Support for User-friendly Customization of Knowledge Network Visualization

WS 2006/2007

Avaré Stewart

Dr. Claudia Niederée

Fraunhofer IPSI

Prof. Dr. Techn. Johannes Fürnkranz

TU Darmstadt, Department of Computer Science

Bachelor Thesis

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 user-friendly definition and adaptation of such Visual Features a special Editor will be built. It is the task of this Bachelor thesis to design and implement two components for this purpose – the Semantic Visualization Editor and the Semantic Visualization Factory. The design and the development of these will be described in the written part of the Bachelor thesis.

The Semantic Visualization Editor offers a user friendly interface that enables the user to specify and adjust the Visuals Features used for the visualization of an SRN View. The Editor provides a working environment in a wizard that leads the user step-by-step through the process of gathering the required preferences. Among the features that can be exploited and adapted for bringing semantics to the visualization are: size; shapes; colors; human understandable labels; etc.

For applying the selected visual features to the targeted SRN View the Semantic Visualization Factory is used. The Semantic Visualization Factory accesses an SRN View and applies the Visual Settings to it. The result of this application is the targeted standard-based visual counterpart of the SRN View that can be visualized with an appropriate visualization component.

The aforementioned components work in collaboration with another set of components: an SRN View Definition Editor, an SRN View Factory (Toshev, 2007) and a Visualization Application (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).

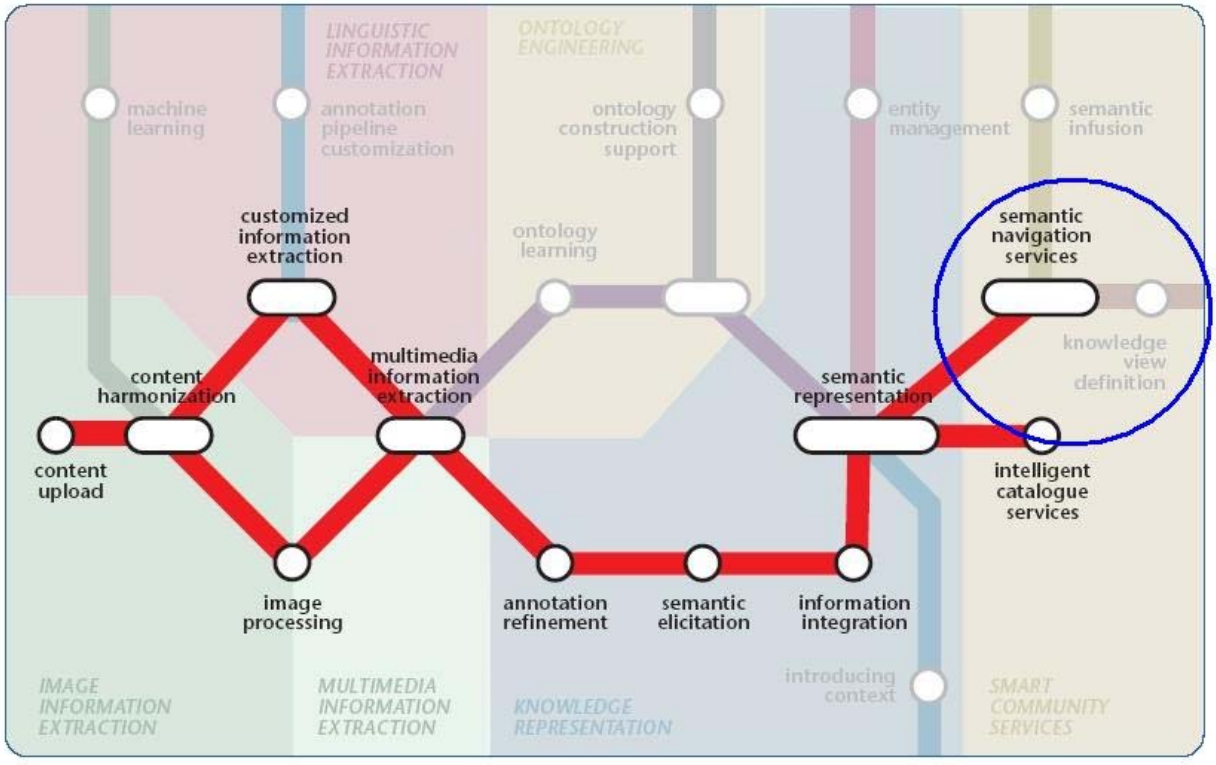

Figure 1: VIKEF Pipeline Overview

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 Creation phase.

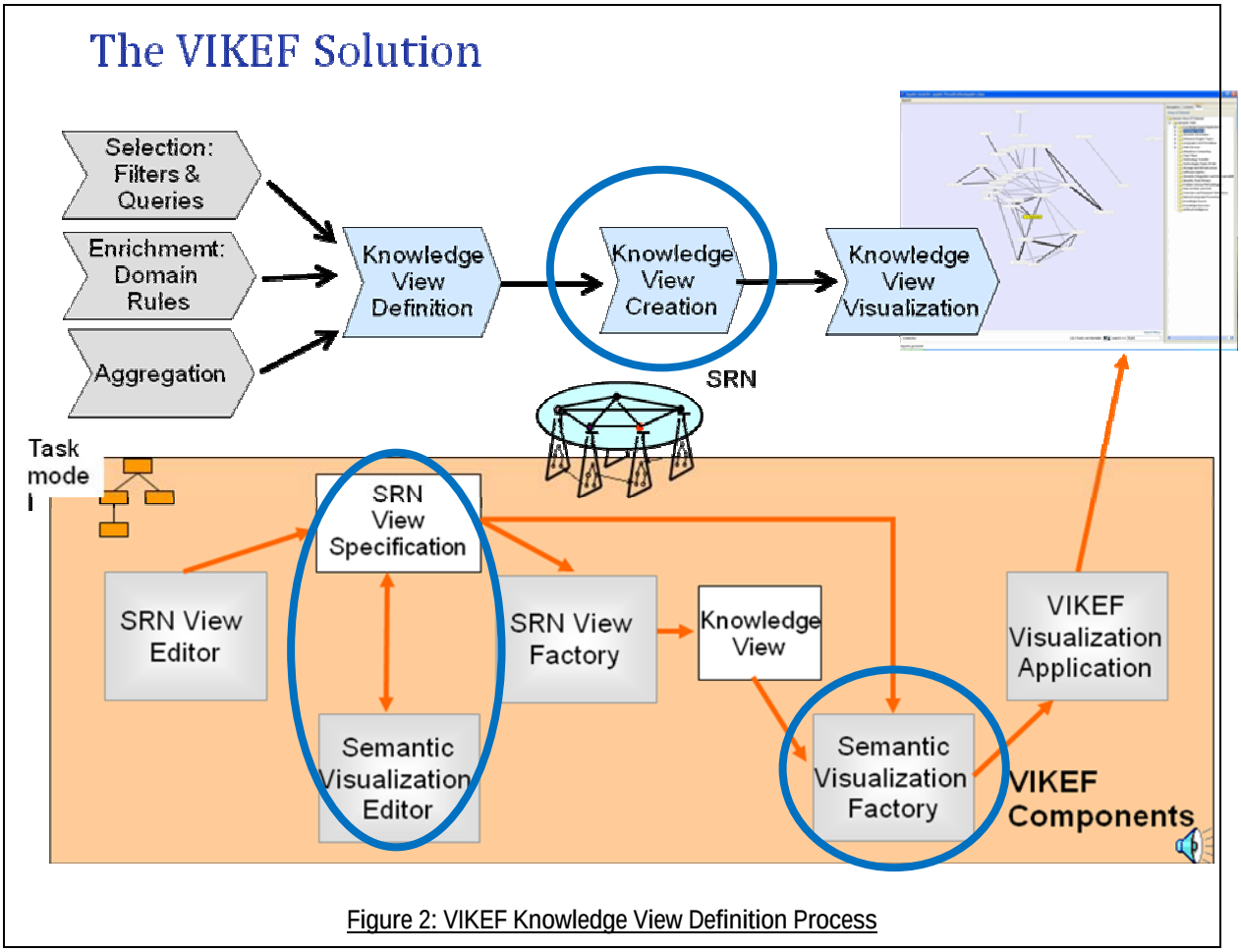

Figure 2: VIKEF Knowledge View Definition Process

1.2 Problem – Customization Support in a Complex Setting

In VIKEF the user should not only be supported in the use of such Knowledge Views, but also in their creation. This should be achieved by providing user-friendly tools.

In the current work the following issues are addressed:

1. Providing flexibility for different types of users. Many different types of users with different levels of expertise (e.g. knowledge engineers, ontology experts, conference organizers, conference participants) play a role in the VIKEF pipeline and Knowledge View Definition Process. They are expected to differ in their expertise and therefore it is important to provide a tool that can support all different types of users.

2. Providing support for different SRN Views. Many different views can be created in the Knowledge View Definition Phase. A key problem is to be solved by this work is to provide flexibility for adapting a different look and feel for each of the created Views.

3. Providing support for adapting the resulting Visual Features. For the same View, a different emphasis may be needed depending on the size of the data, the different requirements to the View by the user creating it or the end user preferences. Different details of a View may need to be emphasized at for different reasons. For example, for the same view, it may be important to show a subset of the View (based on range of years) in a different color and for the same view to later emphasize the size of the nodes in this range instead. These variations depend on the user's preferences and current task at hand.

4. Providing a means for handling the Information Overload. The previous steps of the VIKEF knowledge supply chain create and integrate a rich knowledge space, which is interlinked with the underlying content. This is a valuable source of information for members of scientific and business communities. A direct explorative access to this knowledge space will most probably, however, overwhelm the user with information. Figure 3 displays two graphs featuring the same data. However, the first graph displays the full knowledge space and the second one filters out and displays only the nodes of the relevant type.

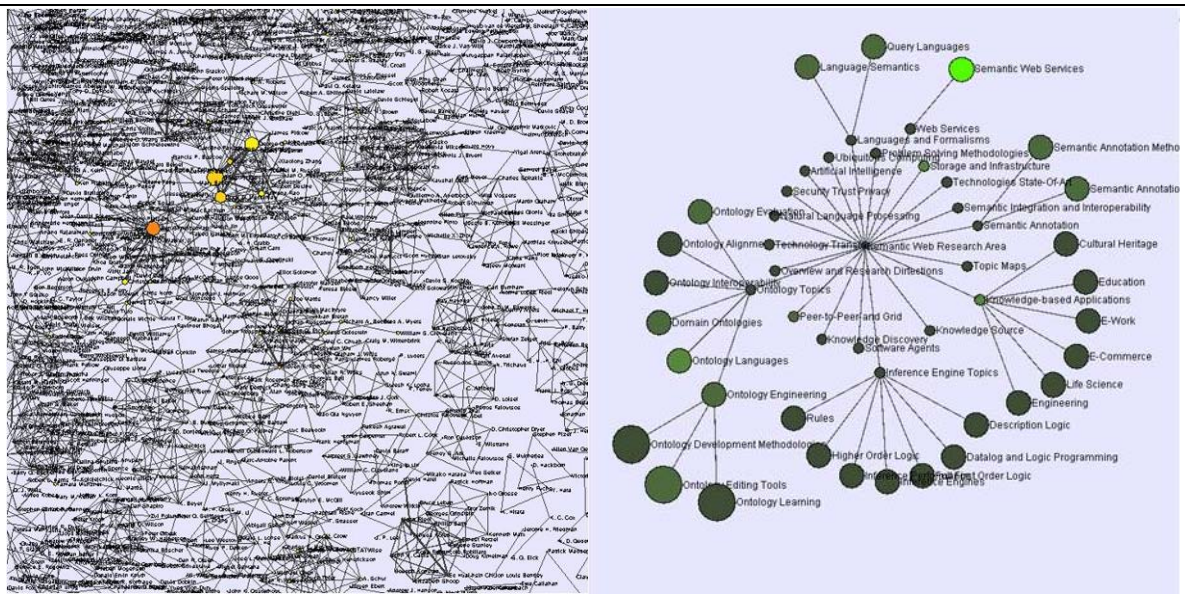

Figure 3: Different Views: Showing Everything and Filtering only the Relevant Information

In order to address these issues and the user-friendly definition and adaptation of such Visual Features a special editor will be built. It is the task of this Bachelor work to design and implement two components for this purpose – the Semantic Visualization Editor and the Semantic Visualization Factory. The design and the development of these will be described in the written part of the Bachelor work.

The Semantic Visualization Editor offers a user friendly interface that enables the user to specify and adjust the Visuals Features used for the visualization of an SRN View. The Editor provides a working environment in a wizard that leads the user step-by-step through the process of gathering the required preferences. Among the features that can be exploited and adapted for bringing semantics to the visualization are size, shapes, colors, human understandable labels, etc.

For applying the selected visual features to the targeted SRN View the Semantic Visualization Factory is used. The Semantic Visualization Factory accesses a SRN View and applies the Visual Settings to it. The result of this application is the targeted standard-based visual counterpart of the SRN View that can be visualized with an appropriate visualization component.

1.3 Structure of this Work

This paper describes the VIKEF components that allow the user to configure how the Visualization looks.

Section 2 lists the components of the project and their purpose.

Section 3 describes the Semantic Visualization Editor. It presents the design decisions taken while creating it and lists the functionality this editor offers its user. It also presents the format the data is stored in and how the Editor collaborates with the Semantic Visualization Factory.

Section 4 describes the implementation process of the Semantic Visualization Components: the technologies employed; a review of the different releases; the architecture of the components and the way the data is parsed and stored. It also describes the data formats used.

Section 5 attempts to evaluate the last (fifth) 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 thesis.

Three appendices are included to describe the technical aspects of the development:

Section 9 (Appendix A) explains the format used to transfer the settings from the Editor to the Factory and the file naming convention used to ensure compatibility.

Section 10 (Appendix B) describes the Editor files.

Section 11 (Appendix C) describes the Factory files.

2 VIKEF Knowledge View Creation

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 (Herman, et al., 2007) 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. The focus of this work is the Semantic Visualization Editor.

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. The created visual specification is inserted into the SRN View specification.

The Editor is complemented by a factory component, which applies the settings and specifications defined with the Editor. The Semantic Visualization Factory prepares the Knowledge View for visualization.

It transforms the Knowledge View into a format, which can be consumed by the respective Visualization Application. In this transformation the selected visual features are applied. These features were defined with the Semantic Visualization Editor and stored as part of the SRN View Specification.

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 the Knowledge View. This includes searching, browsing, and inspecting detail information. For further information concerning the VIKEF Visualization Application refer to (Nikolov, 2007).

3 Semantic Visualization Editor Design

3.1 Overview of the Design Process

In the current Semantic Visualization Editor an important decision influencing the design and implementation was the technology that should be used for creating the Editor and what aspects should be configurable. The issues related to the choice of technology are discussed first, followed by the requirements and issues related to the editor's design.

3.1.1 Issues Related to Technology

There were three different technologies considered in the implementation:

1. JSP / Servlets
2. .NET - ASP
3. HTML with Java Applet

Option 3 was overthrown as the comfort it would offer during implementation was not enough in exchange for the web-based server-side processing of the data in the Editor.

The decision to use Option 1 – JSP (JSP06) over Option 2 – ASP was motivated by ease of integration. First because Java was the preferred language used in the project and second the use of Option 1 would allow easy integration with other components such as the Semantic Infusion Component also developed within VIKEF. Additionally the time needed to get the prototype up and running was crucial.

The first release deadline was set just over a month after the project was started. Time prior to the start of the project to get acquainted and comfortable with the web technology used for implementation was very brief. The learning curve had to be as simple as possible and work done quickly. Providing the necessary functionality from the start had a higher priority than employing high-end Web Frameworks like Struts or JSF that allow clearer coding style and much better separation of logic from content.

In the early stages of the project mixing Server-side JSP/Servlet programming and Client-side JavaScript programming was experimented with. It was desired to be able to create dynamic changing menus de-

pending on the value chosen in some other menu without using form submittal. However, this approach turned out to be naïve and was abandoned for a number of reasons:

1. All the data had to be processed twice and transferred between the two languages.
2. JavaScript does not offer a great part of the functionality that comes with Java and this would have to be implemented as well
3. The design of the web pages was not in its final state and each requirement change came with numerous changes to the code
4. It would be time consuming (about four times longer) to implement certain features.

3.1.2 Requirements and Issues Related to Page Design

In contrast to choosing the technology, the final design of the pages required several iterations on its requirements. In some cases, elements changed significantly from the initial idea. One example is reflected in the design of the edge labels.

As an initial requirement, all edges were supposed to be drawn black and the user was supposed to choose an edge label similarly to the way node labels are chosen or write the label himself. This functionality was replaced by edge colors for a few reasons:

1. It is very improbable that there would be so many differently labeled edges as nodes in a given dataset. Each node represents a resource instance. Edges represent relationships between instances. Having more relationships between instances than the number of instances there are is not feasible.
2. Giving each node in a big graph a label already results in a Visualization Graph with a lot of labels in it. Adding edge labels to each edge makes the graph look bad and the readability drops.
3. It was initially thought that implementing labels was easier than colors, and offering both options for the edges was considered too much. However, as work on the Visualization Application (Nikolov, 2007) began, it turned out that this assumption was false.

Only one problem arose as a result of using colors instead of labels for the edges. In a graph with more than a couple of different edge types, it could easily become unclear what edge type each color represents. The solution to this and similar problems (e.g. becoming unclear what some settings represent if many nodes and filters are used) is including a legend listing the settings made in the Semantic Visualization Editor. The data for this legend is gathered by the Semantic Visualization Factory

and stored in its output file in a format that is optimized for reading by the Visualization Application. This is described in Section 4.4. In this and a few other cases the different components successfully collaborate to fix a problem.

3.1.3 Exemplified Design Tradeoffs

Some tradeoffs were faced during the design. One example is represented with the use of Profile Names. Initially, when an SRN View Specification File was processed with the Semantic Visualization Editor, it was not overwritten but a new file – containing an SRN View Specification with Visual Settings was created. Figure 2 shows that both the SRN View Editor and the Semantic Visualization Editor store data in the same file. This is described in detail in Section 3.3. The Semantic Visualization Editor augments its input file every time it is completed. In addition to the Pretty Name (file name given by the user in the SRN View Editor) a (Visualization) Profile Name was given to the newly created file. For simplicity and integration reasons, it was considered more important to have a single view name represent one view. There is some tradeoff limiting this approach – such as losing the ability to save different Visualizations (containing different colors, nodes, filters, etc.) for the same View.

Limitations to the current solution are namely:

1. Settings previously stored in the input file might easily get irrecoverably lost. Imagine the following scenario: 1. Running the Semantic Visualization Editor and creating numerous filters for different Resource Types. (See Section 3.2 for explanation what these are). 2. Running the Semantic Visualization Editor deselecting some Resource Types. While going back within one run of editor and deselecting some Resource Types after creating filters for these does not affect the filters and these will reappear if the Resources are selected by going back once again, this is not true AFTER the file is stored. Filters (and settings altogether) for nodes selected not to appear, are not stored in the output file.

2. When trying to run the Semantic Visualization Factory without running the Semantic Visualization Editor in advance a problem occurs. This is possible because, at the moment the factory is run, there is no way to know whether an SRN View Specification File (See Figure 2), which is one of the files used by the factory as input, has Visualization data in it. However, once the factory is run, it detects the missing Visualization data and produces an error message. This error message could be avoided by using some default settings. However, the requirements explicitly prohibit making any assumptions on details that are configurable by the user. Therefore it is illegal (but possible) to run the Semantic Visualization Factory before assigning Visualization data to the View using the Semantic Visualization Editor.

3. The above problem might occur even if the user has completed the Semantic Visualization Editor. Imagine the following scenario: 1. SRN View Editor, 2. Semantic Visualization Editor, 3. SRN View Editor just reviewing (not changing anything) but still completing (file being stored). The last step would cause the SRN View Specification File to be overwritten without Visualization data. This could have bad consequences as it could easily lead to data loss.

If Profile Names were kept 2. and 3. would not appear at all and 1. would be quite rare as Profile Name would be selected in the same step, where the Resource Types for visualization are chosen.

The reason this functionality was dropped was the file naming convention is too strict. Different components rely on the already introduced file naming convention and since exploiting Profile Names requires changing it, this setting had to be dropped. See Section 9.2 for information on the file naming convention in use.

Apart from this tradeoff the Semantic Visualization Components are an invaluable and robust tool built for simplicity and ease of use in tackling a part of a complex task.

3.1.4 More on Requirements

After defining the logic and structure of the Knowledge View an adequate visualization of this View is required. A wide variety of possible visualizations should be possible for a Knowledge View depending on the selected visualization metaphor and the associated Visual Features. Furthermore it is required that the Semantic Visualization Editor has the ability to author the widgets of the Visualization Application. This requirement introduced important design issues for both Semantic Visualization Components (Editor and Factory).

However, functionality and usability are not everything that is necessary for the Editor to be a success. Its appearance is important too. The look and feel is created using the same style as the other VIKEF Components and especially the SRN View Editor, which the Semantic Visualization Editor integrates with.

The next section describes each of the current steps (elements) of the Editor as well as motivation behind them.

3.2 Specification of the Semantic Visualization Editor

This section describes the Semantic Visualization Editor in detail. It reviews each page and the elements that can be customized. There are three different tasks that the Editor completes:

1. Setting up the appearance of nodes and edges.

2. Setting up filters on the different nodes.

3. Customizing the functionality of some of the widgets of the Visualization Application.

3.2.1 Editor Page Listing

The Editor has 5 pages (Figure 4) and 8 steps overall. Each of the steps contributes to one of the three tasks. The pages are listed here and will be referred to throughout the remainder of this section.

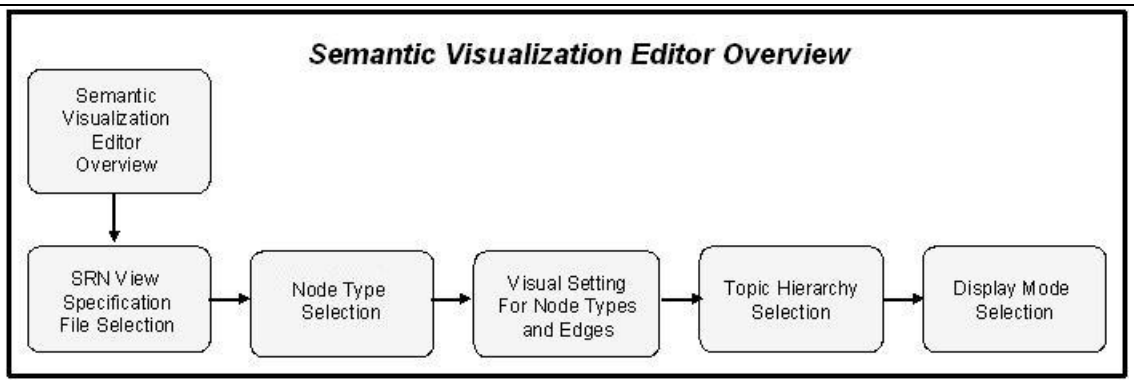

Figure 4: Semantic Visualization Editor Overview

Figure 5: Page 1 – Step 1 – shows the input file selection

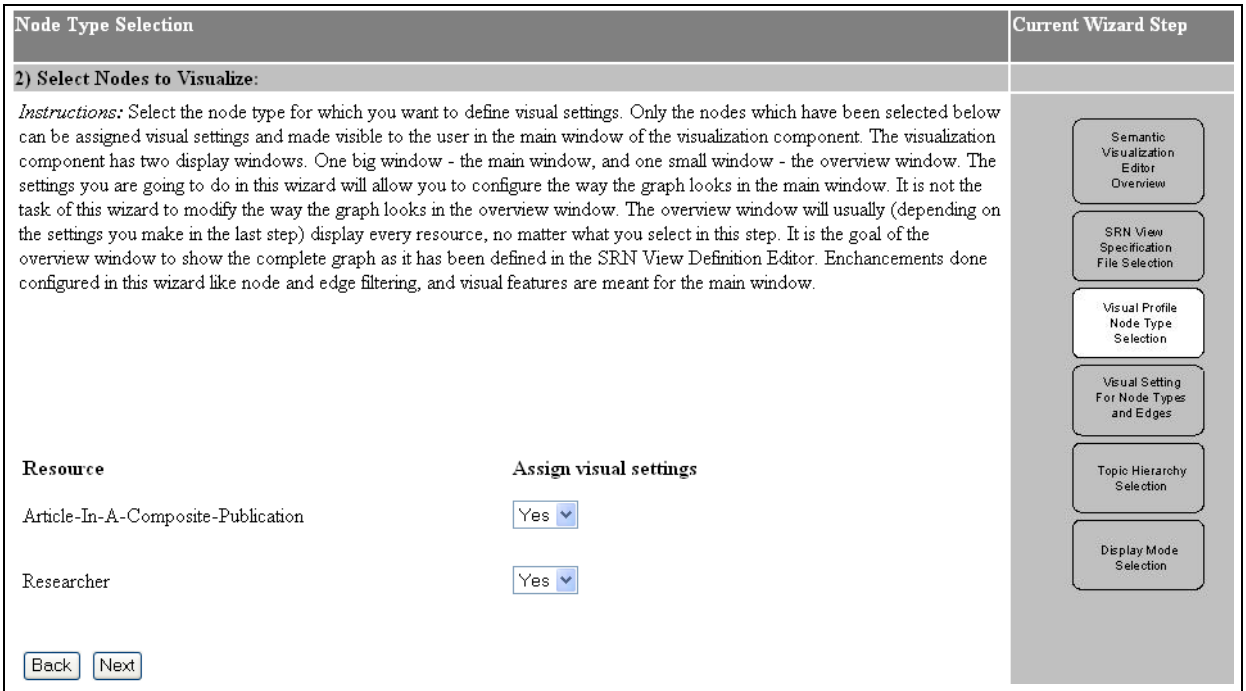

Figure 6: Page 2 – Step 2 – Allows the user to choose nodes to visualize

Figure 7: Page 3 – Step 3 – Visual Settings for Nodes and Step 4 a) – Filters

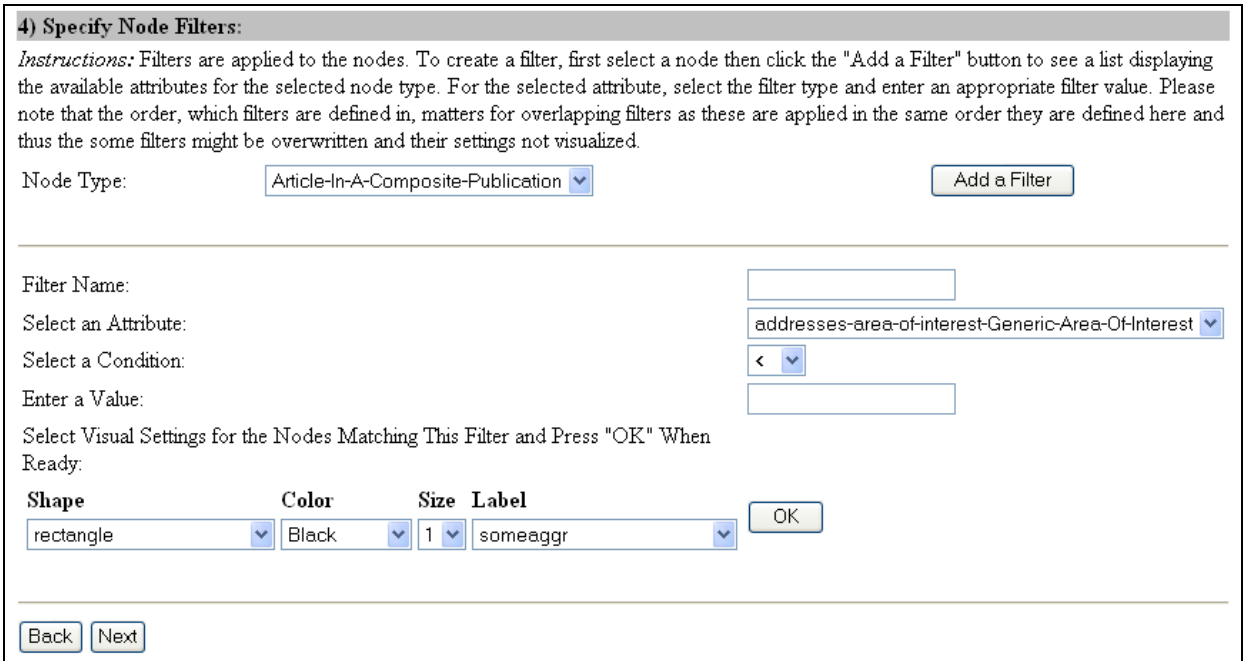

Figure 8: Page 3 – Step 4 b) – Filter definition

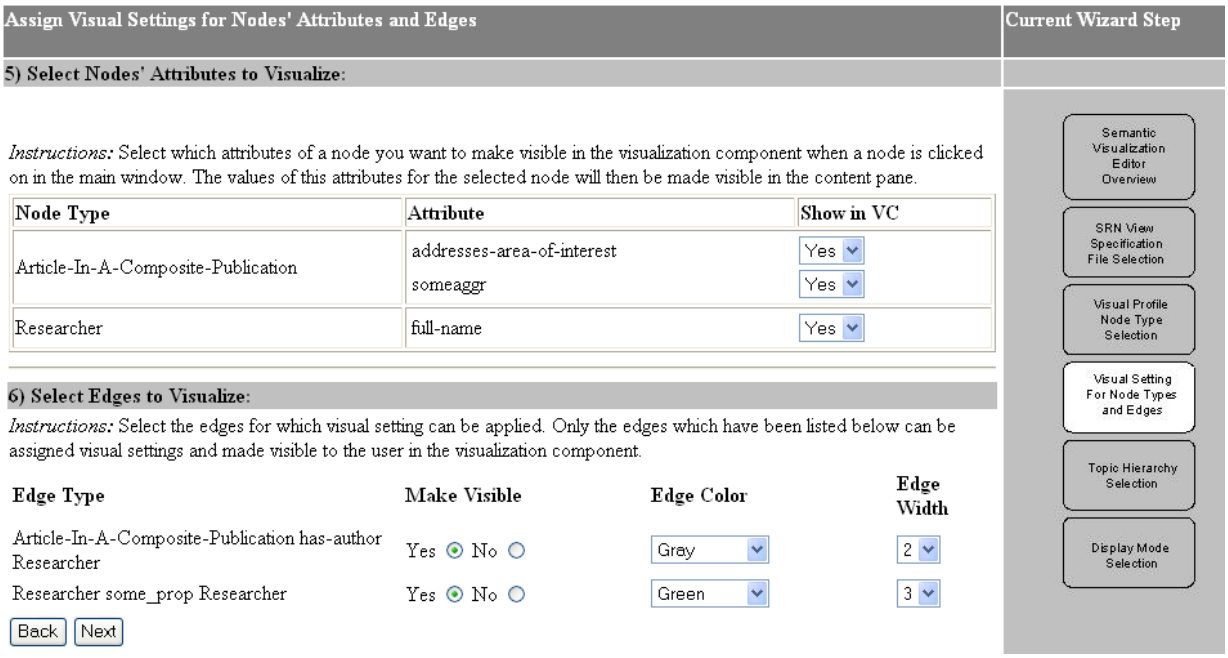

Figure 9: Page 4 – Step 5 – Node Data widget of the Visualization Application authoring and Step 6 – Visual Settings for edges

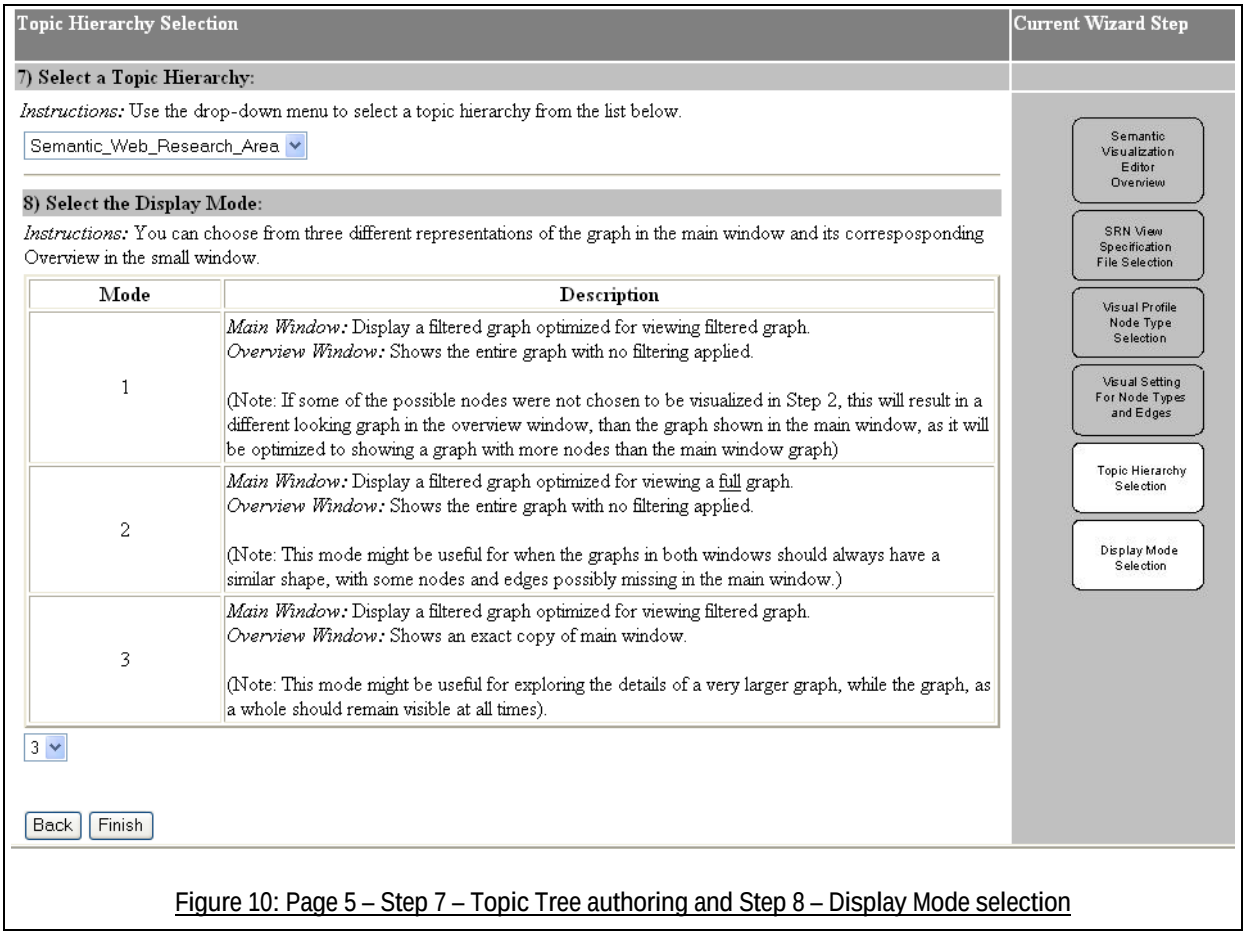

Figure 10: Page 5 – Step 7 – Topic Tree authoring and Step 8 – Display Mode selection

3.2.2 Appearance

There are two main elements in a graph – nodes and edges. The nodes in our graphs represent instances of resource elements (see Figure 6). The edges represent relationships between the resources (see Figure 9).

3.2.2.1 Node Appearance

Figure 7 shows that in the current example there are two different node types. We want to allow the user to define how each of these node types looks in the graph. Currently there are four different visual settings that can be customized: Shape/Icon, Color, Size and Label.

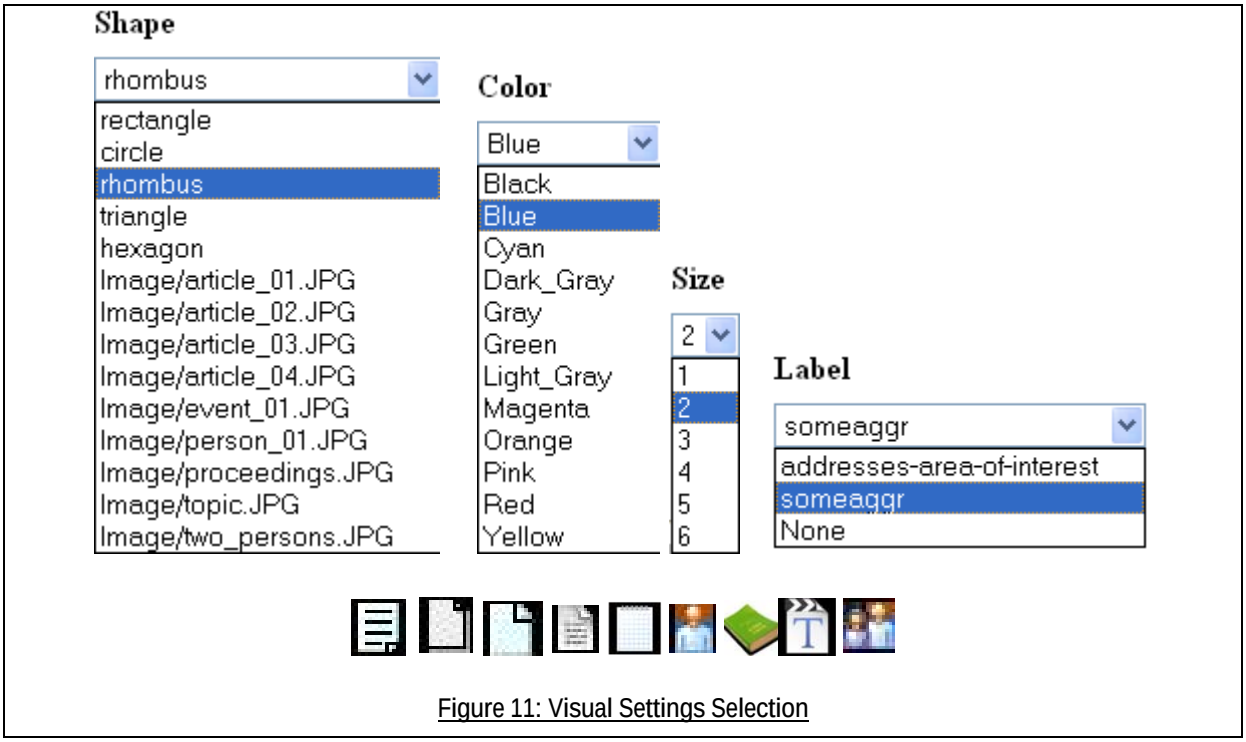

Figure 11: Visual Settings Selection

Figure 11 shows a list of the Shape/Icon, Color and Size elements that the User can currently choose from. The Label elements are Resource and View dependant. This step could use some further development to improve the way the elements are presented to the user. The current version supports preview only of the Icons (Images) after the user clicks the <OK> button. Further work could be done to improve this step in two directions:

1. Eliminating the <OK> button, as it is perceived unnatural clicking a button that causes no obvious change. The role of this button is to submit the settings the user has chosen to the Data Bean. The reason this button was implemented was to speed up the development. This was fully achieved as the easiest way to program a form submittal is by using a button.

2. Designing a preview for all settings the user applies. E.g. showing the User how a triangle in cyan and size 4 with some label of proper type would look like.

3.2.2.2 Edge Appearance

Figure 9 shows that in the current example there are two different Edge Types:

1. has-author – defined as a directed Edge between an Article-In-A-Composite-Publication Node and a Researcher Node

2. some_prop – defined as a directed Edge between two Researcher Nodes.

For each of these Edges the User can choose whether this edge will be visualized (in the Main Window of the Visualization Application) and define a color and width (the same options as in Figure 11 are used for these two Visual Settings).

3.2.3 Design of Information Filters

The second main responsibility of the Semantic Visualization Editor is Filters definition. A Filter is a predicate that is applied to Nodes and different Visual Settings are applied to the Nodes that satisfy this predicate. These Filters are not to be confused with the Filters in the SRN View Specification Editor that filter OUT the Nodes that do not satisfy their predicate. The Filters in the Semantic Visualization Editor do NOT filter out, but only change the appearance of the Nodes satisfying their predicate. This caused some confusion with first time users as discussed in Section 5.2.2. Figure 7 and Figure 8 show the Filter functionality.

3.2.3.1 Different types data filtering

The current version of the Semantic Visualization Components support filtering only on properties of type Integer and String. If a property with some different type is used, the values will be treated as if they were Strings. Strings are sorted lexicographically. E.g. “aaa” is “<” than “b”. Cases are ignored.

Lexicographical filtering on topics, however, would cause confusion. Further work is necessary to implement proper application of Filters on topics in a way similar to the way these are applied in the Visualization Application. Currently filtering on topics is disabled and the User gets a warning if he attempts to create such a filter (Figure 12).

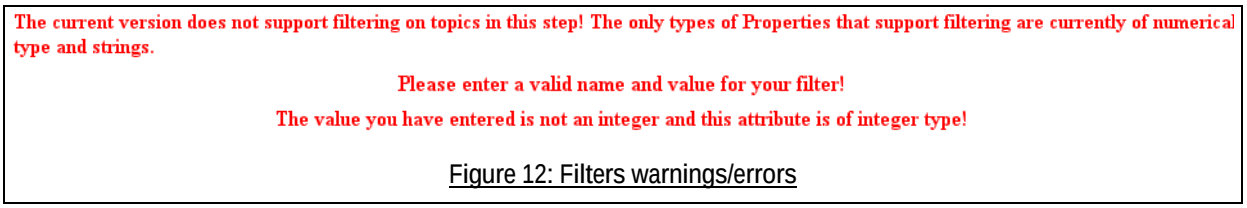

Figure 12: Filters warnings/errors

3.2.3.2 Error Prevention

Fig. 8 shows the dialog for creation of a new Filter. This dialog is displayed only after the User has clicked on the button <Add a Filter>. A successful attempt is made to prevent possible errors in this step. E.g. if one of the fields “Filter Name” or “Enter a Value” is left blank, an error message is displayed (Figure 12). Attempt to put a String value for an Integer type attribute results in an error message as well (as in Figure 8). However, if the filter name entered already exists (a filter exists with this name in the list “Current Filters:”) the filter is overwritten without displaying a warning (as in Figure 7 and Figure 8).

3.2.3.3 Ideas for Further Development

The table “Current Filters” is not always visible. Only when there are filters present. Further work is needed to create a preview of the node in the table instead of or in addition to listing the selected Visual Settings (Shape, Color, Size and Label).

Currently filters only have a single condition predicate. It is impossible to create complex predicate with multiple conditions. This might be implemented in the next release.

3.2.3.4 Logic and application of filters

Filters are “physically” (the settings actually get associated with the nodes) applied to the Nodes in the Semantic Visualization Factory. Filters may be defined in such a way that they partly overlap. Filters are applied in the Semantic Visualization Factory. They are applied in the same order they were defined in and are shown in the Current Filters list. The way filters are applied might be changed in a future release due to the fact that the following observed behavior can be confusing. Consider the list of filters in Figure 13:

Filter Name	Node Type	Attribute	Filter Type	Filter Value	Shape	Color	Size	Label
Filter Name	Article-In-A-Composite-Publication	someaggr	=	0	rectangle	Black	6	someaggr
Filter2	Article-In-A-Composite-Publication	someaggr	>=	0	circle	Blue	5	addresses-area-of-interest

Figure 13: List of filters

The result of these settings looks like shown in Figure 14:

Figure 14: Blue Circle in Size 5

However, if the list is as shown in Figure 15:

Filter Name	Node Type	Attribute	Filter Type	Filter Value	Shape	Color	Size	Label
Filter2	Article-In-A-Composite-Publication	someaggr	>=	0	circle	Blue	5	addresses-area-of-interest
Filter Name	Article-In-A-Composite-Publication	someaggr	=	0	rectangle	Black	6	someaggr

Figure 15: List of filters 2

The result changes like shown in Figure 16:

Figure 16: Black Rectangle in Size 6

Refer to Section 11.2.2 for information on the application of filters.

3.2.4 Visualization Component Specific

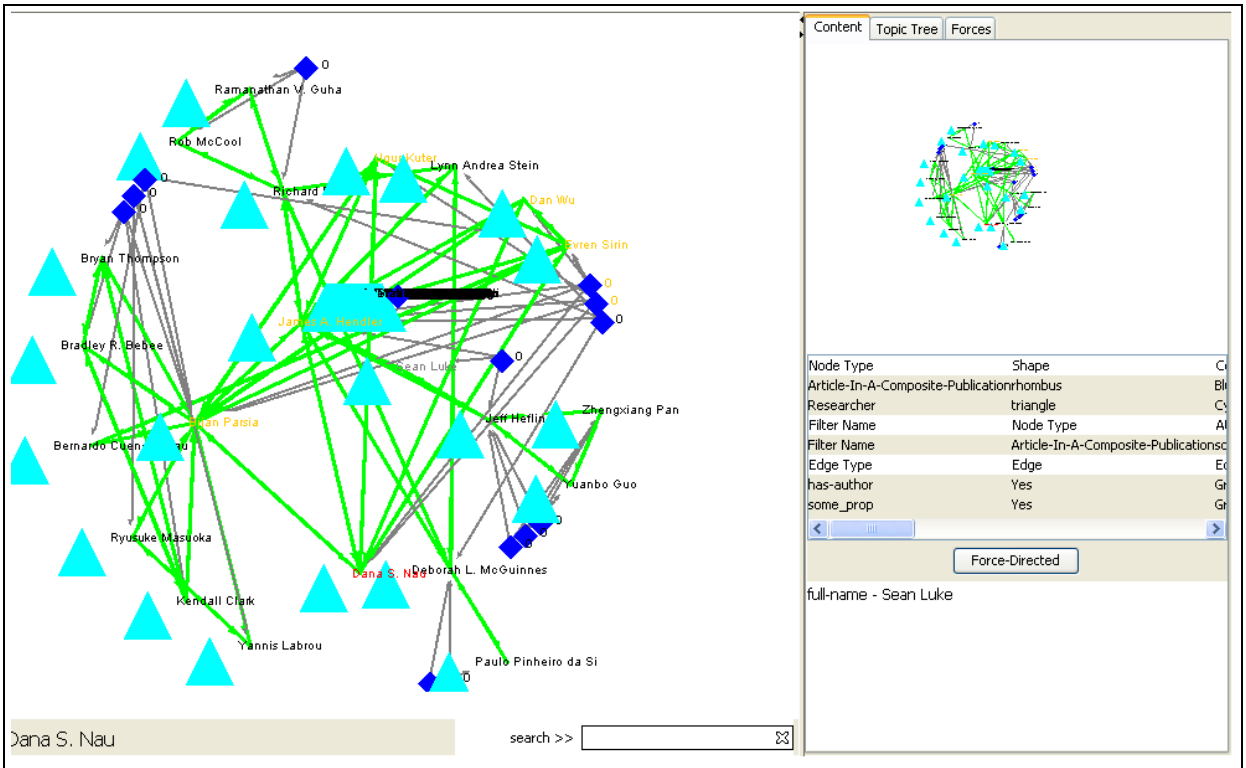

Figure 17: Visualization Application – Display Mode 3

The third and last responsibility of the Semantic Visualization Editor is the authoring of the widgets of the Visualization Application. The Visualization Application is an Applet containing a JTabbedPane (Java, 2004). This part of the Editor configures the Content and Topic Tree Tabs.

3.2.4.1 Context Sensitive

Step 5 of the Editor (see Figure 9) is used to configure the widget at the bottom of the Content tab of the Visualization Application (see Figure 17). Here the user can select which attributes of a node will get displayed when a node is clicked. The possible attributes are context specific and are defined in the SRN View Specification.

3.2.4.2 Layout options

Step 8 of the Editor (see Figure 10) configures the Overview Window at the top of the Content Pane. Figure 17 shows how Mode 3 appears. Figure 18 shows Mode 2. The Main Window of Mode 1 looks like Mode 3 and the Overview Window like Mode 2. The difference between the Display Modes gets even more apparent if not all nodes were selected to be displayed in Step 2 (see Figure 6) of the Editor.

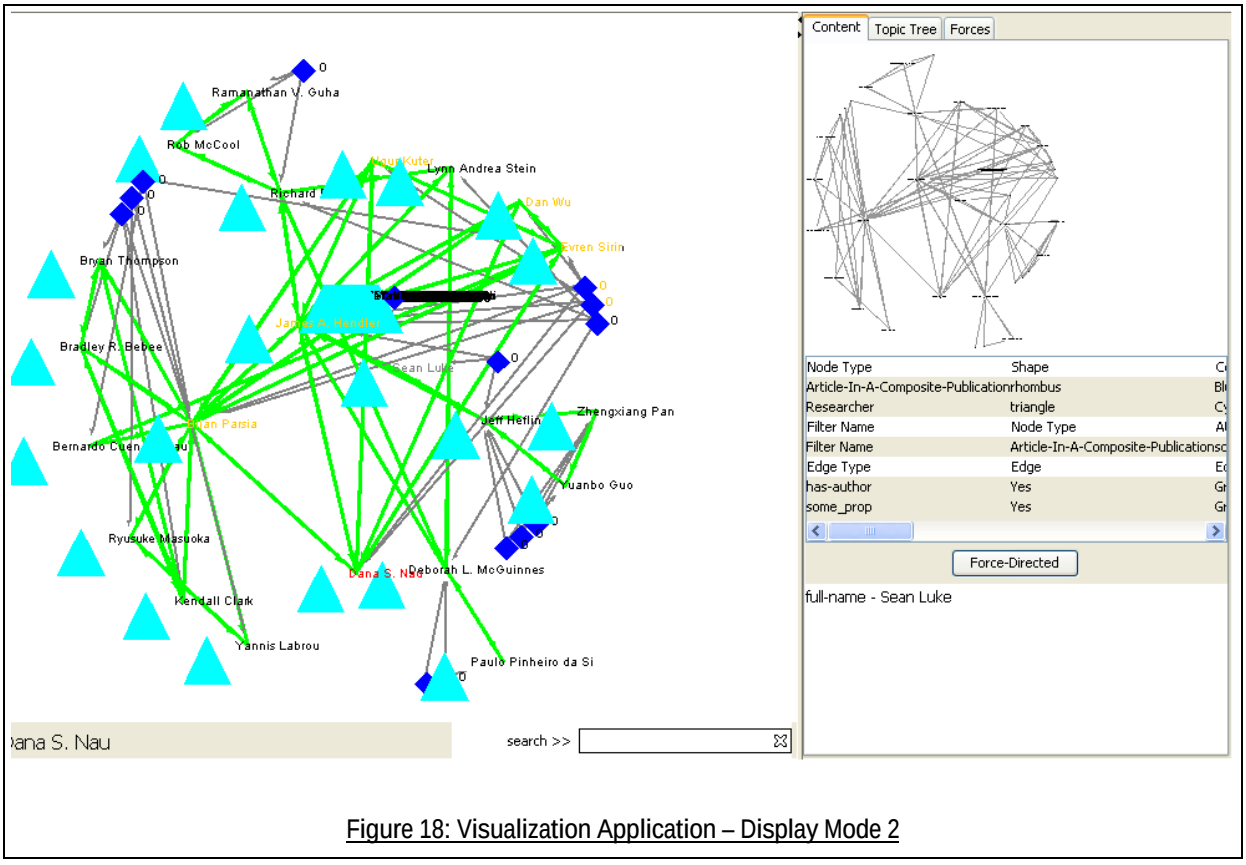

Figure 18: Visualization Application – Display Mode 2

3.2.4.3 Classification Hierarchy (Topic Tree)

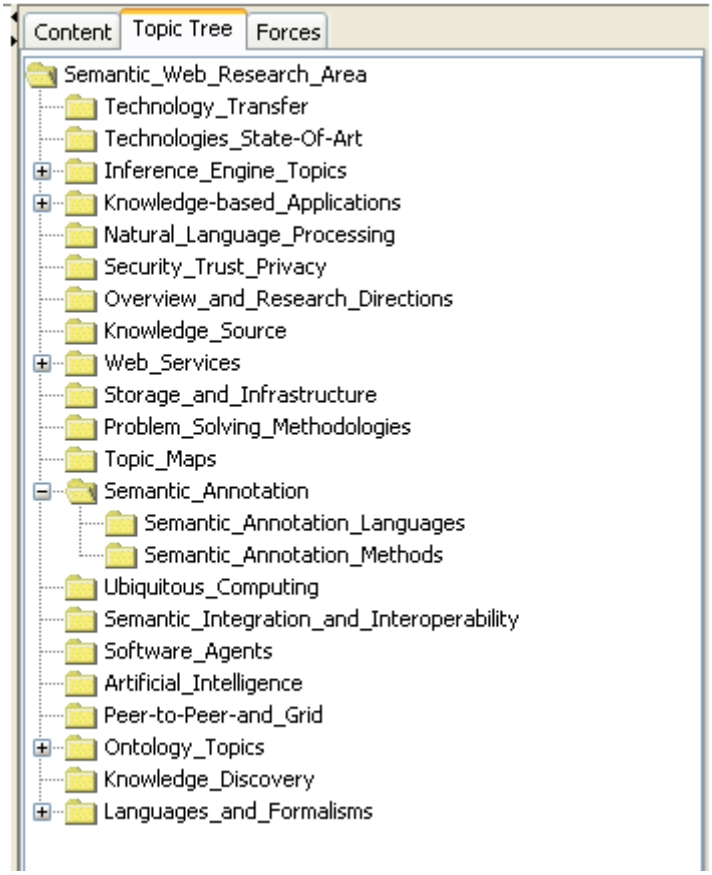

Figure 19: Topic Tree

```
<rdf:Description rdf:about="http://www.ipsi.fraunhofer.de/~stewart/vikef/rn#conf/semweb/HiramatsuAS04">
  <j.0:addresses-area-of-interest>http://trinity.dit.unitn.it/vikef/swc#Knowledge-based_Applications</j.0:addresses-area-of-interest>
  <j.1:has-author rdf:resource="http://www.ipsi.fraunhofer.de/~stewart/vikef/rn#Jun-ichi_Akahani" />
  <j.1:has-author rdf:resource="http://www.ipsi.fraunhofer.de/~stewart/vikef/rn#Kaoru_Hiramatsu" />
  <rdf:type rdf:resource="http://www.aktors.org/ontology/portal#Article-In-A-Composite-Publication" />
  <j.1:has-author rdf:resource="http://www.ipsi.fraunhofer.de/~stewart/vikef/rn#Tetsuji_Satoh" />
  <j.0:someaggr>0</j.0:someaggr>
</rdf:Description>
```

Figure 20: SRN View File Extract

Step 7 of the Editor (see Figure 10) allows the User to choose a Topic Hierarchy according to which to filter the nodes in the Visualization Application. Clicking on a Topic in this Tree filters OUT all nodes that HAVE a topic value attribute defined (see Figure 20) AND the value of this attribute is DIFFERENT from the value of the selected topic and all its subtopics. All other nodes are left unaffected. Step 7 is not extensively tested, because the current ontology (Connolly, et al., 2004) in use has only one Hierarchy. The topics data is extracted from the ontology by the Visualization Application using the selected Topic Hierarchy as a reference element.

3.3 Representation for Visualization Specifications

The Visual Information is stored within one Tag (Visuals) of the XML (Quin, 2006) View Specification file. This Tag only is edited in the Semantic Visualization Editor. The rest of the file (except for the XML Schema) remains unchanged from the original version coming from the SRN View Editor. The View Definition is described as RDF statements.

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#" xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
xmlns:owl="http://www.w3.org/2002/07/owl#" xmlns:daml="http://www.daml.org/2001/03/daml+oil#">
    <rdf:Description rdf:about="http://www.aktors.org/ontology/portal#has-author">
        <rdfs:range rdf:resource="http://www.aktors.org/ontology/portal#Generic-Agent"/>
        <rdfs:domain rdf:resource="http://www.aktors.org/ontology/portal#Article-In-A-Composite-Publication"/>
        <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#ObjectProperty"/>
    </rdf:Description>
    <rdf:Description rdf:about="http://www.ipsi.fraunhofer.de/~stewart/vikef/rn#some_prop">
        <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#ObjectProperty"/>
        <rdfs:domain rdf:resource="http://www.aktors.org/ontology/portal#Researcher"/>
        <rdfs:range rdf:resource="http://www.aktors.org/ontology/portal#Researcher"/>
    </rdf:Description>
    <rdf:Description rdf:about="http://www.aktors.org/ontology/portal#Article-In-A-Composite-Publication">
        <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#Class"/>
    </rdf:Description>
    <rdf:Description rdf:about="http://www.ipsi.fraunhofer.de/~stewart/vikef/rn#addresses-area-of-interest">
        <rdfs:range rdf:resource="http://www.aktors.org/ontology/portal#Generic-Area-Of-Interest"/>
        <rdfs:domain rdf:resource="http://www.aktors.org/ontology/portal#Publication"/>
        <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#ObjectProperty"/>
    </rdf:Description>
    <rdf:Description rdf:about="http://www.aktors.org/ontology/portal#full-name">
        <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
        <rdfs:domain rdf:resource="http://www.aktors.org/ontology/portal#Person"/>
        <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#DatatypeProperty"/>
    </rdf:Description>
    <rdf:Description rdf:about="http://www.ipsi.fraunhofer.de/~stewart/vikef/rn#someaggr">
        <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#DatatypeProperty"/>
        <rdfs:domain rdf:resource="http://www.aktors.org/ontology/portal#Article-In-A-Composite-Publication"/>
        <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#integer"/>
    </rdf:Description>
    <rdf:Description rdf:about="http://www.aktors.org/ontology/portal#Researcher">
        <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#Class"/>
    </rdf:Description>
</rdf:RDF>
```

Figure 21: XML View of the RDF describing a View

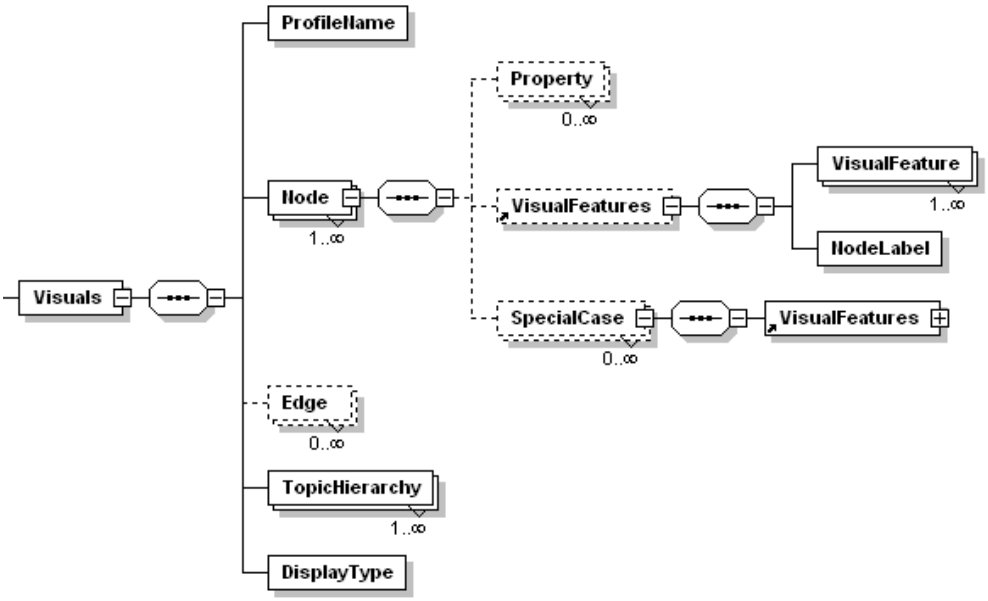

Figure 22: Visuals Schema

```
<Visuals>
    <Node NodeName="Article-In-A-Composite-Publication">
        <Property PropertyName="addresses-area-of-interest" PropertyType="Generic-Area-Of-Interest" ShowInVC="Yes"/>
        <Property PropertyName="someaggr" PropertyType="integer" ShowInVC="Yes"/>
        <VisualFeatures>
            <VisualFeature VFName="Shape" VFValue="rhombus"/>
            <VisualFeature VFName="Color" VFValue="Blue"/>
            <VisualFeature VFName="Size" VFValue="2"/>
            <NodeLabel>someaggr</NodeLabel>
        </VisualFeatures>
        <SpecialCase CompareValue="Semantic Web" Condition="=" OnPropType="Interest" OnProperty="addresses-area-of-interest-
Generic-Area-Of" SCName="Filter Name">
            <VisualFeatures>
                <VisualFeature VFName="Shape" VFValue="Image/article_01.JPG"/>
                <VisualFeature VFName="Color" VFValue="Black"/>
                <VisualFeature VFName="Size" VFValue="5"/>
                <NodeLabel>addresses-area-of-interest</NodeLabel>
            </VisualFeatures>
        </SpecialCase>
    </Node>
    <Node NodeName="Researcher">
        <Property PropertyName="full-name" PropertyType="string" ShowInVC="Yes"/>
        <VisualFeatures>
            <VisualFeature VFName="Shape" VFValue="triangle"/>
            <VisualFeature VFName="Color" VFValue="Cyan"/>
            <VisualFeature VFName="Size" VFValue="4"/>
            <NodeLabel>full-name</NodeLabel>
        </VisualFeatures>
    </Node>
    <Edge EdgeColor="Gray" EdgeName="has-author" EdgeSize="2" Node1Name="Article-In-A-Composite-Publication"
Node2Name="Researcher" Visible="Yes"/>
    <Edge EdgeColor="Green" EdgeName="some_prop" EdgeSize="3" Node1Name="Researcher" Node2Name="Researcher" Visible="Yes"/>
    <TopicHierarchy Selected="Yes" Value="http://trinity.dit.unitn.it/vikef/swc#Semantic_Web_Research_Area"/>
    <DisplayType DisplayType="0"/>
</Visuals>
```

Figure 23: Visual Schema realization

The Schema has been developed iteratively and was modified to support new findings in the process and revised features. Most of the elements are self-explanatory. The only exceptions are the elements “SpecialCase” and “Property”. Initially Filters used to be called Special Cases. Attributes are Datatype Properties in the Ontology. Therefore they were initially known as “Property”. These were renamed midway through the project. A careful examination of the technical aspect of this schema (see Appendix A) is necessary prior to starting to work with it as a lot of data is stored in attributes. The reason why attributes are preferred over sub-elements is that to be parsed this file is loaded into a DOM Document object (Le Hegaret, et al., 2006) and reading attributes is considered easier in this case. However, some data is still stored directly in sub-elements, so that work with the schema requires constant reference to it. It might be a good idea refactoring the Schema for the next release. However, time was insufficient to completely refactor all external sources (such as XML Files) along with the Java sources.

3.4 Application of the Semantic Visualization Editor Design on the Semantic Visualization Factory

The goal of the Semantic Visualization Factory is to apply the settings made with the Editor to an SRN View File. This file is generated by the SRN View Factory in accordance to the SRN View Specification File. The output of the Factory are two graph files containing the same information in both XGMML (Punin, et al., 2001) and GraphML (GraphML Team, 2004) formats. The GraphML file is the input of the Visualization Application as a part of the VIKEF pipeline.

The Factory is closely related and dependant on the Editor. It has two input files (see Figure 2). The factory needs to read and use exactly the settings defined in the Editor. For this task all main classes of the Editor were reused – copies of all the classes of the Editor are present in the Factory. Although this approach introduces some features that are not used by the Factory (e.g. SVEOutput, which is responsible for writing the SRN View Specification File), it is much better than writing new classes for the concrete task, because this way everything remains the same from the programmer’s point of view and everything is used exactly the same way. Another benefit is visible when some element is changed, added to or removed from the Editor Specification. In this case a change has to occur to the classes that handle the input. After the change has been done the classes can just be copied to the Factory. For information on the design of the Factory specific elements and classes refer to Section 4.4.

4 Implementation Issues

This part discusses the software design (code design) of the Semantic Visualization Components. It describes the technologies that were learned and used for the implementation of the components. 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.

For a detailed discussion on the web technology choice for creating the Semantic Visualization Editor refer to Section 3.1.1. After the choice to use Java (Java, 2004) and JSP (JSP06) was made, the implementation was started. Now that the project is in its final phase I can comment on the use of the JSP technology for web development. This comment might be one sided due to my lack of experience with other competing technologies like ASP and PHP, however, after the initial frustration of using a new technology JSP proved to be a comfortable to use and fast for development technology that fully justified its choice. To the weaknesses of this technology I would like to list its close relationship to HTML thus suffering from the same weaknesses HTML suffers from. This weakness is arguably eliminated by JSF, which introduces new tags replacing the HTML tags and represents a framework reportedly developed to fully satisfy the needs of the web developer (unlike mixing up programming and static HTML as in JSP), but also requires a far more serious learning effort before web software can be produced. Another weakness that I experienced using JSP is that experimenting with the two arguably best environments for JSP development on the market today (MyEclipse (MyEclipse, 2007) and BEA Workshop (BEA, 2007)), I found frustrating bugs and weaknesses in both of them that hamper and slow down the development.

The components draw input from an XML File (Quin, 2006), Ontology (Connolly, et al., 2004), (Ontology, 2007) and RDF Document (Herman, et al., 2007). The use of XML Files requires usage of DOM, while parsing Ontology and RDF requires Jena (Jen07). The use of Jena was mostly pleasant and productive. However, one feature was found missing. Transitive relationships are not expressible. In the current components, however, this functionality is needed in XMLHandler and was implemented using error

prone and unnatural recursive methods (being the only means to program the missing functionality using the provided base).

The components directly use two other VIKEF Components: File Manager Client and Ontology Manager Client. The File Manager Client is needed for accessing the input files and storing the output files. 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 Semantic Visualization Components Ontology is accessed for two reasons: Getting a hierarchy of the Resource Types (Node Types) and listing the Topic Hierarchies in step 7 (see Figure 10).

The output of the Factory consists of two files containing the same data in two different formats: XGMML and GraphML. Output in both formats is needed because the Semantic Infusion Component uses both file types. A direct comparison between the two formats (in my opinion) reveals strengths and weaknesses of both. While XGMML has a much more complicated schema that allows for faster development, shorter files and easier extension, if the programmer using it is comfortable with the schema, it also makes it much more difficult to build a parser for. However, a parser fully supporting the XGMML schema is likely going to be very powerful. GraphML is a lot simpler. Parsers for GraphML are lightweight but also mostly going to be abstract and not applicable without customization for the current task. Extending a GraphML file to contain new elements is unhandy, because all these need to be declared in advance prior to being used, but for the same reason likely to be more robust than XGMML. I find that the GraphML schema contains an unnecessary duplication of data in the declaration of a new element in the "key" tag's attributes "attr.name" and "id". These are supposed to contain the same data but are both required by the schema.

Web Services are used for starting the Factory from within VIKEF.

4.2 Component Architecture

Best effort is made to create the software architecture of the Semantic Visualization Components following the principles of Software Engineering. An attempt is made to keep possible (and probable) extension points like the Visual Features open for extension.

I developed The Semantic Visualization Components and this document during the development. A successful effort was made to produce and release usable components for five different review deadlines (four have already passed by the time this document was started), changing and extending the functionality as required for each release. A description of each release follows.

1. Basic Functionality – Nodes and Edges with their Visual Features and Filters. Very basic and asymmetric looking Editor pages. Basic back step functionality. The XML View Specification File parsed using a SAX parser (Megginson, 2004). Only XGMML output of the Factory. Many features are hardcoded. Main goal of the release was to produce something that could be tested and improved.

2. Extending and changing the Visual Features. Instructions for the Editor steps written. Icons (Images) incorporated. Back step functionality improved. Use of the DataBean class improved and extended. XML View Specification File parser rewritten to use a DOM object. First effort made to separate logic from display in the JSP pages. First refactoring. Error Messages for not allowed input introduced. First File Naming Convention introduced. Use of the File Manager Client incorporated. GraphML output format added to the Factory. Web Services (Web Services, 2007) added. Visualization Application project started. Numerous small changes and bug fixes occurred after this release.

3. Edge labels replaced by edge colors. Profile Names dropped. Current file naming convention introduced (see Section 9.2). Visualization Application authoring included. Look and Feel of the Editor pages developed. The Ontology Manager Client and Jena used to access the ontology directly to fix a major problem occurring with the classes hierarchy of the nodes in the RDF part of the SRN View Specification file describing the nodes and edges of the View (see the next part for a detailed description). The Factory changed to use Jena. Authoring of the Visualization Application Legend included in the GraphML output.

4. Code refactored. The elements that were hardcoded and are subject to change were extracted to Constants.java. Only string elements that are not allowed to change (or the formats used will no longer be valid) are left hardcoded (e.g. a node is named “node” in the graph formats, it cannot be renamed). Many bugs fixed. The reference to the ontology changed from using the SRN ID to using the Ontology Version ID. Look and Feel improved. Some error messages. Step 5 (see Figure 9) of the Editor added.

5. Minor bug fixes in the Editor. Minor changes in the Factory to support late changes in the SRN View File. This release comes out at the same time this document is finished. It has more than 4500 LoC and represents a robust and complete beta system fulfilling all major goals set and ready for the challenge of real use.

6. For a summary of possible future release ideas, please refer to Section 7.2.

The next two parts describe the Editor and the Factory: how they were developed; difficulties overcome and problems solved; how the code is structured; how the data is processed and stored. For a technical description of each class see Section 10.

4.3 Semantic Visualization Editor

Despite looking small and simple with its just five functional (seven overall) pages, the Semantic Visualization Editor is a complicated component incorporating many different technologies and fulfilling a complex task. Reaching the current design has been a long and difficult process. Describing it thoroughly and completely is not the goal of this document. This part describes only important decisions and changes that occurred during the implementation and the understanding of which is necessary to understand the whole component.

4.3.1 Editor Pages

The Editor is built as a wizard that leads the User through the necessary steps for extension of the SRN View to a View with defined Visual Features that can be visualized. This occurs by using active web pages from within a browser. The pages use exclusively server-side programming and processing employing the JSP technology.

During the development very important and strict time constraints had the highest priority and affected the work more than anything and at times created tense and stressful working environment that I think we managed to handle very well. The software release deadlines were all successfully held and requirements met. Although new technologies had to be learnt quickly a strong effort is made to produce code that is easy to understand and read. A best attempt is made to separate logic from display without overdoing and creating many new servlets, jsp files and beans with the sole purpose of taking all logic handling elements away from the jsp files that represent the Editor pages. Instead logic processing is separated among the pages (for pages' specific logic and form processing), a bean class (DataBean) keeping the visual data throughout the Editor and a parser class (XMLHandler) that reads the input file.

The refactoring work, which took place for two of the releases of the Editor, aimed to improve the code design. While the goal was achieved, one further improvement can be considered for a future release. Currently although all pages share the same visual design and are constructed very similarly, only the first two lines are abstracted in a separate jsp file – TableHead.jsp, that uses a separate VisualBean for its configuration. This approach can be taken one step further and a new jsp file – e.g. PageTable can be abstracted (instead of TableHead.jsp) that will represent the whole page and not just the first two lines of the page. However, time is insufficient to change this for the fifth release.

While the refactoring done for the fourth release extracted each of the first six steps in a separate jsp file, this work may be considered unfinished, because steps seven and eight still share the same jsp file and the names of the jsp files are confusing as the term “page” is not introduced and the term “step” is used in the meaning of “page” in this document. The reason for this is that the initial design planned a different file for each page. This approach resulted in a couple of very long and complex files. Therefore the design was changed to having a different file for each complex form. It can be further improved by creating a different file for each step of the process.

4.3.2 Input Data Processing and Integration Issues

There has been a single class (XMLHandler) since the beginning of the project parsing the SRN View Specification File. It was initially implemented using a SAX parser for XML and parsing RDF as XML. The SAX parser was abandoned in favor of a DOM document in the second release of the Editor. Jena was introduced for parsing the RDF description of the View in the third release.

This class works in close collaboration with the class handling the data (DataBean) throughout the Editor. This is why these two classes were extracted in a separate package during the refactoring for release number four.

A critical point of the development of the project (and probably its only significant weakness) is related to the integration. It was crucial to not have a stable schema and format enforced on every transition point of the collaborating components throughout the collaborating projects within the Semantic Navigation Services part of VIKEF. This was even more challenging given the tradeoffs between the need to use stable formalisms and quick code development. However, in this case it backfired.

Two further issues with the RDF description of the View arose. They were both solved.

1. Treating RDF as XML causes trouble and the View description that is being used by the Semantic Visualization Editor is in RDF format. An easier and faster solution to this problem would have been changing the way the View was represented from RDF to some XML representation. This would have been possible, because this representation of the graph is only used by the Semantic Visualization Components and could (and should) be in a format that is easy for the latter to process. This way the stress of having to learn a new technology (Jena) under time pressure would have been avoided, the base for solving the second issue below would have been set, the speed of the Editor would have been kept, and the graph would have been represented in a more natural format (than RDF).

2. The abstract of the ontology inserted as RDF statements in the XML and representing the graph (Figure 21) did not always represent a valid graph on its own (if the relationships between the resource classes present in the ontology are not considered).

However, the solutions to these problems were resolved in the Semantic Visualization Components, instead of following the suggestion given in 1. due to external matters. Fixing the first issue employing Jena proved easy and a good approach. While accessing the ontology fixed the malfunction of the Editor when dealing with an invalid graph, the approach is heuristic and it cannot be guaranteed that the result of the solution delivers the intended View as the meaning of the View is not definite for every possible input. For a detailed discussion on this topic refer to Section 10.2.2.2. This is a negative example where one component had to change in order to implement functionality needed by another component. Fortunately, the positive examples of this approach within the project, where the overall complexity was reduced as an effect, outnumber the negative. Overall the components incorporate and collaborate successfully.

4.3.3 Processing and Storing the Data

Each page of the Editor uses the same Bean (DataBean) for storing the settings defined by the user. This class works in close collaboration with XMLHandler. For a technical description of this class refer to Section 10.2.2.1.

After clicking the <Finish> button on page 5 (see Figure 10) the input file is overwritten storing the Visual Settings configured in the Editor run. For writing the output the SVEOutput class is used.

4.4 Visualization Factory

The Factory uses two related input files – an SRN View Specification File (already enhanced with Visual Settings by the Semantic Visualization Editor) and an SRN View File generated according to the specification. It is important to know that the View Specification data is needed in the same way it is needed in the Editor and therefore all classes used for this task could be reused without any changes. This speeded up the creation of the Factory as the effort was only on creating the graph files.

The Factory is supposed to be called through a Web Service within VIKEF. For development purposes a simple JSP page is supported.

The creation of both graph files is similar and follows the same pattern. There are two different classes creating the two different output formats. When the project was started it was set as a requirement that the Factory is built open for addition of new output formats. Initially the only required format was XGMML. However, this extension point was soon made use of in the second release when GraphML was added. The architecture of both output classes is outlined in the abstract OutputFormat class that can be used as

an extension point if further formats need to be added in future. It is general enough to support two very different file formats like XGMML and GraphML. The technical side and pattern of the creation is described in Section 11.

Figure 24, Figure 25, Figure 26 and Figure 27 show how the output files look like. Conforming to the GraphML schema results in a heavy and complicated file, however this file is compatible with the prefuse GraphML parser used in the Visualization Application and thus the overall development time of both Components (Factory and Visualization Application) is much shorter (than it would have been if a new format was used and a new parser had to be developed). Using proper formal formats here has also the advantage of easy use of these files in the Semantic Infusion. Possible future uses of the legal files will also be favored as conforming to the schema guarantees compatibility with general use parsers for these formats that can then be customized to fit the current needs (instead of having to develop new parsers or possibly modify existing ones in a deteriorating way). Refer to [Nikolov 2007] for further discussion on this topic as well.

```
<graph xmlns="http://www.cs.rpi.edu/XGMML" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" Rootnode="1" directed="1"
xsi:schemaLocation="http://www.cs.rpi.edu/XGMML http://www.cs.rpi.edu/~puninj/XGMML/xgmml.xsd">
    <att name="TopicHierarchy" value="http://trinity.dit.unitn.it/vikef/swc#Semantic_Web_Research_Area"/>
    <att name="OntologyVersionId" value="vikefuser____http://www.ipsi.fraunhofer.de/__ONTO__scienceOntologyV3.00__Version__Mon Jan
22 15:00:12 GMT 2007"/>
    <att name="DisplayType" value="1"/>
    <node id="0" label="Sean Luke" name="Researcher" weight="4">
        <att name="full-name" value="Sean Luke"/>
        <att name="uri" value="http://www.ipsi.fraunhofer.de/~stewart/vikef/rn#Sean_Luke"/>
        <graphics fill="Cyan" type="triangle" visible="1"/>
    </node>
    <node id="1" label="Javier Gonzalez-Castillo" name="Researcher" weight="4">
        <att name="full-name" value="Javier Gonzalez-Castillo"/>
        <att name="uri" value="http://www.ipsi.fraunhofer.de/~stewart/vikef/rn#Javier_Gonzalez-Castillo"/>
        <graphics fill="Cyan" type="triangle" visible="1"/>
    </node>
    <node id="2" label="Pascal Auillans" name="Researcher" weight="4">
        <att name="full-name" value="Pascal Auillans"/>
        <att name="uri" value="http://www.ipsi.fraunhofer.de/~stewart/vikef/rn#Pascal_Auillans"/>
        <graphics fill="Cyan" type="triangle" visible="1"/>
    </node>
```

Figure 24: XGMML Visualization Application Authoring and Node Display Extract

```
<edge label="P. R. Smart some_prop Nigel Shadbolt" source="17" target="820" weight="3">
    <graphics fill="Green" visible="1"/>
</edge>
<edge label="0 has-author Timothy W. Finin" source="19" target="1012" weight="2">
    <graphics fill="Gray" visible="1"/>
</edge>
```

Figure 25: XGMML Edge Display Extract

```
<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 26: GraphML Visualization Application Authoring Extract

```
<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 27: GraphML Node and Edge Display Extract

4.5 Integration within VIKEF

Figure 28 draws the relationships and collaborations of the different components in VIKEF's Semantic Navigation Support.

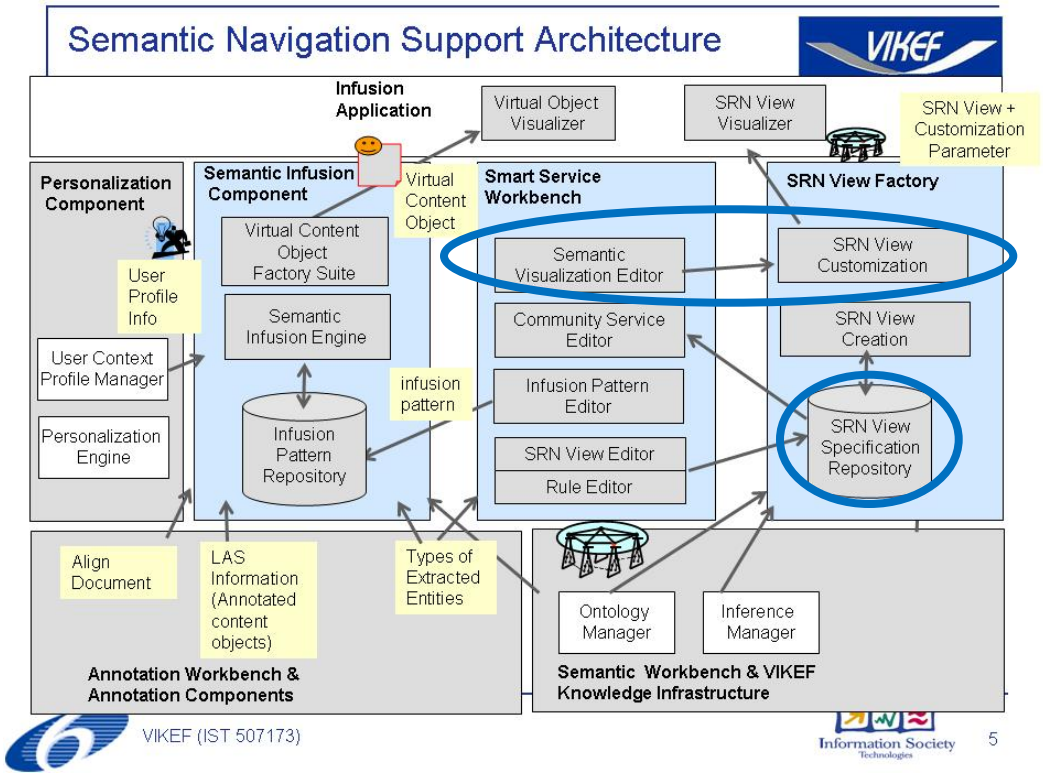

Figure 28: Semantic Navigation Support Architecture

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 Software Design Evaluation

In this part I try to evaluate the software (code) design of the components I developed in accordance to the software design principles. I try to consider the five modular properties, some class-level design principles and some package-level design principles. Of course, evaluation of the own design might be subjective.

5.1.1 Modular-* Criteria

The criteria are generally held and the overall design can be rated as good. According to these criteria a weakness of the design of this project can be found in its rigidity.

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 some of 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 Semantic Visualization Components the systems can be modularly decomposed into three somewhat independent parts: data input processing; data processing during run and output creation communicating with each other using interfaces that remained mostly constant throughout the project although the inner functionality of the classes changed considerably.

5.1.1.2 Modular Composability

The goal of the project was in a clash with Modular Composability. Development targeting at fulfilling a certain task does not transfer well to the requirements of this criterion. The only elements that can be

reused (but only within the project) are the reused Editor classes in the Factory. However, this should not be seen as a weakness of the components.

5.1.1.3 Modular Understandability

This work fully supports Expanded Modular Reasoning and in many aspects, supports Modular Reasoning. Although it is difficult to understand the different classes of the Semantic Visualization Components if examined one by one, considering the two components separately is easy and they are easily understood without knowing the SRN View Components. Additionally, the input and output files essentially act as the "glue" across components. It is easy to understand the components once these files are well understood.

5.1.1.4 Modular Continuity

Modular Continuity was not considered to a large extent in the first releases. However in later releases, a targeted and successful attempt was made to more fully incorporate Modular Continuity through refactoring and the creation of the Constants class. Suggestions have been given in Sections 4.4, 11.3.1 and (Nikolov, 2007) on how to further improve the design in this direction. Such improvements would support specification changes better.

5.1.1.5 Modular Protection

Modular Protection is partially supported. The design of the whole project to be open for Visual Settings extension has significantly increased the durability against propagation of changes.

An attempt was made to further improve it in the Factory so that changes do not get propagated to the Visualization Application as well. Although success was achieved, it can be expected that changes in the SRN View Components may propagate to the Semantic Visualization Components. The main reason for this is that the schema for the output/input files is subject to evolutionary changes.

5.1.2 Class-level Design

The design of the classes in the last two releases 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 fourth release. The positive effect completely justifies this effort.

5.1.2.1 General Principles

5.1.2.1.1 Information hiding

Information hiding is paid close attention to in the class design of the project. All classes have clearly defined public interfaces and field and method visibilities are carefully considered. In the current design of the Editor only DataBean and XMLHandler (being the only two classes in the data package) use package visibility for their fields. This is considered safe, because the two classes work in close collaboration.

Package visible fields are used in the OutputFormat class of the Factory too. This is motivated by the fact that this class is abstract and to make the access to these fields easier from the extending classes that need to have full access to these fields anyway. Abstracting the fields behind methods would introduce too much needless complexity. Some needless complexity has been introduced by providing information hiding throughout the other classes of the project already, but the gains of doing this are considered greater.

5.1.2.1.2 Coupling and cohesion

The classes DataBean and XMLHandler have extremely high coupling. They depend on almost all other classes of the project. They also depend closely and complexly on each other and use each other. The cohesion of both classes is moderate to low as they act as coordinators for both Semantic Visualization Components. However, the cohesion of DataBean is lower than that of XMLHandler as it is responsible for keeping data that can constantly change during a run of the Editor, while XMLHandler only parses and keeps the data read from the input file.

However, it is wrong to view these classes as God Objects. To avoid this Anti-Pattern, the major part of the logical computations during runtime has been delegated to the JSP Pages, in spite of bringing logic and display together to some extent. Creating extra classes or servlets for computations is possible but will introduce a few new small classes. This change might be considered if further decoupling of logic and display is desired in future.

All other classes of the Editor have low coupling and high cohesion as they have only a single responsibility and depend on only DataBean (and XMLHandler) at the most.

The Factory classes have low coupling. They depend on XMLHandler and DataBean. There is high method cohesion. E.g. the class OutputFormat has many responsibilities: it defines the workflow for the creation of an output file; reads in the data of Node and stores it in a format ready to use by the extending classes; applies filters and delivers the Visual Settings for each node; creates the output file. However, each of these responsibilities is in its own method with no interaction between the methods.

5.1.2.2 Assigning Responsibilities

Responsibilities are not assigned to the best extent throughout the Editor. The JSP pages (interface) act not only as role controllers for the events that happen in them but do calculations and implement business logic as well. This is a generally accepted “bad” practice. The reason this approach was taken was the pressure on producing the first release as soon as possible. Having a clearly structured pages’ responsibilities and a clear separation of the logic and display parts within the JSP pages has made this problem only formal and therefore this has not been changed in the last refactoring phase. The DataBean class can be seen as a Façade between the JSP Pages and the Java Sources.

The SVEFactory and OutputFormat classes can be seen as controllers in the Factory.

The Expert and Creator responsibilities are spread among the JSP Pages, DataBean and XMLHandler.

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, complex relationships between them and slows down the production of software.

5.1.2.4 OCP

OCP is met on the design of the Visual Settings for Nodes (see Section 4.2 and Section 10.2.2.1). Another expected extension point is the addition of new output formats in the Factory (see Section 4.4). Extending the Editor with new functionality is possible and requires changes only to DataBean and XMLHandler.

The Components are not closed against ordering of the filters (see Section 3.2.3.4).

5.1.2.5 Other Principles

LSP, DIP and ISP are not considered as mostly not applicable.

The only package design principle that has been considered is CRP. However, the implementation of this principle is not complete. Suggestions are given in Section 4.3.2 and Section 10.2 on how to further redesign the packages to follow this principle completely. The other package design principles are considered not applicable.

5.2 User Involvement

VIKEF end-user consortium members were involved in assessing the SRN View Definition and Semantic Visualization Editors. This assessment phase took place prior to the third release. This was done to allow the possibility for some feedback from the user to be incorporated in later releases.

The goal of assessing the Semantic Visualization Editor was to get user feedback on its ability to support the user in completing steps which are common in preparing a View for visualization. Through the assessment the following points were obtained:

1. An overall impression from end-users engaged in creating views and assigning visual settings for the scientific congress domain.
2. Feedback about how the prototypes could be extended.
3. Feedback on the usability of the Editors in supporting the users in completing a set of predefined tasks.

5.2.1 Set-Up

During the preparation phase, a demonstrator (audio-accompanied slide show), which provides the background and motivation for the knowledge view process, was given to the users. In the demonstrator, each phase in the Knowledge View definition and creation process was explained in detail. In addition to the demonstrator, a user involvement guidebook was given to the users in advance. The purpose of the guidebook was to prepare the participants for the evaluation day.

The assessment was structured in four phases. During the first phase, an overall explanation was given to the users. In the second phase, the users were guided in completing a predefined set of tasks. 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 tools.

5.2.2 Results

In general, the Editor interfaces were seen as simple, easy to learn, practical and quite useful for the purpose for which they were designed. The users found that it was easy to get the system to do what they wanted. The users also found the use of the tools easier for the knowledge view process, than the methods they currently use. The users were pleased with the ability to use icons to represent concepts or nodes in the graph, found that the tools nicely clarify the necessary aspects of the Knowledge View creation process.

End-user feedback was received 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.
- Exploit mouse-over to provide additional text for the reader.
- Describe how the system could integrate into an environment in which the data is stored in the back-end in a relational database.
- Create “step tracking” or make the current wizard step interactive so the user can “jump” to different states during the editing process. Users found it difficult to reenter information after they navigated backwards to edit and then wanted to resume the current step.
- Found some terminology confusing. For example, in the Semantic Visualization Editor, the application of filters implies assigning different visual settings to a subset of the view, not eliminating a subset.

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. The extensions that have not been handled are as follows:

- Add preview functionality for the Visualization Application within the Semantic Visualization Editor – easy to implement.
- Providing a pane in the Semantic Visualization Editor providing an overview of the parts of the Visualization Application configurable in the Editor – easy to implement.
- Replacing the long Instructions texts with mouse-over popups – moderately difficult to implement due to inexperience with HTML popups.
- 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 data input from data bases – difficult to implement.

- Rename filters again – easy to implement, however no suggestion has been given for a new name.
- Make color unavailable when an image has been selected – before this can be implemented the <OK> button in step three must be dropped – difficult.
- Add preview for all different settings – easy to moderate, however time consuming.
- It is possible to do wrong due to the dropping of the Profile Names. Warn the user about the possible danger – easy to implement.
- Change the requirements and make the editor extendable by the user – very difficult to implement.
- Make the “Current Wizard Step” interactive – moderately difficult – however, this would require applying default settings if the user decides to skip steps.
- Offer “Edit Filter” functionality to make it easier to fix errors in the filters – easy to moderately difficult.

6 Related Work

The tools developed in this work focus on requiring adequate support for the navigation of information space views and supporting the creation of services that enable community members to make better use of the opportunities in their domain. In this sense conference management systems (such as (Zakon Group, 2007), (ACTA, 2007) and (IASTED, 2007)) can been seen as related. They support different types of users and allow them to modify and use the domain knowledge that they have access to. Data Mining and Semantic Web applications of these systems bring them in the same line with VIKEF by giving the user means of browsing and accessing their underlying domain knowledge.

This project can be seen as related to every project built around the use of visualization of resources (using XGMML, GraphML and prefuse) for the purpose of providing services to end users in a scientific conference domain. The prefuse visualization gallery (see (prefuse, 2007)) provides an overview of such projects.

Another line of relationship can be seen in the use of active web pages (using jsp, asp, php, etc.). All these such projects use similar component architecture as the components built in this work.

However, the stress in the these components is on creating an Editor that allows a user with no programming experience to create and modify completely new Views according to his needs and not just to try and take the most from already existing predefined Views. In this sense little related work is known. Allowing the user not just to visualize the existing relationships between the data elements he is working with, but also to choose the way the elements and their relationships are visualized (which is the exact task of the featured components) has to our best knowledge not yet been done.

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 two of these components: the Semantic Visualization Editor and the Semantic Visualization Factory. These take a View defined by the SRN View Components, lead the user through the process of defining Visual Settings for how the View should be displayed and create an output file that can be displayed by the Visualization Application.

The design, implementation issues, problems (both solved and unsolved) with the components, concerns and ideas for future further development were presented in this document. The three appendices at the end deliver technical details on the software and the formats used.

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 Editor.

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. This list might appear long, however for a project of this size, given the time constraints, requirements changes and circumstances the achieved can be considered a great overall success.

1. Form submittal in step three of the Editor can be removed. See Section 3.2.2.1.
2. A preview for the settings chosen can be added. See Section 3.2.2.1.
3. Filters functionality should be extended: filtering on topics should be made available (see Section 3.2.3.1 and Section 11.2.2); complex predicates should be possible (see Section 3.2.3.3); defining the order, which filters are applied in, should be configurable or filter priorities should be added for the case of overlapping filters (see Section 3.2.3.4).

4. The XML schema for the Visual Settings should be refactored. See Section 3.3 and Section 9.1.
5. The JSP pages should be further refactored. See Section 4.3.1 and Section 10.2.4.
6. HTML Style formatting can be introduced to replace the nested tables. See Section 10.1.
7. The table is step five can be changed for the case of having nodes with no attributes. See Section 10.1.4.1.
8. JSP pages only designed to use throughout the development should be removed from the final product. See Section 10.1.6 and Section 4.4.
9. The packages can be further refactored. See Section 10.2.
10. Error logging can be further developed. See Section 10.2.4.
11. The Visualization Application authoring elements should either be fully present or fully removed from the XGMML output file of the Factory. See Section 11.3.1.
12. Logic can be further decoupled from display. See Section 5.1.2.1.2.
13. Extensions based on user comments. See Section 5.2.3.

8 Bibliography

[ACTA, 2007] [Online] // ACTA Press - Scientific & Technical Publications - Journals, Proceedings & Papers. - ACTA Press, 2007. - 03 2007. - http://www.actapress.com/.

[BEA, 2007] BEA - Business Software, Business Process Management, Service Bus, Service Oriented Architecture [Online] // BEA Workshop, Eclipse development, JSP, Struts, JSF, EJB, Spring framework, IDE, Integrated Dev. - BEA Systems, Inc, 2007. - 2006. - http://www.bea.com/framework.jsp?CNT=index.htm&FP=/content/products/workshop/.

[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] Knowledge Network Visualization [Report] / auth. Nikolov Dimitar. - Darmstadt : TU Darmstadt, 2007. forthcoming

[MyEclipse, 2007] [Online] // MyEclipse J2EE IDE - Easy and affordable eclipse plugin development tools. - Genuitec, LLC, 2007. - 2006. - http://www.myeclipseide.com/.

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

[Zakon Group, 2007] OpenConf Conference Management Peer-Review Software System & Service [Online] // Coaching, OpenConf, Web & Technology - Zakon Group LLC. - Zakon Group LLC, 2007. - 03 2007. - http://www.zakongroup.com/technology/openconf.shtml.

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

[Publications, 2006] Publications [Online] // VIKEF. - Information Society, 2006. - 03 2007. - http://vikef.net/publications.php.

[Herman, et al., 2007] Resource Description Framework (RDF) / W3C Semantic Web Activity [Online] / auth. Herman Ivan, Swick Ralph and Brickley Dan // World Wide Web Consortium. - W3C, 01 29, 2007. - 11 2006. - http://www.w3.org/RDF/.

[Megginson, 2004] SAX [Online] / auth. Megginson David. - SourceForge, 04 27, 2004. - 11 2006. - http://www.saxproject.org/.

[IASTED, 2007] [Online] // The International Association of Science and Technology for Development. - IASTED, 2007. - 03 2007. - http://www.iasted.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/.

[Web Service, 2007] Web service [Online] // Wikipedia, the free encyclopedia. - Wikimedia, 03 29, 2007. - 12 2006. - http://en.wikipedia.org/wiki/Web_services.

[Punin, et al., 2001] XGMML (eXtensible Graph Markup and Modeling Language) [Online] / auth. Punin John and Krishnamoorthy Mukkai. - Dept of Computer Science RPI, 08 24, 2001. - 11 2006. - http://www.cs.rpi.edu/~puninj/XGMML/.

9 Appendix A – XML Visuals Data and File Naming Convention

9.1 Visuals

Figure 22 shows the current picture of the XML Schema of the Visuals Element, used for storing the data gathered by the Semantic Visualization Editor.

The element ProfileName is no longer used.

The element VisualFeatures only contains the elements VisualFeature and NodeLabel.

The element NodeLabel stores the label of the node (selected in the Editor).

The elements listed below use Attributes to store data (not visible in the big picture above).

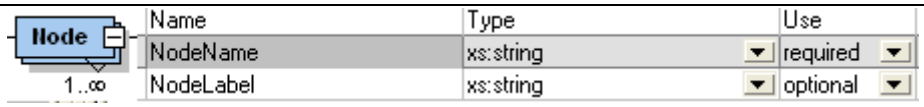

Figure 29: Node Tag

Each Node element represents a type of node (e.g. Researcher). The type is stored in the attribute NodeName. The attribute NodeLabel is not used any longer. It should be removed when the Schema is refactored.

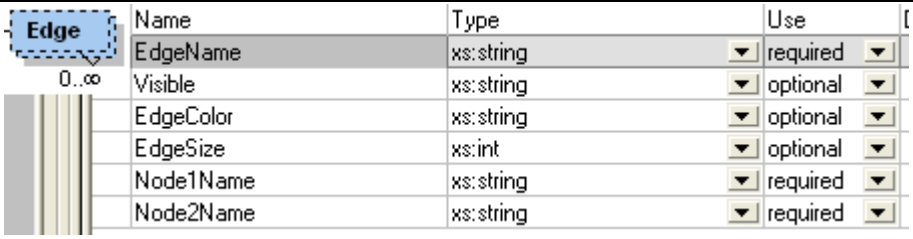

Figure 30: Edge Tag

Each Edge element represents a type of edge (e.g. has-author). The type is stored in the attribute EdgeName. Visible is of type Yes/No and corresponds to the field Make Visible in the Editor (see Figure 9),

EdgeColor and EdgeSize correspond to the respective fields. Node1Name and Node2Name store the types of nodes this edge has as a source and target.

Remark: The Yes/No type is not declared explicitly. It is informally adopted to replace the Boolean type. The reason for using such a notation instead of Boolean is motivated by the desire to make this option easy to change. By keeping it of String type it can be easily changed to something else should it happen that more than two options have to become available.

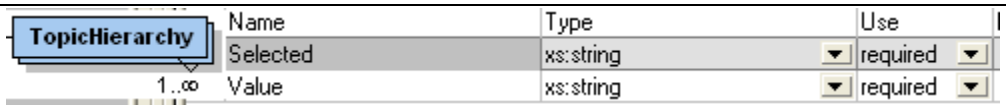

Figure 31: Topic Tree Tag

Each TopicHierarchy element represents a possible Topic Hierarchy (which to choose from in step 7 of the Editor – Figure 10). Selected is of type Yes/No and Value is the name of the hierarchy (e.g. *http://trinity.dit.unitn.it/vikef/swc#Semantic_Web_Research_Area*).

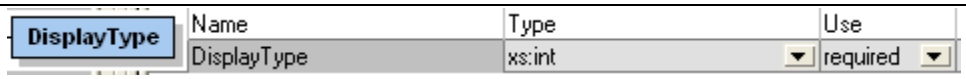

Figure 32: Display Mode Tag

Stores the selected Display Mode in step 8 of the Editor. Currently there are three available options. However, these are stored in the following way: Mode 1 – 0, Mode 2 – 1, and Mode 3 – as a 2 in the XML File.

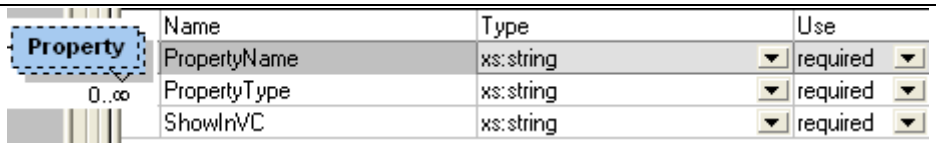

Figure 33: Attribute Tag

Each Property element stores a node attribute. PropertyName stores the type (e.g. full-name). Property-Type stores the format, which the data is stored in (e.g. string). ShowInVC is of type Yes/No and stores the settings of step 5 of the Editor (Figure 9).

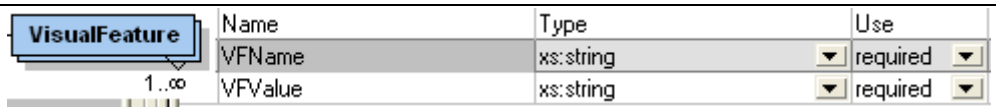

Figure 34: Visual Feature Tag

An example of a Visual Feature is Color. The Visual Features for nodes were implemented in such a way to support easy extensibility. See the discussion in Section 4.2. They are open for extension. Adding (and removing) a Visual Feature was required to be easy for this project. Therefore a conscious attempt was made to abstract over the visual features.

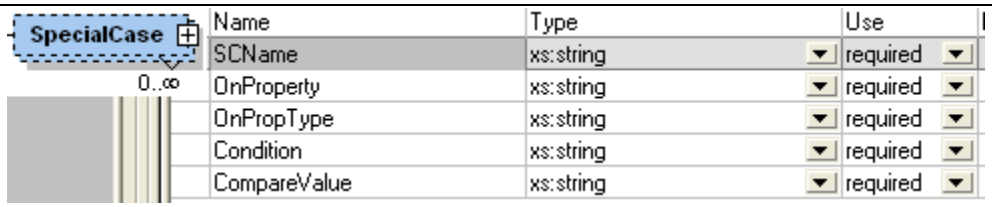

Figure 35: Filter Tag

A SpecialCase represents a Filter as specified in step 4 of the Editor (see Figure 7 and Figure 8). Refer to the Current Filters table in Figure 7 to recognize the data elements described next. SCName stores the Filter Name. OnProperty stores the Attribute. OnPropType stores the format, which the data is stored in (see Figure 33). Condition stores the Filter Type. CompareValue stores the Filter Value.

9.2 File Naming Convention

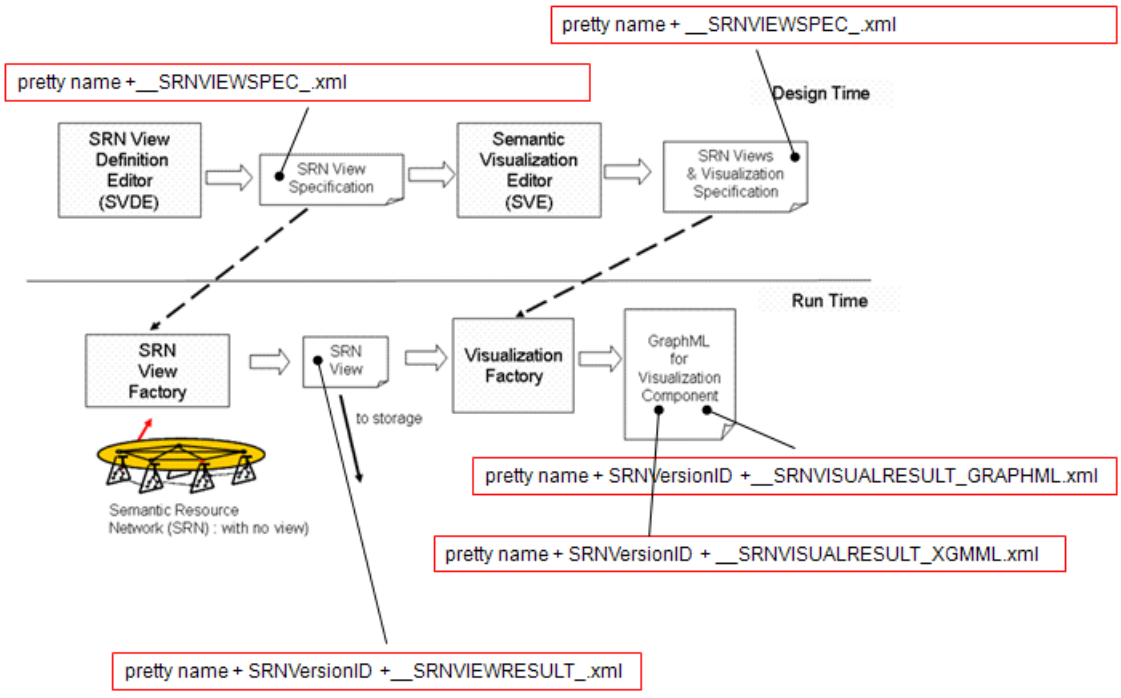

Figure 36: File Naming Convention

10 Appendix B – Technical Description of the Semantic Visualization Editor Files

10.1 JSP Pages

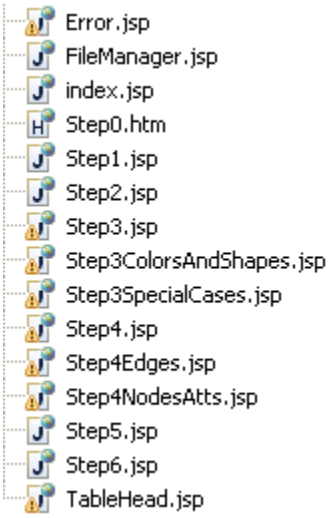

Figure 37: List of Editor JSP Pages

An Editor run starts by calling index.jsp, which forwards the browser to Step0.htm showing an overview of the Editor process (see Figure 4).

All pages are formatted using (cascading) HTML tables. A possible improvement point is applying some high end HTML styling technology (e.g. CSS).

10.1.1 Page one

The actual wizard starts with Step1.jsp (see Figure 5). This page accesses the File Manager Client and creates the form displayed in Figure 5 listing alphabetically all possible input files for the Editor.

10.1.2 Page two

After a file is selected the logic in Step2.jsp (see Figure 6) sets up the DataBean with the predefined start settings for the session and calls the XMLHandler to parse the input file. If the input file contains Visual

Settings (has been the output of the Editor in the past) the settings in the Bean are updated to the previously saved settings.

10.1.3 Page three

Having selected nodes for visualization on page two, the wizard proceeds with Step3.jsp (see Figure 7). This file contains exclusively business logic and includes the files Step3ColorsAndShapes.jsp and Step3SpecialCases to display the forms for processing the nodes' Visual Settings and Filters that can be seen in Figure 7.

A request to this page can occur through submission of six different forms. Therefore the logic in this file is separated in five different parts using if-else clauses. The difference in the numbers here is described in 6. These are the following:

1. <Back> on page four – in this case no processing takes place and the forms are simply displayed.

2. Coming from page two. This is the first page of the Editor that should allow back step functionality (shifting the action to the step before the calling page – in this case page one – Figure 5).

   Back step functionality is always implemented the same way – in the "if" clause checking to see where the request to this page came from. If it came from the previous page and the argument to this request is "Back" redirect to two pages earlier and skip the rest of the page.

   In case of a normal proceeding from page two the visibility of the nodes is updated directly to the DataBean.

3. Clicking the <OK> button in step 3 (see Figure 7). In this case the updated Visual Settings for the Nodes are saved to the Bean.

4. Clicking the <Delete> button for a Filter (only visible when there are filters present). This filter's name is removed from the Bean's filters list.

5. New filter creation has been confirmed (<OK> button in step 4 clicked – only visible when the form for adding a new filter has been made visible by clicking on the <Add a Filter> button). A new SpecialCase object is created with the data defining a filter (name, node, attribute, filter type, filter value, new Visual Settings and label). This object is added to the filters list in the Bean.

6. <Add a Filter> button clicked. This request is handled in Step3SpecialCases.jsp.

10.1.3.1 Step three

Step3ColorsAndShapes.jsp. This page includes no logic processing, only a single check whether nodes were selected to be visualized. If no nodes were selected in step two a warning is displayed instead of the form in Figure 7.

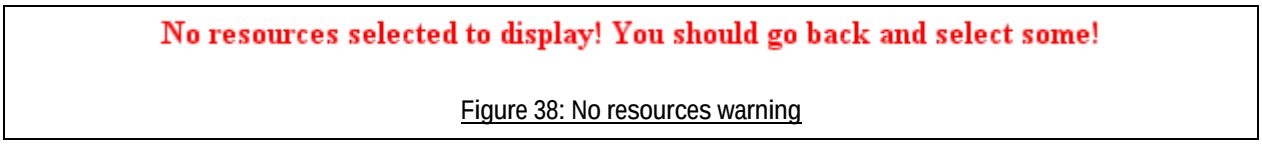

Figure 38: No resources warning

The settings displayed are the settings stored currently in the Bean.

10.1.3.2 Step four

Step3SpecialCases.jsp. This is a complex file drawing the Current Filters table, handling the form submission initiated by clicking the <Add a Filter> button. It has three parts.

1. The table Current Filters is only displayed if it is not empty (filters have been added already or were present in the input file and the nodes for which they were defined were chosen to be displayed in step two).
2. The rest of the form (as seen in Figure 7) is displayed only if nodes were selected to be displayed. (Otherwise there would be no node types available to list and add a filter for.)
3. Clicking on the <Add a Filter> button causes the form seen in Figure 8 to be displayed and the user is requested to enter the necessary values to define a filter. It is possible that a node was defined without attributes in the SRN View Editor. If this is the case for the selected node only a warning is displayed.

10.1.4 Page four

Step4.jsp – Figure 9. This is a simple page combining step five (Step4NodesAtts.jsp) and step six (Step4Edges.jsp) in a single page and doing no logic processing. The logic on page three is quite complex and handles all settings that can be applied on that page. Clicking the <Next> button on page three only sends the browser to page four and does not submit any data that needs to be processed.

10.1.4.1 Step five

This is a step for authoring the Visualization Application. The table is only displayed if nodes were selected to be displayed in step two. Due to the fact that this step was the last element included in the Editor the specific condition of having a node with no attributes is not handled optimally. It would have been

best not to list this node in the table at all. However, currently such nodes are listed with an empty list of attributes. This is a minor issue that might be changed in a future release.

10.1.4.2 Step six

This is the last step dealing with Visual Settings. The table is only displayed if the View defines edges for the nodes selected to be displayed in page two. Otherwise a warning is displayed:

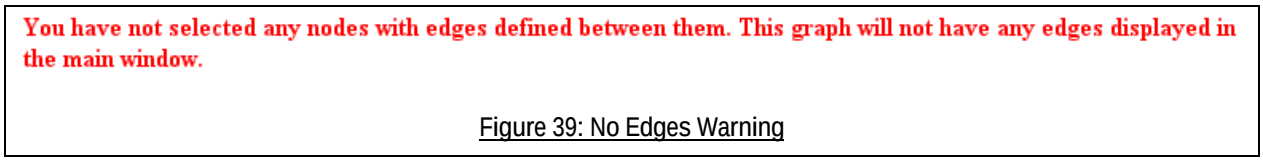

Figure 39: No Edges Warning

10.1.5 Page five

Step5.jsp. This is the last page of the Editor. The logic part processes the settings applied on page four. The display is straight forward easy built and easy to understand. There are only two drop down menus, only the values for the first (step seven) are delivered by the Bean and the options are already present as such having been extracted and stored by the XMLHandler. The second pull down menu lists the Display Modes and its values are hardcoded in the JSP Page.

10.1.6 Additional JSP Files

Only Error.jsp, Step6.jsp and TableHead.jsp belong to the flow of the Editor.

The error page is currently only displayed if there is some error accessing the file list in page one.

The logic part of Step6.jsp processing the settings made on page five and displays a confirmation message that the settings have been stored and offers the user to begin the wizard again.

Figure 40: The End!

TableHead.jsp is included in every page of the Editor and displays the first two gray rows on each page.

FileManager.jsp is an easy to use and powerful form that allows accessing the most important features of the File Manager Client through a GUI. It was created only for supporting purposes during the development and should be removed when the project comes out of the testing phase.

10.2 Java Sources

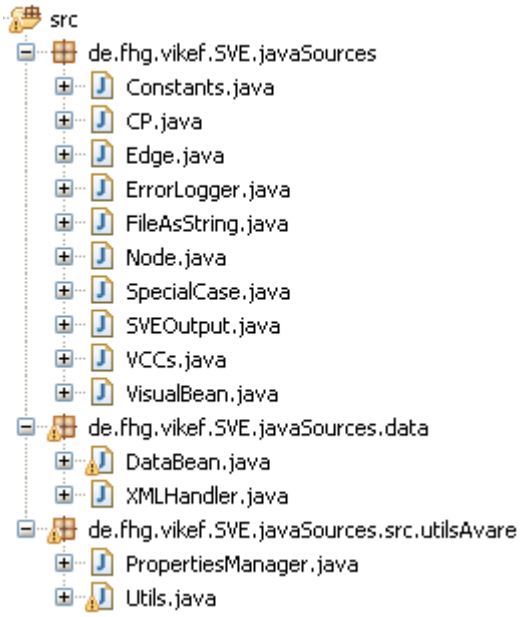

Figure 41: Java Sources

The source files of the Editor can be separated in four categories. They are currently separated in three packages. The package structure might be changed in a future release by creating a new package (visualElements) and moving classes from the javaSources package to the new package to more precisely represent the responsibilities of the classes.

10.2.1 Visual Elements

The first category consists of classes that correspond to the data elements in the Editor. There are currently nodes (Node.java), edges (Edge.java) and filters (SpecialCase.java).

Nodes and edges could be further subcategorized to data elements coming from the View Definition. These classes are only responsible for representing these elements that cannot be changed in the Editor. For nodes the data stored includes the type (e.g. Researcher) and the attributes and their types (e.g. fullname: String). For edges the type of the edge (e.g. has-author) references to the Node objects being a source and target of the edge and references to the types of these nodes as strings for easier access. The Visual Settings that can be modified are stored within the DataBean.

Although filters are created by the Editor, once created they cannot be modified, but only deleted. Therefore it is handy having a class in this category representing a filter storing the following parameters: name; node it is defined for; attribute of this node it is filtering according to; type of this attribute; filter condition; filter value (for comparison); the Visual Settings (as these cannot be changed after the filter is created it is a good idea keeping them in the class and not handling them from within the DataBean as in the case with nodes).

10.2.2 Data Handling

The two classes in this package lie at the core of the Editor and will be described in greater detail. These are DataBean.java used as a bean throughout an Editor session and keeping track of all changes and Visual Settings applied in the Editor and working in close collaboration with XMLHandler.java used for parsing the View Specification File, creation and initialization of the DataBean. These two classes are not closed against changes in each other and changes in one always propagate to the other. They should be comprehended as a whole.

10.2.2.1 DataBean

This class uses almost exclusively LinkedHashMap objects to operate and store all Visual Settings applied in the Editor. It is also one of the two files to change if additional Visual Settings should be added (the other one being VCCs.java). The rest of the components are configured to use the Visual Settings as they have been defined here. No guarantees are made that this extensibility works flawlessly, because

although it was a requirement from the beginning of the project, such a change never actually occurred and it has never been tested.

After the initialization of the Maps done within the file, changes to the Visual Settings are applied exclusively from the outside by the JSP pages using it.

10.2.2.2 XMLHandler

This class parses the View Specification File using a DOM Document. The action flow in this class is separated in two parts: parsing the RDF statements defining the View and where applicable parsing previously stored Visual Settings (if the file contains such).

As of third release the RDF statements are correctly parsed employing a Jena Model. It is also the release that saw a big error handling routine for handling the second issue described in Section 4.3.2 introduced. This routine is described next.

A close observation of this sample View Definition in Figure 21 reveals a problem with the ObjectProperty-s "has-author", "addresses-area-of-interest" and "full-name" (e.g. the range of has-author – "Generic-Agent" – is not declared as a class in this abstract, which in terms of nodes and edges means that the edge has-author has a node as a target that is not defined). However, looking at the big picture (having the ontology in mind) reveals that "Generic-Agent" is a super element of "Researcher". With this additional information the View makes sense. Here is how this problem is handled in XMLHandler (please consider the java comments):

```
Node node1 = nodes.get(extractName(source));
Node node2 = nodes.get(extractName(target));

node == null means that this type of node has not been declared in the classes
//this usually means that this property is declared to work with some super node
//of the node declared in one of the classes (one of the
//declared nodes has to be a subnode of the source node). The next step in trying to fix this problem
//is to access the ontology and try to find which of the declared nodes is a subnode
//of the super node that the property works with, and then use this concrete subnode with it
if (node1 == null) {
    node1 = getFromHierarchy(source);
```

```
/** This method attempts to access the ontology and find out which of the nodes declared in the
 * rdf part of the input file has the passed as uri node as a super node. It takes and returns
 * the first such node it finds. However, it is possible that there are multiple nodes that satisfy this
 * criterium (or all in case of Thing). This might lead to unexpected behaviour and wrong looking
 * graphs. For example it might have been meant by Thing to substitude only one or multiple nodes.
 * The way this is processed here, only a single node is recognized as a Thing, and so some
 * edges that were meant to be shown will not be recognized as edges at all.
 *  The time for the development of this component is not long enough to fix all
 * encountered unexpected problems. The problem of having edges declared using super nodes in the ontology,
 * but only passing concrete subnodes of these super nodes in the rdf abstract defining the graph
 * was encountered at a relatively late stage, when
 * the component was already in the testing phase. Extensive testing and investigation of the behavior
 * that might occur in this case is not planned. Fixing the problem completely in this component is
 * very difficult or maybe impossible. It already caused a huge change in it
 * - it introduced using the ontology and is
 * the only reason why the ontology is being accessed directly from this component.
 * A better approach of handling this problem would be changing the way the output of the
 * SRNViewEditor component is created. The complexity of the parsing part of this component
 * can be decreased significantly if the data passed from the SRNViewEditor component to use
 * with this component is created with a thought of what is supposed to happen with this data here!
 * What this method and the recurse method do, should better be moved to the SRNViewEditor, as the 'Thing'
 * problem described above can most probably be solved there - it is there where the edge (object property)
 * is defined and at definition time it must be known for which nodes it was meant to be declared. */
private Node getFromHierarchy(String uri) {
    OntClass superClass = ontologyModel.getOntClass(uri);
    //for each of the declared nodes
    for (Iterator<String> i = fullUriNodes.keySet().iterator(); i.hasNext();) {
        OntClass node = ontologyModel.getOntClass(i.next());
        CP.out("uri " + uri);
        //see if superClass is a super class of it
        Node ret = recurse(superClass, node, node);
        if (ret != null) {
            return ret;
```

Figure 42: getFromHierarchy Method

Figure 21 reveals one more problem with the consistency of the data. The element “addresses-area-of-interest” is declared as an ObjectProperty. This element represents the topic of an article and should be an attribute of a node and not an edge (the ObjectProperties in the RDF View Definition represent edges and the DatatypeProperties – node attributes. See Figure 21) as topics are not declared as nodes. Fixing this problem was easy and only required refactoring the code and extracting the methods in the if-else clause as separate methods:

```
/** this parses the object properties of the rdf to edges */
private void getEdges() {
    ResIterator ri = rdfSpec.listSubjectsWithProperty(RDF.type, object);
    while (ri.hasNext()) {
        Resource r = ri.nextResource();
        String name = extractName(r.getURI());

        //this fixes the problem of having the topics sometimes as object properties.
        //their behavior is supposed to be as datatype properties
        if (name.equalsIgnoreCase(Constants.topicName)) {
            getAtt(r, name);
        } else {
            getEdge(r, name);
        }
    }
} //end getEdges
```

Figure 43: Edges Creation

10.2.2.3 SVEOutput

This class should be moved to the data package. It creates the extended View Specification File as output. The code is straightforward using the DataBean and XMLHandler and creating a DOM Document containing the Visual data and finally uploading the output file to the File Manager Client.

10.2.3 Reused Code

The only code pieces reused from other projects are the following for simple XML DOM operations methods:

```
public static org.w3c.dom.Document newXML(String uri) throws Exception {

        public static void exportDom(org.w3c.dom.Document document,
                String annotationFile) {
```

Figure 44: Reused Methods

These can be found in Utils.java

10.2.4 Other Sources

```
/**
 * This class is used for logging and for holding different often used String values.
 * The values defined here only appear using the consts defined here. So changing a value
 * requires only changing it here.
 *
 * @author The JimmyTaker
 */
public class Constants {
```

```
/**
 * 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 {

/**
 * Uses a log file to write in the possible exceptions.
 *
 * @author The JimmyTaker
 */
public class ErrorLogger{
```

Figure 45: Other Sources 1

ErrorLogger is quite simple and has not been tested. It might be a good idea replacing it with a structured logging mechanism in a future release.

```
/**
 * The purpose of this class is to pass a file as a String as required by the file manager.
 *
 * @author The JimmyTaker
 */
public class FileAsString {


/**
 * Used for holding the values used in the Visual Features.
 * The values defined here only appear using the consts defined here. So changing a value
 * requires only changing it here.
 *
 * @author The JimmyTaker
 */
public class VCCs {

/**
 * This JavaBean is used for displaying the pages' headers
 *
 * @author The JimmyTaker
 */
public class VisualBean {
```

Figure 46: Other Sources 2

It was already discussed that this class represents a nice idea for refactoring that can be extended and improved without a great effort.

11 Appendix C – Technical Description of the Semantic Visualization Factory Files

This part describes the creation of the XGMML and GraphML output files of the Semantic Visualization Factory. The rest of the files are reused from the Semantic Visualization Editor.

11.1 SVEFactory.java

This is just a small class with a single method called by the Web Service starting the Factory. Its purpose is to start the transformation, provide the transformation files with the input they need and to coordinate the creation of the graph files.

11.2 OutputFormat.java

11.2.1 Workflow

This is an abstract class creating the backbone of the transformation workflow. The different classes implementing different output formats extend this class. The constructor of this class defines the workflow of creation of an output file. It is the following:

```
createRoot();
createNodes();
createEdges();
createFile();
```

Figure 47: Workflow

Each of the first three methods is abstract and is implemented differently depending on the format that should be created. However, the goal of each method remains the same. `createRoot()` is used to create the root element of the graph XML. It is this element where the structure of the whole file is described and the Visualization Application authoring data is stored. The node and edge elements are created next and finally the created DOM Document is exported as an XML file to the File Manager Client.

11.2.2 Implemented Methods

A small class – NodeData – used just to hold the data of a node (the Jena Resource, and XML attributes that should be included in this element) is included here as well. An object of this type is created for each node after the `NodeData readInNodeData(Resource r)` method is called in a loop over the node elements from within the extending classes.

The application logic for the filters is implemented in this class. This method is called from within the extending classes for each node.

```
/** this method delivers the visual settings for the node it is invoked in. It considers the special cases.
*if special cases overlap, then each time the settings are overwritten */
LinkedHashMap<String, String> getVisuals(String type, LinkedHashMap<String, String> atts) {
```

Figure 48: getVisuals Method

This is a long and complex method. The computation occurs in a loop over the Visual Settings (as defined in the options map in the DataBean). Before anything is computed the node, the method is called for, is checked whether it was chosen to be displayed in Step two of the Editor. If this check fails the starting values for the Visual Settings (as found in the DataBean) are returned. Note that the issue described in Section 3.1.3 2. is different as the nodes are attached these default values just to make their definition complete. These nodes will NOT be displayed by the Visualization Application using ANY settings.

Otherwise a loop over the defined Filters, in the order they were defined, is started. For each Filter is checked whether it was declared on the current node and if it should be applied (Filter Value is compared according to the Filter Type to the Value of the current node) the defined Visual Settings for the current node are replaced by the Visual Settings of the Filter. This method currently works only with Filter Value types of integer and string (strings are sorted alphabetically as discussed in Section 3.2.3.1). It should be extended to support topics filtering as well. This will require creating the Topic Hierarchy in the Factory in a similar way to the way it is created in the Visualization Application.

11.3 XGMML and GraphML

11.3.1 Differences

XGMML output is shorter and simpler than GraphML. The reasons for this is that most of the settings that are stored (with the only exception of the Visualization Application authoring Topic Hierarchy and Display Type and the necessary Ontology reference) have their specific spots in the XGMML schema and since XGMML is not used in the Visualization Application it does not need to carry the data for authoring the

legend in the Visualization Application (see Figure 17). However, the XGMML files include some Visualization Application authoring elements. These can be dropped too. The data for the legend takes up a large part of the implementation of the first two methods of the workflow (see Section 11.2.1). Compare the node elements in Figure 26 and Figure 27. The node element in Figure 26 contains authoring information as well. This data is needed only once and refers not to the node but the Visualization Application. Storing this data in an XML conforming to the GraphML Schema proved to be heavy and unhandy, but needed. This parts of the GraphML.java file should be studied profoundly before any changes are attempted. It should be noted that changes to this part will propagate to the Visualization Application and therefore are strongly advised against. Refer to (Nikolov, 2007) for further information. One further difference between XGMML and GraphML is that in GraphML each sub-element of the node and edge elements needs to be declared in the root element before it can be added in the body. XGMML extension elements are declared and initialized simultaneously using the “att” tag.

11.3.2 Nodes

The nodes are added in a loop over the Jena Subjects (representing the nodes) of the SRN View (RDF) file (see Figure 20). In the fifth release an extra element with a predefined URI is added by the SRN View Editor to the View file. This element does not represent a Node and is ignored in the loop. For each Node a “node” element is added to the DOM Document and the URI of this Node is kept in a Vector (ids) for later reference when creating the Edges as Edges are defined between Node URIs in the View but need the ids of node elements in the graph files. The Visual Settings are added to the graph file next. The last element added to each node element is the data corresponding to Step five of the Editor – Nodes’ Attributes to Visualize (see Figure 9 and Figure 26).

11.3.3 Edges

The edges are added in three nested loops:

1. A second loop over the same elements as the loop for creating the nodes. This is necessary because the nodes need to all have integer ids to be able to get referenced by the edges. If this is done in the first run (when the nodes are added) the target node might not have been parsed yet. It is important for an edge in a graph file to be defined AFTER both its nodes have been defined. Otherwise the GraphML parser of the Visualization Application breaks. The approach taken to make sure this is always true is creating all edges only after all nodes have been created. As of the fifth release filtering out of nodes has been finally implemented in the SRN View Components and results in an inconsistent View referencing elements in the edges not present as Resources.

This inconsistency is fixed here by skipping edges that reference a node that has not been defined.

2. A loop over the Edge types (e.g. has-author).

3. A loop over Jena Properties of the node – Jena Resource of the current element from the first loop.

Therefore the edges are ordered by two criteria:

1. Order of their source nodes.

2. Edge type (e.g. has-coauthor). (E.g. first all edges “has-coauthor” having node with id = 0 as a source are added, followed by the edges “is-author-of” having node with id = 0 as a source, etc. Next all edges “has-coauthor” having node with id = 1 as a source are added, followed by the edges “is-author-of” having node with id = 1 as a source, etc.)

The Visual Settings are added first, followed by the id of the source node (which is the id of the current node in the loop). These remain unchanged for all edges over the third loop. To complete the creation of the edge only the id of the target node is needed and only this element is added within the third loop. The rest of the settings are copied.

Release four added an edge label to each edge. This label is not configurable and for each edge it consists of the label of its source node (if one exists, otherwise the type), the type of the edge and the label of its target node (if one exists, otherwise the type). This label is included to make it simpler for the Visualization Application to display a description of the edge that the mouse currently points at in the Info Box at the bottom of the screen. This is another positive example of the different components collaborating to spread the implementation of some functionality in a way that it is easiest to do.

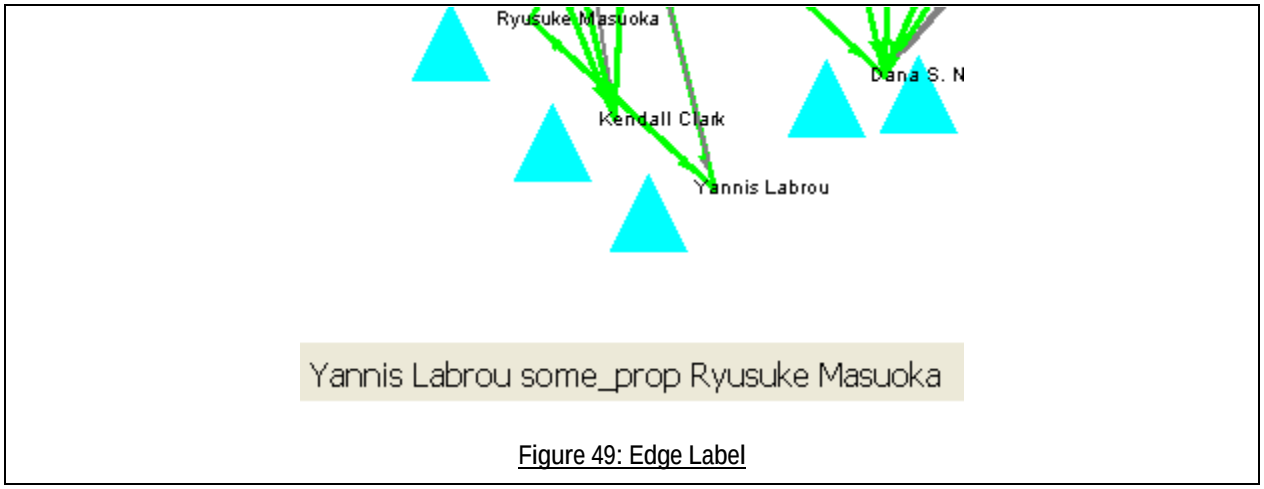

Figure 49: Edge Label