return curr_node;
         }
      }

       /* ======================================
     Koga programata stignuva ovde, k ne e vo BST
         ====================================== */
      return prev_node;        // Vrati go prethodniot (roditelski) jazel
     }

   /* ================================================================
      get(k): njadi go jazelot so kluc k I vrati assoc. value
      ================================================================ */
   public int get(String k)
   {
      BSTEntry p;   // pomosna promenliva

       /* ----------------------------------------
     Najdi go jazelot so kluc key == "k" vo BST
         ---------------------------------------- */
      p = findEntry(k);

      if ( k.equals( p.key ) )
         return p.value;
      else
         return null;
   }

   /* ================================================================
      put(k, v): smesti go parot (k,v) vo BST
```

```
      1. ako klucot "k" e najden vo BST, se zamenuva so vrednosta
         Koja e asocirana so klucot "k"
      1. ako klucot "k" ne e najden vo BST, se dodava nov jazel so
     Par od vrednosti (k, v)
   ================================================================ */
public void put(String k, int v)
{
   BSTEntry p;   // pomosna promenliva

   if ( root == null )
   {  // Vmetni vo prazno BST

      root = new BSTEntry( k, v );
      root.height = 1;
      return;
   }

    /* ------------------------------------------
  Najdi jazel so key == "k" vo BST
       ------------------------------------------ */
   p = findEntry(k);

   if ( k.equals( p.key ) )
   {
      p.value = v;            // Azuriraj ja vrednosta
      return;
   }

    /* ------------------------------------------
  Vnesi nov jazel kako vnes so par od vrednosti (k,v) kako dete na p !!!
       ------------------------------------------ */
   BSTEntry q = new BSTEntry( k, v );
   q.height = 1;

   q.parent = p;

   if ( k.compareTo( p.key ) < 0 )
      p.left = q;               // dodadi go q kako levo dete
   else
      p.right = q;              // dodadi go q kako desno dete

    /* ------------------------------------------------------------------
       Napravi nova presmetka za visinite na site roditelski jazli...
       ------------------------------------------------------------------ */
   recompHeight(p);

    /* ------------------------------------------
       Proverka za mozna promena na visinata (height)
       ------------------------------------------ */
   BSTEntry x, y, z;
   x = y = z = q;      // zapocni prebaruvanje od q (noviot jazel)

   while ( x != null )
   {
      if ( diffHeight(x.left, x.right) <= 1 )
      {
         z = y;
         y = x;
         x = x.parent;
      }
      else
         break;
   }


   if ( x != null )
   {
       /* ------------------------------------------
          Ispecati go stebloto posle vnesuvanjeto
          ------------------------------------------ */
```

```
   /*
       System.out.println("*****************************************");
       System.out.println("Unbalanced AVL tree after insertion !!!");
       System.out.println("*****************************************");
       System.out.println("Tree before rebalance:\n");
   printBST();
       System.out.println("-----------------------------------------");
   */

      tri_node_restructure( x, y, z );
   /*
       System.out.println("Tree after rebalance:\n");
   printBST();
       System.out.println("*****************************************");
   */
   }
 }

/* ================================================
   tri_node_restructure(x, y, z):

         x = parent(y)
         y = parent(z)
   ================================================ */
public BSTEntry tri_node_restructure( BSTEntry x, BSTEntry y, BSTEntry z)
{
  /* ****************************************************************
     Opredeli ja vrskata dete roditel pomegu (y,z) i (x,y))
     **************************************************************** */
   boolean zIsLeftChild = (z == y.left);
   boolean yIsLeftChild = (y == x.left);

  /* ================================================
Opredeli go izgledot:

   Najdi koi jazli se na pozicija a,b i c
   Spored dadenata skica:

                          b
                         /  \
                        a    c
================================================ */
   BSTEntry a, b, c;
   BSTEntry T0, T1, T2, T3;

   if (zIsLeftChild && yIsLeftChild)
   { /* Configuration 1 */
   //     System.out.println("Use tri-node restructuring op #1");

      a = z;                    //          x=c
      b = y;                    //         /  \
      c = x;                    //       y=b  T3
      T0 = z.left;              //      /  \
      T1 = z.right;             //    z=a  T2
      T2 = y.right;             //   /  \
      T3 = x.right;             //  T0  T1
   }
   else if (!zIsLeftChild && yIsLeftChild)
   { /* Configuration 2 */
   //     System.out.println("Use tri-node restructuring op #2");

      a = y;                    //       x=c
      b = z;                    //      /  \
      c = x;                    //    y=a  T3
      T0 = y.left;              //   /  \
      T1 = z.left;              //  T0 z=b
      T2 = z.right;             //     /  \
      T3 = x.right;             //    T1  T2
   }
   else if (zIsLeftChild && !yIsLeftChild)
```

```
 { /* Configuration 4 */
 //     System.out.println("Use tri-node restructuring op #4");

    a = x;                      //      x=a
    b = z;                      //     /  \
    c = y;                      //    T0  y=c
    T0 = x.left;                //       /  \
    T1 = z.left;                //      z=b  T3
    T2 = z.right;               //     /  \
    T3 = y.right;               //    T1  T2
 }
 else
 { /* Configuration 3 */
 //     System.out.println("Use tri-node restructuring op #3");

    a = x;                      //       x=a
    b = y;                      //      /  \
    c = z;                      //     T0  y=b
    T0 = x.left;                //        /  \
    T1 = y.left;                //       T1 z=c
    T2 = z.left;                //          /  \
    T3 = z.right;               //         T2  T3
 }

/* ----------------------------------------------------------------------
   Stavi go b na mestoto na x (napravi go b koren na novoto poddrvo !)
   ---------------------------------------------------------------------- */
 if ( x == root )
 {  /* Ako x  e koren, spravi se so zamenta na razlicen nacin.... */

    root = b;                    // b sega e koren
    b.parent = null;
 }
 else
 {
    BSTEntry xParent;

    xParent = x.parent;   // najdi go roditelot na x

    if ( x == xParent.left )
    { /* Povrzi go b so levata granka na roditelot na x */
       b.parent = xParent;
       xParent.left = b;
    }
    else
    { /* Povrzi go b so desnata granka na roditelot na x */
       b.parent = xParent;
       xParent.right = b;
    }
 }

/* ------------------
   Pravi:   b
          / \
         a   c
   ------------------ */
 b.left = a;
 a.parent = b;
 b.right = c;
 c.parent = b;


/* ------------------
   Pravi:   b
          / \
         a   c
        / \
       T0 T1
   ------------------ */
 a.left = T0;
```

```
   if ( T0 != null ) T0.parent = a;
   a.right = T1;
   if ( T1 != null ) T1.parent = a;

  /* ------------------
     Pravi:   b
            / \
           a   c
              / \
             T2 T3
     ------------------ */
   c.left = T2;
   if ( T2 != null ) T2.parent= c;
   c.right= T3;
   if ( T3 != null ) T3.parent= c;

   recompHeight(a);
   recompHeight(c);

   return b;
}

/* ================================================
   remove(k): izbrisi go jazelot koj go sodrzi klucot k
   ================================================ */
public void remove(String k)
{
   BSTEntry p, q;     // Help variables
   BSTEntry parent;   // parent node
   BSTEntry succ;     // successor node

    /* ------------------------------------------------------
       Najdi go jazelot so kluc key == "k" vo BST
       ------------------------------------------------------ */
   p = findEntry(k);

   if ( ! k.equals( p.key ) )
      return;            // ne e najden ==> nema nisto za brisenje....


    /* ================================================
  Hibbard  Algorithm
  ================================================ */

   if ( p.left == null && p.right == null ) // Slucaj 0: p nema deca
   {
      parent = p.parent;

   /* --------------------------------
     Izbrisi go p od roditelot na p
     -------------------------------- */
      if ( parent.left == p )
         parent.left = null;
      else
         parent.right = null;

       /* ------------------------------------------------
          Nova presmetka na visinata na site rodtelski jazli...
          ------------------------------------------------ */
      recompHeight( parent );

       /* ---------------------------------------------------------------
         Re-balansiranje na AVL pocnuvajki od tekovnata pozicija (ActionPos)
          ---------------------------------------------------------- */
      rebalance ( parent );    // Rebalansiranje na AVL po brisenje na roditelot

      return;
   }

   if ( p.left == null )                // Slucaj 1a: p ima 1 (desno) dete
```

```
{
   parent = p.parent;

/* ---------------------------------------------
  Postavi go desnoto dete na p kako dete na roditelot na p
       --------------------------------------------- */
   if ( parent.left == p )
      parent.left = p.right;
   else
      parent.right = p.right;

    /* ------------------------------------------------------
       Nova presmetka na visinata na site rodtelski jazli...
       ------------------------------------------------------ */
   recompHeight( parent );

    /* ---------------------------------------------------------------
       Re-balansiranje na AVL pocnuvajki od tekovnata pozicija (ActionPos)
       --------------------------------------------------------------- */
   rebalance ( parent );    // Rebalansiranje na AVL po brisenje na roditelot

   return;
}

if ( p.right== null )                // Slucaj 1b: p ima 1 (levo) dete
{
   parent = p.parent;

    /* ---------------------------------------------
       Link p's left child as p's parent child
       --------------------------------------------- */
   if ( parent.left == p )
      parent.left = p.left;
   else
      parent.right = p.left;

    /* ---------------------------------------------------------------
       Novo presmetuvanje na visinata na site roditelski jazli...
       --------------------------------------------------------------- */
   recompHeight( parent );

    /* ---------------------------------------------------------------
       Re-balansiranje na AVL pocnuvajki od tekovnata sostojba (ActionPos)
       --------------------------------------------------------------- */
   rebalance ( parent );  // Rebalancirranje na AVL po brisenje na roditelot

   return;
}

  /* ===============================================================
Jazelot ima 2 deca - najdi go naslednikot (successor) na p
1 cekor desno I se natamu levo do karj
zabelska: mozno e succ(p) da nema levo dete!
     =============================================================== */
 succ = p.right;            // p ima 2 deca....

 while ( succ.left != null )
    succ = succ.left;

 p.key = succ.key;        // zamena na p so naslednikot (succ)
 p.value = succ.value;


  /* --------------------------------
     Brisenje na succ od roditelot na succ
     -------------------------------- */
 parent = succ.parent;        // Prepare for deletion
 parent.left = succ.right;    // Link right tree to parent's left

  /* ----------------------------------------
```

```
        Nova presmetka na visinata na site roditelski jazli…
        ------------------------------------------- */
    recompHeight( parent );

     /* -------------------------------------------
        Re-balanciranje na AVL pocnuvajki od tekovnata pozicija (ActionPos)
        ------------------------------------------- */
    rebalance ( parent );     // Rebalansiranje na AVL po brisenje na roditelot

    return;

}


public void rebalance(BSTEntry p)
{
    BSTEntry x, y, z, q;

    while ( p != null )
    {
       if ( diffHeight(p.left, p.right) > 1 )
       {

          x = p;
          y = tallerChild( x );
          z = tallerChild( y );

          System.out.println("After tri_node_restructure: " + x + y + z);
          System.out.println();
          p = tri_node_restructure( x, y, z );

          printBST();
          System.out.println("=========================================");

       }

       p = p.parent;
    }
}

public BSTEntry tallerChild(BSTEntry p)
{
    if ( p.left == null )
       return p.right;

    if ( p.right == null )
       return p.left;

    if ( p.left.height > p.right.height )
       return p.left;
    else
       return p.right;
}


/* ================================================
    Prikaz na toa kako izgleda BST…
    ================================================ */
public void printnode(BSTEntry x, int h)
{
    for (int i = 0; i < h; i++)
       System.out.print("               ");

    System.out.print("[" + x.key + "," + x.value + "](h=" + x.height + ")");

    if ( diffHeight( x.left, x.right) > 1 )
       System.out.println("*");
    else
       System.out.println();
}
```

```
   void printBST()
   {
      showR( root, 0 );
      System.out.println("================================");
   }

   public void showR(BSTEntry t, int h)
   {
      if (t == null)
         return;

      showR(t.right, h+1);
      printnode(t, h);
      showR(t.left, h+1);
   }


   /* ============================================================
      maxHeight(t1,t2): presmetuva maksimalna visnina na dve poddrva
      ============================================================ */
   public static int maxHeight( BSTEntry t1, BSTEntry t2 )
   {
      int h1, h2;

      if ( t1 == null )
         h1 = 0;
      else
         h1 = t1.height;

      if ( t2 == null )
         h2 = 0;
      else
         h2 = t2.height;

      return (h1 >= h2) ? h1 : h2 ;
   }

   /* ============================================================================
      diffHeight(t1,t2): presmetuvanje na razlika vo visinata pomegu 2 poddrva
============================================================================ */
   public static int diffHeight( BSTEntry t1, BSTEntry t2 )
   {
      int h1, h2;

      if ( t1 == null )
         h1 = 0;
      else
         h1 = t1.height;

      if ( t2 == null )
         h2 = 0;
      else
         h2 = t2.height;

      return ((h1 >= h2) ? (h1-h2) : (h2-h1)) ;
   }

   /* ============================================================
      recompHeight(x): nova presmetka za visinata (height) pocnuvajki od x
      ============================================================ */
   public static void recompHeight( BSTEntry x )
   {
      while ( x != null )
      {
         x.height = maxHeight( x.left, x.right ) + 1;
         x = x.parent;
      }
   }
}
```

Додаток 3

```
//Avl.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define INSERT 1
#define DELETE 2
#define MAX 100

struct avl_nodeData {
    void *data;
    void *key;
    int rc;
};
typedef struct avl_nodeData avlNodeData;

struct avl_node {
    avlNodeData *data;
    int height;/*height of the subtree rooted at thsi node*/
    struct avl_node *lChild;
    struct avl_node *rChild;
    struct avl_node *parent;
};
typedef struct avl_node avlNode;

/*available methods*/
avlNode* newAvlTree (avlNodeData *val);
avlNode* insert (avlNode **root, avlNodeData *val, int (*cmp)(void *key1, void *key2));
avlNode* locate (avlNode *root, avlNodeData *val, int (*cmp)(void *key1, void *key2));
void removeNode (avlNode **node, void (*freeNodedata)(avlNodeData **data));
avlNode* deleteNode (avlNode **root, avlNodeData *val, int (*cmp)(void *key1, void *key2),
void (*freeNodedata)(avlNodeData **data));

void inorder (avlNode *root, void (*display)(avlNodeData *nodeData));
void postorder (avlNode *root, void (*display)(avlNodeData *nodeData));
void preorder (avlNode *root, void (*display)(avlNodeData *nodeData));
avlNode* allocAvlNode (avlNodeData *val)
{
    avlNode *node;

    node = (avlNode*)malloc(sizeof(avlNode));
    node->data = val;
    node->data->rc = 1;
    node->height = 0;
    node->lChild = NULL;
    node->rChild = NULL;
    node->parent = NULL;
    return node;
}

avlNode* newAvlTree (avlNodeData *val)
{
    avlNode *root = NULL;

    root = allocAvlNode(val);
    return root;
}

void freeNodeData (avlNodeData **data);
void freeAvlNode (avlNode **node, void (*freeNodeData)(avlNodeData **data))
{
    if ((*node)->data->rc <= 1)
        freeNodeData(&((*node)->data));
    else
        --((*node)->data->rc);
    free(*node);
    *node = NULL;
```

```
    return;
}

int getHeight (avlNode *node)
{
    int h_lChild = (node->lChild) ?  node->lChild->height + 1: 0;
    int h_rChild = (node->rChild) ?  node->rChild->height + 1: 0;

    return ((h_lChild >= h_rChild) ? h_lChild : h_rChild);
}

void leftRotate (avlNode **node)
{
    avlNode *father = (*node)->parent;
    avlNode *rChild = (*node)->rChild;
    avlNode *grandSon = rChild->lChild;
    avlNode *tmp = *node;

    if (father) {
        if (father->lChild == *node)
            father->lChild = rChild;
        else
            father->rChild = rChild;
        rChild->parent = father;/**node gets changed at this point, because of the
recursive nature of avlTree_fixup*/
    } else
        rChild->parent = NULL;
    rChild->lChild = tmp;
    tmp->rChild = grandSon;
    if (grandSon != NULL)
        grandSon->parent = tmp;
    tmp->parent = rChild;
    /*update heights*/
    tmp->height = getHeight(tmp);
    rChild->height = getHeight(rChild);

    return;
}

void rightRotate (avlNode **node)
{
    avlNode *father = (*node)->parent;
    avlNode *lChild = (*node)->lChild;
    avlNode *grandSon = lChild->rChild;
    avlNode *tmp = *node;

    if (father) {
        if (father->lChild == *node)
            father->lChild = lChild;
        else
            father->rChild = lChild;
        lChild->parent = father;
    } else
        lChild->parent = NULL;
    lChild->rChild = tmp;
    tmp->lChild = grandSon;
    if (grandSon != NULL)
        grandSon->parent = *node;
    tmp->parent = lChild;
    /*update heights*/
    tmp->height = getHeight(tmp);
    lChild->height = getHeight(lChild);

    return;
}

inline int balanceFactor (avlNode *node)
{
    int h_lChild = (node->lChild) ?  node->lChild->height + 1: 0;
    int h_rChild = (node->rChild) ?  node->rChild->height + 1: 0;
```

```
    return h_rChild - h_lChild;
}

avlNode* getRoot(avlNode *node)
{
    if (node->parent)
        return getRoot(node->parent);
    else
        return node;
}

void avlTree_fixup (avlNode **node, int action)
{
    int bf = 0;
    int ht = -1;
    avlNode *father = NULL;

    if (*node == NULL)
        return;
    ht = getHeight(*node);
    (*node)->height = ht;
    bf = balanceFactor(*node);
    if (bf == 0 && action == 1) /*for the case of insertion - no change in height*/
        return;
    if (bf == 0 && action == 2) {/*for the case of deletion*/
        avlTree_fixup(&((*node)->parent), action);
        return;
    }
    if (bf == 1 || bf == -1) {/*=> that one node has been added to the right/left subtree,
can disturb the balancing at the parent*/
        if (action == 1) {
            avlTree_fixup(&((*node)->parent), action);
            return;
        }
        else /*no change in height*/
            return;
    }
    if (bf == 2) {
        int bf_rChild = balanceFactor((*node)->rChild);
        if (bf_rChild == 1) {
            avlNode *tmp = (*node)->rChild;
            leftRotate(node);
            /*node is now by itself set to parent of node before
             * rotation*/
            if (action == 1)
                avlTree_fixup(node, action);
            else
                avlTree_fixup(&((*node)->parent), action);
            return;
        } else if (bf_rChild == -1) {
            rightRotate(&((*node)->rChild));
            (*node)->height = getHeight(*node);
            if (action == 1)
                leftRotate(&((*node)->parent));
            else
                leftRotate(node);
            avlTree_fixup(&((*node)->parent), action);
            return;
        } else if (bf_rChild == 0) {/*for the case of deletion*/
            leftRotate(node);
            avlTree_fixup(&((*node)->parent), action);
            return;
        }
    } else if (bf == -2) {
        int bf_lChild = balanceFactor((*node)->lChild);
        if (bf_lChild == -1) {
            rightRotate(node);
            /*node is now by itself set to parent of node before
             * rotation*/
```

```
            if (action == 1)
                avlTree_fixup(node, action);
            else
                avlTree_fixup(&((*node)->parent), action);
            return;
        } else if (bf_lChild == 1) {
            leftRotate(&((*node)->lChild));
            (*node)->height = getHeight(*node);
            if (action == 1)
                rightRotate(&((*node)->parent));
            else
                rightRotate(node);
            avlTree_fixup(&((*node)->parent), action);
            return;
        } else if (bf_lChild == 0) {/*for the case of deletion*/
            rightRotate(node);
            avlTree_fixup(&((*node)->parent), action);
            return;
        }
    }
}

/*cmp is the function pointer provided by the user
 * return codes of cmp expected by the insert function
 * key1 = key2 return 0
 * key1 < key2 return 1
 * key1 > key2 return 2
 * */
void add (avlNode **root, avlNodeData *val, int (*cmp)(void *key1, void *key2))
{
    avlNode *node = NULL;
    avlNode *tmp = NULL;
    int dir = 0;

    if (!(*root)) {
        *root = newAvlTree(val);
        (*root)->parent = NULL;
        //return *root;
        return;
    }
    if (cmp(val->key, (*root)->data->key) <= 1) {
        tmp = (*root)->lChild;
        dir = 1;
    } else {
        tmp = (*root)->rChild;
        dir = 2;
    }
    if (tmp == NULL) {
        tmp = allocAvlNode(val);
        tmp->parent = *root;
        if (dir == 1) {
            (*root)->lChild = tmp;
            avlTree_fixup(root, INSERT);
        } else {
            (*root)->rChild = tmp;
            avlTree_fixup(root, INSERT);
        }
    } else
        add(&tmp, val, cmp);

    //return ((*root)->parent) ? (*root)->parent : *root;
    return;
}

avlNode* insert (avlNode **root, avlNodeData *val, int (*cmp)(void *key1, void *key2))
{
    add(root, val, cmp);
    return getRoot(*root);
}
```

```
avlNode* search (avlNode *root, avlNodeData *val, int (*cmp)(void *key1, void *key2))
{
    if (root == NULL)
        return NULL;
    if (cmp(val->key, root->data->key) == 0)
        return root;
    else if (cmp(val->key, root->data->key) == 1)
        return search(root->lChild, val, cmp);
    else
        return search(root->rChild, val, cmp);
}

void replace (avlNode **src, avlNode **dest)
{
    avlNode *father = (*src)->parent;

    if (*dest != NULL)
        (*dest)->parent = (*src)->parent;
    if (father->lChild == *src)
        father->lChild = *dest;
    else
        father->rChild = *dest;

    return;
}

void removeNode (avlNode **node, void (*freeNodedata)(avlNodeData **data))
{
    avlNode *child = NULL;
    avlNode *father = NULL;

    father = (*node)->parent;
    if ((*node)->lChild != NULL)
        child = (*node)->lChild;
    else if ((*node)->rChild != NULL)
        child = (*node)->rChild;
    else
        child = NULL;

    replace(node, &child);
    avlTree_fixup(&((*node)->parent), DELETE);
    freeAvlNode(node, freeNodedata);

    return;
}

avlNode* largestInSubtree (avlNode *root)
{
    if (!(root->rChild))
        return root;
    return largestInSubtree(root->rChild);
}

avlNode* smallestInSubtree (avlNode *root)
{
    if (!(root->lChild))
        return root;
    return smallestInSubtree(root->lChild);
}

avlNode* deleteNode (avlNode **root, avlNodeData *val, int (*cmp)(void *key1, void *key2),
void (*freeNodedata)(avlNodeData **data))
{
    avlNode *target = NULL;
    avlNode *toRemove = NULL;

    target = search(*root, val, cmp);
    if (!target)
        return;
    if (target->lChild != NULL)
```

```
        toRemove = largestInSubtree(target->lChild);
    else if (target->rChild != NULL)
        toRemove = smallestInSubtree(target->rChild);
    else
        toRemove = target;
    if (target != toRemove) {
        ++(toRemove->data->rc);
        target->data = toRemove->data;
    }
    removeNode(&toRemove, freeNodeData);

    return getRoot(*root);;
}


avlNode* locate (avlNode *root, avlNodeData *val, int (*cmp)(void *key1, void *key2))
{
    return search(root, val, cmp);
}

void inorder (avlNode *root, void (*display)(avlNodeData *nodeData))
{
    if (root == NULL)
        return;

    inorder(root->lChild, display);
    display(root->data);
    inorder(root->rChild, display);
    return;
}

void postorder (avlNode *root, void (*display)(avlNodeData *nodeData))
{
    if (root == NULL)
        return;

    postorder(root->lChild, display);
    postorder(root->rChild, display);
    display(root->data);
    return;
}

void preorder (avlNode *root, void (*display)(avlNodeData *nodeData))
{
    if (root == NULL)
        return;

    display(root->data);
    preorder(root->lChild, display);
    preorder(root->rChild, display);
    return;
}
int intCompare (void *val1, void *val2)
{
    if (*((int *)val1) == *((int *)val2))
        return 0;
    if (*((int *)val1) < *((int *)val2))
        return 1;
    else
        return 2;
}

void intDisplay (avlNodeData *data)
{
    printf("%d ", *((int *)(data->key)));
    return;
}

void freeNodeData (avlNodeData **data)
{
```

```
    /*free((int *)((*data)->data));
      free((int *)((*data)->key));*/
    free(*data);
    *data = NULL;

    return;
}

int main (int argc, char **argv)
{
    /*sample use of avlTree
     * avlNodeData will be just integers*/
    avlNode *root = NULL;
    int i = 5;
    int val[] =  {5, 10, 2, 12, 11, 15, 4, 3};
    int len = 8;
    avlNodeData *data[MAX];

    for (i = 0; i < len; i++) {
        data[i] = (avlNodeData*)malloc(sizeof(avlNodeData));
        data[i]->data = (void *)(&val[i]);
        data[i]->key = (void *)(&val[i]);
        root = insert(&root, data[i], intCompare);
        printf("\n********** ITERATION %d **********\n", i);
        printf("Inorder:\t");
        inorder(root, intDisplay);
        printf("\n");
        printf("Postorder:\t");
        postorder(root, intDisplay);
        printf("\n");
        printf("***********************************\n");
    }
    printf("\n\nDELETION\n\n");
    root = deleteNode(&root, data[4], intCompare, freeNodeData);
    printf("Inorder:\t");
    inorder(root, intDisplay);
    printf("\n");
    printf("Postorder:\t");
    postorder(root, intDisplay);
    printf("\n");

    printf("\n\nDELETION\n\n");
    root = deleteNode(&root, data[5], intCompare, freeNodeData);
    printf("Inorder:\t");
    inorder(root, intDisplay);
    printf("\n");
    printf("Postorder:\t");
    postorder(root, intDisplay);
    printf("\n");

    printf("\n\nDELETION\n\n");
    root = deleteNode(&root, data[6], intCompare, freeNodeData);
    printf("Inorder:\t");
    inorder(root, intDisplay);
    printf("\n");
    printf("Postorder:\t");
    postorder(root, intDisplay);
    printf("\n");

    printf("\n\nDELETION\n\n");
    root = deleteNode(&root, data[3], intCompare, freeNodeData);
    printf("Inorder:\t");
    inorder(root, intDisplay);
    printf("\n");
    printf("Postorder:\t");
    postorder(root, intDisplay);
    printf("\n");
    //system("pause");

    return 0;}
```

Користена литература

[1] A. Kerren and John T. Stasko „ Algorithm Animation “,Springer 2001.

[2] A. Zeller, “DDD-A Free Graphical Front-End for UNIX Debuggers” ACM SIGPLAN Notices, Pages 22-27, Volume 31, January 1996

[3] A. Gonzalez, R. Theron, A. Telea, Francisco J. Garcia „ Combined Visualization of Structural and Metric Information for Software Evolution Analysis “ACM New York, NY, USA, 2009.

[4] A. Marcus, L. Feng, J. I. Maletic, „ 3D Representations for Software Visualization “ACM New York, NY, USA, 2002.

[5] A. Moreno, N. Myller, E. Sutinen “Visualizing Programs with Jeliot 3”, AVI’04 Proceedings of the working conference on Advanced visual interfaces, Pages 373-376, ACM New York, NY, USA ,2004

[6] A. Pérez-Carrasco, J. Á. Velázquez-Iturbide, J. Urquiza-Fuentes, “SRec: An animation system of recursion for algorithm courses”, 13rd Annual Conference on Innovation and Technology in Computer Science Education, ACM Press, 2008

[7] A. Turing. "On computable numbers, with an application to the Entscheidungsproblem".Proceedings of the London Mathematical Soci-ety, 2(42):230–265, 1936

[8] A. W. Lawrence, Albert N. Badre, John T.Stasko “Empirically Evaluating the Use of Animations to Teach Algorithms” –Technical Report GIT –GVU-94-07

[9] A. Zeller “Visual Debugging with DDD” Dr. Dobb's Journal, Pages 21-28, Volume 322, March 2001

[10] A. Zeller “Animating data structures in DDD” Proc. SIGCSE/SIGCUE Program Visualization Workshop, Finland, July 2000

[11] B. A. Myers. Taxonomies of visual programming and program vi-sualisation. Journal of Visual Languages and Computing, 1(1):97–123, 1990

[12] B. A. Price , R.M. Baecker and I. S. Small ``A Principled Taxonomy of Sofware Visualization" Journal of Visual Languages and Computing Vol.4, 1993

[13] C. Chen, „Information Visualization“,Second Edition, Springer, pp.1-26, 2006.

[14] C. Chen, W. Hardle, A. Unwin „Handbook of Data Visualization“, Springer, 2008.

[15] C. Collberg, S. Kobourov, J. Nagra, J. Pitts, K. Wampler „ A System for Graph-Based Visualization of the Evolution of Software “, ACM New York, NY, USA, 1998

[16] C. Demetrescu, Irene Finocchi2, and John T. Stasko „ Specifying Algorithm Visualizations “,Springer 2001.

[17] C. Gutwenger, M. JËunger, G. W. Klau3, Sebastian Leipert, „ Graph Drawing Algorithm Engineering with AGD “, Springer 2002.

[18] C. Halverson, Jason B. Ellis, C. Danis, Wendy A. Kellogg „ Designing Task Visualizations to Support the Coordination of Work in Software Development “, ACM New York, NY, USA, 2006.

[19] C. M. Kehoe, J. T. Stasko “Using Animations to learn about Algorithms: An Ethnographic Case Study”- Technical Report GIT –GVU-96-20, September 1996.

[20] C. M. Kehoe, J. T. Stasko Ashley Taylor “Rethinking the Evaluation of Algoritham Animations as Learning Aids: An Observation Study”- Technical Report GIT –GVU-99-10, Mart 1999.

[21] C. Probst. "A Demand-Driven Solver for Constraint-Based Con-trol Flow Analysis". PhD thesis, University of Saarland, Saarbr¨ ucken, Germany, 2002.

[22] C. Ravindranath Pandian, „Software Metrics- A Guide to Planning, Analysis, and Application “,Auerbach publications, pp. 39-56, 2004.

[23] E. J. Chikofsky and J. H. Cross II. Reverse engineering and design recovery: A taxonomy.IEEE Software, 7(1):13–18, 1990

[24] F. Nielson, H. R. Nielson, and C. Hankin.,"Principles of Program Analysis". Springer, Berlin, Heidelberg, New York, 1999

[25] G. Cattaneo, G. F. Italiano, U. Ferraro-Petrillo, “Catai: Concurrent Algorithms and Data Types Animation over the Internet”, Journal of Visual Languages and Computing. 2002

[26] G. Di Battista, P. Eades, R. Tamassia, I. G.Tollis, „Graph Drawing- Algorithms for the Visualization of Graphs“, 1999.

[27] H. Agrawal, J. R. Horgan, S. London and W. E. Wong, ``Fault Localization using Execution Slices and Dataflow Tests," in Proceedings of the 6th IEEE International Symposium on Software Reliability Engineering, pp 143-151, Toulouse, France, October 1995.

[28] H. Biermann , R. Cole „Comic Strips for Algorithm Visualization“, February 17, 1999.

[29] H.A. Muller, M. A. Orgun, Scott R. Tilley, and James S. Uhl.A reverse engineering approach to subsystem structure identification. Journal of Software Maintenance: Research and Practice, 5(4):181–204, December 1993

[30] I. Nassi and B. Shneiderman "Flowchart techniques for structured programming". SIGPLAN Notices, 8(8):12–26, August 1973.

[31] J. Á. Velázquez-Iturbide, A. Pérez-Carrasco, J. Urquiza-Fuentes, “Interactive visualization of recursion with SRec”, 14th Annual Conference on Innovation and Technology in Computer Science Education, ACM Press, 2009

[32] J. C. Grundy and J. G. Hosking. SoftArch: Tool support for integrated software architecture development. International Journal on Software Engineering and Knowledge Engineering, 13(2):125–151,2003

[33] J. Domingue, „ Software Visualization and Education “, Springer 2002.

[34] J. Francik, „ Algorithm Animation Using Data Flow Tracing “, Springer 2002.

[35] J. K. Gilbert, Miriam Reiner, Mary Nakhleh, „ Visualization: Theory and Practice in Science Education “ Vol.3 pp.1-5, 2008.

[36] J. Noble „Visualising Objects: Abstraction, Encapsulation, Aliasing, and Ownership“, Springer 2002.

[37] J. T. Stasko “Using Student-Built Algorithm Animations as Learning Aids” Technical Report GIT –GVU-96-19, Atlanta, ACM Technical Symposium on Computer Science Education (SIGCSE '97), San Jose, CA, February 1997

[38] J. T. Stasko, Carlon Reid Turner “Tidy Animations of Tree Algorithms”- Technical Report GIT –GVU-92-11, Atlanta, GA, Technical Report GIT-GVU-92-11, June 1992.

[39] J. T. Stasko. TANGO: A framework and system for algorithm ani-mation.Computer, 23(9):27–39, 1990.

[40] T. Stasko. The path-transition paradigm: A practical methodology for adding animation to program interfaces. Journal of Visual Languages and Computing, 1(3):213–236, 1990

[41] J. X. Chen „Guide to Graphics Software Tools“, Springer, 2008.

[42] J.E. Baker, I.F.Cruz, G.Liotta, and R. Tamassia “A New Model for Algorithm Animation Over the WWW” ACM Computing Surveys. Vol. 27 No.4, December ,1995

[43] J.H.Cross “Using the new jGrasp canvas of dynamic viewers for program understanding and debugging in Java courses” Journal of Computing Sciences in Colleges, ACM, Volume 29 Issue 1, October 2013

[44] J.H.Cross, T.D. Hendrix “jGRASP: an integrated development environment with visualizations for teaching Java in CS1, CS2, and beyond”, ACM, Journal of Computing Sciences in Colleges, Volume 23 Issue 3, January 2008.

[45] J.T.Stasko, C Patterson, Understanding and characterising software visualization systems, Proceedings of IEEE Workshop on Visual Languages 1992.

[46] K. Knowlton. Bell telephone laboratories low-level linked list language. 16-minute black and white film, 1966

[47] K. Koskimies, T. Syst¨ a, and Jyrki Tuomi. Automated support for modeling OO software.IEEE Software, 15(1):87–94, 1998.

[48] L.Voinea, A. Telea „ Multiscale and Multivariate Visualizations of Software Evolution “, ACM New York, NY, USA, 2006

[49] M. Balzer and O. Deussen, „Hierarchy based 3D Visualization of Large Software Structures“, University of Konstanz, Germany, 2004.

[50] M. Brown and R. Sedgewick. A system for algorithm animation. InProceedings of ACM SIGGRAPH’84, Minneapolis, MN, pages 177–186, New York, NY, ACM Press, 1984.

[51] M. Brown, J. Domingue, B. Price, J. Stasko „ Software Visualization “ MIT Press, Cambridge MA,1998.

[52] M. Dahm, "Byte Code Engineering with the BCEL API" Technical Report B-17-98, April 2001

[53] M. Lanza, „ CodeCrawler — Lessons Learned in Building a Software Visualization Tool “,Software Composition Group - University of Berne, Switzerland, 2002.

[54] M. Petre and Ed de Quincey „ A gentle overview of software visualisation “,PPIG Newsletter – September 2006.

[55] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape-analysis via 3-valued logic. InProceedings of the 26th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, San Antonio, TX, pages 105–118, New York, NY, 1999. ACM Pres

[56] M. H. Brown and M.Najork. Algorithm animation using 3D inter-active graphics. InProceedings of ACM Symposium on User InterfaceSoftware and Technology, Atlanta, pages 93–100, New York, NY, 1993. ACM Pres

[57] M. H. Brown, Zeus: A System for Algorithm Animation and Multi-view Editing, SRC Research Report 75, February 1992

[58] M. Jackson. Principles of Program Design. Academic Press, 1975.

[59] N. Myller, R.Bednarik and A. Moreno "Integrating Dynamic Program Visualization into BlueJ: the Jeliot 3 Extension", Seventh IEEE International Conference on Advanced Learning Technologies, ICALT 2007.

[60] N. Ourusoff, "Introduction to Jackson Design Method: JSP and a little JSD", Proceedings of 2003 InformingScience and Information Technology Education, Pori, Finland, June 2003. pp 5-15.

[61] P. Mutzel and P. Eades, P. Mutzel and P. Eades, Graphs in Software Visualization, Springer 2002.

[62] P. Crescenzi C. Demetreseu, I.Finocchi, R.Petreschi "LEONARDO: a software visualization system" Rome, Italy, 2000.

[63] R. Baecker. Sorting Out Sorting. 30 minute color film (developed with assistance of Dave Sherman, distributed by Morgan Kaufmann,University of Toronto), 1981

[64] R. Fleischer and L. Kuˇcera, „ Algorithm Animation for Teaching ", Springer 2002.

[65] R. Koschke, „ Software Visualization for Reverse Engineering ", Springer 2002.

[66] R. Wettel and M. Lanza "Visualizing Software Systems as Cities",Faculty of Informatics - Visualizing Software for Understanding and Analysis, VISSOFT 2007. 4th IEEE International, Workshop on June 2007.

[67] R. Wilhelm and D. Maurer. "Compiler Design". Addison Wesley Longman, Redwood City, CA, 1995.

[68] R. Wilhelm, T. Müldner, and R. Seidel, „ Algorithm Explanation: Visualizing Abstract States and Invariants ", Springer 2002.

[69] R. Bednarik, A.Moreno and N.Myller "Jeliot 3, an Extensible Tool for Program Visualization", 5th Annual Finnish / Baltic Sea Conference on Computer Science Education. November 17 - November 20, 2005.

[70] R. Laddad. "Aspectj in Action: Practical Aspect-Oriented Programming" Second Edition,Chapter 1, Manning Publications Co. 2010

[71] S. Diehl, „Future Perspectives", Springer 2002.

[72] S. Diehl, „Software Visualization – Visualizing the Structure, Behaviour and Evolution of Software ", University Trier, Germany, Springer, Mart 2007.

[73] S. Diehl, Carsten GËorg, and Andreas Kerren, „ Animating Algorithms Live and Post Mortem " , Springer, 2001.

[74] S. Ellershaw and M. Oudshoorn „ Program Visualization – The State of the Art“,Department of Computer Science, University of Adelaide ,1999.

[75] S. G.Eick, Todd L. Graves, Alan F. Karr, Audris MOckus, and Paul Schuster „Visualizing Software Changes“, IEEE Transaction on software engineering, Vol.28, No.4, April 2002.

[76] S. P. Reiss, Visualizing Java in action, in: Proceedings of the IEEE Conference on Software Visualization, pp. 123–132" 2002

[77] S. Khuri “Designing Effective Algorithm Visualizations” , San Jose USA, 2001

[78] T. Genssler, V. Kuttruff, and F. Brosig. "Inject/J Software Transformation Language" - Language Specification, November 2007

[79] T. Systa, K. Koskimies, and H. M uller, “Shimba – an environ-ment for reverse engineering Java software systems”.Software – Practice and Experience, 31(4):371–394, 2001.

[80] T. Zimmermann and A. Zeller „ Visualizing Memory Graphs “, Springer 2002.

[81] T. Khan, H. Barthel,A.Ebert, and P.Liggesmeyer, "Visualization and Evolution of Software Architectures"-Visualization of Large and Unstructured Data Sets Workshop 2011

[82] V. Averbukh, A. Konovalov, V. Vorzopov „ An Approach to Evaluation of Software Visualization “ , ACM New York, NY, USA, 1997.

[83] W. de Pauw and G. Sevitsky. Visualizing reference patterns for solving memory leaks in Java. InProceedings of European Symposium on Object-Oriented Programming ECOOP’99, pages 116–134. Springer, Berlin, Heidelberg, New York, 1999

[84] W.De Pauw, E. Jensen, N. Mitchell, G Sevitsky, J. Vlis-sides, and J. Yang. Visualizing the execution of Java programs. In Proceedings of Dagstuhl Seminar on Software Visualization [Die02b], pages 151–162. 2002.

[85] http://cs.joensuu.fi/jeliot/

[86] http://stellar.cleanscape.net/products/testwise/tools_xslice.html -Xslice

[87] http://users.dcc.uchile.cl/~rbaeza/cursos/visual/aa/quicksort.html

[88] http://www.cc.gatech.edu/gvu/ii/softvis/3dcv/3dcv.html

[89] http://www.dis.uniroma1.it/~demetres/Leonardo/

[90] http://www.di.unisa.it/cattaneo.dir/CATAI/node10.html

[91] http://www.gnu.org/software/ddd/

[92] http://www.jgrasp.org/

[93] http://www.lite.etsii.urjc.es/srec/