Knowledge Network Visualization

SS 2007

Avaré Stewart

Dr. Claudia Niederée

Fraunhofer IPSI

Prof. Dr. Techn. Johannes Fürnkranz

TU Darmstadt, Department of Computer Science

Student Research Project

by

Dimitar Nikolov

31. March 2007

ABSTRACT

The European project VIKEF (Virtual Information and Knowledge Environment Framework, see www.vikef.net) targets the effective acquisition, organization, processing, sharing, and use of knowledge implicitly available in scientific and business documents. A major goal of the project is designing and developing a set of advanced semantic-enabled community services, i.e. services that reuse semi-automatically extracted semantic information to provide more effective and intelligent services to members of communities.

One major VIKEF service class is *Semantic Navigation* support. It focuses on enabling the seamless transition between the semantic and the content layer. In doing so, the semantics are exploited for an improved, task-specific user experience, for fostering a better understanding of the domain and for improving the presentation and content digestion. Semantic Navigation relies on Semantic Resource Networks (SRNs) and the definition of SRN Views. The so-called SRNs are knowledge networks consisting of an RDF representation of domain knowledge together with links to the underlying annotated content. SRN Views are task-specific and partly enriched subsets of the SRN. To make the SRN Views useful they have to also be visualized in a flexible, task-specific manner.

For the display of such views a special Visualization Application will be built. It is the task of this student research project to design and implement a component for this purpose – the VIKEF Visualization Application. Its design and the development will be described in the written part of the project.

The Visualization Application offers a user friendly interface that enables the user to browse through and reshape the displayed graph used for the visualization of an SRN View. The application provides a working environment in a java applet that offers the user different means for interaction with the View. Among the features that can be exploited and adapted for interaction with the visualization are: overview; drag & drop, search, filtering on topics, etc. The Visualization Application uses the targeted standard-based visual counterpart of the SRN View created by another VIKEF Component – the Semantic Visualization Factory – as input for the creation and visualization of the graph.

The aforementioned component works in collaboration with a set of components: an SRN View Definition Editor, an SRN View Factory (Toshev, 2007), a Semantic Visualization Editor and a Semantic Visualization Factory (Nikolov, 2007) to enable the next generation of semantic enables community services.

Table of Contents

1 Introduction

VIKEF (Virtual Information and Knowledge Environment Framework, see www.vikef.net) was started three years ago. During this time the project has evolved from a bright idea through a concrete specification, gradual and at times rapid changes in the details of this specification and development of further elements to reach its current advanced and mostly complete and successful state. Also refer to (VIKEF 1, 2007) and (VIKEF 2, 2007).

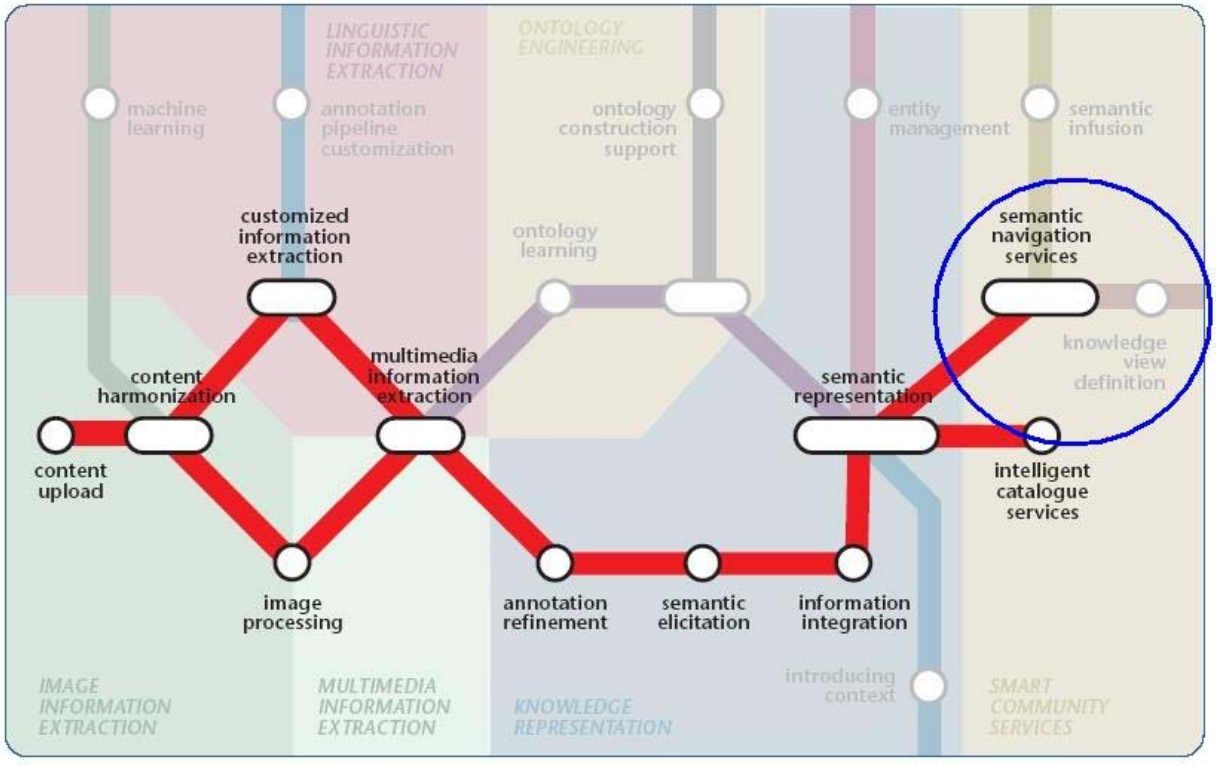

Figure 1: VIKEF Pipeline Overview

1.1 Motivation: VIKEF Semantic Enabled Community Services

Figure 1 shows an overview of the phases in the VIKEF project: from image information extraction to the smart community services. A major goal of the project is the design and development of a set of ad-

vanced semantic-enabled community services i.e., services that reuse semi-automatically extracted semantic information to provide more effective and intelligent services to members of communities.

One major VIKEF service class is *Semantic Navigation* support services. Semantic Navigation relies on Semantic Resource Networks (SRNs) and the definition of SRN Views. The so-called SRNs are knowledge networks consisting of an RDF representation of domain knowledge together with links to the underlying annotated content.

SRN Views are constructed in a three-stage Knowledge View Definition Process (Figure 2). In this process, what a View consists of (Knowledge View Definition) – is logically separated from how the view will look (Knowledge View Creation) and how this view will be rendered to the user (Knowledge View Visualization). The focus of this work is on the Knowledge View Visualization phase.

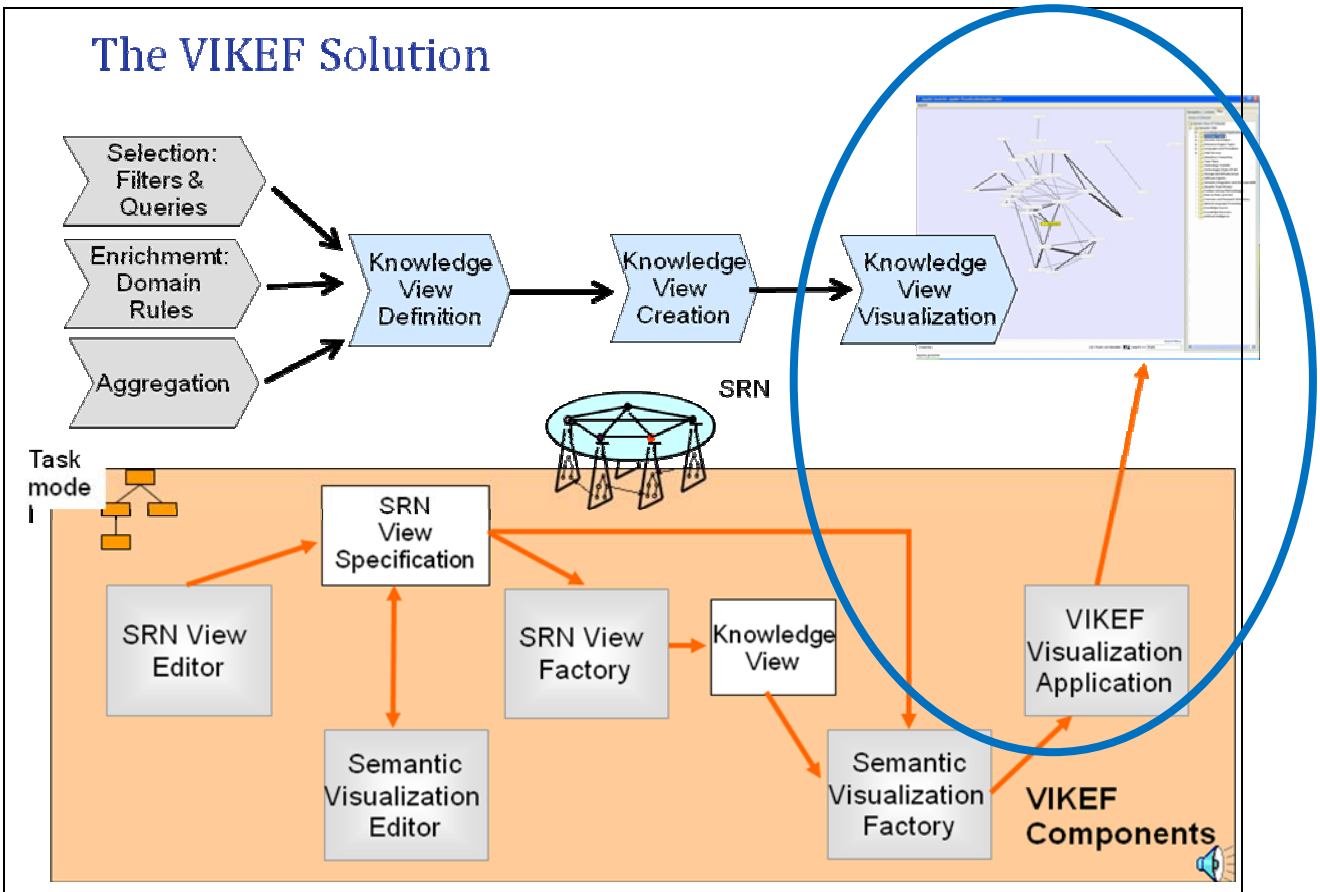

Figure 2: VIKEF Knowledge View Definition Process

1.2 Problem – Visualization

In VIKEF the user should be supported in the use of such Knowledge Views. This should be achieved by providing a user-friendly tool for their visualization.

In the current work the following issues are addressed:

1. Providing an application that can display a View. VIKEF, being a Web based Framework, can only provide web applications and applets to its users. Therefore such an application can only be built using an applet.

2. Providing means to the user for interaction with the display. Targeting functionality and user-friendliness, the VIKEF concept requires from each of its components to assist the user as much as possible and provide him with rich functionality to satisfy his needs. Therefore it is necessary to not just show the user the View he has created but to also allow him to interact and play with it after it has been displayed.

3. Providing widgets that assist the user in exploring and studying the data. Being nice and user-friendly is not enough. The application should be useful. It is important to provide the user with the information he wants to have in an easy to access and use format.

In order to address these issues and the user-friendly display a special applet will be built. It is the task of this project work to design and implement a component for this purpose – the VIKEF Visualization Application. Its design and development will be described in the written part of the project work.

The Visualization Application offers a user friendly interface that enables the user to easily interact with and explore the visualization of an SRN View. It provides a working environment in an applet that offers the user numerous widgets that assist him in his work. Among the features that can be exploited are: an overview, additional data display, basic search capability, filtering according to appropriate topics, different layouts for the display, etc.

1.3 Structure of this Work

This paper describes the VIKEF component that displays the Visualization.

Section 2 describes the process and purpose of the View Creation and Visualization.

Section 3 describes the Visualization Application. It presents the design decisions taken while creating it and lists the functionality this component offers its user. It also presents the format the input data is stored in.

Section 4 describes the implementation process of the Visualization Application: the technologies employed; a review of the different releases; the architecture of the component and the way the data is parsed. It also describes the data format used and the specific challenges overcome during the development.

Section 5 attempts to evaluate the last (third) release of the software according to two criteria: software design principles and goals satisfaction and usability.

Section 6 discusses related work.

Section 7 sums up the document and the recommendations for further development.

Section 8 list the materials used while working on this project.

Section 9 (Appendix) describes the technical aspects of the development – a description of the component's files.

2 VIKEF Knowledge View Visualization

2.1 Background: Supporting Semantic Enabled Community Services

Various technologies can and should be employed in collaboration to create Knowledge Views. If the knowledge base is, for example, represented by RDF statements, then SPARQL queries and Jena (Jen07) rules can be combined to create Knowledge Views. In addition, counting functions encoded in Java or another programming language might be necessary to realize knowledge aggregation options. The result of such a View definition can be encapsulated as a Web service, visualized and made available. Thus, a variety of different complex technologies have to be mastered for creating knowledge views. The goal of VIKEF is to find a more user-friendly solution for creating Knowledge Views. Also refer to (Publications, 2006) for further information on VIKEF.

As mentioned in Section 1.1 the definition and display of Knowledge Views is be a three step process, consisting of View definition, View Creation and View Visualization. For this process, VIKEF technology offers convenient tools, called Editors, which ease the creation of Knowledge Views (see Figure 2). This enables persons other than knowledge experts or IT-experts: such as conference and trade fair organizers, to tailor the Knowledge Views exactly to the needs of their community members. In VIKEF two such Editors are provided for this purpose.

2.2 View Creation – From Definition to Visualization

The SRN View Editor (see (Toshev, 2007)) is responsible for defining the logic and structure of the Knowledge View or SRN View as it is called in VIKEF. It supports the step of Knowledge View definition.

The Semantic Visualization Editor is used to determine the „look and feel" of how the Knowledge View is displayed to the user. It is used to define Visual Features for the Knowledge View display such as color and size of nodes or, the layout of the Knowledge View information as a graph display and as widgets. For further information concerning the Semantic Visualization Components refer to (Nikolov, 2007).

In the final stage, Knowledge View Visualization, the created format is processed by the VIKEF Visualization Application and displayed to the user. The Visualization Application enables the user to interact with

and study the Knowledge View. This includes searching, browsing, and inspecting detail information. This Application is the component described in this paper.

3 Visualization Application Design

3.1 Overview of the Design Process

The work on the Visualization Application began shortly after the other four collaborating components (SRN View Components and Semantic Visualization Components) were started. The main goal of the component was defined from the start – build a prefuse (prefuse, 2007) application that is capable of displaying the graph file produced by the Semantic Visualization Factory.

3.1.1 Technology

The component is built using the prefuse visualization toolkit beta. This toolkit is based on the Java programming language using applets for the creation of a GUI. To fulfill its task the component was extended with new functionality a few times, which required employing further technologies. For a discussion on these refer to Section Implementation Issues4.

3.1.2 Window Design

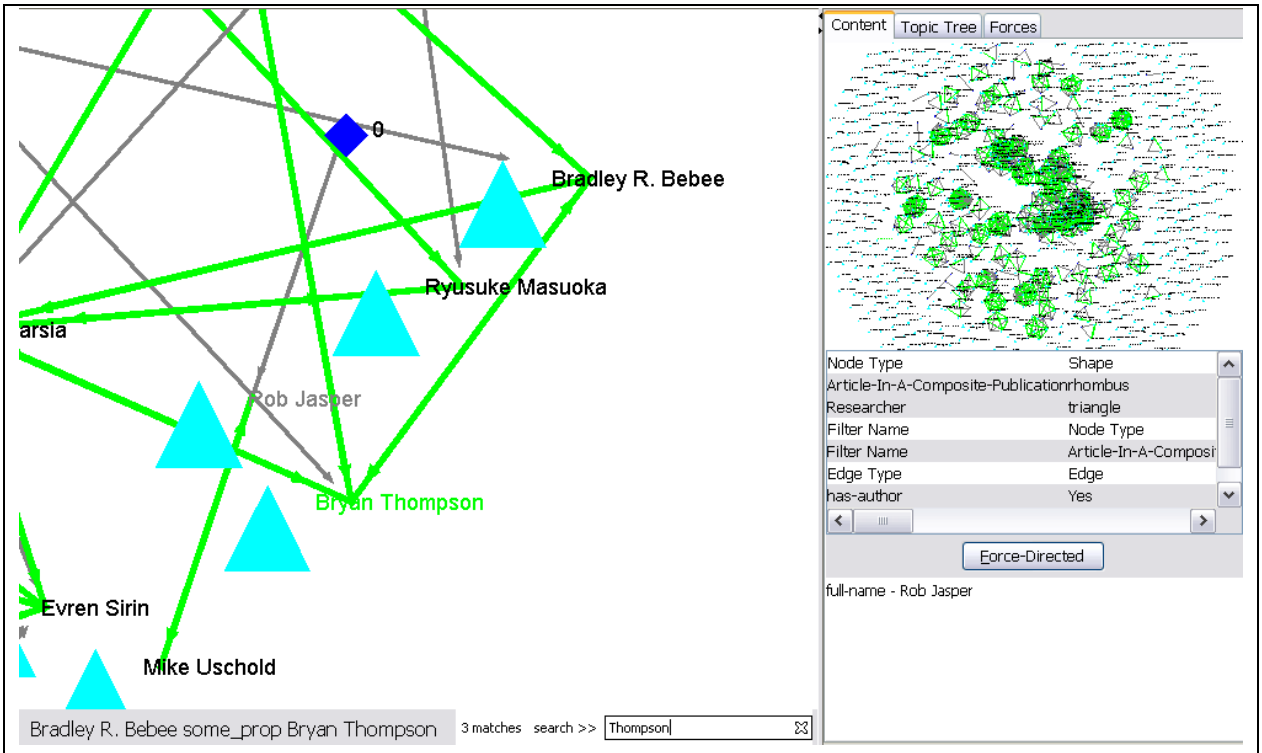

Figure 3: Visualization Application – Display Mode 3, Content Pane Tab

The design of the window was mostly predefined in the requirements. Two graph display windows were required – a big one to display the graph applying to it the settings defined in the Semantic Visualization Editor and a small one to display the whole graph without necessary paying attention to these settings. However, one major change to the way the component was designed was introduced after its first release – its appearance was supposed to be party configurable by the Semantic Visualization Editor.

3.2 Specification of the Visualization Application

This section describes the Visualization Application in detail. It reviews each widget and its responsibilities. There are two different tasks that the Application completes:

1. Display the View to the user and allow the user to study the relationships between the different nodes.

2. Allow easy to use filtering of the graph nodes on topics.

The component is built using a JSplitPane (Java, 2004) consisting of the pane containing the Main Window and two other widgets and a tabbed pane with three tabs and six widgets, resulting in nine widgets

altogether. Eight of these widgets contribute to the first task and only one (Topics Tree - Figure 12) to the second task. Six of the widgets are interactive. The rest are static and are used to display some information to the user. Each of these widgets is described in detail in Sections 3.2.3 - 3.2.11.

3.2.1 Semantic Visualization Editor

This part lists the elements of the Semantic Visualization Editor that directly correlate to the Visualization Application. The settings displayed in the figures in this section are the same as the settings visualized in the Main Window of Figure 3.

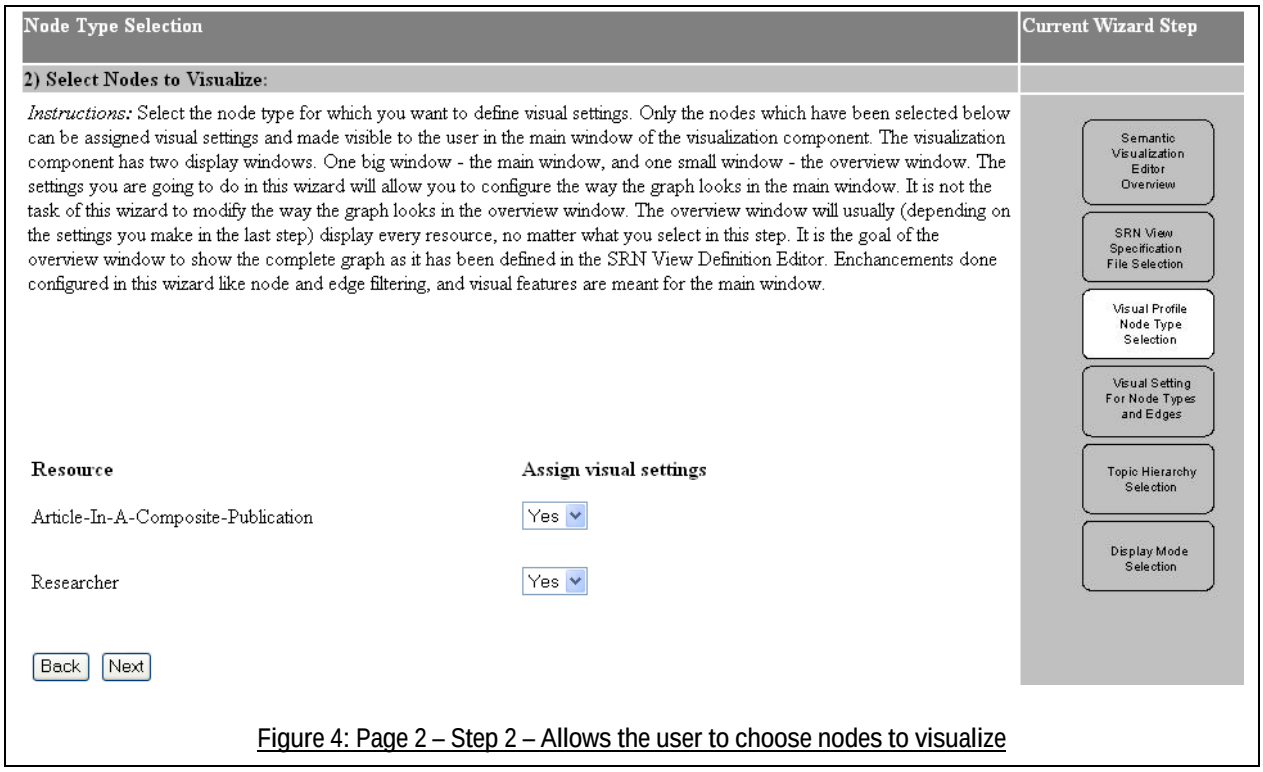

Figure 4: Page 2 – Step 2 – Allows the user to choose nodes to visualize

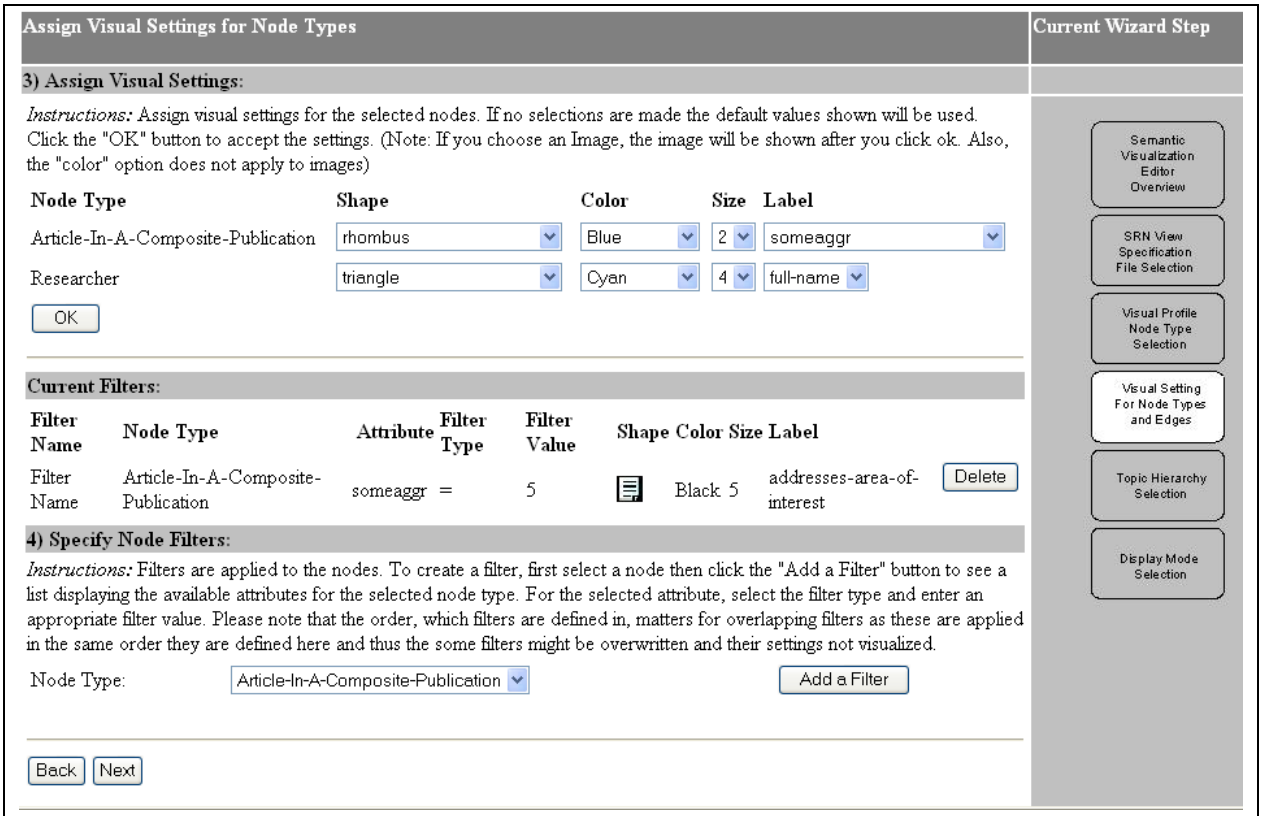

Figure 5: Page 3 – Step 3 – Visual Settings for Nodes and Step 4 a) – Filters

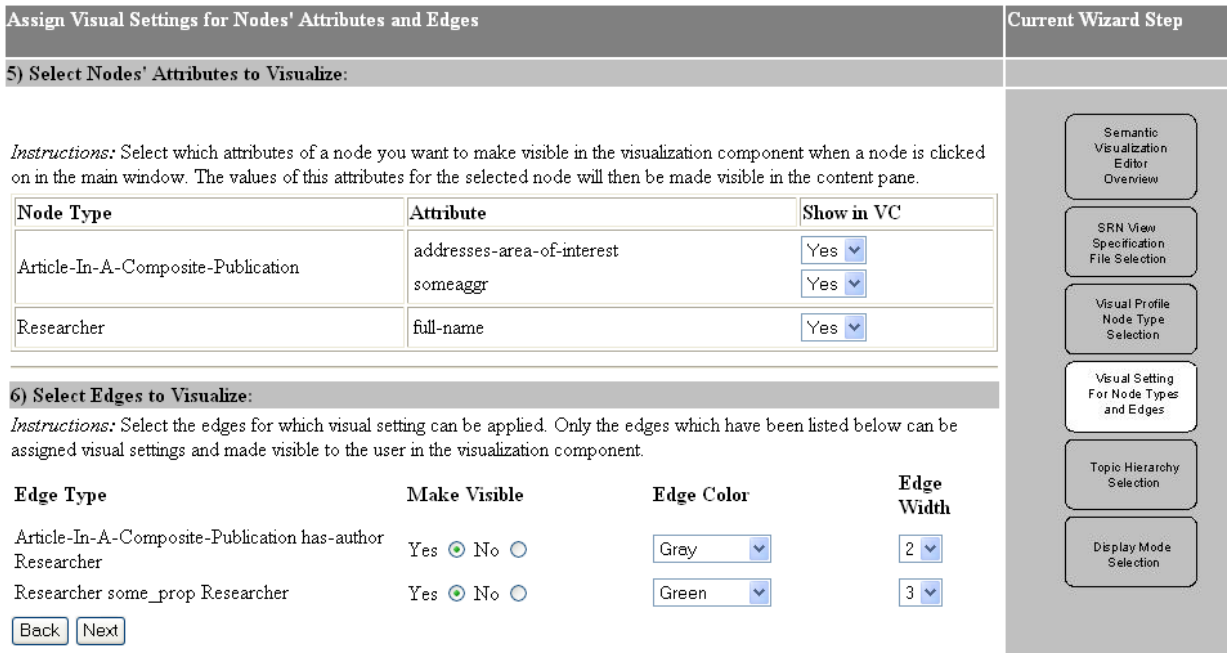

Figure 6: Page 4 – Step 5 – Node Data widget of the Visualization Application authoring and Step 6 – Visual Settings for edges

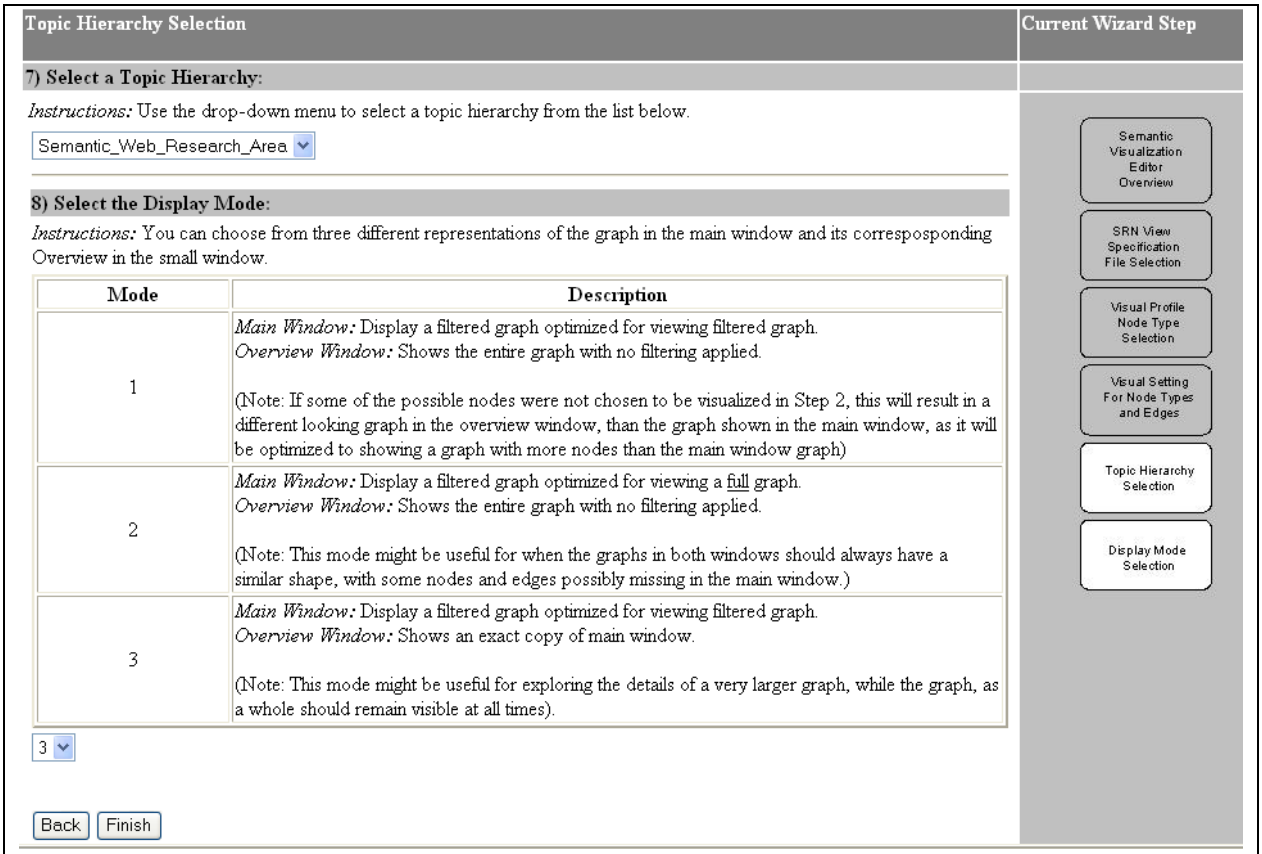

Figure 7: Page 5 – Step 7 – Topic Tree authoring and Step 8 – Display Mode selection

3.2.2 JSP Start Page

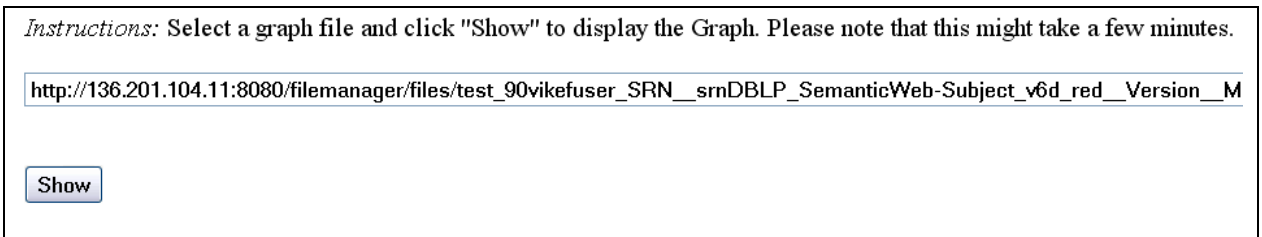

Figure 8: JSP Start Page

The Visualization Application is started using a JSP Page. The purpose of this page is to allow the user to choose a file for visualization and pass the necessary data to the applet. It shows a simple form listing the displayable files available. Refer to Section 4.3.1 for a discussion on the data passed by the JSP Page to the applet.

3.2.3 Main Window

Figure 3 shows the Main Window. This window is interactive and has one purpose – display the View as it has been set up by the Semantic Visualization Editor. This Editor currently defines the following elements that affect the way the graph in this picture looks (see Figure 4, Figure 5, Figure 6 and Figure 9):

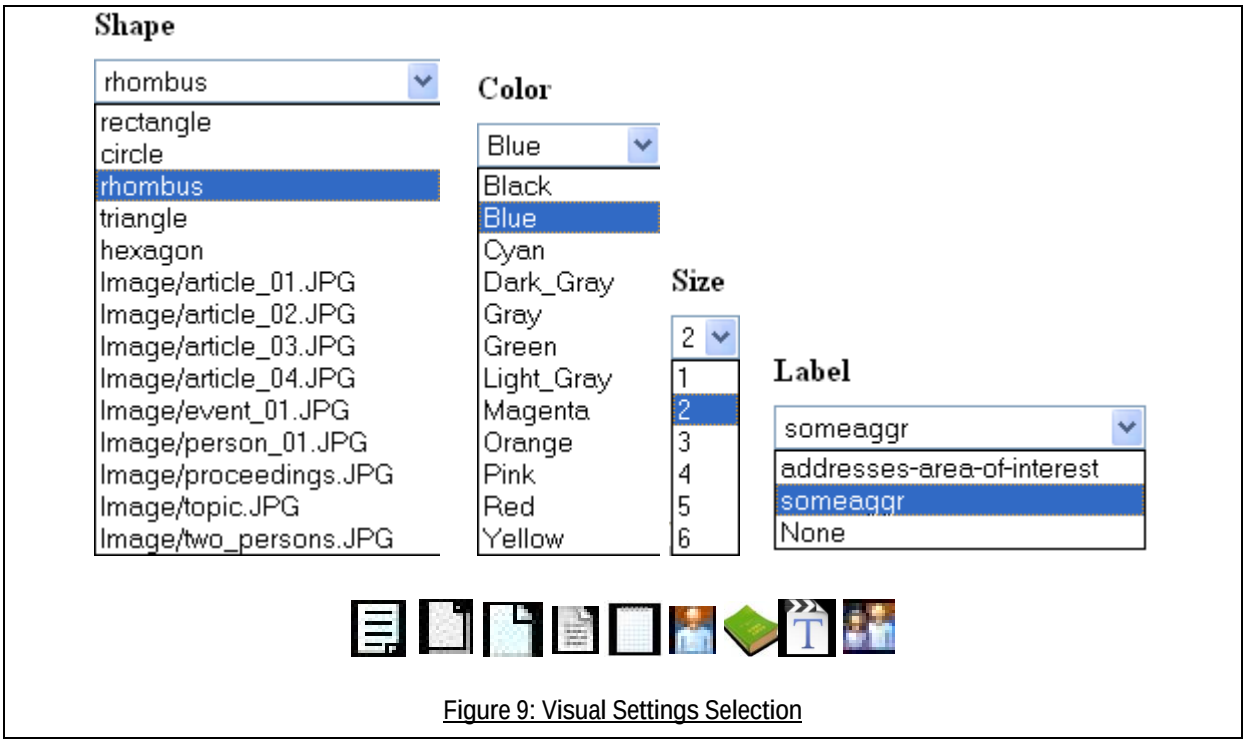

Figure 9: Visual Settings Selection

1. Number of different types of nodes (in the shown graph there are two types of nodes displayed – Researchers and Articles).

2. Shapes for the display of the nodes (in the shown graph only triangle and rhombus are used). Please note that the application of filters on some nodes will cause them to look different from other nodes of the same type. It is also possible to use one of the predefined images listed in Figure 9. In this case the color option is not applicable.

3. Colors of the nodes (in the shown graph only blue and cyan). Not applicable for images.

4. Sizes of the nodes. This attribute is relative regarding the displayed graph due to the fact that the Window supports zooming. However, the ratio between the different sizes is kept.

5. Labels of the nodes. The labels available to the user to choose from are View specific and normally differ from one node type to the other. The list of labels shown to the user to choose from is the same as the list of attributes this node has.

6. Number of different types of edges (in the shown graph there are two edge types available and both are displayed: has-author in gray and some_prop in green).
7. Colors of the edges.
8. Sizes of the edges. This attribute is relative like the sizes of the nodes.

Adding to or changing the Visual Features will require changing the Component. Refer to Section 5.1.2.4 for a discussion on this.

The Main Window offers some nice functionality for interaction with the graph:

1. Pointing the mouse over a node highlights it and its neighbors.
2. Clicking on a node moves it to the center of the screen and repositions the nodes connected to it around it in a way that makes them easy to observe.
3. The graph can be freely moved around by dragging the mouse.
4. Nodes can be freely dragged around with the mouse.
5. The graph can be zoomed in and out by dragging the mouse with the right mouse button or by rolling the mouse wheel.

The Main Window has two optimization modes for displaying of the graph:

1. Optimization for showing the filtered graph (applied if Display Mode 1 or 3 is selected in step eight of the Editor – Figure 7)
2. Optimization for showing the full graph (applied if Display Mode 2 is selected).

The difference between the two modes can be seen only if some nodes have been chosen not to be displayed in step two (Figure 4). The more nodes are chosen not to be displayed, the greater the difference between the two different graphs will be. The notes shown in Figure 7 describe in what sense the difference between the graphs in the two Windows will be observed.

3.2.4 Label Display

The Label Display is the small text box in the bottom left corner of the screen. It is static and its purpose is to display the label of the element the mouse is currently pointing at. If this element is a node the widget displays its label. If it is an edge – the label of the source node, the edge type and the label of the

target node are displayed. Figure 10 shows an example of the mouse pointing at a node and Figure 3 – an edge.

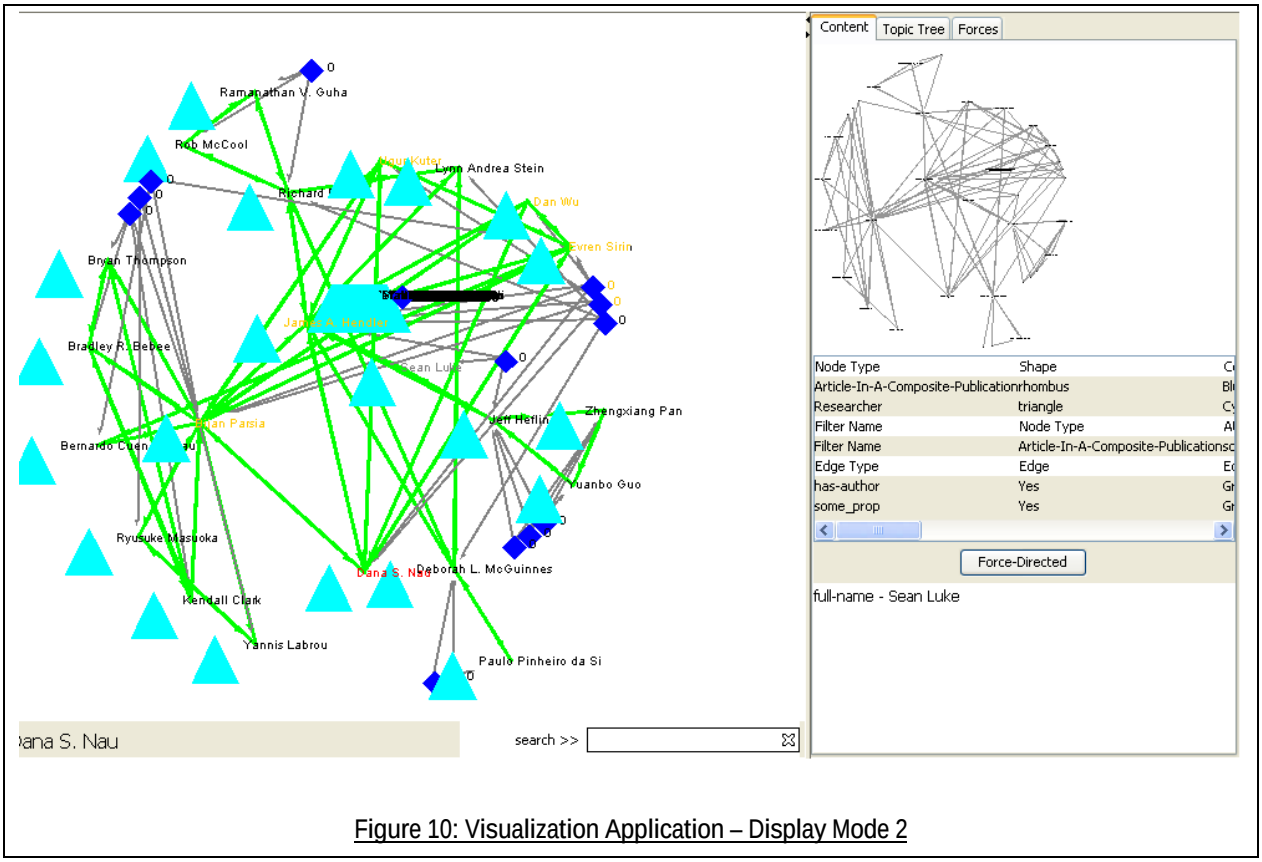

Figure 10: Visualization Application – Display Mode 2

3.2.5 Search Box

The Search Box is interactive and allows searching for nodes having the string written in the Search Box as a label. Figure 3 shows a highlighted node.

3.2.6 Overview Window

This widget is interactive and is situated on the top of the Content tab. The goal of the Overview Window initially was to show the complete View as a graph without any filtering or Visual Settings applied. This was changed for the second release of the Visualization Application allowing the user to choose what he wants to see in the Overview Window. This is accomplished in step eight of the Semantic Visualization Editor (see Figure 7). Currently the Overview Window has two modes:

1. Display the same graph as the Main Window and thus allow the user to zoom in on a small part of the graph, while still being able to see the big picture (as in Figure 3). This operation mode of the Overview Window is only available if option 3 for the Display Mode is selected. Changes

made to the graph in the Main Window (e.g. element dragged around) translate to the graph in the Overview Window and vice versa.

2. Display the entire graph with no filtering applied (Figure 10). In this case the graphs in the two Windows are normally quite different. Changes in one of the Windows do not translate to the other. Also see Section 4.3.6.

3.2.7 Legend

Node Type	Shape	Color	Size	Label				
Article-In-A-Composite-Publication	rhombus	Blue	2	someaggr				
Researcher	triangle	Cyan	4	full-name				
Filter Name	Node Type	Attribute	Filter Type	Filter Value	Shape	Color	Size	Label
Filter Name	Article-In-A-Composite-Publication	someaggr	=	5	Image/article_01.JPG	Black	5	addresses-area-of-interest
Edge Type	Edge	EdgeColor	EdgeSize					
has-author	Yes	Gray	2					
some_prop	Yes	Green	3					

Figure 11: Legend

The Legend is static and is situated in the middle of the Content tab. In the first release this widget was not present. However, it turned out to be quite easy to forget what settings have been applied to each node, filter and edge. To remember the Visual Settings applied, the user had to start the Semantic Visualization Editor again just to review the Visual Settings applied. This widget was added in the second release to list the Visual Settings that have been applied to the graph shown in a table similar to the table the user has used to define them.

3.2.8 Force-Directed

This widget is interactive – a button – and is situated in the middle of the Content tab. There are two modes available for graph display:

1. Static – based on a Radial Tree Layout and applied to the graph every time a node is clicked. Figure 10 shows a graph that has been applied this layout to (it has been applied to the highlighted in gray node “Sean Luke” situated in the center of the graph).

2. Dynamic – based on a Force Directed Layout – can be run and stopped by using the button. The Overview Window in Figure 3 shows the complete graph after it has been applied this layout to. The application of this layout is slow and uses up the computer resources. When it is turned on the Application runs slower and usually (due to the fact that the graphs are usually huge) never reaches a stable state – which means that the nodes of the graph keep jittering forever. The best way to use it is to turn it on for a short time every time it is needed to reposition the node according to this layout and then turn it off while working with the graph.

3.2.9 Node Data

This widget is situated at the bottom of the Content tab. Its task is to display additional information about the node that has last been clicked on in the Main Window. This widget is configurable in step five of the Semantic Visualization Editor (Figure 6).

3.2.10 Topic Tree

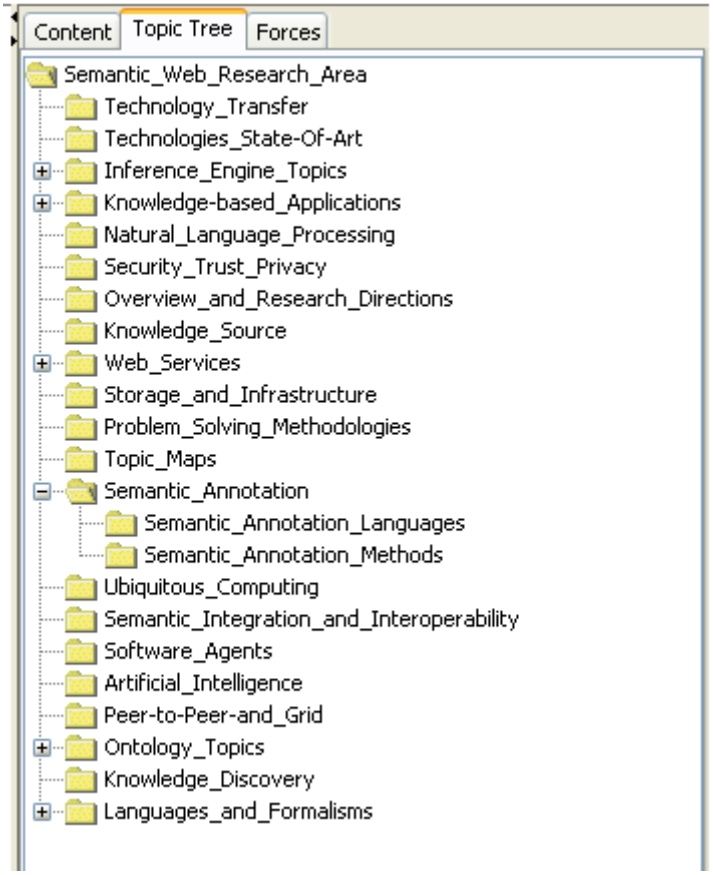

Figure 12: Topic Tree Tab

This widget is situated in its own tab. It is interactive and its task is to allow the user to easily filter the nodes on the topics they have been given. The topic hierarchy used can be selected in step seven of the Semantic Visualization Editor (see Figure 7). Clicking on a topic in the tree shown hides all nodes that have a topic attribute defined (see Figure 13), but this topic is different from the selected and all its sub-topics. The nodes that do not have a topic attribute defined are not affected (see Figure 16).

```
<node id="19">
  <data key="name">Article-In-A-Composite-Publication</data>
  <data key="Size">2</data>
  <data key="visible">Yes</data>
  <data key="Shape">rhombus</data>
  <data key="Color">Blue</data>
  <data key="label">0</data>
  <data key="someaggr">0</data>
  <data key="addresses-area-of-interest">http://trinity.dit.unitn.it/vikef/swc#Semantic_Web_Services</data>
  <data key="uri">http://www.ipsi.fraunhofer.de/~stewart/vikef/rn#conf/semweb/DenkerKFPS03</data>
  <data key="textdata">addresses-area-of-interest - Semantic_Web_Services;someaggr - 0;</data>
</node>
```

Figure 13: Node having a topic (addresses-area-of-interest)

3.2.11 Forces

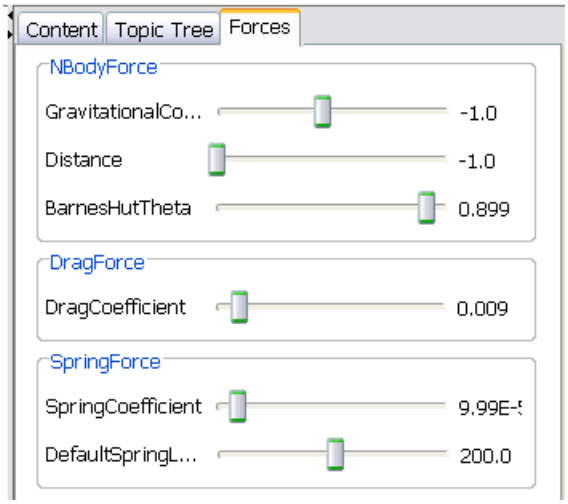

Figure 14: Forces Tab

This widget is situated in its own tab. It is interactive and its task is to customize the way the graph looks when Force-Directed Layout is started. The settings applied in this widget can only be seen when this layout is turned on. The sliders will not be described in detail, however their purpose is to customize different aspects of the nodes' and edges' interaction among themselves. It is only worth noting that the most important slider is the last one – Default Spring Length – affecting the length of the edges.

4 Implementation Issues

This part discusses the software design (code design) of the Visualization Application. It describes the technologies that were learned and used for the implementation of the component. It also describes the more important problems encountered and solved during coding and studying the used technologies.

4.1 Technologies Employed

This part extends Section 3.1.1 and lists the technologies that were technically required to complete the project.

The component draws input from a GraphML (GraphML Team, 2004) File and Ontology. The use of Ontology (Connolly, et al., 2004), (Ontology, 2007) requires Jena (Jen07).

The JSP (JSP06) Start Page of the component directly uses two other VIKEF Components: File Manager Client and Ontology Manager Client. The File Manager Client is needed for accessing the GraphML input file. Within VIKEF data files are stored on a central server, and the File Manager Client is used exclusively for accessing these files. For managing data in form of Ontology and SRN the Ontology Manager Client is used. In the Visualization Application Ontology is accessed in order to create the Topic Tree (see Figure 11).

4.2 Component Architecture

Best effort is made to create the Software Architecture of the Semantic Visualization Components following the principles of Software Engineering.

I developed the Visualization Application and this document during its development. An effort was made to produce and release a usable component for three different review deadlines, changing and extending the functionality as required for each release. A description of each release follows.

1. Basic Functionality – Read in and display a GraphML file produced by the Semantic Visualization Factory. Show the full graph with no filtering in the Overview Window.
2. Extend the component adding the following elements: Label Display, Search Box, Legend, Radial Tree Layout, Topic Tree and Forces. Fix a few bugs.

3. Refactor the component extensively. Add the Node Data widget. This is the current release of the component. It contains over 2000 LoC.

4. For possible future release ideas, please refer to the Section 7.2.

The next part describes the development, difficulties overcome and problems solved, the structure of the code, and the way the data is processed in the Visualization Application. For a technical description of each class see Appendix.

4.3 Development

This section describes the experience of working with prefuse. The overall impression of using this graphical package for Java is very good and work with it proved pleasant and effective. The only weakness found is discussed in Section 9.2.2.2.

4.3.1 Applet restrictions

The component is built around a prefuse applet. Using java applets comes with the security restrictions to applets. Two problems introduced by the applet technology had to be solved in order to complete the project. These were passing the input files to the applet and making the java libraries available to the applet.

4.3.1.1 Input

Two files need to be input to the applet in order for it to process them and display the graph. The first one is a GraphML file generated by the Semantic Visualization Factory and the second one is an .owl – ontology – file needed to extract the topic tree from. Due to the applet security restrictions it was impossible to access the File Manager Client directly from the applet to allow the user to choose an input graph file. Similarly it was impossible to access the Ontology Manager Client to extract a topic tree.

To solve this problem two JSP Pages were created. The user could now select an input file in a HTML (HTML, 2007) form. Upon submission the graph file is copied and stored locally, the Ontology Manager Client contacted and the ontology extracted and stored locally as a file. Finally the applet is started and the filenames of the two files are passed to it as parameters.

4.3.1.2 Java Libraries

The ontology has to be accessed in order to extract the topic tree. This is fastest implemented by storing a copy of the ontology locally as a file and passing the name of this file as a parameter to the applet so that the topic tree can be extracted from within the applet and used directly. However, this requires copy-

ing jena.jar and many other jars required by jena.jar to the directory containing the JSP pages, which is the only place where they can be accessed by the applet. However, this results in clumsy directory structure.

Another approach would have been extracting the topic tree from within the JSP page, storing it as an XML file and passing the name of this file to the applet. The applet could then use simple XML processing to extract the topic tree without the need for additional libraries.

The second approach is now considered slightly better, however it was not taken at the time the component was developed due to inexperience with using complex applets requiring input files and numerous jars. Nevertheless, the currently implemented solution would always be the preferred solution if the application did not have the applet restrictions, as it requires processing the data only once and the second here suggested approach processes the same information three times (1. Read the Ontology; 2. Store it as XML; 3. Read the XML). Whether the tradeoff between the additional overhead of this second approach and the possible jar problems with the implemented solution is really in favor of the second approach cannot be proven as unforeseen problems might appear.

4.3.2 Extending prefuse

The project requirements exceeded the standard capabilities of the default renderers. In order to draw the nodes with shapes or icons and labels a new renderer for the nodes had to be developed. Drawing differently sized and colored edges is not possible using the default renderers either. A considerable effort in the implementation phase was needed to implement these features that resulted in over 850 LoC (almost half the code length of the whole component) in three classes. See the Appendix for a detailed description of each of these classes.

4.3.3 Legend

```
<graphml xmlns="http://graphml.graphdrawing.org/xmlns">
    <key attr.name="TopicHierarchy" attr.type="String" for="node" id="TopicHierarchy"/>
    <key attr.name="OntologyVersionId" attr.type="String" for="node" id="OntologyVersionId"/>
    <key attr.name="DisplayType" attr.type="int" for="node" id="DisplayType"/>
    <key attr.name="nodeCount" attr.type="int" for="node" id="nodeCount"/>
    <key attr.name="nodeVisCount" attr.type="int" for="node" id="nodeVisCount"/>
    <key attr.name="0.nodeVisName" attr.type="String" for="node" id="0.nodeVisName"/>
    <key attr.name="1.nodeVisName" attr.type="String" for="node" id="1.nodeVisName"/>
    <key attr.name="2.nodeVisName" attr.type="String" for="node" id="2.nodeVisName"/>
    <key attr.name="0.node.nodeType" attr.type="String" for="node" id="0.node.nodeType"/>
    <key attr.name="0.node.0.visValue" attr.type="String" for="node" id="0.node.0.visValue"/>
    <key attr.name="0.node.1.visValue" attr.type="String" for="node" id="0.node.1.visValue"/>
    <key attr.name="0.node.2.visValue" attr.type="String" for="node" id="0.node.2.visValue"/>
    <key attr.name="0.node.label" attr.type="String" for="node" id="0.node.label"/>
    <key attr.name="1.node.nodeType" attr.type="String" for="node" id="1.node.nodeType"/>
    <key attr.name="1.node.0.visValue" attr.type="String" for="node" id="1.node.0.visValue"/>
    <key attr.name="1.node.1.visValue" attr.type="String" for="node" id="1.node.1.visValue"/>
    <key attr.name="1.node.2.visValue" attr.type="String" for="node" id="1.node.2.visValue"/>
    <key attr.name="1.node.label" attr.type="String" for="node" id="1.node.label"/>
    <key attr.name="specialCaseCount" attr.type="int" for="node" id="specialCaseCount"/>
    <key attr.name="0.specialCase.filterName" attr.type="String" for="node" id="0.specialCase.filterName"/>
    <key attr.name="0.specialCase.nodeType" attr.type="String" for="node" id="0.specialCase.nodeType"/>
    <key attr.name="0.specialCase.attribute" attr.type="String" for="node" id="0.specialCase.attribute"/>
    <key attr.name="0.specialCase.filterType" attr.type="String" for="node" id="0.specialCase.filterType"/>
    <key attr.name="0.specialCase.filterValue" attr.type="String" for="node" id="0.specialCase.filterValue"/>
    <key attr.name="0.specialCase.0.visValue" attr.type="String" for="node" id="0.specialCase.0.visValue"/>
    <key attr.name="0.specialCase.1.visValue" attr.type="String" for="node" id="0.specialCase.1.visValue"/>
    <key attr.name="0.specialCase.2.visValue" attr.type="String" for="node" id="0.specialCase.2.visValue"/>
    <key attr.name="0.specialCase.label" attr.type="String" for="node" id="0.specialCase.label"/>
    <key attr.name="edgeCount" attr.type="int" for="node" id="edgeCount"/>
    <key attr.name="edgeVisCount" attr.type="int" for="node" id="edgeVisCount"/>
    <key attr.name="0.edgeVisName" attr.type="String" for="node" id="0.edgeVisName"/>
    <key attr.name="1.edgeVisName" attr.type="String" for="node" id="1.edgeVisName"/>
    <key attr.name="2.edgeVisName" attr.type="String" for="node" id="2.edgeVisName"/>
    <key attr.name="0.edge.edgeType" attr.type="String" for="node" id="0.edge.edgeType"/>
    <key attr.name="0.edge.0.visValue" attr.type="String" for="node" id="0.edge.0.visValue"/>
    <key attr.name="0.edge.1.visValue" attr.type="String" for="node" id="0.edge.1.visValue"/>
    <key attr.name="0.edge.2.visValue" attr.type="String" for="node" id="0.edge.2.visValue"/>
    <key attr.name="1.edge.edgeType" attr.type="String" for="node" id="1.edge.edgeType"/>
    <key attr.name="1.edge.0.visValue" attr.type="String" for="node" id="1.edge.0.visValue"/>
    <key attr.name="1.edge.1.visValue" attr.type="String" for="node" id="1.edge.1.visValue"/>
```

```
    <key attr.name="1.edge.2.visValue" attr.type="String" for="node" id="1.edge.2.visValue"/>
    <key attr.name="name" attr.type="String" for="node" id="name"/>
    <key attr.name="uri" attr.type="String" for="node" id="uri"/>
    <key attr.name="addresses-area-of-interest" attr.type="String" for="node" id="addresses-area-of-interest"/>
    <key attr.name="someaggr" attr.type="String" for="node" id="someaggr"/>
    <key attr.name="full-name" attr.type="String" for="node" id="full-name"/>
    <key attr.name="textdata" attr.type="String" for="node" id="textdata"/>
    <key attr.name="visible" attr.type="String" for="node" id="visible"/>
    <key attr.name="Shape" attr.type="String" for="node" id="Shape"/>
    <key attr.name="Color" attr.type="String" for="node" id="Color"/>
    <key attr.name="Size" attr.type="String" for="node" id="Size"/>
    <key attr.name="label" attr.type="String" for="node" id="label"/>
    <key attr.name="visibleEdge" attr.type="String" for="edge" id="visibleEdge"/>
    <key attr.name="EdgeSize" attr.type="String" for="edge" id="EdgeSize"/>
    <key attr.name="EdgeColor" attr.type="String" for="edge" id="EdgeColor"/>
    <key attr.name="label" attr.type="String" for="edge" id="label"/>
    <graph edgedefault="directed" id="G">
        <node id="0">
            <data key="TopicHierarchy">http://trinity.dit.unitn.it/vikef/swc#Semantic_Web_Research_Area</data>
            <data key="OntologyVersionId">vikefuser____http://www.ipsi.fraunhofer.de/__ONTO__scienceOntologyV3.00__Version__Mon Jan
22 15:00:12 GMT 2007</data>
            <data key="DisplayType">1</data>
            <data key="nodeCount">2</data>
            <data key="nodeVisCount">3</data>
            <data key="0.nodeVisName">Shape</data>
            <data key="1.nodeVisName">Color</data>
            <data key="2.nodeVisName">Size</data>
            <data key="0.node.nodeType">Article-In-A-Composite-Publication</data>
            <data key="0.node.0.visValue">rhombus</data>
            <data key="0.node.1.visValue">Blue</data>
            <data key="0.node.2.visValue">2</data>
            <data key="0.node.label">someaggr</data>
            <data key="1.node.nodeType">Researcher</data>
            <data key="1.node.0.visValue">triangle</data>
            <data key="1.node.1.visValue">Cyan</data>
            <data key="1.node.2.visValue">4</data>
            <data key="1.node.label">full-name</data>
            <data key="specialCaseCount">1</data>
            <data key="0.specialCase.filterName">Filter Name</data>
            <data key="0.specialCase.nodeType">Article-In-A-Composite-Publication</data>
            <data key="0.specialCase.attribute">someaggr</data>
            <data key="0.specialCase.filterType">==</data>
            <data key="0.specialCase.filterValue">5</data>
            <data key="0.specialCase.0.visValue">Image/article_01.JPG</data>
            <data key="0.specialCase.1.visValue">Black</data>
            <data key="0.specialCase.2.visValue">5</data>
            <data key="0.specialCase.label">addresses-area-of-interest</data>
            <data key="0.edgeVisName">Edge</data>
            <data key="1.edgeVisName">EdgeColor</data>
            <data key="2.edgeVisName">EdgeSize</data>
            <data key="edgeCount">2</data>
            <data key="edgeVisCount">3</data>
            <data key="0.edge.edgeType">has-author</data>
            <data key="0.edge.0.visValue">Yes</data>
            <data key="0.edge.1.visValue">Gray</data>
            <data key="0.edge.2.visValue">2</data>
            <data key="1.edge.edgeType">some_prop</data>
            <data key="1.edge.0.visValue">Yes</data>
            <data key="1.edge.1.visValue">Green</data>
            <data key="1.edge.2.visValue">3</data>
            <data key="name">Researcher</data>
            <data key="Size">4</data>
            <data key="visible">Yes</data>
            <data key="Shape">triangle</data>
            <data key="Color">Cyan</data>
            <data key="label">Sean Luke</data>
            <data key="full-name">Sean Luke</data>
            <data key="uri">http://www.ipsi.fraunhofer.de/~stewart/vikef/rn#Sean_Luke</data>
            <data key="textdata">full-name - Sean Luke;</data>
        </node>
```

Figure 15: GraphML Visualization Application Authoring Extract

Figure 15 shows an extract of a sample GraphML file generated by the Semantic Visualization Factory that can be displayed by the Visualization Application. This extract contains the full widget authoring data. All elements from “NodeCount” to “m.edge.n.visValue” are only responsible for authoring the legend. The pattern used for these elements should be profoundly studied before any attempts are made to change or extend it (it is strongly advised against such changes as these will require technical changes in both the Semantic Visualization Factory and the Visualization Application). It proved unhandy fitting this data as legal elements in the GraphML schema. The main reason, why this element proved unhandy to author, is that the data that is going to be put in it is unknown at compile time and only available at runtime.

A different (probably better) approach might have been following a similar pattern as the pattern used for authoring the Node Data widget (see below). It is advised that this part is rewritten using this approach in a future release.

4.3.4 Node Data

This is another widget that should be authored using data not available at compile time. It was added in the third release of the component. The approach taken for authoring the legend proved unhandy and it would have been even less applicable for this widget as the number of elements that need to be included for each type of node could differ greatly. Therefore a new approach was developed – storing the whole data as a string and using an escape character (;) to separate the different elements from each other (see “textdata” in Figure 15 and Figure 16). This approach proved fast and easy to use.

4.3.5 Creating the Graph

```
<node id="1214">
    <data key="name">Researcher</data>
    <data key="Size">4</data>
    <data key="visible">Yes</data>
    <data key="Shape">triangle</data>
    <data key="Color">Cyan</data>
    <data key="label">Matthew Quinlan</data>
    <data key="full-name">Matthew Quinlan</data>
    <data key="uri">http://www.ipsi.fraunhofer.de/~stewart/vikef/rn#Matthew_Quinlan</data>
    <data key="textdata">full-name - Matthew Quinlan;</data>
</node>
<node id="1215">
    <data key="name">Researcher</data>
    <data key="Size">4</data>
    <data key="visible">Yes</data>
    <data key="Shape">triangle</data>
    <data key="Color">Cyan</data>
    <data key="label">Wang , L.</data>
    <data key="full-name">Wang , L.</data>
    <data key="uri">http://www.unitn.it/transformationv1#LE113826812056940720731614778605 45</data>
    <data key="textdata">full-name - Wang , L.;</data>
</node>
<edge source="0" target="937">
    <data key="EdgeSize">3</data>
    <data key="EdgeColor">Green</data>
    <data key="visibleEdge">Yes</data>
    <data key="label">Sean Luke some_prop Jeff Heflin</data>
</edge>
<edge source="0" target="972">
    <data key="EdgeSize">3</data>
    <data key="EdgeColor">Green</data>
    <data key="visibleEdge">Yes</data>
    <data key="label">Sean Luke some_prop James A. Hendler</data>
</edge>
<edge source="2" target="1180">
    <data key="EdgeSize">3</data>
    <data key="EdgeColor">Green</data>
    <data key="visibleEdge">Yes</data>
    <data key="label">Pascal Auillans some_prop Bernard Vatant</data>
</edge>
```

Figure 16: GraphML Node and Edge Display Extract

Figure 16 shows the GraphML representation of a few sample node and edge elements. Parsing and displaying graph elements stored in this format proved easy and straight forward using the prefuse internal GraphML parser.

4.3.6 Relationship between the two Display Windows

The relationship between the two Display Windows in the different Display Modes is not consistent. While changes to the graph (applied by the user – e.g. dragging) in one of the Windows translate to the other in Display Mode 3, this is not the case in the other two Display Modes. This difference might introduce

some confusion, however it is implementation dependant how the two Windows are interconnected. While it is possible to make the two Windows independent in Display Mode 3, making the two Windows dependant in the other Display Modes is next to impossible. The reason for this is that the displayed graphs in these cases are required to be different and thus are implemented using different visualizations (in prefuse terms), which makes them independent. Display Mode 3, on the contrary, features two displays of the same visualization.

4.3.7 Topic Tree

Extracting the data for the Topic Tree from the ontology and displaying it in the Topic Tree tab proved technically unhandy. The reason for this is the inability of Jena to make a transitive check whether a class is a subclass of another class. A recursive method had to be used to make up for the lack of this functionality. Apart from this and the fact that it relies on many additional jars (which makes it unhandy for using in applets), Jena was found to be a powerful and easy to use tool. Refer to the Appendix for further technical information.

4.3.8 Interactive Elements

All of the widgets except for the Legend are either interactive or change during execution (Label Display and Node Data) to reflect the user's interaction with the Main Window. Ideas for the implementation of these elements were drawn from the prefuse demos coming with the prefuse package. Somewhat new and not featured in any of the prefuse demos is only the implementation of the Topic Tree.

4.4 Integration within VIKEF

Figure 17 draws the relationships and collaborations of the different components in VIKEF's Semantic Navigation Support.

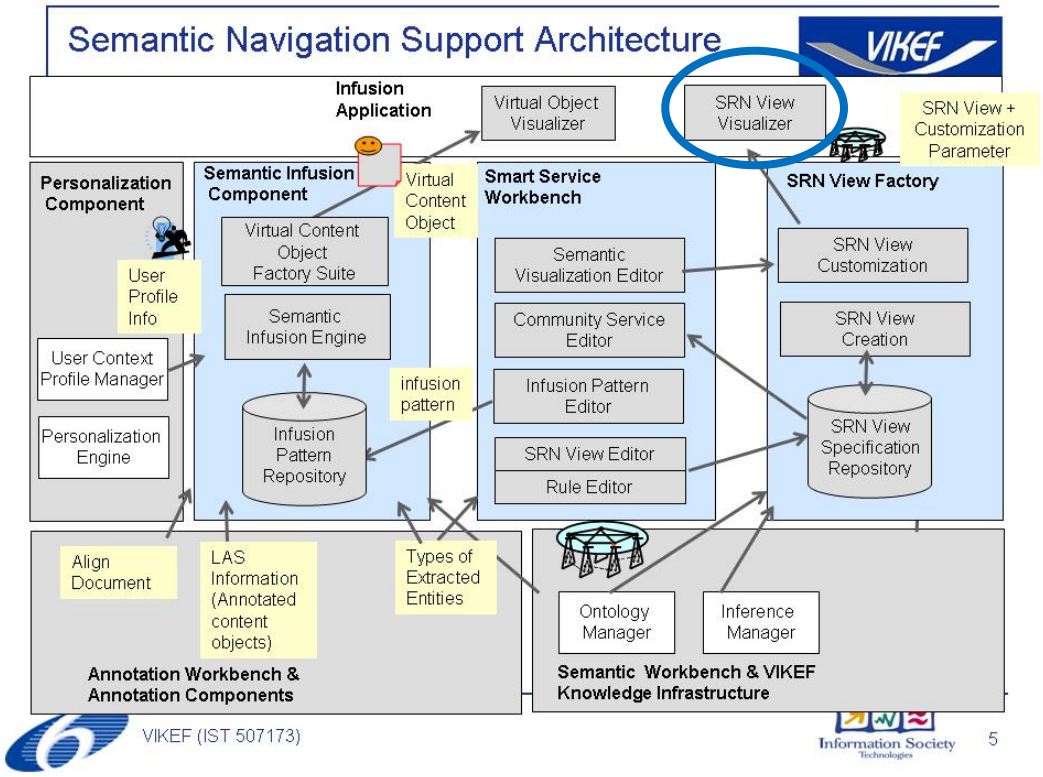

Figure 17: Semantic Navigation Support Architecture

5 Preliminary Evaluation

This part has two goals: first to evaluate the software design of the project from the author's own point of view and second to sum up the experience different people had using the software.

5.1 Evaluation Design

This part attempts to evaluate the software (code) design of the project in accordance to the software design principles. It considers the five modular properties, some class-level design principles and some package-level design principles.

5.1.1 Modular-* Criteria

According to these criteria the component has no well formed weaknesses.

5.1.1.1 Modular Decomposability

The big picture of the project is divided into five parts: SRN View Editor and Factory and Semantic Visualization Editor and Factory and Visualization Application. According to the plan and in practice work can continue on each of these components independently of the state of the others. The lack of strict formats for communication among them spoils this to some extent as it is often the case that features have to be considered in the big picture to ensure operability.

Within the Visualization Application the system can be modularly decomposed into many parts: main application, a part for each widget (here only the Main Window, the Overview Window and the Topics Tree depend somewhat on each other as they use a common Visualization element), a node renderer and an edge renderer. The work on these modules can continue independently of each other and they were developed and tested independently of each other.

5.1.1.2 Modular Composability

Modular Composability has been given thought to while developing the component. The node and edge renderers can be considered stand alone modules that can be reused in other prefuse applications. The other elements are all different Panels and as such limited situations can be found where they can be

reused. They are mostly (as described in Section 4.3.8) elements whose features and implementations were to a big extent reused from prefuse demos.

5.1.1.3 Modular Understandability

Support for Modular Decomposability and Composability also resulted in good Modular Understandability. Modular reasoning is decently supported.

5.1.1.4 Modular Continuity

Modular Continuity was not considered to a large extent in the first releases. However in the last release, a targeted attempt was made to fully incorporate Modular Continuity through refactoring and the creation of the Cs class. The suggestion given in Section 4.3.3 will further improve the design in this direction. Provided this change is implemented, the only classes that will still show some weaknesses, having this criterion in mind, would be the renderers. Extending the functionality of the Cs class could help "improve" these classes too, but abstracting structures like the shapes and colors away, from the only spots that they are used at, just in order to possibly slightly increase Modular Continuity, is considered too much overhead and a too big decrease in Modular Understandability to do.

5.1.1.5 Modular Protection

Support for Modular Protection is good. The application has been developed supporting this criterion on purpose. Small changes and extensions (even in the graph input file specification) will most probably remain restricted within one module. A contribution to the good mark this component gets on this criterion is made by the standardization of the input graph file targeted during the development of the Semantic Visualization Factory.

5.1.2 Class-level Design

The design of the classes in the last release has improved significantly after they were thoroughly refactored. A lot of time and effort was spent on improving this part of the project prior to the third release.

5.1.2.1 General Principles

5.1.2.1.1 Information hiding

Information hiding is not at the level it should be at. The refactoring work did not focus on this aspect. A lot of the elements are declared using package visibility although they should not be accessible from outside the classes they are declared in. Further refactoring should be done to improve this issue.

5.1.2.1.2 Coupling and cohesion

Coupling and cohesion are paid attention to while developing the Visualization Application. All classes have low to moderate coupling and moderate to high cohesion. ContentPane is the only class where cohesion might be seen as a problem, however high method cohesion is still maintained. Further refactoring might extract each widget in its own class, rather than handling the widgets in different classes depending on their positioning in the application window.

5.1.2.2 Assigning Responsibilities

Responsibilities are not assigned to the best extent throughout the Application according to general practices. The Applet class acts as a Façade Controller and an Expert on the creation of the displays depending on the Display Mode and the graph. This is a generally accepted “bad” practice. The reason this approach was taken was that in my opinion creating a class without any responsibilities is even worse practice as it introduces needless classes used just for passing data to other classes. Logic is not separated from display in the design of the classes. A major reason for this is that only little and simple business logic processing is required in an application built for visualization purposes that aims to display data to the user, but not to process and output anything.

The classes act as Expects on the creation of any widgets created by them. They also act as Controllers on the interactive widgets created by them managing listeners and coordinating possible changes to the graph.

There are no Creator responsibilities within the component as no objects are created after the initialization.

5.1.2.3 SRP

SRP has been considered, however not always followed during the development of the system. A system fully supporting SRP introduces a greater number of smaller classes, a lot of relationships between them and slows down the production of software.

5.1.2.4 OCP

The Visualization Application is open for the addition of new widgets and their authoring. This can be achieved with only the overhead needed to create those widgets. However, closure against new Visual Features is not fully supported and might require changes in the renderer classes. Further improving this criterion in this direction is not possible without making blind assumptions or great overhead.

5.1.2.5 LSP

The refactoring work done prior to the third release targeted and achieved LSP compatibility for the new renderer classes used in the component.

LSP can be fully supported if some further refactoring is done to create constructors in the other classes with the same signatures as the constructors of the classes they extend. However, this is useless as the super classes of these classes are framework classes designed to be extended before they are used.

5.1.2.6 Other Principles

DIP and ISP are not considered as mostly not applicable.

The only package design principle that has been considered is CCP for the renderers package. Changes to the current package design cannot be motivated by any of the package design principles. Such changes might become applicable only if considerable refactoring is done on class design level and new classes are introduced and the current classes changed or completely removed.

5.2 User Involvement

General users were involved in assessing the Visualization Application. This assessment phase took place prior to the release of this document.

The goal of assessing the Visualization Application was to get user feedback on its ability to support the user in exploring and using a visualization for better studying of the data contained in the domain. Through the assessment the following points were obtained:

1. An overall impression from end-users engaged in using a set of predefined visualizations.
2. Feedback about how the prototype could be extended.
3. Feedback on the usability of the component in supporting the users in completing a set of predefined tasks.

5.2.1 Set-Up

The assessment was structured in four phases. During the preparation phase, the purpose of the application and the predefined tasks were presented to the users. A set of instructions on how to use the Visualization Application and a small practical tutorial using a small predefined graph were given to them in order to introduce them to the application. In the second phase, the users were guided in completing a predefined set of tasks with gradually increasing difficulty and decreasing assistance. In the third phase,

the users were asked to complete a questionnaire. During this time they received less guidance and were free to refer back to the tool whenever they needed while completing the questionnaire. In the final phase, the users were given the opportunity to openly discuss aspects of the tool.

5.2.2 Results

In general, the application interface was seen as simple, easy to learn, practical and quite useful for the purpose for which it was designed. Most users found that it was easy and natural to get the system to do what they wanted from the very beginning, yet others stated it requires a considerable amount of time to get used to the unnatural controls.

Most users were quick and required little or no help to become confident with moving separate nodes or connected graphs around. Few still found it confusing at the end of the session.

Most users found the relationships are visualized in an obvious and very easy to use way; still there were exceptions who could not figure out the basic meaning of an edge between two nodes.

The only widget that was found confusing by all users was the legend; most users easily oversaw the scrollbar and after they were told that it existed few found it heavy moving and slow; the structure and labels were seen as inconsistent. While some were able to interpret and become confident with using it without assistance, some required some help and explanation and few said they were not sure if they will be able to get used to using the legend in its current format.

We were able to receive end-user feedback on how the prototype could be extended. Comments included the following:

- Extend the tools functionality to allow users to define other structures, such as pie charts, as output.
- Add more layouts. E.g. grid.
- Improve the search functionality. Make the nodes found by the search box easier to see. E.g. make the found nodes bigger; mark the found nodes with a circle or an arrow. Mark the found elements in the Overview Window as well.
- Offer the user to choose a color, which the different highlighted nodes should appear in, and change the color of not just the label, but also the node itself when highlighting it.

- Improve the Legend: Better separation of column title and column value. Add column borders. Make the scroll bar more apparent. Hide the Filters Row from the legend if no filters defined. Rename some column titles.
- Make it possible to apply the Radial Tree Layout to the same node twice in a row.
- Add the functionality to choose, where to reposition a node to after clicking on it, instead of repositioning it in the center of the window.
- Improve the layout of the JSP Page to match the layout of the Semantic Visualization Editor and make the file selection box smaller so that it fits in the browser window.

5.2.3 Assessing User Comments

In assessing the user comments, some items are easy and others more difficult to include. Some items are also outside the scope of this tool or are not appropriate for this context. The expected difficulty having the project design in mind is also listed:

- Extending the functionality to support Pie Charts – moderately difficult to support in the Editor, very difficult to support in the Visualization Application.
- Extending the functionality to support more layouts – easy to implement as long as using predefined prefuse layouts.
- Make the nodes found by the search box easier to see – easy to implement.
- Give the user the ability to choose highlight colors – easy to implement.
- Improve the legend – moderately difficult – the way the legend data is transmitted in the GraphML file should be changed too.
- Make it possible to apply the Radial Tree Layout to the same node twice in a row – easy to implement.
- Choose where to reposition a node to after clicking on it, instead of repositioning it in the center of the window – difficult to implement.
- Improve the JSP page – easy to implement.

6 Related Work

This project is related to every project built around GraphML and prefuse. The prefuse visualization gallery (prefuse, 2007) provides an overview of such projects. There are numerous projects there and in the prefuse demos that are closely related to the current application and have been used to gather ideas for the realization. Vizster is a good example of a project using a similar graph to display similar relationships between different instances. The only new aspect implemented in the VIKEF Visualization Application is the filtering according to a topic hierarchy and the fact that it uses Jena for the extraction of this hierarchy from the ontology. The task to develop this component was not to develop something new seen as a separate component but to complete the process started by the other components in the VIKEF pipeline (see Figure 2). In this sense little related work is known. Refer to (Nikolov, 2007) for further information.

7 Conclusion

The Components developed within the Semantic Navigation Services of VIKEF provide a useful and easy to use tool for the enrichment, aggregation and visualization of enormous datasets – an invaluable aid to members of communities working with this information.

7.1 Summary

This paper describes one of these components: the Visualization Application. It takes a GraphML file produced by the Semantic Visualization Factory and displays it.

The design, implementation issues, problems (both solved and unsolved) with the component, concerns and ideas for future further development were presented in this document. The Appendix at the end delivers technical details on the software.

A preliminary evaluation section is also included in this paper discussing the software design according to the software engineering principles and the usability and level of satisfaction of the component.

7.2 Limitations of this Work and Future Work

This part summarizes the ideas given throughout the document on further possibilities for development and improvement.

1. The Legend should be redesigned. See Section 4.3.3.
2. The classes should be further refactored. See Sections 5.1.2.1 and 5.1.2.6.
3. Extensions based on user comments. See Section 5.2.3.

8 Bibliography

[Toshev, 2007] Design & Implementation of framework for constructing / tailoring task specific views on knowledge bases [Report] / auth. Toshev Yasen. - Darmstadt : TU Darmstadt, 2007. forthcoming

[Quin, 2006] Extensible Markup Language [Online] / auth. Quin Liam // World Wide Web Consortium. - W3C, 09 11, 2006. - 11 2006. - http://www.w3.org/XML/.

[HTML, 2007] HTML Tutorial [Online] // W3Schools Online Web Tutorials. - W3 Schools, 2007. - 10 2006. - http://www.w3schools.com/html/.

[Java, 2004] Java 2 Platform SE 5.0 [Online] // Java Technology. - Sun Microsystems, 2004. - 11 2006. - http://java.sun.com/j2se/1.5.0/docs/api/.

[Jen07] Jena Semantic Web Framework [Online]. - SourceForge. - 01 2007. - http://jena.sourceforge.net/.

[JSP06] JSP Tutorial [Online]. - 10 2006. - http://www.jsptut.com/.

[Nikolov, 2007] Support for User-friendly Customization of Knowledge Network Visualization [Report] / auth. Nikolov Dimitar. - Darmstadt : TU Darmstadt, 2007. forthcoming

[Ontology, 2007] Ontology (computer science) [Online] // Wikipedia, the free encyclopedia. - Wikimedia, 04 02, 2007. - 01 2007. - http://en.wikipedia.org/wiki/Ontology_(computer_science).

[prefuse, 2007] prefuse | interactive information visualization toolkit [Online]. - SourceForge, 02 11, 2007. - 01 2007. - http://prefuse.org/.

[GraphML Team, 2004] The GraphML File Format [Online] / auth. Team the GraphML. - 09 29, 2004. - 01 2007. - http://graphml.graphdrawing.org/.

[VIKEF 1, 2007] VIKEF Knowledge Supply Chain [Online] // VIKEF. - Information Society Technologies, 2007. - 03 2007. - http://www.vikef.net/downloads/presentations/SemanticInfusion.pps.

[VIKEF 2, 2007] VIKEF Knowledge Supply Chain [Online] // VIKEF. - Information Society Technologies, 2007. - 03 2007. - http://www.vikef.net/downloads/presentations/KnowledgeView.pps.

[Le Hegaret, et al., 2006] W3C Document Object Model [Online] / auth. Le Hegaret Philippe, Whitmer Ray and Wood Lauren // World Wide Web Consortium. - W3C, 06 12, 2006. - 12 2006. - http://www.w3.org/DOM/.

[Connolly, et al., 2004] W3C Web Ontology (WebOnt) Working Group (OWL) (Closed) [Online] / auth. Connolly Dan, Hendler Jim and Schreiber Guus // World Wide Web Consortium. - W3C, 06 15, 2004. - 01 2007. - http://www.w3.org/2001/sw/WebOnt/.

9 Appendix – Technical Description of the Visualization Application Files

9.1 JSP Pages

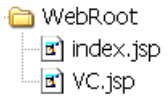

Figure 18: JSP Pages

The Visualization Application is started by calling index.jsp.

9.1.1 Index.jsp

Figure 8 shows the simple form drawn by this file. It creates a connection to the File Manager Client and alphabetically lists the graph files that can be displayed by the component. The form is then submitted to VC.jsp.

9.1.2 VC.jsp

This page contains the applet, however, it also does some important file processing needed to compensate for the applet restrictions (see Section 4.3.1.1). It copies the two input files for the applet – the GraphML graph file selected by the user in index.jsp from the File Manager Client and the .owl ontology file referenced by this graph file from the Ontology Manager Client – from the VIKEF storage space (see Figure 17) to the local tomcat directory (specified in Cs.java) on the server, where the Visualization Application is running. This is the only directory containing input files that the applet can access. After the files are copied, the applet is started and the filenames of the two temporary files are passed to it as arguments.

9.2 Java Sources

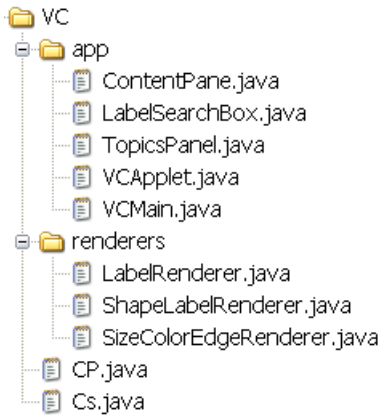

Figure 19: Java Sources

9.2.1 Widgets

The `VCApplet` class is the applet derivative class that is used to start the Visualization Application. It calls the constructors of the other classes initializing the Panels with the widgets and is responsible for initializing the proper pair of displays depending on the Display Mode. For Display Mode 3 it is responsible for creating a Visualization, initializing the renderers and then passing this Visualization to the constructors of VCMain and ContentPane for the creation of the two displays. For Display Modes 1 and 2 it passes the necessary data to the two constructors. This allows them to create two different Visualizations for the two displays.

9.2.1.1 VCMain.java

This class is used to display the Main Display Window. It has two constructors called by `VCApplet` depending on the Display Mode. The method `prepareVis()` is invoked by the constructors and implements the prefuse logic for the initialization of a Display – Actions – drawing of the graph elements, different colors and highlights – Layouts, Panel size, Listeners, etc. It should be noted that in the case of Display Mode 3 what this method does also applies to the Overview Window.

This file also contains a small class – TreeRootAction – reused from a prefuse demo for repositioning the nodes in accordance with the Radial Tree Layout after a node has been clicked on.

9.2.1.2 LabelSearchBox.java

```
/**
 * This is a small class that extends Box adding to it the widgets that are used in the VC.
 * These widgets include showing the label of the element the mouse is currently pointing at and
 * Simple search for searching a node according to its label.
 * The search and label are implemented by reusing code from prefuse demos.
 * @author The Jimmytaker
 *
 */
public class LabelSearchBox extends Box {
```

Figure 20: Class initializing the widgets Label Display and Search Box

This class has two methods initializing each of the two widgets.

9.2.1.3 ContentPane.java

This class represents the Content Pane. It contains the Overview Window, the Legend, the Force-Directed Layout button and the Node Data widgets. It has two constructors invoked by `VCApplet` depending on the Display Mode in use.

For Display Mode 3 the Visualization already initialized and set up by `VCApplet` and `VCMain` is used for the Display of the Overview Window. In this case refer to `VCMain` for the Actions started and operating. For Display Modes 1 and 2 the method `prepareVis()` initializes the Actions and Layouts to be used and is only invoked in these two modes.

The method `prepareDisp()` is invoked by the constructors and sets up the Listeners associated with the Display of the Overview Window.

```
/** Prepares the widged that displays the Legend. This method does not use Cs for hardcoded Strings.
 * The reason is that the strings used in this method are used only here and were defined for usage here.
 * Abstracting over them would cause more complication. One should better leave this Strings untouched
 * both here and in the Factory. */
void prepareLegend() {
```

Figure 21: Method preparing the Legend widget

The method `prepareLegend()` does serious logic processing to extract the data for the creation of the Legend from the format it is stored in. The complexity of this method can be decreased significantly if the approach described in Section 4.3.3 is followed.

This file also includes a small class – `ColumnTitle` – used in the creation of the Legend. This class will most probably be dropped if the approach of creating the legend is changed as described above.

The method `addButton()` adds the Force-Directed Layout button to the Content Pane and sets up its functionality.

The method `prepareNodeData()` adds the last widget displaying the Node Data to the Content Pane.

9.2.1.4 TopicsPanel.java

```
/** This class represents the topics panel in the JTabbedPane.
 * This panel extracts the Topics hierarchy with name the name selected by the user in the SVEEditor from
 * the ontology and represents it as a directory structure.
 * Clicking on an element in this structure causes all nodes (and their corresponding edges) that have a
 * topic property defined to disappear from the graph. No changes are applied to nodes that do not have
 * the topic property defined.
 * @author The Jimmytaker
 *
 */
public class TopicsPanel extends JPanel {
```

Figure 22: Class creating the Topics Tree

This class is responsible for the creation and the functionality of the Topic Tree Tab. This class attempts to extend the functionality Jena offers by extending a relationship between two elements (in this case the "subclass" relationship) to a transitive relationship. This is achieved through a recursive method that is used to extract the necessary data from the ontology and keeping a record of this data in the map shown in Figure 23.

```
/**this maps each topic name to a list of its subtopics. This is needed to allow for all the nodes
 * that have a topic that is a subelement of the selected element to remain visible. */
LinkedHashMap<String, ArrayList> topics = new LinkedHashMap<String, ArrayList>();
```

Figure 23: Topics map

```
/** The valueChanged method of the listener is invoked when a new element in the tree is selected.
 * @author The Jimmytaker
 *
 */
class TSListener implements TreeSelectionListener {

    public void valueChanged(TreeSelectionEvent arg0) {
```

Figure 24: Tree Selection Listener

This file also contains the small class TSListener. Every time a new node in the Topics Tree is selected, all items are set visible and then in a while loop over all node items it is checked whether these items should remain visible. An item should remain visible if it has no topic attribute defined (seeFigure 16: GraphML Node and Edge Display ExtractFigure 16) or if its topic (see addresses-area-of-interest in Figure 13) is the same as or a sub-element of the selected topic in the Topics Tree. The map shown above contains a quick reference of the nodes that should be visible for each topic (see also Figure 25). When a node is found that should not be visible, its visibility field and the visibility fields of all adjacent edge items are set to "false".

```
/** this checks whether the current node is the same as the selected node
 * or its subelement */
boolean isElementOf(String nodet, String selectedt) {
    if (topics.get(selectedt).contains(nodet)) {
        return true;
    }
    return false;
}
```

Figure 25: Check visible

9.2.2 Renderers

New renderers were needed for both the nodes – to support shapes/images with labels – and the edges – to support different colors and sizes. Both these were built around the sources of the already existing prefuse renderers, however the changes/extensions are not just superficial.

9.2.2.1 LabelRenderer.java

```
/** The only change this class introduces to the original
 * prefuse LabelRenderer is that the maximal length
 * of a node label is 20 chars.
 * The reason for the existence of this class is that
 * it is the renderer used in ContentPane
 * @author The Jimmytaker
 *
 */
public class LabelRenderer extends prefuse.render.LabelRenderer {

    public LabelRenderer(String label) {
        super(label);
    }

     /** To keep the labels readable and the graph looking good, limit labels to 20 chars. This method
     * is invoked in getRawShape. */
    protected String getText(VisualItem item) {
        String s = null;
        if ( item.canGetString(m_labelName) ) {
            s = item.getString(m_labelName);
        }
//      maximal label length - 20
        if (s.length() > 20) {
            s = s.substring(0, 20);
        }
        return s;
    }
```

Figure 26: LabelRenderer

See the comments in the code of the file.

9.2.2.2 ShapeLabelRenderer.java

```
/**
 * This class is built around prefuse's LabelRenderer and ShapeRenderer.
 * In the current application it is needed to mark each node with its label and a visual element,
 * which might be a shape or an image. What is used is determined in the graphml input file.
 * @author The Jimmytaker
 *
 */
public class ShapeLabelRenderer extends LabelRenderer {
```

Figure 27: New Node Renderer

In the implementation of this class the needed functionality is achieved by overriding the two main methods for the creation of node elements and adding new methods to support the new features.

```
/**
 * If the display pair in this graph (as defined in the SVEEditor) should include
 * invisible nodes, when a VisualItem is discovered that should not be displayed,
 * its start visibility attribute is set to false, so that the place of the node is kept in the shape
 * of the graph but it is not displayed. This is achieve by the VisibilityFilter applied.
 * Otherwise, if the invisible elements should be dropped, null is returned.
 *
 * Further changes include:
 * - this method no longer handles images, since some kind of shape is expected to be
 * always present.
 * - the boxes are now simpler. They are always the same (no RoundRectangle2D).
 * @see prefuse.render.AbstractShapeRenderer#getRawShape(prefuse.visual.VisualItem)
 */
protected Shape getRawShape(VisualItem item) {

    if (m_displayType != 1 && !isVisible(item)) {
        return null;
    }
    //else
    this is the case 1 - invisible elements for the elements that should be filtered out
    if (!isVisible(item)) {
        item.setStartVisible(false);
    } else {
        item.setStartVisible(true);
    }
        m_text = getText(item);
        int size = getSize(item);
```

```
/**The new look of the nodes is achieved mainly by overriding this method.
 * Changes:
 * - no fill - the nodes are always white and include a graphic and text
 * - the recognition of the type of shape or image happens here
 * - not only images but also shapes get drawn here
 * @see prefuse.render.Renderer#render(java.awt.Graphics2D, prefuse.visual.VisualItem)
 */
public void render(Graphics2D g, VisualItem item) {
    RectangularShape shape = (RectangularShape)getShape(item);
    if ( shape == null ) return;


//create the proper shape or an image. Not all of the cases can happen, as some of these
//shapes are left out of the SVEEditor
int stype = getShap(item);
switch (stype) {
case Constants.SHAPE_NONE:
    //this is an image
    img  = getImage(item);
    break;
case Constants.SHAPE_RECTANGLE:
    //used
    sshape = sr.rectangle(x, y, width, width);
    break;
case Constants.SHAPE_ELLIPSE:
    //used - represents a circle.
    sshape = sr.ellipse(x, y, width, width);


     //fill the shape with the proper color
     if (stype != Constants.SHAPE_NONE) {
         item.setFillColor(getColor(item).getRGB());
         GraphicsLib.paint(g, item, sshape, getStroke(item), RENDER_TYPE_FILL);

         //add an image - code reused from super method
     } else {
```

Figure 28: Interesting code abstracts from the two main drawing overridden methods

```
//new methods
/** Passed a VisualItem as a parameter, this method attempts to read its shape field and return
 * the integer code of the found basic shape as defined in prefuse.Constants.
 */
int getShap(VisualItem item) {
    String s = null;
    if ( item.canGetString(m_shapeName) ) {
        s =  item.getString(m_shapeName);
        if (s.toLowerCase().equals("rectangle")) {
            return Constants.SHAPE_RECTANGLE;
        } else if (s.toLowerCase().equals("circle")) {
            return Constants.SHAPE_ELLIPSE;

/** Passed a VisualItem as a parameter, this method attempts to read its size field and return
 * the number found there. This number has been defined as a visual setting in the SVEEditor.
 */
int getSize(VisualItem item) {
    if (item.canGetString(m_sizeName)) {
        return Integer.parseInt(item.getString(m_sizeName));
    }

/** Passed a VisualItem as a parameter, this method attempts to read its color field and return
 * the corresponding Color to the string found there.
 * This color has been defined as a visual setting in the SVEEditor.
 */
Color getColor(VisualItem item) {
    if (item.canGetString(m_colorName)) {
        String c = item.getString(m_colorName);
        if (c.toLowerCase().equals("black")) {
            return Color.black;
        } else if (c.toLowerCase().equals("blue")) {
            return Color.blue;

/** Passed a VisualItem as a parameter, this method attempts to read its visibility field and return
 * a boolean value of whether this node should be displayed. Nodes can be chosen not to be
 * displayed in the main window in the SVEEditor.
 */
boolean isVisible(VisualItem item) {
    if (item.canGetString(m_visName)) {
        return item.getString(m_visName).equals(Cs.pos);
    }
```

Figure 29: New methods

```
//overriden methods
/**The only change in this method is from private to protected. It is originally declared private,
 * and therefore cannot be reused!
 */
protected String computeTextDimensions(VisualItem item, String text,
        double size) {

    /** The only change here is that the images used are given with relative path.
     * The used images are coming with the VC and stored in the images folder.
     * The super method expects full path.
     */
    protected String getImageLocation(VisualItem item) {
        return item.canGetString(m_imageName)
                ? "/" + item.getString(m_imageName)
                : null;
    }

/** The only change is from private to protected. Super method could not be reused! */
protected final void drawString(Graphics2D g, FontMetrics fm, String text,
        boolean useInt, double x, double y, double w)
{
```

Figure 30: Overridden methods

The signature of a couple of methods has been changed from private to protected, as these are declared as private in the prefuse jar and therefore cannot be accessed by extending classes. This is the only feature of prefuse that was found to be implemented badly and should be changed in a future prefuse release.

Getter and Setter methods for the GraphML fields are also included to make the new class more portable and easy to reuse.

```
//elements of the graphml file used thoughout this class. Note that shape and image are the same
//element in this application as it is either or, what is going to be used, but still it is
//a good idea to be able to make difference between them, as they are used in a different way
protected String m_shapeName = "Shape";
protected String m_colorName = "Color";
protected String m_sizeName = "Size";
protected String m_imageName = "Shape";
protected String m_visName = "visible";

int m_displayType;
```

Figure 31: GraphML elements used throughout the class

9.2.2.3 SizeColorEdgeRenderer.java

```
/**
 * In the current application the different types of edges need to have different colors and sizes.
 * This is achieved in this extension of EdgeRenderer
 * @author The Jimmytaker
 *
 */
public class SizeColorEdgeRenderer extends EdgeRenderer {
```

Figure 32: SizeColorEdgeRenderer

In the implementation of this class the needed functionality is achieved by overriding the two main methods for the creation of node elements and adding new methods to support the new features. However, the needed changes are much fewer and much more superficial than in ShapeLabelRenderer.

```
//overriden methods
/** Not all edges should be visible at all times. Return null unless the edge SHOULD be visible */
protected Shape getRawShape(VisualItem item) {
    if (isVisible(item)) {
        return super.getRawShape(item);
    }
    return null;
}

/**
 * Do nothing if the edge should not be visualized in the main display window.
 * Otherwise invoke the super method for rendering an edge but changing the input
 * so that the edge is drawn in the desired way.
 * The stroke and fill color for the edge are set to the value found in the graphml input file
 * and the getLineWidth method is overriden to return the appropriate value from the graphml file
 * prior to invokation of the super method.
 */
public void render(Graphics2D g, VisualItem item) {
    if (isVisible(item)) {
        //read the graphml file
        int color = getColor(item).getRGB();
        item.setStrokeColor(color);
        item.setFillColor(color);
        // render the edge line
        super.render(g, item);
        // render the edge arrow head, if appropriate
        if ( m_curArrow != null ) {
            g.setPaint(ColorLib.getColor(item.getFillColor()));
            g.fill(m_curArrow);
        }
    }

/**
 * This method is used in EdgeRenderer.getRawShape, which is invoked from within its overriding method
 * in this class. It delivers the size for the current edge chosen by the user in SVEEditor
 * @param item the VisualItem for which to determine the line width
 * @return the desired line width, in pixels
 */
protected double getLineWidth(VisualItem item) {
    int s;
    if (item.canGetString(m_sizeName) && item.getString(m_sizeName) != null) {
        s = Integer.parseInt(item.getString(m_sizeName));
        return item.getSize()*s;
    } else {
        return item.getSize();
    }
```

Figure 33: Overridden methods

```
//new methods
/**
 * This method gets the decodes the string representation of the desired color (as defined in the
 * graphml file) to a Color object.
 */
protected Color getColor(VisualItem item) {

        if (item.canGetString(m_colorName)) {
            String c = item.getString(m_colorName);
            if (c.toLowerCase().equals("black")) {
                return Color.black;
            } else if (c.toLowerCase().equals("blue")) {
                return Color.blue;
```

```
/** This method determines whether an edge should be visualized in the main display window.
 * The criteria include both:
 * 1. The edge has not been explicitly chosen not to be displayed in the SVEEditor
 * 2. Both the source and target nodes of this edge should be displayed.
 */
boolean isVisible(VisualItem item) {
    EdgeItem   edge = (EdgeItem)item;
    VisualItem item1 = edge.getSourceItem();
    VisualItem item2 = edge.getTargetItem();
    if (item.canGetString(m_edgeVisName) && item1.canGetString(m_nodeVisName)
        && item2.canGetString(m_nodeVisName)) {
        return item.getString(m_edgeVisName).equals(Cs.pos) && item1.getString(m_nodeVisName).equals(Cs.pos)
            && item1.getString(m_nodeVisName).equals(Cs.pos);
    }
```

Figure 34: New methods

Getter and Setter methods for the GraphML fields are also included to make the new class more portable and easy to reuse.

```
//  elements of the graphml file used thoughout this class.
    protected String m_sizeName = "EdgeSize";
    protected String m_nodeVisName = "visible";
    protected String m_edgeVisName = "visibleEdge";
    protected String m_colorName = "EdgeColor";

    int displayType;
```

Figure 35: GraphML elements used throughout the class

9.2.3 Other Sources

```
/**
 * This class contains all the strings used throughout
 * the VC and assigns const names to them
 * @author The Jimmytaker
 *
 */
public class Cs {


   /**
    * This is a small class to help in debugging.
    * Its sole purpose is to provide easy to turn on
    * and off console printing
    * @author The Jimmytaker
    *
    */
   public class CP {
```

Figure 36: Other Sources