UNIVERZITET U BEOGRADU

FAKULTET ORGANIZACIONIH NAUKA

ZAVRŠNI RAD

KREIRANJE VIZUELNIH ALATA NA NETBEANS PLATFORMI

Mentor:

Prof dr. Vladan Devedžić

Student:

Damir Kocić 169/06

Beograd, 2010 godine

Sadržaj

1. Uvod 3
2. Pregled relevantnih oblasti 4
2.1 NetBeans platforma 4
2.2 Nodes API 6
2.3 Visual Library API 9
2.4 Common Palette 11
3. Zahtevi i analiza 12
3.1 Scena 12
3.2 Widget-i 13
3.3 Paleta 13
4. Projektovanje 14
4.1 GraphScene 14
4.2 NeuralNetworkWidget 15
4.3 Palette 17
5. Implementacija 19
5.1 Implementacija scene 19
5.2 Implementacija widget-a 24
2.2.1. Implementacija provajdera akcija 29
5.3 Implementacija palete 34
6. Evaluacija 43
7. Zaključak 45
Literatura 46

1. Uvod

Osnovni cilj ovog rada je razvoj vizuelnog alata za rad sa neuronskim mrežama, koji će u isto vreme biti intuitivan i u potpunosti ispuniti potrebe korisnika koji se tek upoznaju sa radom sa neuronskim mrežama.

Kako se širi oblast primene neuronskih mreža, tako i interesovanje za neuronske mreže raste. Postoji nekoliko alata za rad sa neuronskim mrežama, Encog, Joone, JNNS. Većina ovih alata ima korisnički interfejs koji nije intuitivan, i korisnicima je teško da se naviknu na rad sa tim alatima. Kako bi se prevazišao ovaj problem, razvijen je korisnički interfejs za aplikaciju zasnovanu na „Neuroph“ *framework-u*, koji je nalik na interfejse aplikacija koje se danas široko koriste. Krajnji cilj je razvoj aplikacije koja će omogućiti kreiranje i rad sa neuronskim mrežama.

Ovaj alat razvijen je kao modul aplikacije koja je zasnovana na NetBeans platformi. NetBeans platforma predstavlja okruženje zasnovano na *framework-u* koji omogućava kreiranje modularne strukture.

Razvijeno rešenje ima veliku ulogu u upoznavanju zainteresovanih korisnika, pre svega studenata koji se bave inteligentnim sistemima, sa neuronskim mrežama. Ovaj alat omogućava korisnicima da na jednostavan način kreiraju neuronske mreže, i izvršavaju određene akcije nad njima, i samim tim se upoznaju sa njihovom strukturom i načinom funkcionisanja.

U drugom poglavlju dat je pregled oblasti od ključnog značaja za razvoj vizuelnih alata na NetBeans platformi.
U trećem poglavlju definisani su zahtevi i izvršena je analiza korisničkih zahteva.
Četvrto poglavlje obuhvata projektovanje strukture alata za vizuelizaciju, koja je objašnjena na primerima dijagrama klasa.
U petom poglavlju opisana je implementacija strukture koja je opisana u četvrtom poglavlju, na primerima koda.
Šesto poglavlje obuhvata evaluaciju razvijenog alata, i poređenje sa sličnim alatima.
U poslednjem poglavlju sagledan je rezultat i mogućnosti daljeg razvoja ovog alata.

2. Pregled relevantnih oblasti

U ovom poglavlju dat je kratak opis NetBeans platforme i pregled ključnih API-a koji omogućavaju kreiranje vizuelnih alata u aplikacijama zasnovanim na NetBeans platformi.

2.1 NetBeans platforma

Veličina i složenost desktop aplikacija koje danas koristimo tokom vremena postajala je sve veća. U isto vreme bilo je potrebno obezbediti fleksibilnost aplikacija, kako bi se mogle lako nadograđivati, pa je bilo poželjno podeliti aplikaciju na više nezavisnih blokova, koji bi na kraju činili jednu modularnu arhitekturu, koja bi se mogla lako nadograđivati. Svi blokovi moraju biti nezavisni, a funkcije koje su definisane u javnim interfejsima moraju biti dostupne ostalim blokovima.

NetBeans Platforma predstavlja framework koji omogućava kreiranje složenih desktop aplikacija. Ova platforma obezbeđuje napredni korisnički interfejs, koji omogućava programerima da se fokusiraju na razvoj same logike aplikacije, bez potrebe da sami kreiraju pojedinačne elemente korisničkog interfejsa, što značajno ubrzava razvoj jedne ovakve aplikacije. NetBeans platforma, kao i sve aplikacije razvijene pomoću nje, je podeljena na module. Modul predstavlja kolekciju funkcionalno povezanih klasa. Pored klasa, modul sadrži i interfejs pomoću kojieg drugi moduli komuniciraju sa njim bez direktne zavisnosti. Moduli su opisani pomoću manifest fajlova i podataka u XML fajlovima.

Modul predstavlja jednostavnu Java arhivu, koja sadrži:

- manifest.mf fajl koji sadrži opis modula i veza sa drugim modulima,
- layer.xml fajl, koji opisuje šta je sve kreirano u modulu,
- funkcionalno povezane klase,
- resurse (ikone, fajlovi sa svojstvima...).

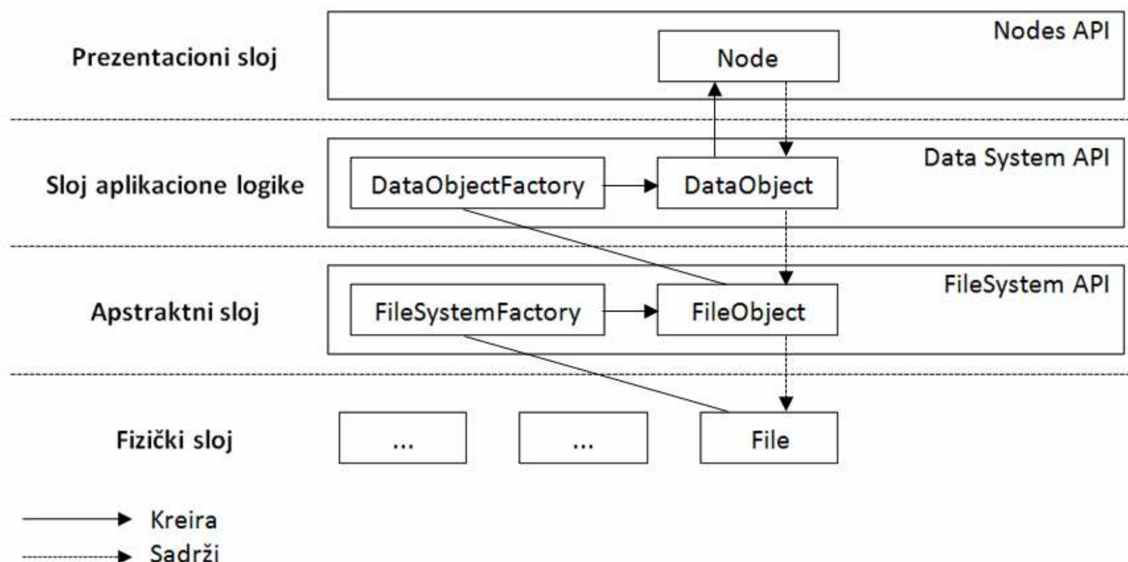
Osnovne karakteristike NetBeans platforme:

- **Sistem modula** – modularnost omogućava organizovanje koda u potpuno odvojenim modulima, pri čemu svaki modul može koristiti samo klase iz onih modula prema kojima je definisana zavisnost *(dependency)*. Ovakva organizacija aplikacije je ključna za velike aplikacije koje su razvijane u distribuiranom okruženju, kako kod razvoja, tako i kod održavanja.

- **Upravljanje životnim ciklusom aplikacije** – kao što aplikacioni serveri pružaju servise web aplikacijama i povezuju web module, EJB module... tako i *NetBeans runtime container* ima zadatak da poveže NetBeans module u jednu *Swing* aplikaciju.
- **Podrška za plugin-ove** – NetBeans platforma omogućava korisnicima da instaliraju dodatke (*plugin-ove*) u svoje aplikacije preko *Update* centra, jer se svi plugin-ovi mogu instalirati, deinstalirati, aktivirati i deaktivirati prilikom izvršavanje same aplikacije.
- **Servisna infrastruktura** – NetBeans platforma obezbeđuje infrastrukturu za registrovanje i pozivanje implementacije servisa, i samim tim smanjuje potrebu za direktnom zavisnošću između modula. Ovim se omogućava visoka povezanost aplikacije bez direktnih veza (zavisnosti) između modula.
- **Sistem prozora** – Složenije desktop aplikacije zahtevaju više od jednog radnog prozora. Gotov sistem prozora NetBeans platforme omogućava minimizaciju, maksimizaciju i premeštanje prozora u samoj aplikaciji.
- **Standardizovani UI alat** – *Swing* biblioteka komponenti korisničkog interfejsa predstavlja osnovni UI alat i osnovu svake Netbeans aplikacije. Prednost je lako menjanje izgleda same aplikacije, dodavanje internacionalizacije i Java 2D efekata aplikaciji.
- **Jedinstveni prezentacioni sloj** – NetBeans *Nodes API* obezbeđuje jedinstveni model za prikazivanje podataka, NetBeans *Explorer API* i *Property Sheet API* pružaju nekoliko naprednih swing komponenti za prikazivanje podataka u vidu stabla. Neki od tih komponenti su *Property Sheet*, Paleta, Menadzer plugin-a i *Output window*.
- **Integracija Java Help-a –** *JavaHelp API* je sastavni deo NetBeans platforme. Korisnici mogu da pišu help dokumenta za svaki modul posebno, a ovaj API će ih automatski povezati u jedinstveni set Help dokumenata.

API (*Application Programming Interface*) predstavlja interfejs preko kojeg neka aplikacija komunicira sa skupom metoda koje su definisane u samom API-ju. Reč interfejs pokazuje da API živi između najmanje dva različita subjekta, npr. na jednoj strani je unutrašnja struktura aplikacije, a na drugoj strana aplikacija koja je poziva, ili programer koji razvija jednu aplikaciju i njen API, koji sa druge strane koriste programeri u svojim aplikacijama. Programiranje API-a omogućava različitim timovima, koji se ne poznaju i međusobno ne sarađuju, da razvijaju nezavisne projekte koji mogu međusobno da komuniciraju.

NetBeans platforma omogućava lako kreiranje, upravljanje, manipulisanje i prikazivanje podataka, korišćenjem *Nodes , Data System* i *FileSystem API-a*.



Slika 1 – Arhitektura prikaza podataka u NetBeans platform

Na slici 1 je prikazana slojevita arhitektura koju NetBeans platforma koristi za prikaz podataka. Sa slike se vidi da je ključni API za prikazivanje podataka na korisničkom interfejsu *Nodes API*.

2.2 Nodes API

Nodes API kontroliše upotrebu i kreiranje nodova, omogućava akcije i *cookies* i kontroliše predstavljanje podataka u *Explorer* prozoru. Nodovi su neposredno odgovorni za vizuelnu reprezentaciju i odgovarajuće ponašanje većine objekata u NetBeans-u. Nodovi ne bi trebalo da se koriste za čuvanje podataka, već samo za njihovo prezentovanje. Podaci se obično čuvaju u data objektima ili u drugim skladištenim mehanizmima.

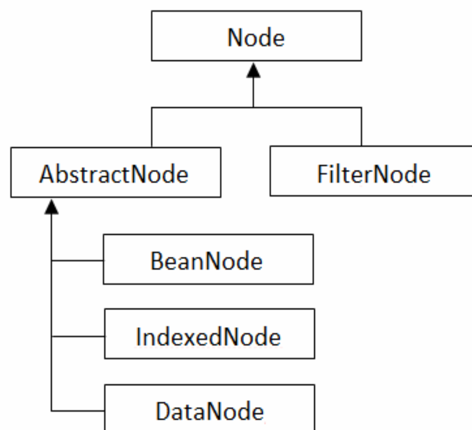
Nodovi predstavljaju *žive* komponente, jer svaka promena će inicirati promenu prikaza na korisničkom interfejsu. Svaki nod poseduje osobine po kojima se razlikuje od ostalih nodova, to su:

- **Naziv** – svaki nod ima svoj naziv
- **Opis** – koji se može prikazati kao tooltip tekst.
- **Ikona** – po kojoj se nodovi vizuelno razlikuju.
- **Akcije** – koje se pozivaju preko popup menija
- **Deca** – svaki nod može imati više dece
- **Atributi** – koji se mogu podešavati preko Property Sheet-a

Najčešće korišćeni tipovi nodova su sledeći:

- ***Data* nodovi** – ovaj tip nodova zasnovan je na *data* objektima, jednostavan primer predstavlja reprezentacija XML fajla na disku, moguća je njegova izmena, pregledanje i premeštanje. Java klasa koja nasleđuje *JFrame* bio bi složeniji primer, koji se sastoji od dva stabla, na jednom bi bile predstavljene klase, metode i polja, a na drugom hijerarhija Swing komponenti.
- ***Data* folder nodovi** – ovaj tip nodova predstavlja kontejner za *data* nodove, i analogan je java paketima.
- **Komponente palete** – svaka kategorija na paleti predstavlja nod, a elementi svake kategorije predstavljaju decu nodove.
- ***Breakpoint*** – svaki *breakpoint* kojim se definiše tačka pauziranja prilikom debug-ovanja aplikacije predstavlja jedan nod, pa je samim tim folder koji sadrži informacije o svim breakpoint-ima nod roditelj.
- **Nod projekta** – predstavlja nod čija deca predstavljaju različite fajlove i akcije nad njima koje su specifične za dati projekat.

Osnovna ponašanja nodova su definisana *Node* klasom. Na slici 2 prikazane su podklase klase *Node*.



Slika 2 - Hijerarhija Node podklasa

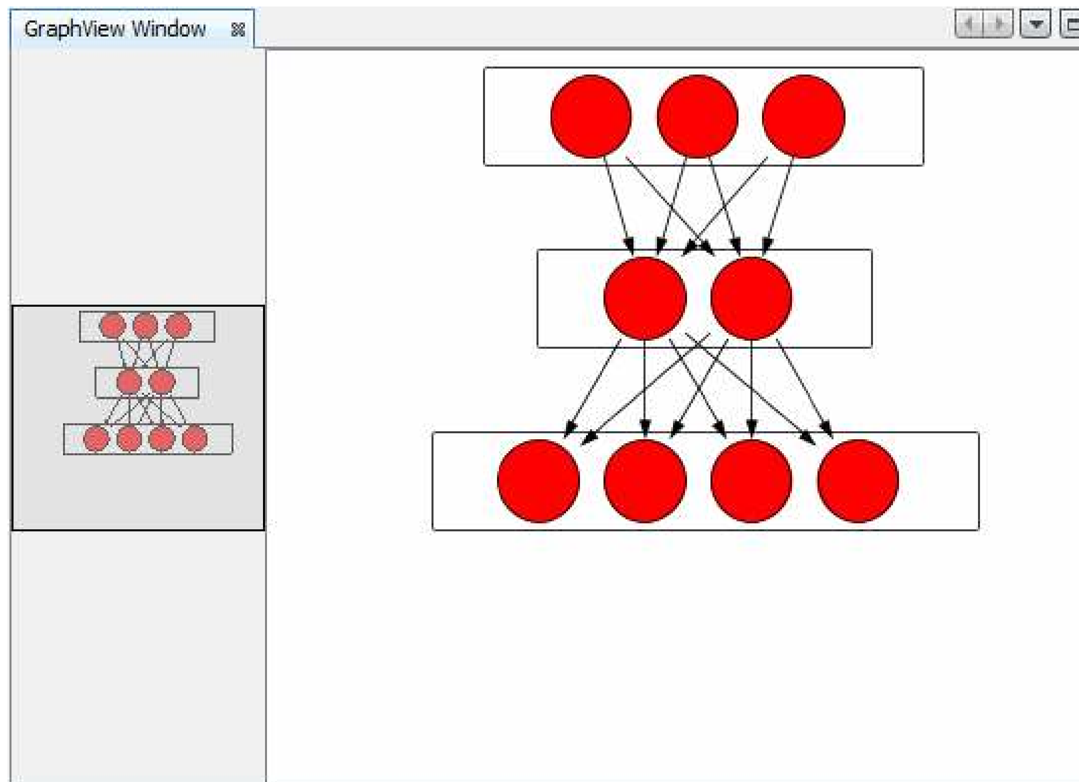
- ***AbstractNode*** klasa obezbeđuje osnovne osobine Node klase, koristi se za instanciranje noda bez potrebe za implementiranjem i proširivanjem Node klase.
- ***FilterNode*** je zadužen za kreiranje proxy noda koji delegira pozive metoda do originalnog noda. Ova vrsta nodova se koristi kada je potrebno prikazati podatke na više različitih načina.
- ***BeanNode*** omogućava kreiranje noda od java objekta. BeanNode će pronaći sva svojstva JavaBean objekta i prikazati ih kao svojstva noda pomoću PropertySheet-a.
- ***IndexedNode*** omogućava uređivanje elemenata po datim indeksima, odnosno indeksiranje elemenata.
- ***DataNode*** se koristi za reprezentovanje data objekata.

Svaki objekat klase Node poseduje objekat Children, koji obezbeđuje kontejner za podnodove. Objekat *Children* je odgovoran za kreiranje, uređivanje i strukturu nodova dece. Ovaj objekat se prosleđuje u konstruktoru AbstractNode klase. U najvećem broju slučajeva koristi se *Children.Keys*, međutim ako ne želimo da nod koji kreiramo ima dalje račvanje i podnodove, u konstruktoru se prosleđuje *Children.LEAF*. Pored njih na raspolaganju su i druge klase: *Children.Array* – super klasa za sve *Children* klase, *Children.Map<T>* – nodovi se sortiraju u mapu, *Children.SortedArray* – proširuje *Children.Array* sa *Comparator-om*, *Children.SortedMap<T>* proširuje *Children.Map<T>* sa *Comparator-om*.

Nodovi omogućavaju dodavanje akcija koje su osetljive na trenutno stanje. *DataNode* predstavlja akcije *DataObjecta* koji reprezentuje. *DataLoader* definiše MIME-specifičan folder u *layer* fajlu sa metodom *actionsContext()* gde su akcije registrovane. Kod nodova koji ne predstavljaju objekte klase DataObject pomoću metode *getActions()* u *Node* klasi defenišu se akcije konekst menija. Sve promene na nodovima mogu se pratiti pomoću *PropertyChangeListener-a*, kao i *NodeListener-a*. Pomoću *PropertyChangeListener-a* mogu se pratiti promene svojstava nodova korišćenjem *getPropertySet()* metode, a *NodeListener* se koristi za informisanje o promenama imena noda, ili izmenama na nodu roditelju ili detetu.

2.3 Visual Library API

NetBeans *Visual Library API* predstavlja biblioteku za prikazivanje različitih struktura. U osnovi, ova biblioteka je najpogodnija za prikazivanje grafova. Struktura *Visual Library API-a* predstavljena je stablom, klasa *Widget* predstavlja koren tog stabla, odnosno superklasu svih grafičkih komponenti.



Slika 3. - Primer widget-a

Na slici 3. grafički je prikazana struktura neuronske mreže, svaki deo neuronske mreže: sloj neurona, neuron i veza između neurona, predstavljen je klasom *Widget*. Na slici se vidi da jedan *widget* može služiti kao kontejner drugim widget-ima, i svaki *"Child Widget"* ima poziciju koja zavisi od njegovog roditelja. Klasa *Widget* koristi se i za definisanje granica i pozadine *widget-a*. Kao i *swing* kontejneri i *widget-i* mogu imati *layout* koji definiše poziciju njihove dece.

Widget predstavlja ekvivalent *swing* klasi *JComponent*, pa i sve komponente Visual Library API-a predstavljaju pod-klase klase *Widget*, od kojih su najbitnije, *IconNodeWidget*, *ImageWidget*, *LabelWidget*, *ConnectionWidget*, *LayerWidget*, *Scene*.

- **IconNodeWidget –** Predstavlja kompozitni *widget*, koji se sastoji od *ImageWidget-a* i *LabelWidget-a*.
- **ImageWidget –** *Widget* koji omogućava prikazivanje slika na sceni.

- **LabelWidget –** *Widget* koji omogućava prikazivanje teksta na sceni.
- **ConnectionWidget –** Ovaj *widget* se koristi za prikazivanje linija veza, i definisanje kontrolnih tačaka koji određuju putanju ovih linija.
- **LayerWidget –** Predstavlja transparentni *widget* koji služi za organizaciju i grupisanje *widget-a* na sceni.
- **Scene –** Predstavlja koreni element hijerarhije *widget-a* na sceni, koji je odgovoran za prikaz kompletne renderovane površine.

Osnovna svojstva widget-a :

- Svaki *widget* ima svoj okvir, koji je osnovno definisan klasom *EmptyBorder* i predstavlja prazan okvir. Izgled okvira može se promeniti pozivanjem metode *setBorder()*. Okvir je predstavljen interfejsom *Border*, veliki broj klasa implementira ovaj interfejs, pored klase *EmptyBorder*, tu su još i klase *LineBorder*, *BevelBorder*, *DashedBorder*, *ImageBorder*, *ResizeBorder*, *RoundedBorder* i *SwingBorder*. *ResizeBorder* dodaje svakom *widget-u* mogućnost menjanja veličine. Postoji i klasa *CompositeBorder* koja omogućava kombinaciju bilo kojih navedenih klasa. Sami okviri se ne kreiraju direktno već preko klase *BorderFactory*.

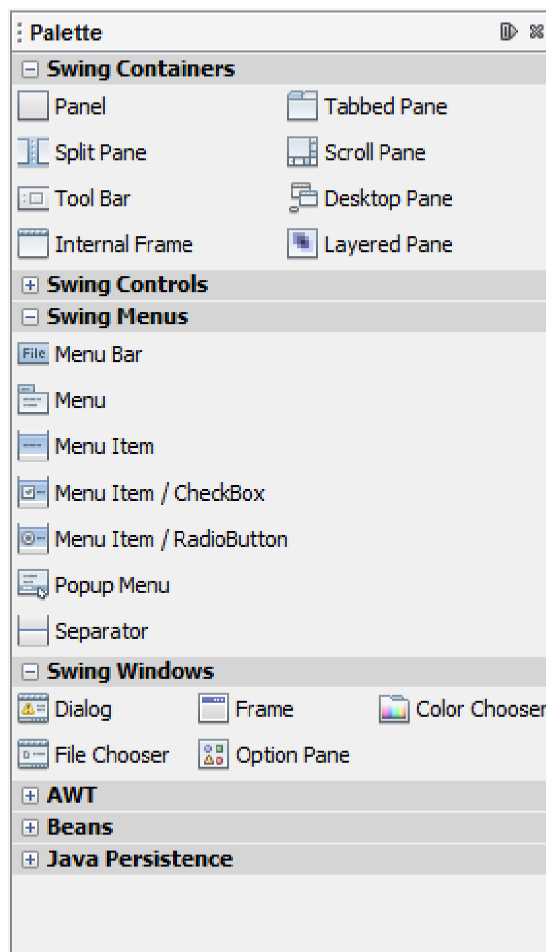
- Svakom *widget-u* može se dodeliti određeni layout, koji je odgovoran za raspored dece *widget-a* . Postoje četiri vrste layout-a, koji se kreiraju od strane LayoutFactory klase, i dodaju se widget-u pozivanjem setLayout() metode. **Absolute layout** koji dozvoljava widgetima da se pozicioniraju pozivom metode setPrefferedLocation(), bez automatskog raspoređivanja. **Flow layout** raspoređuje widget-e sekvencijalno horizontalno ili vertikalno. **Card Layout** prikazuje samo trenutno aktivan widget dok svi ostali imaju veličinu (0,0) pa su praktično nevidljivi. Metodom getActiveCard() određuje se trenutno aktivni widget. **Overlay layout** omogućava da se svi widgeti poređaju jedan preko drugog.

- Ponašanje *widget-a* uslovljeno je akcijama koje mu sami dodajemo. Ove akcije su definisane interfejsom *WidgetAction*. Akcije *widget-a* kreira klasa *ActionFactory*. Akcijama svakog *widget-a* upravlja klasa *WidgetAction.Chain*. Ova klasa prihvata događaje i prosleđuje ih odgovarajućim akcijama. Pozivanjem metoda *addAction()* i *removeAction()* *WidgetAction.Chain* klasa dodaje i briše akcije koje su dostupne nad *widget-om*. Klasa WidgetAction.Chain grupiše određene akcije u alate, i pozivanjem metode *setActiveTool()* aktivira samo one akcije koje su definisane postavljenim alatom, npr. na scenu možemo dodati veliki broj akcija, kao što su zumiranje, pomeranje objekata, *drag and drop*, i druge, ali ako želimo da korisnicima omogućimo samo *drag and drop* aktiviraćemo alat koji je zadužen za tu akciju, dok će ostale akcije postati nedostupne.

2.4 Common Palette

Common palette modul je zadužen za grafičko prikazivanje komponenti koje se mogu prevlačiti na radnu površinu aplikacije koja je predviđena za smeštanje tih komponenti. Glavni zadatak ovog modula je da korisniku aplikacije obezbedi lako kreiranje i manipulaciju neuronskim mrežama uz pomoć palete. Na slici 4. Prikazan je izgled palete NetBeans IDE razvojnog okruženja.

Postoje dva načina za dodavanje komponenti na paletu. Prvi način podrazumeva definisanje komponenti u XML fajlu, a drugi kreiranje stabla komponenti i prikazivanje tog stabla na paleti.



Slika 4 - Primer palete NetBeans IDE razvojnog okruženja

3. Zahtevi i analiza

Vizualni alati predstavljaju alate koji omogućavaju grafički prikaz i editovanje određenih podataka. NetBeans platforma omogućava brzu izradu ovakvih alata rešenjima koja su integrisana u *VisualLibrary API-u*. Cilj ovog rada je kreiranje vizualiznog alata, za „Neuroph“, aplikaciju, koji omogućava kreianje i izmenu neuronskih mreža. Potrebno je obezbediti vizualno kreiranje neuronske mreže, i editovanje.

Ovaj vizuelni sistem obuhvata:

- Scenu koja će služiti kao kontejner za widget-e neuronske mreže.
- *Widget-e* osnovnih komponenti neuronskih mreža, kao što su neuroni i slojevi neurona.
- Paletu osnovnih elemenata, sa *drag and drop* funkcijom.

3.1 Scena

Komponente *VisualLibrary API-a*, kao na primer *widget-i* organizuju se kao stablo, što znači da svaki *widget* može imati decu *widget-e*. Sama klasa *Scene* predstavlja *widget* koji služi kao kontejner za sve ostale *widget-e* koje treba prikazati. Dakle Scena predstavlja koren stabla widget-a . Scena se predstavlja pogledom koji je ustvari instance kalse JComponent sa JScrollPane-om.

Konkretno za „Neuroph“ je potrebno da:

- Scena sadrži *Widget* koji predstavlja neuronsku mrežu (*NeuralNetworkWidget*).
- NeuralNetworkWidget predstavlja kontejner za *widget-e* slojeva neuronske mreže (*NeuralLayerWidget*).
- *NeuralLayerWidget* predstavlja kontejner za *widget-e* neurona (*NeuronWidget*).
- Scena omogući definisanje veza između neurona i njihov grafički prikaz, odnosno mora da sadrži konekcioni lejer. Konekcioni lejer predstavlja *LayerWidget* koji služi za smeštanje i organizovanje *ConnectionWidget-a*.
- Scena omogući interakciju sa *widget-ima* na njoj. Interakciju obezbeđuje interakcioni lejer. Interakcioni lejer predstavlja *LayerWidget* koji služi za smeštanje i organizovanje akcija nad scenom.
- Scena ima umanjeni prikaz, odnosno prikaz scene iz „ptičje“ perspektive, kako bi korisnik imao uvid u sve objekte koji se nalaze na sceni.

3.2 Widget-i

Kao što je ranije rečeno klasa *Widget* predstavlja ekvivalent klasi *JComponent*, samim tim predstavlja osnovni kontejner za sve ostale vizuelne komponente. Kao što smo videli ranije postoji veliki broj različitih implementacije klase *Widget*, kao što su *LayerWidget*, *IconNodeWidget*, *ImageWidget* i druge.

Za kreiranje grafičkih komponenti neuronske mreže potrebno je napraviti sopstvene implementacije klase *Widget*, koje će pored osnovnih osobina *widget-a* sadržati i referencu na objekat neuronske mreže na koji se odnosi. Referenciranjem se stvara veza između komponenti neuronske mreže i njihove grafičke prezentacije, i samim tim se omogućava laka manipulaciju neuronskom mrežom preko korisničkog interfejsa. Pod manipulacijom neuronskom mrežom podrazumeva se dodavanje i brisanje neurona, slojeva neurona i veza između neurona, i promena njihovih svojstava.

Widget-i koje je potrebno definisati:

- **NeuralNetworkWidget** – sadrži referencu na određenu neuronsku mrežu, grafički je prikazuje i služi kao kontejner za sve ostale komponente;
- **NeuralLayerWidget** – sadrži referencu na određeni lejer neuronske mreže, grafički ga prikazuje i služi kao kontejner za widget-e neurona;
- **NeuronWidget** – sadrži referencu na neuron i služi za grafički prikaz neurona. Ovaj *widget* poseduje i listu *widget-a* ulaznih i izlaznih konekcija neurona;
- **NeuronConnectionWidget** – poseduje referencu na određenu vezu između neurona, i grafički je prikazuje.

Takođe je potrebno definisati akcije i menije nad pojedinim komponentama. Kako bi se obezbedila interaktivnost aplikacije

3.3 Paleta

Paleta predstavlja prozor sa komponentama, koje se mogu prevlačiti (drag n drop) na radnu površinu, i nad kojima se mogu izvršavati određene akcije. Paleta se sastoji od nodova koji su organizovani u stablo. Ovakva struktura omogućava grupisanje nodova po kategorijama, tako npr. kod palete NetBeans IDE razvojnog okruženja postoje kategorije: kontejneri, koja sadrži sve komponente koje predstavljaju kontejnere, meni, koja sadrži sve menije i druge.

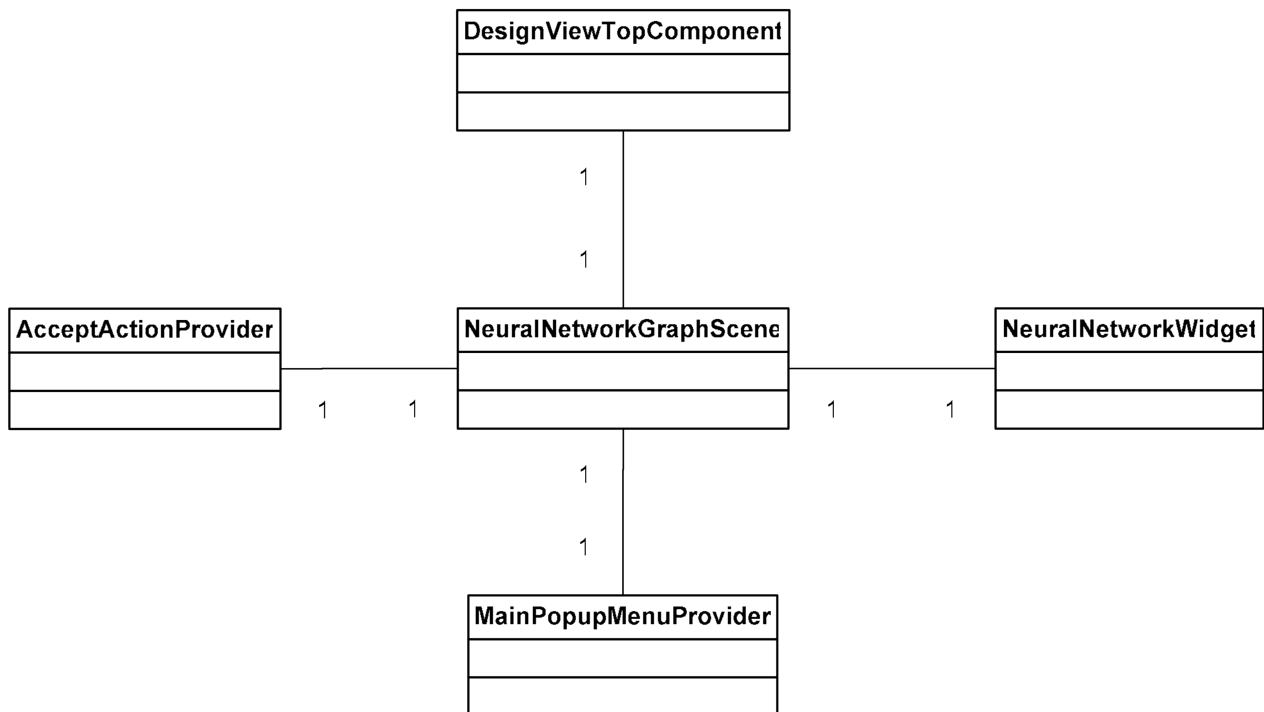
Za „Neuroph“ je potrebno razviti paletu koja će sadržati nodove koji predstavljaju komponente neuronske mreže (neuron i lejer), i koji će moći da se prevlače na scenu. Prilikom prevlačenja potrebno je obezbediti prikaz ikone noda, a posle spuštanja komponente na scene potrebno je osvežiti neuronsku mrežu.

4. Projektovanje

U ovom poglavlju isprojektovane su komponente vizuelnih alata aplikacije *Neuroph* pomoću dijagrama klasa, i objašnjene su prikazane strukture.

4.1 GraphScene

U prethodnom poglavlju naveli smo zahteve koji su se odnosili na kreiranje scene. Potrebno je napraviti interaktivnu scenu koja će služiti za prikaz neuronske mreže i njeno editovanje.



Slika 5 - Dijagram klasa koje čine scenu

DesignViewTopComponent

Svi prozori koje kreiramo predstavljaju se klasom *TopComponent,* koja je deo *Windows System API-a*. Uloga klase *TopComponent* jeste da kreira prozor koji će se integrisati u aplikaciju koja je zasnovana na NetBeans platformi. Svakom prozoru koji dodamo u aplikaciju možemo pristupiti iz menija *Windows*, a naziv stavke menija definišemo u samoj klasi. Klasa *DesignViewTopComponent* predstavlja prozor na kome će se prikazivati scena. Komunikacija između ovog prozora i palete obezbeđuje se pozivanjem metode *associateLookup(),* sa parametrom koji predstavlja *Lookup* palete, u konstruktoru klase.

NeuralNetworkGraphScene

Klasa *NeuralNetworkGraphScene* predstavlja samu scenu. Ova klasa sadrži polja *interactionLayer* i *connectionLayer* koja su tipa *LayerWidget* i predstavljaju kontejnere za widget-e akcija i konekcija. Takođe poseduje polje *neuralNetworkWidget* koje predstavlja neuronsku mrežu. Ova klasa poseduje metode koje su zadužene za kreiranje *widget-a* neurona *createNeuron()* i lejera *createLayer(),* a takođe i metodu *visualiseNetwork()* koja kao parametar prima neuronsku mrežu koju je potrebno prikazati, i prikazuje je na sceni.

MainPopupMenuProvider

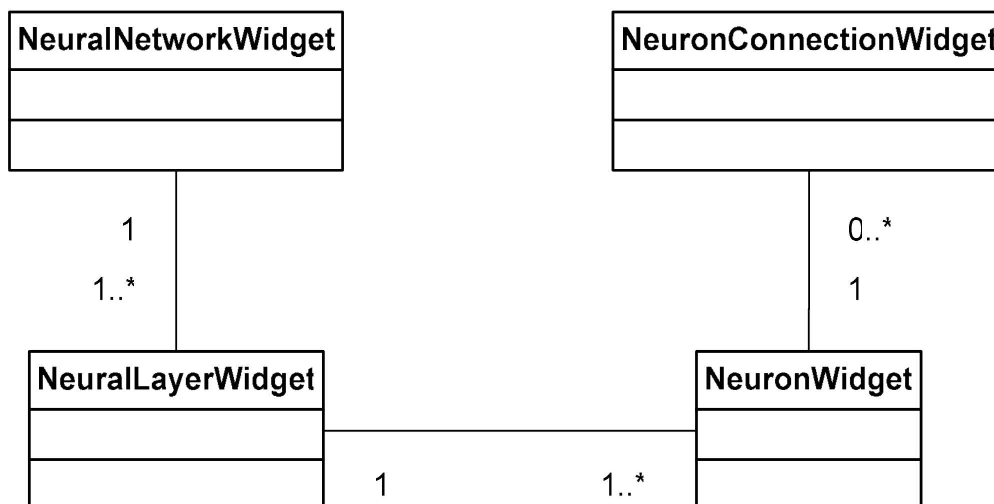
Svi padajući meniji predstavljaju se interfejsom *PopupMenuProvider,* i kreiraju se pozivom metode *getPopupMenu()*. Klasa *MainPopupMenuProvider* implementira ovaj interfejs, a samim tim i metodu *getPopupMenu()* u kojoj se kreira odgovarajući padajući meni.

AcceptActionProvider

Klasa *ActionFactory* zadužena je za kreiranje svih akcija. *ActionProvider* definiše šta se ustvari treba uraditi kada izvršimo određenu akciju. *AcceptActionProvider* definiše šta je potrebno uraditi kada se doda komponenta sa palete na scenu.

4.2 NeuralNetworkWidget

Svi elementi koji se mogu dodati na scenu predstavljeni su klasom *Widget*. *NeuralNetworkWidget* klasa zadužena je za vizuelnu reprezentaciju neuronske mreže. Nasleđuje klasu *LayerWidget* i sadrži referencu ka neuronskoj mreži čime je omogućeno editovanje neuronske mreže preko DesignView prozora. Takođe poseduje metode *addLayer()* i *removeLayer()* koje u isto vreme obrađuju neuronsku mrežu i njenu vizuelnu interpretaciju.



Slika 6 - Struktura vizuelne reprezentacije neuronske mreže

NeuralLayerWidget

Klasom *NeuralLayerWidget* vizuelno se reprezentuju neuronski lejeri. Ova klasa poseduje referencu prema neuronskom lejeru učitane neuronske mreže, a metode *addNeuron()* i *removeNeuron()* omogućavaju dodavanje i izbacivanje neurona iz neuronskih lejera.

NeuronWidget

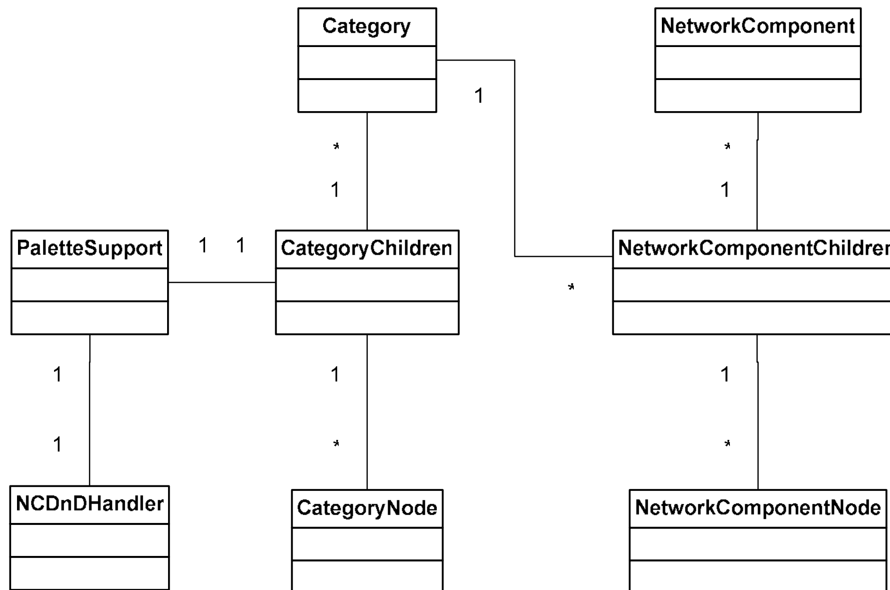
Vizuelna reprezentacija neurona definisana je preko klase *NeuronWidget*, za razliku od prethodno pomenutih klasa, *NeuronWidget* nasleđuje *IconNodeWidget* klasu. Pored reference na neuron poseduje i reference na widget-e svih konekcija koje poseduje taj neuron, što omogućava laku manipulaciju neuronima i vezama koje čine.

NeuronConnectionWidget

Klasa *NeuronConnectionWidget* nasleđuje *ConnectionWidget*, koja obuhvata sve što je potrebno za kreiranje konekcionih widget-a. *NeuronConn*ectionWidget poseduje referencu na stvarnu vezu u neuronskoj mreži, i na neurone koji kreiraju tu vezu. Što omogućava pristup ulaznom i izlaznom neuronu u bilo kom trenutku i brisanje veze.

4.3 Palette

U ovom poglavlju opisana je struktura palete. Na slici 7 prikazan je dijagram klasa koje kreiraju paletu, sa slike se može zaključiti da se paleta sastoji od *nodov-a* koji predstavljaju kategorije i njihove elemente, i provajdera *DragAndDrop* akcije.



Slika 7 - Dijagram klasa koje kreiraju paletu

Category

Komponente palete potrebno je grupisati po određenim kriterijuma, klasa *Category* definiše svaku kategoriju komponenti na paleti. Kategorija ima polje name koje predstavlja naziv, odnosno ime koje se prikazuje na paleti.

NetworkComponent

Komponente palete opisane su u klasi *NetworkComponent*. Svaka komponenta ima naslov, ikonu, identifikacioni broj i kategoriju.

Node klase

U klasama *CategoryNode* i *NetworkComponentNode* definiše se izgled komponenti palete. U njima se podešavaju svojstva komponenti kao što su naslov i ikona.

Children klase

U klasama *CategeryChildren* i *NetworkComponentChildren* definišu se sve kategorije i komponente koje se dodaju na paletu. Klasa *CategoryChildren* poseduje niz kategorija za koje se kreiraju nodovi. *NetworkComponentChildren* klasa poseduje niz komponenti palete, i metodu *initCollection()* koja kao ulazni parametar prima kategoriju, i inicijalizuje sve komponente zadate kategorije. Dakle uloga ovih klasa jeste inicijalizacija komponenti palete.

NCDnDHandler

Klasa *NCDnDHandler* predstavlja instancu klase *DragAndDropHandler*. Osnovna funkcija ove klase jeste povezivanje scene sa paletom i dodavanje DragAndDrop akcija na komponente palete. U klasi NCDnDHandler definisan je protokol preko koga komuniciraju paleta i scena koja prihvata komponente palete.

PaletteSupport

Za kreiranje palete zadužena je klasa *PaletteFactory*. Klasa *PaletteSupport* predstavlja posrednika između komponenti palete, akcija, *DragAndDropHandler-a* i klase *PaletteFactory*, i glavna uloga joj je kreiranje palete.

5. Implementacija

U ovom poglavlju prikazana je implementacija vizuelnih alata pomoću NetBeans platforme na primeru *Neuroph-a*. Kao razvojno okruženje tokom implementacije korišćen je NetBeans IDE 6.9.1.

5.1 Implementacija scene

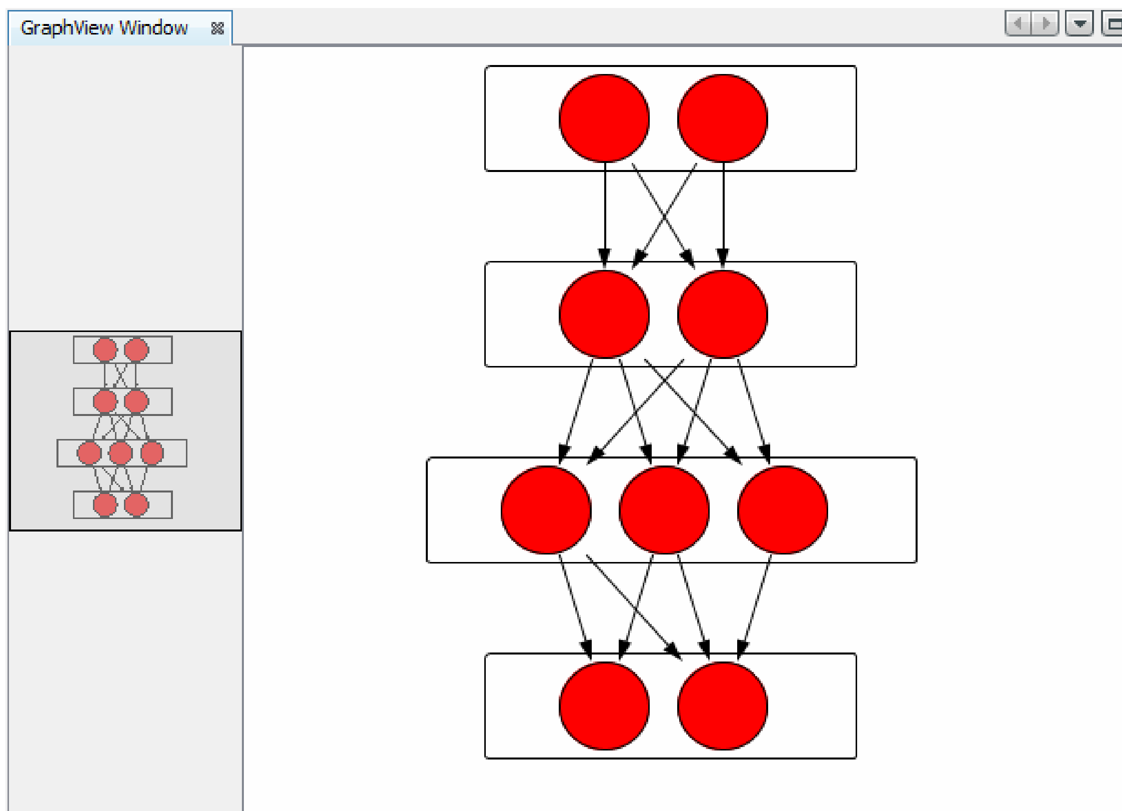
Na početku je potrebno kreirati prozor na kome će scena biti prikazana. U aplikacijama koje su zasnovane na NetBeans platformi svi prozori se predstavljaju klasom *TopComponent*. Dakle klasa *GraphViewTopComponent,* predstavlja komponentu koja će služiti kao kontejner za scenu*.*

```
public GraphViewTopComponent() {
    initComponents();
    …
    nnet =  new NeuralNetwork();
    NeuralNetworkGraphScene scene = new NeuralNetworkGraphScene();
    view = scene.createView();
    viewPane.setViewportView(view);
    add(scene.createSatelliteView(), BorderLayout.WEST);
    associateLookup(Lookups.fixed(new
    Object[]{PaletteSupport.createPalette()}));
}
```

U konstruktoru klase *GraphViewTopComponent* kreira se nova scena i pogled na tu scenu. Pogled se zatim dodaje na panel *viewPane*, koji se prikazuje na radnoj površini. Ovoj kompenenti se na kraju dodaje *lookup* koji omogućava komunikaciju sa paletom. Zatim je potrebno kreirati samu scenu, klasa *NeuralNetworkGraphScene* koja implementira interfejs *GraphScene*, predstavlja scenu neuronske mreže.

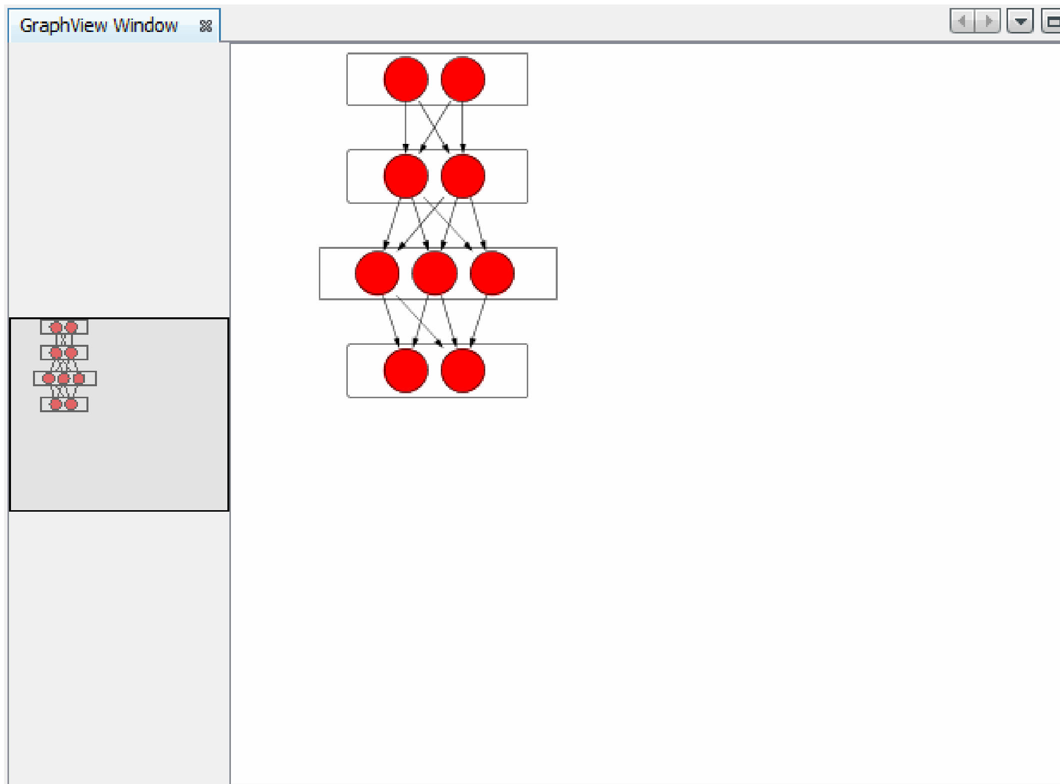
```
public NeuralNetworkGraphScene()   {
    interractionLayer = new LayerWidget(this);
    addChild(interractionLayer);
    nnw = new NeuralNetworkWidget(this);
    nnw.setLayout(LayoutFactory.createVerticalFlowLayout(
    LayoutFactory.SerialAlignment.CENTER, 50));
    nnw.setNnet(GraphViewTopComponent.getNeuralNetwork());
    nnw.setPreferredLocation(new Point(100, 10));
    addChild(nnw);
    connectionLayer = new LayerWidget(nnw.getScene());
    addChild(connectionLayer);
    getActions().addAction(ActionFactory.createMouseCenteredZoomActi
        on(1.1));
    getActions().addAction(ActionFactory.createPanAction());
    getActions().addAction(ActionFactory.createPopupMenuAction(new
        MainPopupMenuProvider()));
}
```

U konstruktoru klase *NeuralNetworkGraphScene* kreira se osnova za vizuelizaciju neuronske mreže, metoda *addChild()* zadužena je za dodavanje *widget-a* na scenu, na početku se sceni dodaje *interactionLayer* koji omogućava interakciju sa objektima na sceni, zatim *nnw*, koji predstavlja widget neuronske mreže i na kraju *connectionLayer* koji je zadužen za organizaciju konekcionih widget-a. Zatim se sceni dodaju funkcije *zoom-a*, i padajući meni.



Slika 8 - Izgled scene pre korišćenja funkcije *zoom-a*

Na slikama 8 i 9 prikazan je izgled scene pre i posle primene funkcije *zoom-a*. Funkcija *zoom-a* omogućava uvećanje i umanjenje objekata na sceni, kako bi kreirana struktura bila preglednija ili kako bi se objekti mogli detaljnije ispitati.



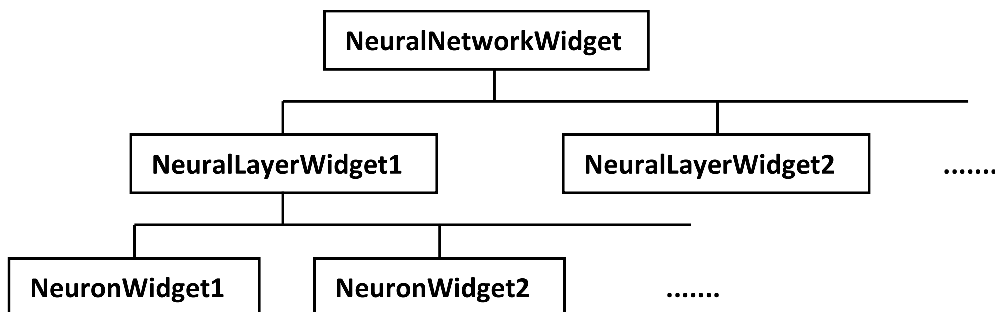
Slika 9 - Izgled scene posle korišćenja funkcije zoom-a

Metoda *visualiseNetwork()* zadužena je za vizuelizaciju neuronske mreže, odnosno za njeno prikazivanje na sceni. Na početku metode je potrebno ukloniti sve komponente koje se nalaze na sceni. Posle svake promene na sceni potrebno je pozvati metodu S*cene.validate(),* kako bi sve komponente koje koriste scenu bile obaveštene o poslednjoj promeni. Metoda *resolveStructure()* zadužena je za određivanje strukture neuronske mreže i njeno iscrtavanje na sceni, analogno tome metoda *resolveConnections()* je zadužena za određivanje svih konekcija i njihovo iscrtavanje na sceni.

```
public static void visualizeNetwork(NeuralNetwork nnet) {
    nnw.removeChildren();
    nnw.getScene().validate();
    connectionLayer.removeChildren();
    connectionLayer.getScene().validate();
    Vector<Neuron> neurons = new Vector<Neuron>();
    Vector<NeuronWidget> neuronws = new Vector<NeuronWidget>();
    resolveStructure(nnet, neurons, neuronws);
    resolveConnections(neurons, neuronws);
}
```

```java
public static void resolveStructure(NeuralNetwork nnet, Vector<Neuron>
                        n, Vector<NeuronWidget> nws) {
   for (int i = 0; i < nnet.getLayersCount(); i++) {
   NeuralLayerWidget nlw = createLayer(nnet.getLayerAt(i));
      for (int j = 0; j < nnet.getLayerAt(i).getNeuronsCount(); j++) {
         Neuron n = nnet.getLayerAt(i).getNeuronAt(j);
         NeuronWidget nw = createNeuron(n);
         resizeLayer(nlw);
         nlw.addChild(nw);
         neurons.add(n);
         neuronws.add(nw);
      }
   nnw.addChild(nlw);
   }
}
```

Detaljnijim pregledom metode *resolveStructure()* dolazi se do zaključka da se u njoj vrši prolazak kroz neuronsku mrežu i za svaku njenu komponentu kreira odgovarajući widget, koji se zatim dodaje u widget roditelj. Metoda *createLayer()* zadužena je za kreiranje vizuelne reprezentacije slojeva neurona, odnosno *NeuralLayerWidget-a*, a *createNeuron()* za kreiranje vizuelne reprezentacije neurona, odnosno *NeuronWidget-a*. Kada se kreira *NeuralLayerWidget* on se smešta u *NeuralNetworkWidget*, a *NeuronWidget* u *NeuralLayerWidget*. Na slici 10 prikazana je struktura vizuelne reprezentacije neuronske mreže, koja predstavlja stablo čiji je koren *NeuralNetworkWidget*, *NeuralLayerWidget-i* predstavljaju njegovu decu, a *NeuronWidget-i* decu *NeuralLayerWidget-a*.



Slika 10 - Stablo vizuelne reprezentacije neuronske mreže

```
public static void resolveConnections(Vector<Neuron> ns,
Vector<NeuronWidget> nws) {
   for (int k = 0; k < ns.size(); k++) {
   for (int m = 0; m < ns.size(); m++) {
      Vector<Connection> inputConns = ns.get(m).getInputConnections();
      for (int n = 0; n < inputConns.size(); n++) {
         if (inputConns.get(n).getConnectedNeuron().equals(ns.get(k)))
         {
            NeuronConnectionWidget cw = new
               NeuronConnectionWidget(nnw.getScene(), inputConns.get(n),
               nws.get(k), nws.get(m));
            cw.setTargetAnchorShape(AnchorShape.TRIANGLE_FILLED);
            cw.setSourceAnchor(AnchorFactory.createRectangularAnchor(nw
               s.get(k)));
            cw.setTargetAnchor(AnchorFactory.createRectangularAnchor(nw
               s.get(m)));
            cw.getActions().addAction(ActionFactory.createPopupMenuActi
               on(new ConnectionPopupMenuProvider()));
            nws.get(k).addConnection(cw);
            nws.get(m).addConnection(cw);
            connectionLayer.addChild(cw);
         }
      }
   }
   }
}
```

Metoda *resolveConnections()* takođe prolazi kroz neuronsku mrežu, ali je zadužena za kreiranje i iscrtavanje konekcionih *widget-a*, odnosno veza između neurona. S obzirom da je konekcioni lejer zadužen za organizaciju konekcionih *widget-a* , na kraju svake iteracije *NeuronConnectionWidget* dodajemo u *connectionLayer.*

5.2 Implementacija widget-a

Kako bi se uspostavila veza između neuronske mreže i njene vizuelne reprezentacije, *widget-a*, potrebno je da svaki *widget* ima referencu na komponentu neuronske mreže koju reprezentuje. Kako ni jedna od postojećih implementacija klase *Widget* ne zadovoljava gore navedeni uslov, potrebno je kreirati sopstvene implementacije.

```
public class NeuralNetworkWidget extends LayerWidget {
   private NeuralNetwork nnet;
   public NeuralNetworkWidget(Scene scene) {
      super(scene);
      nnet = new NeuralNetwork();
   }
   public void addLayer(int position, NeuralLayerWidget nlw) {
      nnet.addLayer(nlw.getLayer());
      addChild(position, nlw);
   }
   public NeuralNetwork getNnet() {
      return nnet;
   }
   public void setNnet(NeuralNetwork nnet) {
      this.nnet = nnet;
   }
}
```

Na početku je potrebno kreirati *widget* koji će služiti kao kontejner za komponente neuronske mreže. U ovom slučaju potrebno je kreirati klasu *NeuralNetworkWidget*. Kako ova predstavlja jednu vrstu kontejnera za druge *widget-e* koji se neće videti na sceni, potrebno je da nasledi klasu *LayerWidget*. Kada je definisana struktura ovog kontejnera, potrebno ga je povezati sa neuronskom mrežom. Definisanjem polja koje se odnosi na neuronsku mrežu i javne *get()* i *set()* metode koje omogućavaju pristup tom polju kreira se veza sa neuronskom mrežom. Metoda *addLayer()* zadužena za dodavanje NeuralLayerWidget-a na scenu i dodavanje sloja neurona u neuronsku mrežu.

Sada je potrebno napraviti *widget* koji će predstavljati decu *NeuralNetworkWidget-a*, a takođe će imati svoju decu koja će reprezentovati neurone. Dakle i ovaj *widget* će predstavljati kontejner za druge *widget-e*, ali će se razlikovati od *NeuralNetworkWidget-a* jer će imati svoj izgled i poziciju na sceni, i samim tim nasleđuje klasu *IconNodeWidget*.

```java
public class NeuralLayerWidget extends IconNodeWidget {
   private Layer layer;
   NeuralLayerType type;
   public NeuralLayerWidget(Scene scene, Layer layer) {
      super(scene);
      this.layer = layer;
   }
   public Layer getLayer() {
      return layer;
   }
   public NeuralLayerType getType() {
      return type;
   }
   public void setType(NeuralLayerType type) {
      this.type = type;
   }
   public void addNeuron(NeuronWidget neuronWidget) {
      layer.addNeuron(neuronWidget.getNeuron());
      addChild(neuronWidget);
   }
}
```

Klasa *NeuralLayerWidget* predstavlja vizuelnu interpretaciju neuronskih lejera, i sadrži referencu na lejer neuronske mreže koji reprezentuje. Metoda *addNeuron()* služi za dodavanje vizuelne reprezentacije neurona na *NeuralLayerWidget*, i samih neurona u mrežu.

Sledeći je na redu widget koji predstavlja neuron. Klasa *NeuronWidget* takođe nasleđuje *IconNodeWidget* klasu, poseduje referencu na neuron na koji se odnosi, ali i na sve veze koje taj neuron poseduje. Metoda *addConnection()* zadužena je za dodavanje novih, vizuelnih i stvarnih veza, *removeAllConnections()* za brisanje svih veza, dok metoda *getConnections()* vraća sve veze koje jedan neuron poseduje.

```java
public class NeuronWidget extends IconNodeWidget {
   private Neuron neuron;
   private List<ConnectionWidget> connections;
   public NeuronWidget (Scene scene, Neuron neuron) {
      super(scene);
      connections = new ArrayList<ConnectionWidget>();
      this.neuron = neuron;
   }
   public Neuron getNeuron() {
      return neuron;
   }
   public void addConnection(ConnectionWidget cw) {
      connections.add(cw);
   }
   public void removeAllConnections() {
      connections.clear();
   }
   public List<ConnectionWidget> getConnections() {
      return connections;
   }
}
```

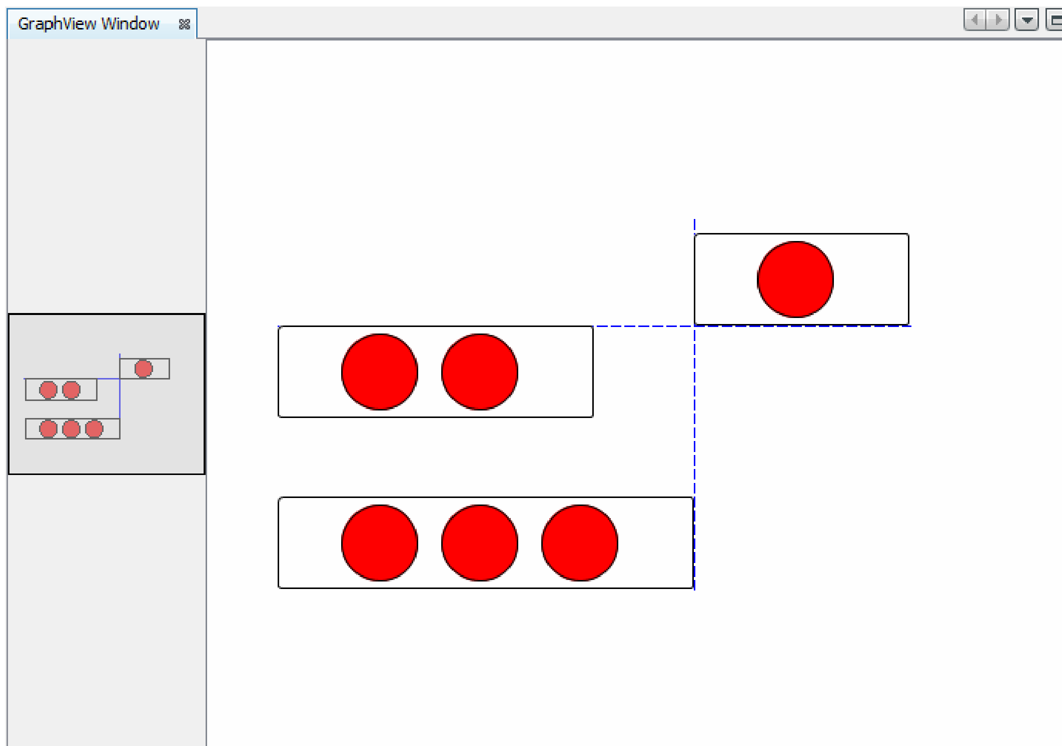
Poslednji *widget* koji je potrebno kreirati treba da reprezentuje vezu između neurona. Na početku se kreira klasa *NeuronConnectionWidget*, za razliku od ostalih implementiranih *Widget* klasa, ova klasa nasleđuje *ConnectionWidget* klasu. Ova implementacija *widget-a* specijalizovana je za kreiranje usmerenih linija veza.

```java
public class NeuronConnectionWidget extends ConnectionWidget {
   private Connection connection;
   private NeuronWidget src;
   private NeuronWidget trg;
   public NeuronConnectionWidget(Scene scene, Connection connection,
                                 NeuronWidget src, NeuronWidget trg) {
      super(scene);
      this.connection = connection;
      this.src = src;
      this.trg = trg;
   }
   public Connection getConnection() {
      return connection;
   }
   public void removeConn() {
      trg.getNeuron().removeInputConnectionFrom(src.getNeuron());
   }
}
```

Kao i kod ostalih implementacija *widget-a* , *NeuronConnectionWidget* sadrži referencu ka stvarnoj vezi dva neurona, ali takođe sadrži i reference na neurone koje povezuje. Metoda *removeConn()* zadužena je za brisanje vezu između neurona.

```
private static NeuralLayerWidget createLayer(Layer layer){
   NeuralLayerWidget nlw = new NeuralLayerWidget(nnw.getScene(),
                                layer);
   Layout layout = LayoutFactory.createHorizontalFlowLayout(
                        LayoutFactory.SerialAlignment.LEFT_TOP, 15);
   nlw.setLayout(layout);
   nlw.setPreferredSize(new Dimension(80, 60));
   nlw.setOpaque(true);
   nlw.setBorder(BorderFactory.createRoundedBorder(5, 5, Color.white,
                        Color.black));
   nlw.getActions().addAction(
       ActionFactory.createAlignWithMoveAction(nnw, interractionLayer,
       ActionFactory.createDefaultAlignWithMoveDecorator()));
   return nlw;
}
```

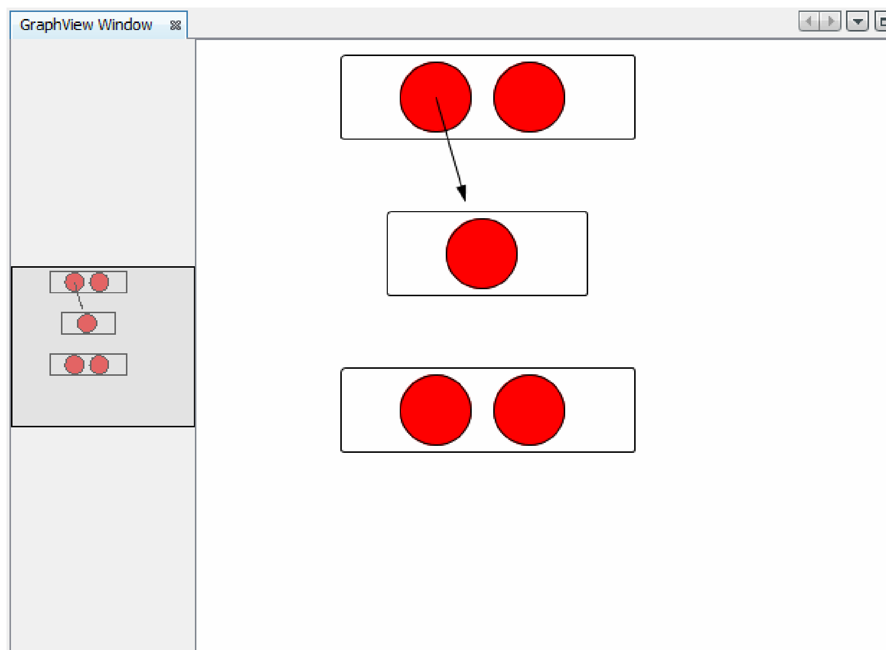
U metodi *resolveStructure()* kreiranje *widget-a* slojeva neurona vrši se pozivanjem metode *createLayer()*. Ova metoda ima zadatak da kreira *NeuralLayerWidget* i podesi njegove parametre kao što su dimenzija, lejout i okvir. Na kraju se lejeru dodaje *AlignWithMove* akcija, koja omogućava lejerima da se poravnaju u liniju prilikom pomeranja na sceni. Na slici 11 pokazano je kako funkcioniše *AlignWithMove* akcija.



Slika 11 - Primer *AlignWithMove* akcije

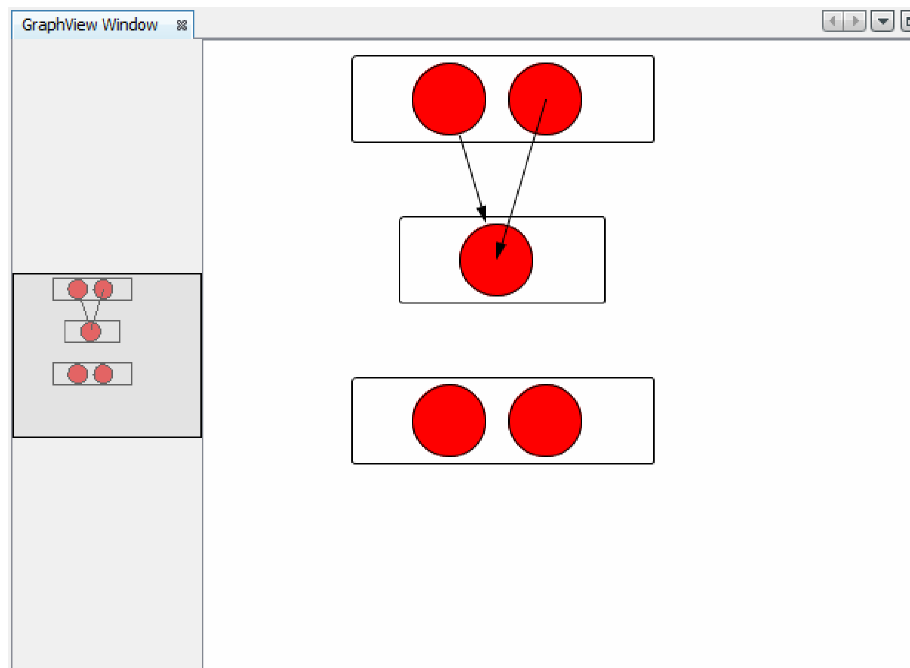
```java
private static NeuronWidget createNeuron(Neuron neuron) {
    NeuronWidget nw = new NeuronWidget(nnw.getScene(), neuron);
    nw.setToolTipText("Hold Ctrl and drag to create connection");
    nw.setPreferredSize(new Dimension(50, 50));
    nw.setBorder(BorderFactory.createRoundedBorder(50, 50, Color.red,
    Color.black));
    nw.getActions().addAction(ActionFactory.createPopupMenuAction(new
    NeuronPopupMenuProvider()));
    nw.getActions().addAction(ActionFactory.createExtendedConnectAction(
    connectionLayer, new FunctionConnectProvider(nnw)));
    nw.setOpaque(false);
    nw.bringToFront();
    return nw;
}
```

Metoda *createNeuron()* zadužena je za kreiranje *NeuronWidget-a*, i definisanje njegovog izgleda. Na *widget* se može postaviti pomoćni tekst, koji ima ulogu da korisniku olakša upotrebu aplikacije, u ovom slučaju obaveštava korisnika na koji način može da kreira veze između neurona.



Slika 12 - Primer kreiranja konekcije 1.

Ukoliko se drži taster ctrl i povuče se kursor sa neurona kreiraće se konekciona liniju kao na slici 12. Kreirana linija može se spojiti sa neuronom koji nije na istom lejeru, kada se poveže sa neuronom linija se centrira kao na slici 13, i puštanjem strelice neuroni se povezuju.



Slika 13 - Primer kreiranja konekcije 2.

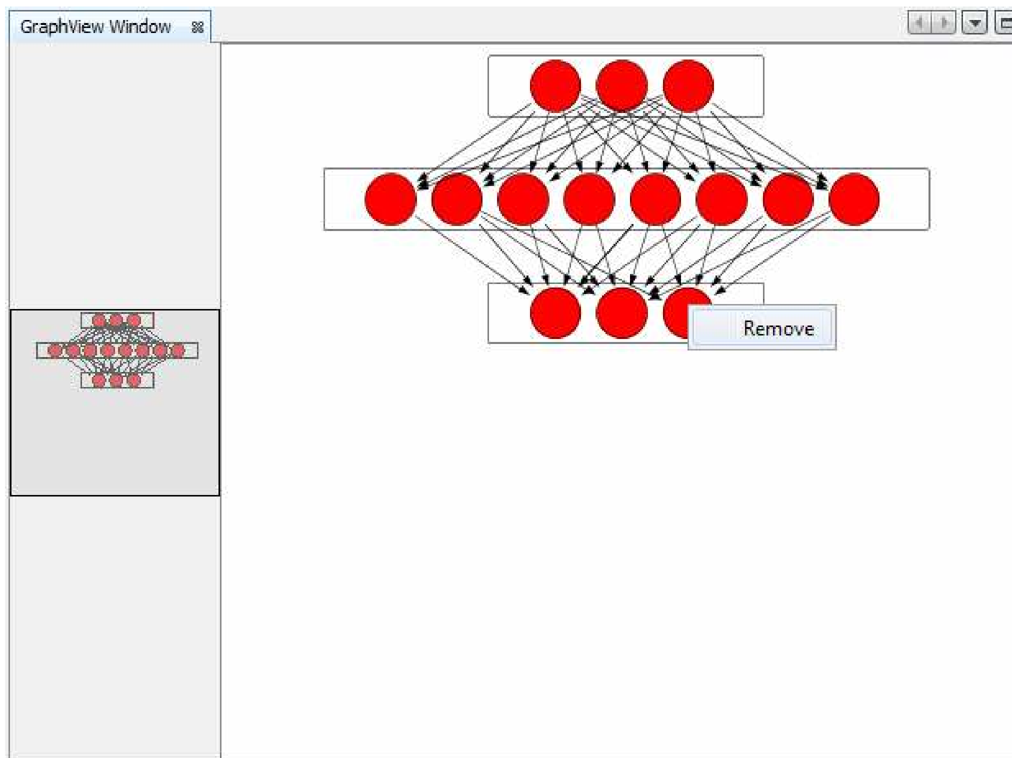
2.2.1. Implementacija provajdera akcija

Kako bi obezbedili potrebnu interaktivnost aplikacije, *Widget-ima* je potrebno dodati akcije kao što su npr. popup meni i mogućnost kreiranje veza. Neke metode klase *ActionFactory* kao parametar zahtevaju provajder akcije. U slučaju neurona, prilikom dodavanja *popup* menija prosleđujemo provajder *NeuronPopupMenuProvider*, u kome je definisan izgled samog *popup* menija i mogućih akcija. Prilikom dodavanja akcije kreiranja veza konstruktoru klase *ActionFactory* prosleđuje se *FunctionConnectProvider* koji definiše vezu.

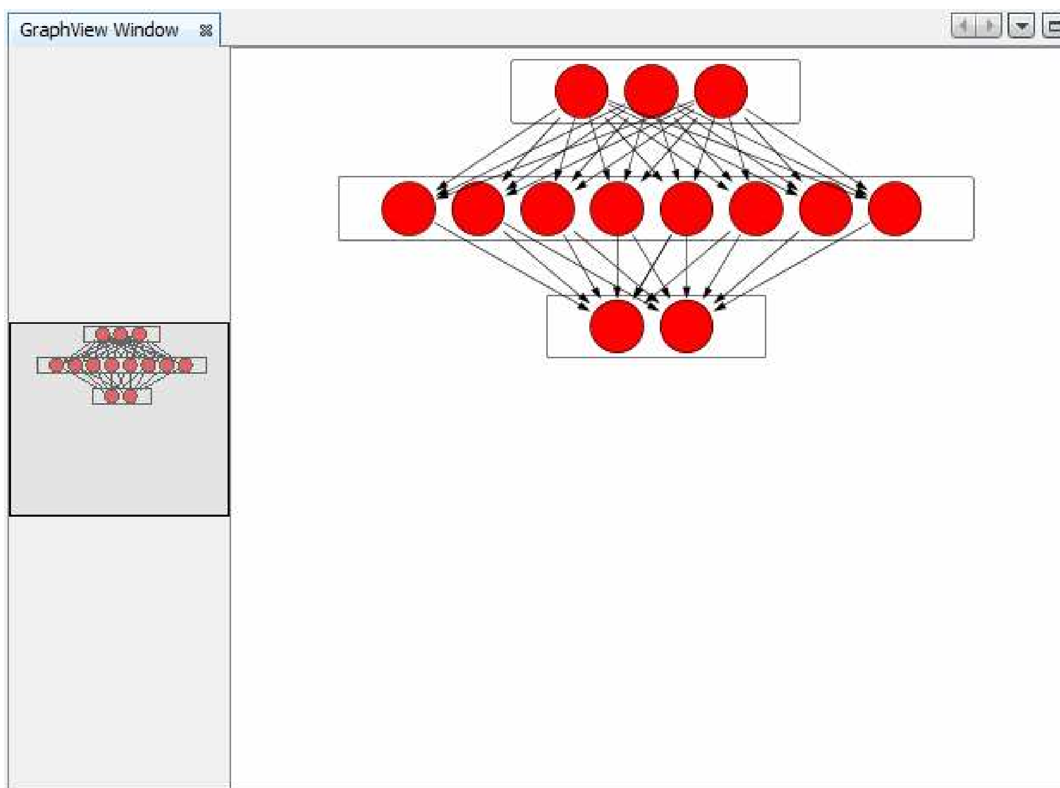
```java
public class NeuronPopupMenuProvider implements PopupMenuProvider {
   JPopupMenu neuronPopupMenu;
   @Override
   public JPopupMenu getPopupMenu(final Widget widget, Point point) {
      neuronPopupMenu = new JPopupMenu();
      JMenuItem removeItem = new JMenuItem("Remove");
      removeItem.addActionListener(new ActionListener() {
      @Override
      public void actionPerformed(ActionEvent e) {
         Widget parent = widget.getParentWidget();
         List<Widget> allConnections =
            NeuralNetworkGraphScene.getConnectionLayer().getChildren();
         List<ConnectionWidget> neuronConnections =
            ((NeuronWidget)widget).getConnections();
         Neuron neuron = ((NeuronWidget) widget).getNeuron();
         Layer layer = neuron.getParentLayer();
         layer.removeNeuron(neuron);
         for (int i = 0; i < neuronConnections.size(); i++) {
            ConnectionWidget cw = neuronConnections.get(i);
            if(allConnections.contains(cw)) {
               NeuralNetworkGraphScene.getConnectionLayer().removeChild(
               cw);
            }
         }
         neuronConnections.clear();
         parent.removeChild(widget);
      }});
   neuronPopupMenu.add(removeItem);
   return neuronPopupMenu;
   }
}
```

U klasi *NeuronPopupMenu kreiran je* provajder padajućeg menija neurona. Svaki *PopupMenuProvider* ima polje koje predstavlja popup meni, i metodu *getPopupMenu()*, koja je zadužena za inicijalizaciju samog menija u kojoj se dodaju stavke menija i akcije koje izvršavaju. *NeuronPopupMenuProvider* omogućava korisnicima da obrišu željeni neuron iz neuronske mreže, a takođe i brisanje svih veza koje je posedovao obrisani neuron.

Na slikama 14 i 15 dat je prikaz neuronske mreže pre i posle pozivanja akcije *Remove* nad *NeuronWidget-om*, rezultat akcije je izbacivanje neurona i svih njegovih veza iz mreže



Slika 14 - Primer popup menija



Slika 15 - Rezultat izvršenja akcije Remove nad NeuronWidget-om

U nastavku je objašnjena klasa *FunctionConnectProvider* koji omogućava kreiranje veza između neurona.

```
public class FunctionConnectProvider implements
ConnectProvider {

   private NeuralNetworkWidget nnet;

   public FunctionConnectProvider(NeuralNetworkWidget nnet) {
      this.nnet = nnet;
   }
   ...
}
```

Na početku se konstruktoru klase prosleđuje objekat klase *NeuralNetworkWidget* koji predstavlja vizuelnu neuronsku mrežu, time je obezbeđena veza između provajdera i neuronske mreže. U nastavku je dat pregled svih metoda ključnih za *ConnectProvider*.

```
public class FunctionConnectProvider implements ConnectProvider {
      …
   @Override
   public boolean isSourceWidget(Widget source) {
      return source instanceof NeuronWidget && source != null ? true :
      false;
   }
   @Override
   public ConnectorState isTargetWidget(Widget src, Widget trg) {
      Neuron srcNeuron = null;
      Neuron trgNeuron = null;
      if (src != trg && src instanceof NeuronWidget && trg instanceof
         NeuronWidget && src.getParentWidget() !=
         trg.getParentWidget()) {
      List<Widget> layers = nnet.getChildren();
      for (int i = 0; i < layers.size(); i++) {
         List<Widget> neurons = layers.get(i).getChildren();
         for (int j = 0; j < neurons.size(); j++) {
            if (neurons.get(j).equals((Object) src)) {
               srcNeuron = ((NeuronWidget)neurons.get(j)).getNeuron();
            }
         }
      }

      for (int i = 0; i < layers.size(); i++) {
         List<Widget> neurons = layers.get(i).getChildren();
         for (int j = 0; j < neurons.size(); j++) {
```

```
            if (neurons.get(j).equals((Object) trg)) {
               trgNeuron = ((NeuronWidget)neurons.get(j)).getNeuron();
            }
         }
      }
      if (srcNeuron != null && trgNeuron != null) {
         ConnectionFactory.createConnection(srcNeuron, trgNeuron);
         connection = new Connection(srcNeuron, trgNeuron);
      }
      return ConnectorState.ACCEPT;
   }
   return ConnectorState.REJECT_AND_STOP;
   }

   …
}
```

Zadatak metode *isSourceWidget()* jeste da utvrdi da li određeni widget može biti polazna tačka veze. Kao što se vidi iz *FunctionConnectProvider* klase *samo NeuronWidget-i* mogu biti polazne tačke konekcije. Metoda *isTargetWidget()* ima zadatak da utvrdi da li određeni *widget* može biti krajnja tačka jedne veze. Iz priloženog koda zaključuje se da krajnje tačke mogu biti samo *NeuronWidget-i* koji nemaju istog roditelja kao i polazni *widget*. Ukoliko neki od ovih uslova nije ispunjen kreiranje konekcije se prekida. Ako su svi uslovi ispunjeni metoda vraća *ConnectorState.ACCEPT* vrednost i počinje izvršavanje *createConnection()* metode. U ovoj metodi kreiraju se *widget-i* veza, i podešava se izgled konekcionih linija, npr. metoda *setTargetAnchorShape(AnchorShape.TRIANGLE_FILLED)* dodeljuje kraju konekcione linije izgled trougla.

```
public class FunctionConnectProvider implements ConnectProvider {
   …
   @Override
   public void createConnection(Widget source, Widget target) {
      NeuronConnectionWidget conn = new
                                 NeuronConnectionWidget(nnet.getScene()
                                 , connection, (NeuronWidget)source,
                                 (NeuronWidget)target);
      conn.setTargetAnchorShape(AnchorShape.TRIANGLE_FILLED);
      conn.setTargetAnchor(AnchorFactory.
                 createRectangularAnchor(target));
      conn.setSourceAnchor(AnchorFactory.
                 createRectangularAnchor(source));
      ((NeuronWidget)source).addConnection(conn);
      ((NeuronWidget)target).addConnection(conn);
      conn.getActions().addAction(ActionFactory.
            createPopupMenuAction(new ConnectionPopupMenuProvider()));
      NeuralNetworkGraphScene.getConnectionLayer().addChild(conn);
      nnet.getScene().validate();}}
```

5.3 Implementacija palete

Jedan od osnovnih ciljeva ovog rada jeste stvaranje okruženja koje će omogućiti kreiranje i editovanje neuronske mreže pomoću vizuelnih alata. Najbolji način za realizaciju ovog cilja je kreiranje palete sa komponentama neuronske mreže koje će moći da se prebacuju na radnu površinu i automatski dodaju u neuronsku mrežu. Dodavanje komponenti na paletu može se vršiti na dva načina, prvi podrazumeva registraciju komponenti palete u XML fajlu, a drugi kreiranje komponenti preko nodova. U ovom projektu smo se fokusirali na drugi način i u nastavku je detaljno objašnjen.

Na početku je potrebno kreirati nod-ove koji predstavljaju komponente palete. Za paletu *„Neuroph“* aplikacije potrebno je definisati dva nod-a, jedan će predstavljati kategorije, a drugi komponente mreže. Za svaki nod je potrebno napraviti JavaBean. Klasa *CategoryBean* sadrži samo parametar *name* koji predstavlja naziv kategorije komponente.

```
public class CategoryBean {
   private String name;
   public Category() {
   }
   public String getName() {
      return name;
   }
   public void setName(String name) {
      this.name = name;
   }
}
```

Zatim je potrebno kreirati klasu u kojoj su definisane sve kategorije, s obzirom da su kategorije ustvari biti „deca“ palete naziv kreirane klase je *CategoryChildren*. U ovoj klasi će biti definisane sve kategorije komponenti koje će se nalaziti na paleti, za sada je to samo kategorija *Components*.

```
public class CategoryChildren extends Children.Keys {
  private String[] categories = new String[]{
  "Components"};
  public CategoryChildren() {
  }
  @Override
  protected Node[] createNodes(Object key) {
     Category obj = (Category) key;
     return new Node[] { new CategoryNode(obj) };
  }
}
```

I na kraju je potrebno kreirati sam nod. Kreira se klasa CategoryNode, čiji konstruktor prima parametar tipa *Category*, i u zavisnosti od ovog parametra kreira odgovarajući nod.

```
public class CategoryNode extends AbstractNode {
   public CategoryNode( Category category ) {
      super( new NetworkComponentChildren(category),
             Lookups.singleton(category));
      setDisplayName(category.getName());
   }
}
```

Struktura komponenti palete opisana je preko klase *NetworkComponentBean.*

```
public class NetworkComponentBean {
   Integer id;
   String icon;
   String title;
   String category;
   public NetworkComponent() { }
   public String getCategory() { return category; }
   public void setCategory(String category){this.category = category;}
   public String getIcon() { return icon; }
   public void setIcon(String icon) { this.icon = icon; }
   public void setId(Integer id) { this.id = id; }
   public void setTitle(String title) { this.title = title; }
   public Integer getId() { return id; }
   public String getTitle() { return title; }
}
```

Sagledavanjem klase *NetworkComponentBean* dolazi se do zaključka da svaka komponenta na paleti poseduje identifikacioni broj, naziv, ikonu i kategoriju. Kako bi se komponente postavile na paletu potrebno je kreirati njihove nod-ove koji će se nalaziti pod određenom kategorijom, i čiji će izgled biti određen ikonom i nazivom.

```
public class NetworkComponentChildren extends Index.ArrayChildren {

   private Category category;

   private String[][] items = new String[][]{
     {"0", "Components",
"org/neuroph/netbeans/visual/support/circle.png", "neuron"},
     {"1", "Components",
"org/neuroph/netbeans/visual/support/rectangle.png", "layer"},
   };
   public NetworkComponentChildren(Category Category) {
      this.category = Category;
   }
   @Override
   protected java.util.List<Node> initCollection() {
      ArrayList childrenNodes = new ArrayList( items.length );
         for( int i=0; i<items.length; i++ ) {
            if( category.getName().equals( items[i][1] ) ) {
               NetworkComponent item = new NetworkComponent();
               item.setId(new Integer(items[i][0]));
```

```
                item.setCategory(items[i][1]);
                item.setIcon(items[i][2]);
                item.setTitle(items[i][3]);
                childrenNodes.add( new NetworkComponentNode( item ) );
            }
        }
        return childrenNodes;
    }
}
```

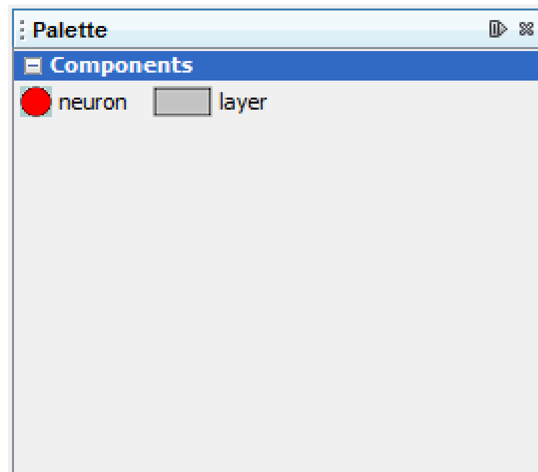
Komponente palete inicijalizovane su u klasi *NetworkComponentChildren*. Polje *items* predstavlja niz svih komponenti koje će se nalaziti na paleti, neuron i lejer. Metoda *initCollection* ima zadatak da za svaku kategoriju kreira odgovarajuće komponente. Klasa *NetworkComponentNode* predstavlja nod komponenti. U konstruktoru se definiše izgled samog nod-a. Metoda *setIconBaseWithExtension()* postavlja ikonu, a *setDisplayName()* naslov nod-a.

```
public class NetworkComponentNode extends AbstractNode {
    private NetworkComponent netItem;
    public NetworkComponentNode(NetworkComponent key) {
        super(Children.LEAF, Lookups.fixed( new Object[] {key} )
        );
        this.netItem = key;
        setIconBaseWithExtension(key.getIcon());
        setDisplayName(key.getTitle());
    }
    public NetworkComponent getNetItem() { return netItem; }
}
```

Kada su kreirani svi potrebni elementi potrebno je kreirati paletu, za to je zadužena klasa *PaletteSupport*.

```
public class PaletteSupport {
    public static PaletteController createPalette() {
        AbstractNode paletteRoot = new AbstractNode(new
        CategoryChildren());
        paletteRoot.setName("Palette Root");
        return PaletteFactory.createPalette(paletteRoot, new
                                           MyActions(), null, null);
    }
    …
}
```

Kreiranje palete se vrši u metodi *createPalette(),* a za kreiranje je odgovorna klasa *PaletteFactory.* Posle kreiranja palette ona izgleda kao na slici 16, sa slike se može primetiti da koren palete *Components* predstavlja nod kategorije, a elementi neuron i layer nodove komponenti neuronske mreže.



Slika 16 - Paleta Neuroph okruženja

Kako bi bilo moguće prevlačenje komponenti sa palete na scenu potrebno joj je prilikom kreiranja dodeliti *DragAndDropHandler.* Klasa *NCDnDHandler* nasleđuje klasu *DragAndDropHandler*, i ona je zadužena za kreiranje *DragAndDrop* akcije. Kako bi se komponentama na paleti dodala *DragAndDrop* akcija, konstruktoru palete je potrebno proslediti instancu klase *DragAndDropHandler*. Uloga *DragAndDropHandler-a* jeste da preko transfer objekta uspostavi vezu između scene i palete.

```
public class PaletteSupport {
   public static PaletteController createPalette() {
      AbstractNode paletteRoot = new AbstractNode(new
      CategoryChildren());
      paletteRoot.setName("Palette Root");
      return PaletteFactory.createPalette(paletteRoot, new
                                          MyActions(), null, new
                                          NCDnDHandler());
   }
   …
}

private static class NCDnDHandler extends DragAndDropHandler {
   @Override
   public void customize(ExTransferable exTransferable, Lookup
   lookup) {
      Node node = lookup.lookup(Node.class);
      final Image image = (Image)
   node.getIcon(BeanInfo.ICON_COLOR_16x16);
      exTransferable.put(new ExTransferable.Single
   (DataFlavor.imageFlavor) {
         @Override
         protected Object getData() throws IOException,
      UnsupportedFlavorException {
            return image;
         }
      });
```

```
        final String title = node.getDisplayName();
        exTransferable.put(new
        ExTransferable.Single(DataFlavor.stringFlavor) {
           @Override
           protected Object getData() throws IOException,
           UnsupportedFlavorException {
              return title;
           }
        });
     }
  }
```

Klasa *NCDnDHandler* omogućava kreiranje dve vrste transfernih objekata. Prvi je tekstualni transferni objekat, čija je uloga u određivanju komponente koja se prevlači, a drugi predstavlja ikonu komponente i ima samo estetsku ulogu. Metode preko kojih scena pristupa transfernom objektu su *getImageFromTransferable()* i *getStringFromTransferable().*

```
private Image getImageFromTransferable(Transferable
transferable) {
   Object o = null;
   try {
      o =
   transferable.getTransferData(DataFlavor.imageFlavor);
   } catch (IOException ex) {
      …
   } catch (UnsupportedFlavorException ex) {
      …
   }
   return (Image) o;
}
```

Metoda *getImageFromTransferable()* pomoću metode *getTransferData()* kupi podatke iz transfer objekta, a koji će podatak pokupiti određuje klasa *DataFlavor.* Postoji nekoliko već definisanih instanci ove klase kao što su *DataFlavor.imageFlavor, DataFlavor.stringFlavor i DataFlavor.javaFileListFlavor.* U ovom slučaju kako bi se pokupila sliku sa transfernog objekta koristi se *DataFlavor.imageFlavour*.

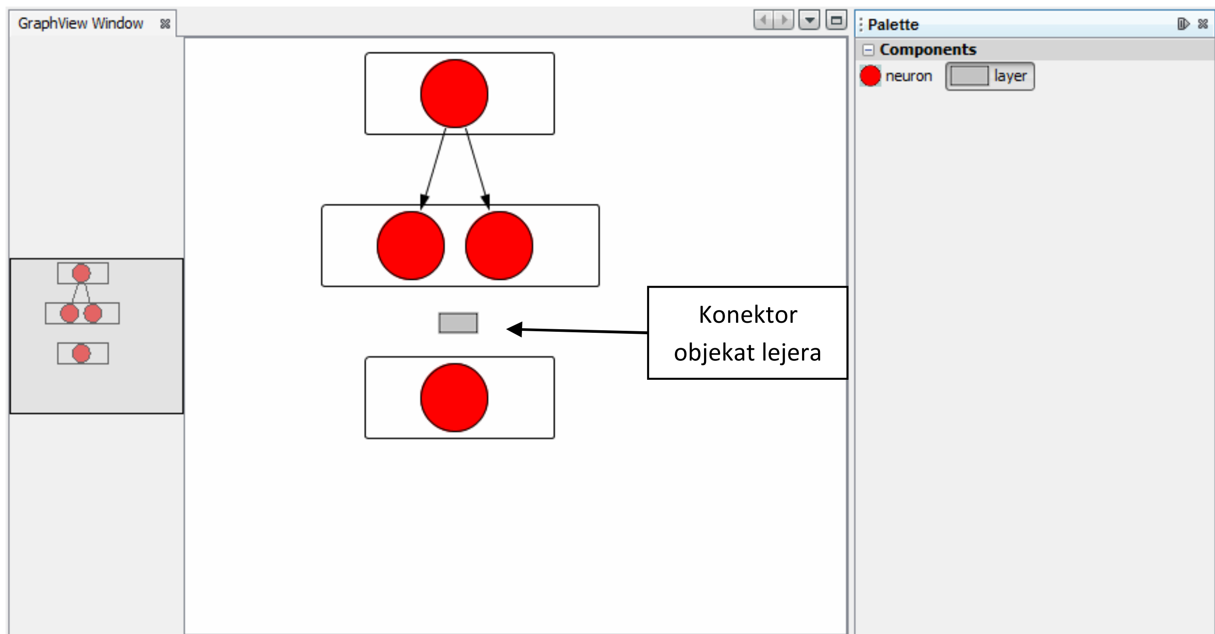
```
private String getStringFromTransferable(Transferable transferable) {

   Object o = null;
   try {
      o = transferable.getTransferData(DataFlavor.stringFlavor);
   } catch (IOException ex) {
   …
   } catch (UnsupportedFlavorException ex) {
   …
   }
   return (String) o;
}
```

Jedina rezlika između metoda *getImageFromTransferable()* i *getStringFromTransferable()* jeste u tipu podatka koji kupe sa transfenog objekta. Dakle umesto *DataFlavor.imageFlavor,* koristi se *DataFlavor.stringFlavor* kako bi se pokupio tekst koji se nalazi na transfernom objektu.

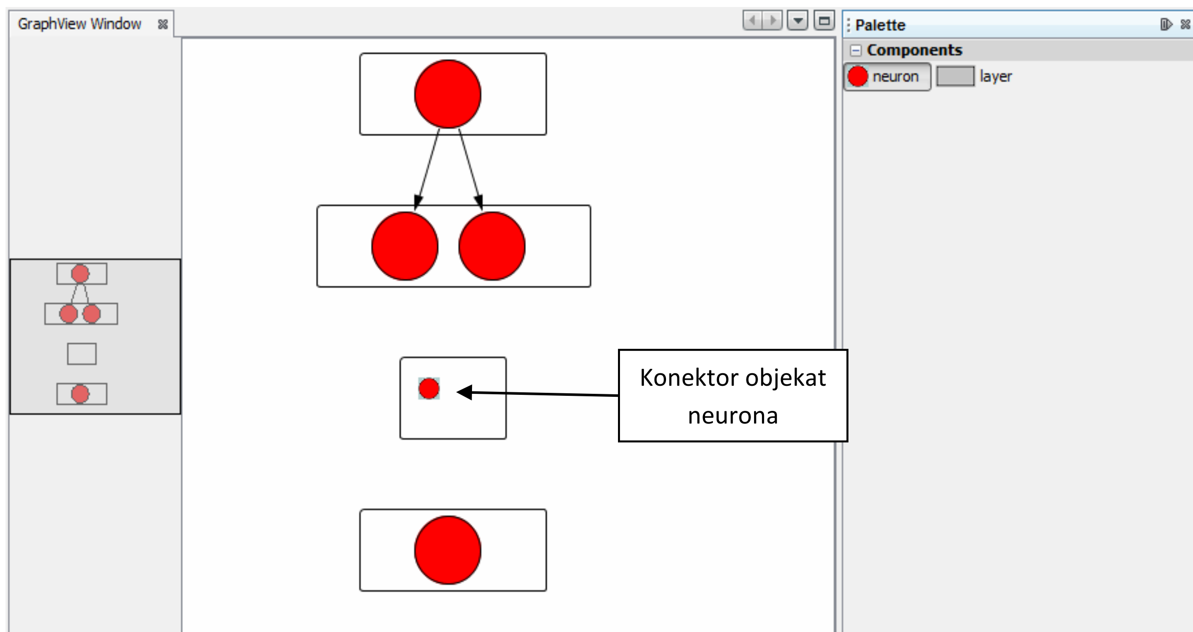
Transferni objekat predstavlja vezu izmeću palete i scene. Kako bi scena mogla da prihvata spoljne objekte u konstruktoru scene potrebno je dodati *AcceptAction,* koju kreira *ActionFactory*. Za ovu akciju je potrebno kreirati provajder *AcceptProvider*, u kome će se definisati način na koji će scena obraditi dolazne objekte.

```
   @Override
   public ConnectorState isAcceptable(Widget widget, Point point,
   Transferable transferable) {
      Image dragImage = getImageFromTransferable(transferable);
      JComponent view = getView();
      Graphics2D g2 = (Graphics2D) view.getGraphics();
      Rectangle visRect = view.getVisibleRect();
      view.paintImmediately(visRect.x, visRect.y, visRect.width,
      visRect.height);
      g2.drawImage(dragImage,
      AffineTransform.getTranslateInstance(point.getLocation().getX(),
            point.getLocation().getY()), null);
      return ConnectorState.ACCEPT;
   }
```

Prva metoda u provajderu jeste *isAcceptable()* metoda. Ova metoda vraća *ConnectorState objekat*, odnosno utvrđuje da li objekat koji se prosleđuje može da se doda na scenu. U ovom slučaju svi objekti sa palete će moći da se dodaju na scenu, pa metoda ima ulogu samo za iscrtavanje ikone komponente koja se prevlači pre samog dodavanja na scenu. A ikonu uzima od prethodno opisanog tranfernog objekta pozivanjem metode *getImageFromTransferable().*



Slika 17 – Primer konektor objekta lejera



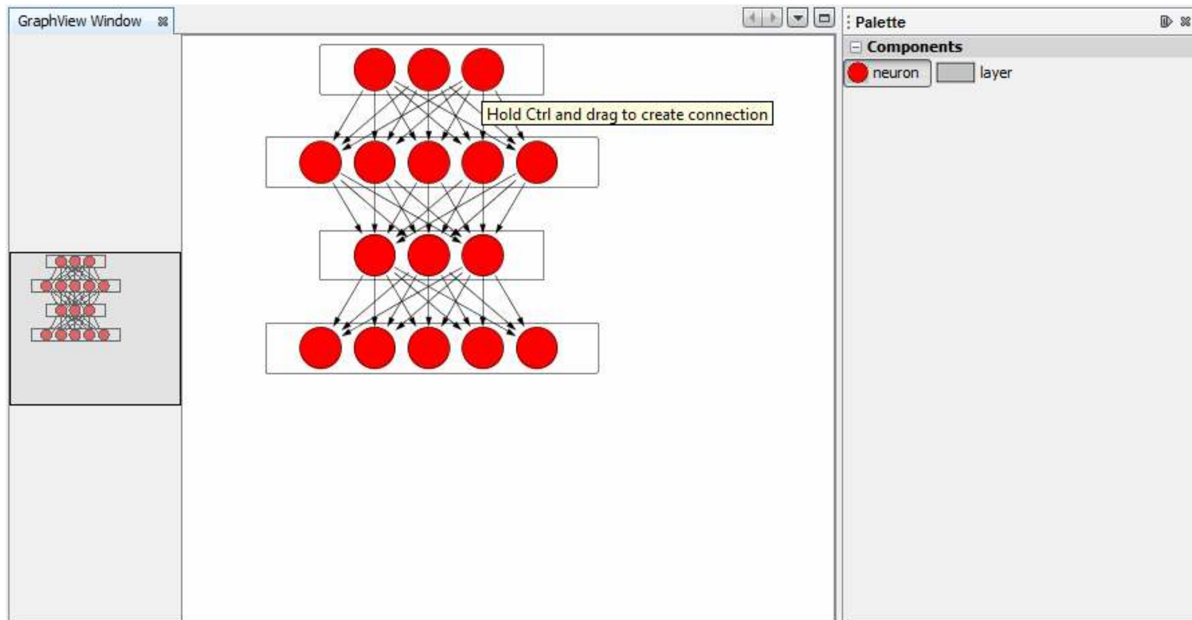
Slika 18 - Primer konektor objekta neurona

Na slikama 17 i 18 dati su primeri konektora objekata za lejer i neuron kojima su dodeljene ikone komponenti na koje se odnose. Konektor objekat se kreira kada se komponenta sa palete prevuče na scenu i postoji sve dok se komponenta ne doda na scenu.

Poslednji korak u povezivanju scene i palete jeste implementacija metode *accept().*

```
    @Override
    public void accept(Widget widget, Point point, Transferable
transferable) {
        Widget w = null;
        if
(getStringFromTransferable(transferable).equalsIgnoreCase("neuron")) {
            Neuron n = new Neuron();
            w = createNeuron(n);
            for (int i = nnw.getChildren().size() - 1; i >= 0; i--
) {
                Point upl =
nnw.getChildren().get(i).getLocation();
                upl.setLocation(upl.getX() + 100, upl.getY() +
10);
                Dimension d =
nnw.getChildren().get(i).getPreferredSize();
                Rectangle r = new Rectangle(upl, d);
                if (r.contains(point)) {
                    resizeLayer(nnw.getChildren().get(i));
                    ((NeuralLayerWidget)
nnw.getChildren().get(i))
                            .addNeuron((NeuronWidget) w);
                }
            }
        } else {

            Layer layer = new Layer();
            w = createLayer(layer);
            int pos = 0;
            for (int i = 0; i < nnw.getChildren().size(); i++) {
                double location =
nnw.getChildren().get(i).getLocation().getY();
                if (point.getY() < location) {
                    pos = i;
                    break;
                } else {
                    pos = nnw.getChildren().size();
                }
            }
            nnw.addLayer(pos, (NeuralLayerWidget) w);

        }

    //w.setPreferredLocation(widget.convertLocalToScene(point));
        nnw.getScene().repaint();
        nnw.getScene().validate();
    }
```

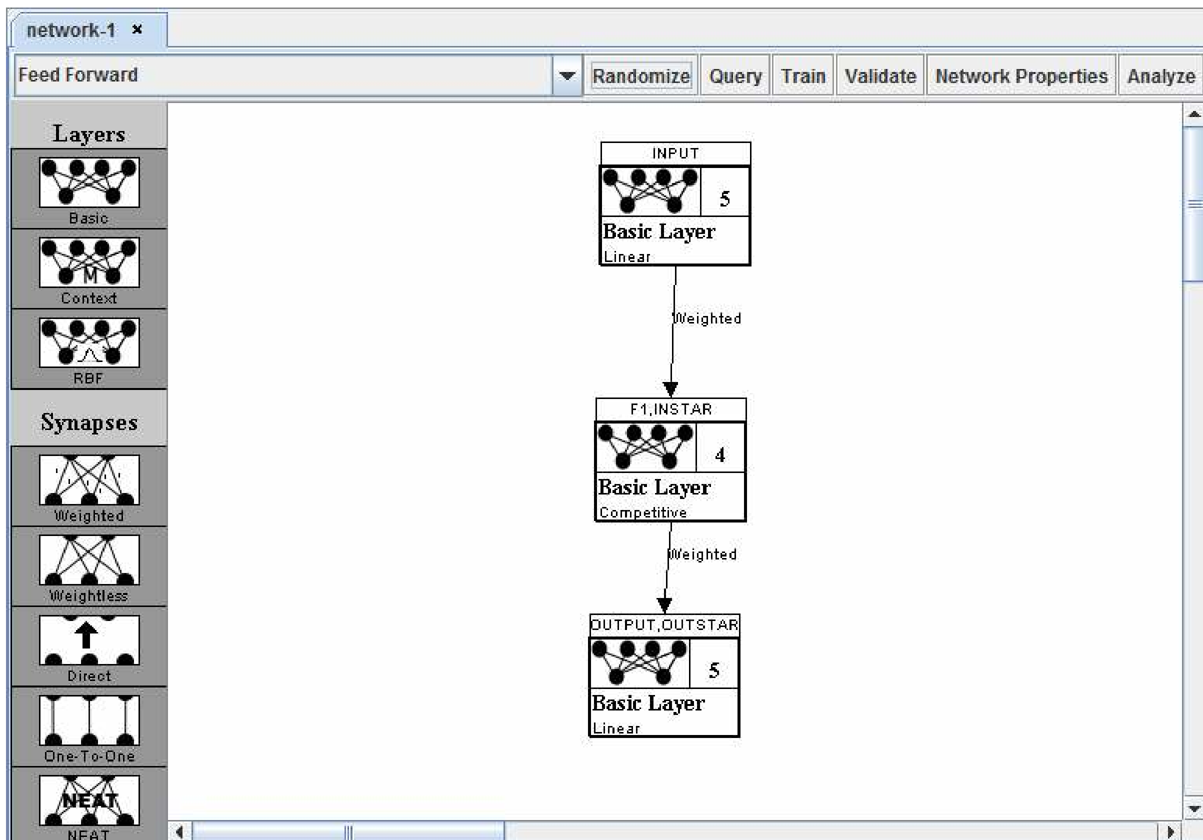
Metoda *accept()* ima zadatak da obradi objekte koji pristižu na scenu. Objekti koji se mogu dodati su neuron i lejer. Pozivanjem metode *getStringFromTransferable()* određuje se koji je objekat u pitanju. U slučaju dodavanja neurona prvo se kreira novi *NeuronWidget* pozivanjem metode *createNeuron()*, zatim se određuje roditelj, odnosno lejer na koji se dodaje kreirani neuron, i na kraju se *NeuronWidget* dodaje na *NeuralLayerWidget*, a samim tim i neuron na neuronski lejer. U drugom slučaju dodaje se lejer, za koji se na početku kreira odgovarajući *widget*, zatim određujemo poziciju u odnosu na ostale lejere, i na kraju se dodaje u mrežu. Zbog osvežavanja scene posle svake promene potrebno je pozvati metodu *Scene.validate().* Metoda *Widget.setPreferredLocation()* ima zadatak da odredi poziciju *widget-a* na sceni. Kako bi se objekat kreirao na mestu na kome je ispušten konektor objekat poziva se metoda *Widget.setPreferredLocation()* sa parametrom *Widget.convertLocalToScene(point);*



Slika 19 - Konačni izgled okruženja

6. Evaluacija

U ovom poglavlju izvršena je analiza razvijenog alata, u odnosu na *Encog WorkBench*. *Encog WorkBench* predstavlja aplikaciju za rad sa neuronskim mrežama i zasnovana je na *Encog framework-u*.



Slika 20 - Izgled Encog visual editora

Na slici 20 prikazan je vizuelni editor *Encog WorkBench-a*. Kao što vidimo paleta ovog alata sadrži različite tipove neuronskih lejera i veza između njih, osnovni problem ovog editora jeste to što ne dozvoljava direktno podešavanje parametara neurona i veza. Svaki lejer sadrži niz neurona, čiji se parametri mogu podešavati prilikom podešavanja lejera, što ne predstavlja veliki problem. Podešavanja veza između neurona vrši se u tabeli svih veza između dva lejera, veliki problem ovog pristupa jeste pronalaženje veze čije je parametre potrebno promeniti. Za razliku od Encog vizuelnog editora, alat koji je razvijen za „Neuroph" omogućava korisniku da jednostavnim klikom na neuron ili vezu podesi odgovarajuće parametre.

Druga bitna razlika jeste radna površina. Kod *Encog-a* veličina radne površine je fiksna, samim tim je i veličina neuronske mreže ograničena tom površinom, takođe ukoliko vizuelna reprezentacija mreže ne može da stane na ekran nije moguće sagledati izgled kompletne mreže, jer ne postoji funkcija zumiranja niti pogled na radnu površinu. Ovim alatom otklonjeni su svi pomenuti nedostaci.

NetBeans platforma i veliki broj *NetBeans API-a* omogućili su lako otklanjanje prethodno pomenutih nedostataka. Za razvoj ovog vizuelnog alata korišćeni su *Nodes API, CommonPalette, Windows System API* i najvažniji *VisualLibrary API. Nodes API i CommonPalette* sadrže veliki broj rešenja koji omogućavaju kreiranje palete komponenti, *Windows System API* je zadužen za organizaciju prozora, dok je *VisualLibrary API* zadužen za vizuelnu reprezentaciju neuronske mreže. Razvoj jednog ovakvog alata bez podrške NetBeans platforme bio bi vrlo spor, i krajnji rezultat verovatno ne bi ispunio sve potrebne zahteve. Ovi moduli NetBeans platforme jedinstveni su, predstavljaju skup rešenja koje veliki broj programera iz celog sveta i danas razvija, i za sada ne postoji alternativno rešenje, i samim tim čine NetBeans platformu najpogodnijim rešenjem za razvoj kako modernih vizuelnih alata, tako i velikih modularnih aplikacija.

NetBeans platforma predstavlja alat za kreiranje modularnih aplikacija. U današnje vreme najpopularnije platforme za razvoj ovakvih aplikacija jesu NetBeans platforma i Eclipse RCP.

Osnovne prednosti NetBeans platforme:

1) sistem modula koji je lako proširiv, i vrlo lak za kreiranje;
2) veliki broj API-a koji obuhvataju najčešće korišćene funkcije;
3) predefinisani korisnički interfejs koji se lako nadograđuje;
4) veliki broj uputstava i odlična korisnička podrška.

7. Zaključak

U ovom poglavlju dat je kratak pregled rezultata ovog rada, mogućnosti primene razvijenog rešenja i navedeni su pravci daljeg razvoja.

Osnovni Doprinos

Cilj ovog rada bio je kreiranje vizuelnog alata koji će omogućiti vizuelnu obradu neuronskih mreža. Ovaj cilj je u potpunosti ispunjen i u nastavku je dat kratak opis razvijenog alata.
Kao krajnji rezultat dobijeno je vizuelno okruženje za razvoj neuronskih mreža, sa korisničkim interfejsom koji je znatno napredniji u odnosu na postojeća rešenja. Kreiran je sistem koji korisnicima pruža fleksibilno vizuelno editovanje neuronske mreže. U osnovi ovaj alat omogućava kreiranje nove neuronske mreže, dodavanje delova mreže kao što su neuroni i neuronski lejeri korišćenjem palete i njihovo brisanje, kreiranje veza, podešavanje parametara i nadograđivanje postojeće neuronske mreže.
U 5. Poglavlju je korak po korak opisana detaljna implementacija vizuelnog alata na primeru *„Neuroph“* aplikacije, ovo poglavlje može služiti kao uputstvo za sve one koji žele bliže da se upoznaju sa mogućnostima koje pruža NetBeans platforma u oblasti kreiranja vizuelnih alata.

Mogućnosti primene

Osnovna primena razvijenog alata predstavlja kreiranje i izmena različitih neuronskih mreža, koje se mogu koristiti u mnogim oblastima, među kojima su prepoznavanje rukopisa i zvuka, predviđanje kretanja vrednosti akcija na berzi, prepoznavanje lica, i otisaka prstiju i druge.

Pravci daljeg razvoja

Postoje brojne mogućnosti za unapređenje razvijenog rešenja od kojih su najinteresantnije:

- Kreiranje predefiisanih šablona za vizuelni prikaz neuronske mreže. Danas postoji veliki broj različitih neuronskih mreža, sa različitom strukturom. Pa je potrebno kreirati vizuelne templejte koji će olakšati kreiranje različitih vrsta neuronskih mreža od nule.

- Razvoj sistema za trodimenzionalni prikaz neuronske mreže. Trodimenzionalna vizuelizacija neuronske mreže bi pomogla korisnicima da lakše shvate strukturu neuronske mreže i kako sama mreža funkcioniše.

Literatura

[1] Heiko Böck, „The Definitive Guide to NetBeans™ Platform“,Apress, 2009.

[2] Tim Boudreau, Jaroslav Tulach, Geertjan Wielenga, „Rich Client Programming Plugging into the NetBeans Platform“, Prentice hall, 2007.

[3] Jürgen Petri, „NetBeans Platform 6.9 Developer's Guide“, Packt Publishing, 2010.

[4] Adam Myatt, Brian Leonard, Geertjan Wielenga, „Pro NetBeans™ IDE 6

Rich Client Platform Edition“, Apress, 2008.

[5] Nodes API Javadoc, http://bits.netbeans.org/dev/javadoc/org-openide-nodes/org/openide/nodes/doc-files/api.html

[6] VisualLibrary API Javadoc, http://bits.netbeans.org/dev/javadoc/org-netbeans-api-visual/org/netbeans/api/visual/widget/doc-files/documentation.html

[7] CommonPalette API Javadoc, http://bits.netbeans.org/dev/javadoc/org-netbeans-spi-palette/overview-summary.html