Master thesis

November 16, 2011

OntoX
A Scriptable Visualization Framework for the Semantic Web

Stefan Zehnder
of Attelwil, Switzerland (02-918-563)

supervised by
Prof. Dr. Harald C. Gall
Michael Würsch and Matthias Hert

University of Zurich
Department of Informatics

s.e.a.l.
software evolution & architecture lab

Master thesis

OntoX
A Scriptable Visualization Framework for the Semantic Web

Stefan Zehnder

University of Zurich
Department of Informatics

s.e.a.l.
software evolution & architecture lab

Master thesis

Author: Stefan Zehnder, stefan.zehnder@uzh.ch

Project period: 17.05.2011 - 17.11.2011

Software Evolution & Architecture Lab
Department of Informatics, University of Zurich

Acknowledgements

I would like to thank my supervisors Michael Würsch and Matthias Hert for all their ideas and support during this work, and I also like to thank Professor Harald Gall for giving me the opportunity to write this interesting thesis.

Abstract

In information science, an ontology represents a shared vocabulary for a domain of interest with the purpose to describe entities of the domain and the relationships between these entities. One of the more common goals in developing ontologies, is to share a common understanding of the domain knowledge among people and software agents. An agent can process large amounts of data in native ontology syntax, whereas human beings need to have the data in a visualized form to be able to detect patterns, structures, and elements in the ontology. But most of today's visualization tools have problems in scalability, cannot include domain specific knowledge, and confront the user with too much visualized details.

The aim of this thesis is to create novel and powerful way for the user to analyse the data integrated into ontologies. Therefore a framework named OntoX has been developed that can read RDF/OWL files and present this data as an interactive information graph. In contrast to traditional tools that only rely on the user interface to interact with the graph, OntoX comes with its own implemented domain specific language. Through this simple language, a user can write ontology specific scripts that filter out elements, modify the design, or change the structure of the graph. Therefore the user gets a tool that assists him in analyzing the domain knowledge, and allows an individual configuration for every ontology.

Zusammenfassung

In der Informatik versteht man unter dem Begriff Ontologie ein gemeinsames Vokabular einer bestimmen Domäne, welches die Entitäten der Domäne und deren Beziehungen zueinander beschreibt. Eines der häufigsten Ziele in der Ontologie-Entwicklung, ist das Teilen des gemeinsamen Verständnisses des Domänenwissens unter Menschen und Agenten. Ein Agent kann riesige Mengen von Daten im nativen Ontologie Syntax verarbeiten, aber Menschen müssen die Daten in visualisierter Form haben, damit sie Muster, Strukturen und einzelne Elemente entdecken können. Die meisten der heutigen gebräuchlichen Visualisierungstools haben aber Probleme mit der Skalierbarkeit, können kein domänenspezifisches Wissen verwenden und konfrontieren den Benutzer mit zu viel Details.

Das Ziel dieser Arbeit ist das Erarbeiten einer neuen Möglichkeit, welche es den Benutzern erlaubt das integrierte Wissen einer Ontologie in einer neuartigen Weise zu analysieren. Zu diesem Zweck wurde eine neue Applikation namens OntoX entwickelt, welche RDF/OWL-Dateien lesen und die Informationen dann in einem interaktiven Informationsgrafen darstellen kann. Im Gegensatz zu herkömmlichen Entwicklungen, welche nur eine Bedienung über die Benutzerschnittstelle zulassen, kommt OntoX mit der eigenen implementierten domänenspezifischen Programmiersprache. Durch diese einfache Sprache kann ein Benutzer einfache Skripte erstellen, welche es ihm erlauben Elemente des Grafen zu filtern, das Design oder die Struktur zu verändern. Dadurch erhält der Benutzer ein Tool, welches ihm bei der Analyse einer Domäne unterstützt und je nach Ontologie individuell konfiguriert werden kann.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

1.1 Motivation

The Semantic Web is a *Web of Linked Data*, an extension to the classic *Web of Documents*, in which information is given a well-defined meaning. Through augmenting Web pages with data targeted at computers, machines can understand the semantics, or meaning of information, in the World Wide Web [BLHL01].

Through ontologies it is possible to enrich the data with additional meaning, defining relationships between semantic data, and specifying logical rules for reasoning about it. The Semantic Web has been proven useful whenever knowledge had to be processed by machines, and gained more and more importance as the need of knowledge and technologies working together started to grow.

Whereas machines can easily process all the information stored in RDF/OWL files, it is for human beings much more cognitively challenging to understand these large and complex data sets. Human beings therefore can benefit from visualization techniques that can visualize the data as a graph, since graphs are often used to visualize relationships and patterns between entities. Such graphs, produced from RDF triples, can contain more information, like implicit information such as the underlying structure of a data model or which instances are most closely connected [MG03].

Visualizing the Semantic Web is not a trivial task. The large data sets result in graphs that are quite connected and overload therefore the graph with too much information. Also irrelevant data and redundant information increase the complexity of the graph. There are other tools available for visualizing the Semantic Web in the form of a graph, but they have problems in scalability, limited filtering, or cannot visualize certain ontology specific characteristics.

1.2 Goal

The goal of this thesis is to develop a novel visualization framework, named ONTOX, for displaying an information graph that is described in RDF/OWL. The visualized graph should allow a user to incrementally explore the graph, by starting with a small set of nodes and then dynamically adding or removing elements.

Additionally, the framework should possess its own domain specific language that allows a user to create simple scripts that can alter the design and structure of the graph for the purpose of graph customization and the use of domain specific knowledge. Another requirement is the ability to filter unwanted node and edge elements from the graph to improve scalability and preventing an information overload in the graph.

1.3 Structure

The remainder of this thesis is structured as follows: *Chapter 2: Related Work* introduces the Semantic Web and gives some basic information about visualization techniques. *Chapter 3: Background* lists some similar work that has been done in the area of Semantic Web visualization and presents some basic visualization frameworks. *Chapter 4: Technological Foundations* gives an overview about the basic technology that is used to build the ONTOX framework. In *Chapter 5: OntoX Framework* the implemented application is presented with all its features, including the implemented domain specific language that allows a user to interact with the application in a more complex way. Some interesting details of the implementation, which helps to understand the structure and functionality of the framework, are presented in *Chapter 6: Implementation Details*. *Chapter 7: Evaluation* list the results of the evaluation that was performed after the implementation of the prototype. *Chapter 8: Final Remarks* concludes this thesis with a summary and some ideas how to improve the ONTOX framework.

Chapter 2

Related Work

This chapter introduces the Semantic Web and some of its technologies, which form the basis for this thesis. Also related work in information visualization is presented in this chapter.

2.1 Semantic Web

What is the Semantic Web? The World Wide Web Consortium [wc3b] states it as follows: *"In addition to the classic "Web of documents" W3C is helping to build a technology stack to support a Web of data, the sort of data you find in databases. The ultimate goal of the Web of data is to enable computers to do more useful work and to develop systems that can support trusted interactions over the network. The term Semantic Web refers to W3Cs vision of the Web of linked data. Semantic Web technologies enable people to create data stores on the Web, build vocabularies, and write rules for handling data. Linked data are empowered by technologies such as RDF, SPARQL, OWL, and SKOS."*

Berners-Lee et al. [BLHL01] describes it as: *"The Semantic Web is not a separate Web but an extension of the current one, in which information is given well-defined meaning, better enabling computers and people to work in cooperation."*

The Semantic Web is about the integration and combination of data drawn from different sources. It is a presentation of information that can be interpreted by machines, so machines become much more capable in processing and understanding the data that they merely display at present.

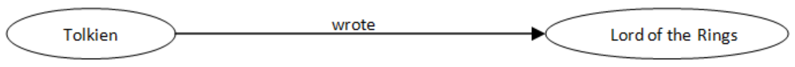
2.1.1 Resource Description Framework

The Resource Description Framework (RDF) has been developed by the World Wide Web Consortium (WC3). It is a language designed to standardize the definition and use of metadata. RDF is designed to support the Semantic Web as a language representing information in the web and providing a model for describing, and creating relationships between resources. The framework describes a resource as an object that is uniquely identifiable by a Uniform Resource Identifier (URI).

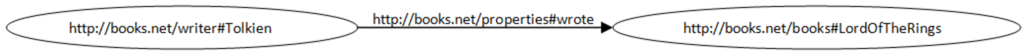
Resources are described in triples (also called statements). A RDF triple consist of the three elements *Subject*, *Predicate*, and *Object*. These three pieces of information are all that is needed to fully define a single bit of knowledge. The RDF triple allows the definition of information in a consistent and human understandable way. The first part of the triple is the subject. It specifies the resource that is being described with a triple. The predicate (also called property) defines the kind of information one wants to express about the subject, e.g. an attribute, a relationship, or a characteristic. The last element of the triple is the object. Within RDF, the object defines the value of the predicate. It can be a literal or another resource. Each triple represents a complete unique fact.

The RDF Core Working Group decided to use a directed labeled graph for describing the RDF data model. A directed graph consists of a set of nodes which are connected by an arc, where the subject is the source node of the directed graph, the predicate is the arc that connects the two nodes, and the object is the target node in the directed graph.

As an example, the sentence "Tolkien wrote Lord of the Rings." can be transformed into an RDF statement: Subject "Tolkien", predicate "wrote", and object "Lord of the Rings". If this information would be visualized, then the graph would look as shown in Figure 2.1.



(a) Visualized triple


http://books.net/writer#Tolkien
http://books.net/properties#wrote
http://books.net/books#LordOfTheRings

(b) Triple with URIs

Figure 2.1: RDF statement example

To access a specific resource object, a unique identifier is needed. URIs provide a common syntax for naming a resource, regardless of the protocol to access the resource. URIs are related to URLs (Uniform Resource Locators), both can either include a complete location or a path to a resource or a partial or relative path. The URI can optionally include a fragment identifier, separated by a "#" character. In the following example

```
http://books.net/writer#Tolkien,
```

is *Tolkien* the fragment and the rest represents the URI.

Namespaces can be used to simplify the URIs. A namespace can be defined in RDF/XML in the form of

```
xmlns:rdf=http://www.w3.org/1999/02/22-rdf-syntax-ns#
```

then an element in RDF, like

```
http://www.w3.org/1999/02/22-rdf-syntax-ns#resource,
```

can be shortened to

```
rdf:resource.
```

RDF Schema (RDFS) is an extension of the Resource Description Framework and provides a higher level of abstraction. It provides a type system for RDF. RDFS allows resources to be defined as classes, properties and values. Classes are also resources but they are also a collection of resources. Resources that belong to a class are called individuals of that class. RDFS can also be used to define relationships among classes and resources [Pow03, Car06].

2.1.2 Ontology

[wc3a] defines an ontology in the following way: *"On the Semantic Web, ontologies define the concepts and relationships used to describe and represent an area of concern. Ontologies are used to classify the terms that can be used in a particular application, characterize possible relationships, and define possible constraints on using those terms."*

Ontologies are the structural frameworks for organizing information. The Web Ontology Language (OWL) is a semantic markup language for publishing and sharing ontologies. It has been developed as a vocabulary extension of RDF. A OWL class is an abstraction mechanism for grouping resources with similar characteristics, its individuals are called instances.

OWL distinguishes two main categories of properties: "Object properties" (instance of class *owl:ObjectProperty*), which links individuals to individuals, and "Datatype properties" (instance of class *owl:DatatypeProperty*), which links individuals to data values.

Data values in OWL are instances of the RDFS class *rdfs:Literal*. Literals can be either plain, they have no datatype, or typed, they have an rdf:datatype attribute. All datatypes are instances of the class *rdfs:Datatype* [OWL04].

2.2 Visualization

In their schematic form, XML/RDF data can barely be understood by human begins. It is possible to read small sets of RDF data, but almost impossible to keep an overview or track relationships between the elements. Human beings need the data in another form to work with it. Here comes the field of Information Visualization into play that describes how to transform the information into a visual form. This way, human beings can explore and understand large amounts of information. To visualize information, users rely on applications that can read the data and create a visual representation out of it. Information is mostly represented in the form of a graph with nodes and edges, where nodes represent single entities and edges represent the relationship between two entities.

Cui [Cui07] states the following reasons why graphs are powerful visualization tools:

- Graphs are very simple models that can be applied to various fields. Every data set, that has internal relationships, can be modeled as graph, e.g. World-Wide-Web.
- Graphs are an abstract concept having a specific definition: The long history of graph theory has a very solid foundation and comes with a set of powerful algorithms for graph processing.
- Human beings have strong visual processing abilities. Information can be directly perceived and used without being interpreted and formulated explicitly.

- The most powerful way, to express that there is a relationship between graphical elements, is to connect them by a line.

Shneiderman [Shn96] presents seven tasks that an information visualization application should support:

1. *Overview:* Gain overview of the entire data collection. This may include a separate zoomed out view that contains a movable field-of-view box to control the contents of the detail view.
2. *Zoom:* Zoom in on items of interest. Smooth zooming helps users to preserve their sense of position and context.
3. *Filter:* Filter out uninteresting items. Users need to control the content of the display, so they can quickly focus on their interests by eliminating unwanted items.
4. *Details-on-demand:* : Select an item or group and get details when needed. A simple example would be a pop-up window that contains details about the selected item.
5. *Relate:* View relationships among items.
6. *History:* Keep a history of actions to support undo, replay, and progressive refinement. Normally, the process of information exploration has many steps, therefore the history of actions allows the user to retrace his steps.
7. *Extract:* Allow extraction of sub-collections and of the query parameter. Store desired subpart of the collection.

Tufte [Tuf01] builds a set of common-sense principles for data visualization:

1. *Show the data:* Show the data in its full complexity and let viewers make their own discoveries.
2. *Tell the truth:* Visual representations of data must tell the truth.
3. *Help the viewer to think about the information rather than the design:* Focus on content of data not the visualization technique.
4. *Encourage the eye to compare the data:* Comparative rather than descriptive visualizations
5. *Make large data sets coherent:* Present huge amounts of information compactly and reveal the data at several levels of detail.

2.2.1 Challenges in graph visualization

Graph visualization is a complex field, since it draws on ideas from several intellectual domains: computer science, psychology, semiotics, graphic design, cartography, and art. This makes the task of analyzing a set of data with relations, full of challenges. One of these challenges is the size of the graph, because large graphs can cause difficult problems: Algorithm complexity, display clutter, readability, and navigation.

- *Algorithm complexity:* The bigger the graph gets, the longer the processing time takes to compute the layout which makes it hard for real-time graph interaction.
- *Display clutter:* The more data items are visualized in the graph, the more the graph becomes cluttered and visually confusing.

- *Readability:* Human perceptual abilities usually require a small graph size.
- *Navigation:* How to navigate through a huge graph without getting lost? Users are having a problem to keep the overview over large graph with only a small display.

Another challenge is the complex data structure. Current data items typically can contain more than three attributes. A graph can have different types of nodes and links, because of the complex data structure, e.g. in an ontology graph, a node could be a resource or literal. To represent all this information at the same time, is a challenging task that can be solved by using visual cues, such as color, shape, or transparency. But these solutions also lead to another question: How should different information be visualized? Because due to the constraint of the human perception, some visual attributes have more representational power than others. For example, some pairs of colors are more distinguishable than others [Cui07].

Novak [Nov02] lists the size of the graph also as a key issue in graph visualization. Large graphs can compromise performance or can even reach the limit of the viewing device. If the number of elements is large then it will become impossible to discern between nodes and edges. Another problem is that the layout algorithms may not scale up when displaying thousand of nodes, and can make an algorithm completely unusable. Navigation and interaction is a very important help for information visualization when dealing with large graphs. One of these interaction tools is zooming and is quite indispensable in exploration of large graph structures. Zooming may lead to the lost of the contextual information, therefore keeping the focus is a very important complement to zooming. Two other important features, that help when dealing with large graphs, are clustering and filtering. Clustering is the process of grouping information to achieve more recognizable presentation of source data. Data Filtering should be used to reduce the amount of visualized information.

Chapter 3

Background

This chapter introduces some existing visualization tools that are used for visualizing the Semantic Web. It also provides an example of a framework that comes with its own scripting language for displaying data models. At the end of this chapter, several basic graph visualization frameworks are presented that can be used to visualize information as graphs.

3.1 Semantic Web Visualization Tools

In this section some tools are presented that provide aid in the visualization of RDF/OWL files.

RDF Gravity

RDF Gravity (RDF Graph Visualization Tool) [GW04] is a tool for visualizing RDF/OWL graphs. The tool allows the user to specify global and local filters. Global filters can hide or show specific edges based on their type. Local filters can be used to hide or show particular instances of nodes or edges. RDF Gravity has a full text search implemented to search over concepts, properties and instanced specified in a RDF file. A user has the ability to select multiple elements in the graph and change their position simultaneously. Also a Zoom functionality is integrated into the tool. The nodes and edges have different colors, styles, and figures according to the type they represent. RDF Gravity allows the user to make RDQL (RDF Data Query Language) queries to the RDF model and show the result directly in the graph. The strengths of the tool are the advanced filter capabilities and the possibility to use queries, but it is not ideal with respect to layouting. When a lot of elements need to be visualized, the graph gets too clustered.

OntoGraf

OntoGraf [pro] is an OWL visualization plug-in for the Protégé application. Protégé is an ontology editor and a knowledge-base editor. OntoGraf supports the interactive navigation of the relationships of an ontology. The plug-in comes with a set of different layout algorithms like spring, tree, radial, and grid. It allows the filtering of nodes and relationships to help the user to create the desired view of the ontology. Additional features are zooming capabilities, and configurable tooltips that show detailed information about nodes and edges. The plug-in allows the user to incrementally explore an OWL file.

GrOWL

GrOWL [KWV07] is a tool developed to visualize and edit OWL ontologies. It has been used in the Ecosystem Services Database, a data and analysis portal to assist the informed estimation of the economic values of ecosystem services. When a user enters a query into the database, the result is displayed with GrOWL. GrOWL also allows the development of ontologies. The tool comes with a set of layout algorithms that can be used for automatic layouting, but also allows the manual layout of individual nodes. The filer mechanism of GrOWL can restrict the view only to show class definition, the subclasses, the superclasses, or all instances associated with a selected node. GrOWL has implemented a prefix search: User can search for nodes using incremental matching.

Jambalaya

Jambalaya [SNM$^+$02] is a visualization tool for Protégé, designed to support ontology evolution and knowledge acquisition. In addition to standard visualization techniques, representing RDF data as nodes and edges, Jambalaya allows the nesting of nodes. A node representing a class can have nested nodes representing subclasses or instances of that class, which in turn can have other nested nodes, representing instances of those instances. To handle scalability issues, Jambalaya loads data incrementally from large data sources, strictly on as needed basis.

3.2 Mondrian

Another approach for the visualization of data is described by Meyer et al. [MGL06]. Instead of using a specific data format that can be a visualized by the tool, an interface has been created that allows the programmer to script the visualization.

To best way to understand Mondrian is to start with a script example (Listing 3.1) that visualizes a data set. The basic element in Mondrian is the view element which allows the programmer to paint on it. The next step is to add the nodes to the view. The example is built on a model of a source code with 38 classes, and the goal is to create a simple overview of the classes and show their relationships. The model object is the source of the classes. Each class is represented in the view as a node with the form of a rectangle with border. To give the rectangles different shapes, and therefore more meaning to the visualized parts: The methods of the object, representing a class, are used to set the width and the height and color of the object. In the example, the method NOM (Number of Methods) returns the number of methods the class has, and its value is used as the width for the rectangle. To show inheritance relationships in the view, the example uses edges.

Listing 3.1: Mondrian example script

```
view := ViewRenderer new.
view nodes: model classes
   using: (Rectangle withBorder width: #NOA;
         height: #NOM; linearColor: #LOC
         within: model classes).
view edges: model inheritances
   using: (Line from: #superclass to: #subclass).
view open.
```

Other features of Mondrian, to further enhance the graph, are *Layouting*, *Nesting*, *Adding inter-edges* and *Decorating Shapes*. Layouting is used to the rearrange the nodes in another form than in a horizontal line, for example a tree. Nesting is used to show more details for a class, like *"What are the methods in the class?"*, as a result, the rectangle of a class representation, has now additional inner nodes to indicate the methods for the class. Another possibility is to add the inter-edges to show the invocation interaction between the classes and their methods. The last feature is used to add some decorations to the shape e.g. adding arrows to the edges to show the inheritance path. The result with the additonal features is shown in Figure 3.1.

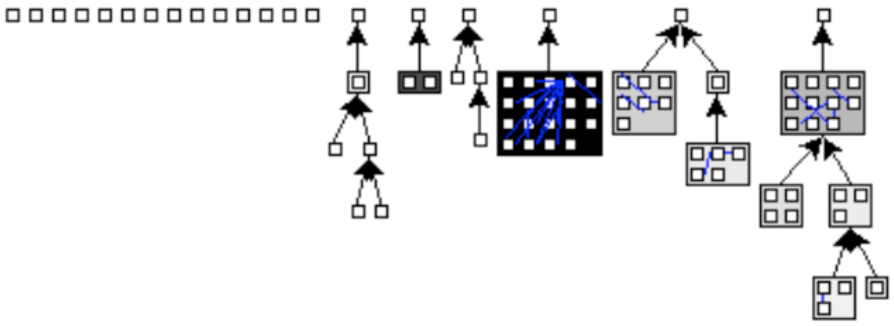


Figure 3.1: Visualized model with Mondrian [MGL06].

Mondrian shows a powerful approach that allows a programmer to quickly draft rich visualizations of a data model. But it does not provide real-time interaction with the visualized data or cannot provide on-demand information about the displayed elements via the user interface. To change the visualization, one needs to change the script. In contrast to Mondrian, the presented framework in this thesis will provide a basic user interface that allows the interaction with the displayed elements, but also allows the creation of specific scripts to improve the visualization of information.

3.3 Visualisation Frameworks

In this section some generic visualization frameworks are presented that can be used to visualize information in the form of graphs.

3.3.1 Jung - Java Universal Network/Graph Framework

Jung [MFN03] is a framework for the modeling, analysis, and visualization of graphs. The framework simplifies the creation of a graph, allows annotating of graph elements with any type of data, has its own drawing system and provides several layout algorithms. It is designed to be easy extendible for any kind of graph. The framework can handle dynamic graphs and also has a filter mechanism integrated.

The framework allows several ways to create a graph. The developer can load a graph from specified file types. The graph can also be built from scratch by specifying all nodes and edges. Another possibility is to use the graph generator. User defined data can be added through key-value pair association with graphs, edges, and nodes.

Jung also comes with set of implemented layout algorithms, like Spring, Fruchterman-Rheingold, and others. The provided rendering engine is based on Java Swing and has different implemented renderers that draw the nodes and edges. Jung also comes with a couple of functions for statistical analysis: Average shortest path, clustering coefficients and others. It is an ideally suited framework for building tools/applications related to network exploration and data mining [MFS^{+}05].

Although it is a very powerful and advanced framework, it does not perform well, when the amount of nodes increases. An early prototype of the ONTOX framework was implemented in Jung, but a couple of hundred nodes already slowed the rendering process considerably down. Some part may be caused by Java Swing, which still does not perform as well as a native implemented graphics engine. Another problem is that Jung renders always the entire graph. If something changes the entire graph is redrawn, it is not possible to create a partial refresh of a couple of elements.

3.3.2 Prefuse

Prefuse [pre06] is a Java user interface toolkit for constructing interactive information visualization applications. It provides support for visualization, animation, and interaction. The toolkit is built in Java using the Java2D graphics library.

The visualization process of Prefuse starts with abstract data set that needs to be visualized. The abstract data can represent a node, edge, tree node, graph or tree. The abstract data is the complete data set. Prefuse also has some functionality implemented to save or load such graph data sets. The next step, after the abstract data has been prepared, is to select which part of the data should be visualized. This process is called filtering in Prefuse. The toolkit creates visual analogues from the abstract data set. These visual analogues are then the subjects of all subsequent processing (e.g. layout, rendering). The last part of the visualization process is the view component which takes control of the screen drawing and interaction with the visualized data set. The implemented renders take care of painting the visual analogues [HCL05].

Prefuse is powerful toolkit to visualize different sorts of data sets, and also comes with a couple of already implemented layout algorithms. It allows configuration of the system on different levels. But Prefuse does not seem to be further developed, the last release was made in 2007.

Chapter 4

Technological Foundations

This chapter is intended to give an overview of the fundamental frameworks that are used to build ONTOX. It will give some basic knowledge about the platform where ONTOX is build on top of it. Also the core process and components of the drawing framework are explained here. The chapter concludes with a presentation of the programming language Groovy and introduces the core elements that are necessary to create a domain specific language.

4.1 Eclipse

Eclipse[1] is an Integrated Development Environment (IDE) that is mostly written in Java. It can be used to develop applications in Java and other programming languages. But Eclipse is also a general tools integration platform and provides the Eclipse Rich Client Platform (RCP) for developing general purpose applications.

Eclipse provides the basic functionality to run or create additional modules that extend the Eclipse platform. These modules, called plug-ins, represent the smallest unit of an Eclipse function that can be developed and delivered separately. Plug-ins are coded in Java, therefore a plug-in consist of Java classes, libraries, and other resources (e.g. images). All plug-ins have a manifest file declaring its interconnections to other plug-ins. An interconnection to a plug-in is defined through a named *extension point*, which can be extended by other plug-ins, otherwise an *extension* is a connection to an extension point in another plug-in [BdR06].

Eclipse provides a useful basis for implementing the ONTOX framework, through its plug-in architecture, the ONTOX framework can be built as an extension to the Eclipse environment or to create a standalone RCP application.

Standard Widget Toolkit (SWT)

The Standard Widget Toolkit (SWT) provides a common OS-independent API for widgets and graphics implementation. All information that is presented to the user, through the user interface on the Eclipse platform, is implemented on top of SWT. SWT uses native widgets to create buttons, lists, menus, etc., when they are available, otherwise SWT emulates these widgets. Through the tight interaction with the native window system, SWT gets an authentic native look and feel, has good performance, portability, and is a robust basis for GUIs [BdR06].

[1]Eclipse - http://www.eclipse.org/

4.2 Jena

Jena[1] is a Java framework for building Semantic Web applications. The framework was initially created by the HP Labs Semantic Web Programme and is open source. Jena comes with an API for RDF, RDFS, OWL, and the query engine SPARQL. It can read and write RDF in RDF/XML, N3 and n-Triples form. The framework also includes an in-memory and a persistent storage system. It is one of the most widely used Java APIs for RDF and OWL.

Jena provides helper methods to read an RDF or OWL file. The data is then stored in a Jena *Model* or *OntModel* object, from where the data can be queried. A basis type, to represent a RDF resource or literal, is the *RDFNode*. Important extended classes of *RDFNode* are therefore *Resource* and *Literal*. RDF statements are presented through objects of type *Statement*. Jena has different methods and classes implemented to query the model. All this methods return a *RDFNodes* or *Statements*, either as single object or as a list [jen].

4.3 Graphical Editing Framework (GEF)

The Eclipse Graphical Editing Framework (GEF) supports the creation of rich graphical editors and views as part of the Eclipse platform. It consists of three frameworks *Draw2D*, *Zest* and *GEF*. The last framework shares the same name as the Eclipse project name. In the rest of this work the name GEF will be used for the framework and not as the project name. The next sections will give a more detailed overview over these three frameworks and show what their main functionalities are.

4.3.1 Draw2D

Draw2D is a lightweight drawing framework for displaying graphical information on a Standard Widget Toolkit (SWT) canvas. The term lightweight means that all graphical components, which are called figures, are simply Java objects, with no corresponding resource in the operating system. On the other hand the SWT canvas is heavyweight because each SWT widget has an OS specific resource associated with it. The basic purpose of the framework is to render figures onto the canvas and to coordinate all aspects of displaying and interaction for a particular Draw2D diagram. Figures can be "nested" or "composed" of other figures in such a way that complex figures can emerge from simple geometric shapes and images. One important component of the Draw2D framework is the *UpdateManager*. It tracks which areas of the diagram have changed and need to be refreshed. Therefore the *UpdateManager* notifies only those figures in the diagram that need to be redrawn. This makes Draw2D more efficient and scalable for interacting with large graphs. Another framework component is the *EventDispatcher* which routes SWT events, such as mouse and keyboard events, to the appropriate figures on the canvas. Whenever a figure receives an event from the *EventDispatcher*, it redirects that event to any listeners attached to the figure which processes the event.

Figures

As mentioned before the Draw2D canvas contains figures, which are the basis elements and can contain additional figures. The z-order and the nesting of each figures determines what part of a figure is visible to the user and need to be rendered and painted. The painting process for each

[1]Jena - A Semantic Web Framework for Java - http://jena.sourceforge.net/

figure is split into several steps: painting the client area, the children, and the border. Draw2D comes with set of common figures such as line, rectangle, polygon and ellipse that can be used.

Connections

Connections are specialized figures that draw a line between two locations on the canvas. They are normally used to *connect* two figures. Since it is important to distinguish between the start and the end of a line, in a directed graph, the beginning is called the source and the end is called the target. Both the source and the target have their own anchors. The anchor is responsible for computing the location where the connection starts or ends. This means that each figure, to which a connection connects to, needs a specialized anchor to compute the correct intersection point between the connection figure and the node figure. Each connection can have a decoration associated with it. The decoration is a rotatable figure that is used to add an arrowhead or some other figure to the connection figure.

Standard connections normally draw a straight line form the source anchor to the target anchor, through using the default connection router. The connection router determines the path a connections takes from the source to the target anchor. Each connection can have its own connection router associated with it. For example the *FanRouter* is useful if there are connections that have the same source and target anchor. As soon as the router detects an additional connection with the same source and target anchor, it adds a bendpoint so that the connections to not overlap (FanRouter example in Figure 4.1.

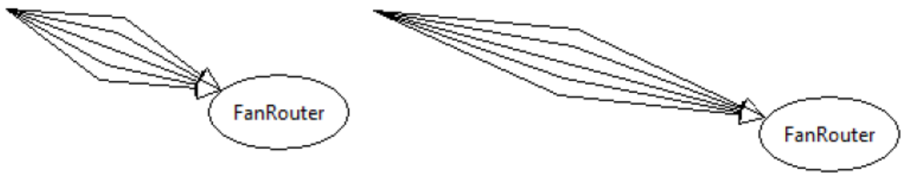


Figure 4.1: FanRouter example [RWC11]

Layers and Viewports

Draw2D uses different layers for different content. All layers are stacked on top of each other. The layer on the lowest level, the primary layer, contains the figures representing the content. On top of the primary layer is the connection layer, containing all the connections in the application. The order of the layers can be switched, for example to hide the connections behind the content figures, otherwise straight line connections would draw over other figures that are in between.

Draw2D creates automatically a viewport for showing only a portion of the underlying layer. If the figure is bigger than the user's interaction window, the viewport automatically adds scrollbars to the window. The *FreeformLayer* with its *FreeformLayeredPane* is a special primary layer that allows the expansion of space in any direction. An enhanced version of the *FreeformLayeredPane*, the *ScalableFreeformLayerdPane* add scaling capabilities to he layer, e.g. to zoom in and out of the layer.

4.3.2 Zest

The previous section showed the Draw2D framework, which provides the basic drawing functions to create advanced graphical items. Zest is another framework that is implemented as a layer on top of Draw2D. It provides an easier way to present the model information in diagram form than just using Draw2D. It can be used to paint a graph based on nodes and edges. On one side Zest facilitates the presentation of model, but on the other side it limits the presentation format and the ability to edit that information.

Zest also comes with its own set of layout algorithms, like a *Tree* or *Spring* layout algorithm for arranging the nodes in a graph. The layout algorithms are not bound to Zest and come in a separate package, therefore they can be used in other implementations that don't need Zest's graphical layer. It also comes with a couple of layout algorithms that can be used in combination with another layout algorithm.

Zest has the following layout algorithms implemented in release version 1.1.0 [RWC11]:

- **CompositeLayoutAlgorithm:** Used to combine multiple layout algorithms in sequence.
- **DirectedGraphLayoutAlgorithm:** Positions nodes in a single vertical column with root nodes on top of one another in the first row, child nodes on top of one another in the second row, and so on.
- **GridLayoutAlgorithm:** Positions nodes in a grid filled left to right, then top to bottom.
- **HorizontalShift:** Repositions nodes horizontally so that they do not overlap and take up the least amount of horizontal space. This algorithm is normally used in combination with other layout algorithms.
- **HorizontalLayoutAlgorithm:** Positions all nodes in a single row, evenly spaced within the current width of the diagram.
- **TreeLayoutAlgorithm:** Positions root nodes in the first row, child nodes in the second row, grandchild nodes in the third row, and so on.
- **HorizontalTreeLayoutAlgorithm:** Similar to a *TreeLayoutAlgorithm* but positions root nodes in the first column, child nodes in the second, grandchild nodes in the third, and so on.
- **RadialLayoutAlgorithm:** Positions nodes similarly to the *TreeLayoutAlgorithm* except with the roots at the center, child nodes in a circular fashion around the root nodes, grandchild nodes in a circular fashion around the child nodes, and so on.
- **SpringLayoutAlgorithm:** positions nodes having more connections toward the center of the diagram and nodes having fewer connections around the edges.
- **VerticalLayoutAlgorithm:** Positions all nodes in a single column.

It is also possible to implement custom layout algorithms or extend the existing ones. The next version of the Zest Layout Package will come with additional layout algorithms, that can be easily integrated into the ONTOX framework through their modular implementation.

4.3.3 GEF

This section will have a look at the Graphical Editing Framework (GEF) plug-in. Both Zest and GEF are built on top of Draw2D whereas Zest is a simpler framework that requires fewer lines of code than GEF. If Zest provides all functionality one needs then there is no reason to use GEF. But GEF provides more customization and flexibility for both rendering a model and interacting with a model. It enables developers to design a visual representation of a model component, using Draw2D figures. Trough GEF a developer can easily add editing rules, listeners and business logic to a GEF diagram.

The framework is based on the Model-View-Controller (MVC) architecture. The MCV architecture has three main components: the *model*, *view*, and *controller*. The model manages the behavior and data of the application domain. It responds to request for information about its state (normally from the view), and responds to instructions to change the state (normally from the controller). The model contains the business logic and information that persists among application sessions. Listeners attached to model objects are notified when state changes occur, sent by the controller. GEF can display man different types of models from Plain Old Java Objects (POJO) model to Eclipse Modeling Framework (EMF) models. The view renders the model into a visual interactive representation, the user interface. It is responsible for drawing the diagram and passing user events to any attached listeners. View elements are normally built out of Draw2D figures. The controller receives user input from the view and initiates a response by making calls on model objects. In the other direction, the controller updates the view when the underlying model changes. Implemented objects of the controller are in GEF called *EditParts*.

GEF provides the developer with additional functions to easily create a graphical editor. One example is commands. A command encapsulates a single change to the model, like create or delete a visual element. After a command is executed, it is placed on the command stack. This allows the user choose an undo or redo operation at a later time. To control a particularly type of behavior, that can be performed on a model element, GEF uses *EditPolicies*. An *EditPolicy* controls the commands that can be performed on a model, defines the feedback a user sees, and can delegate/forward the defined behavior to other *EditParts* and *EditPolicies*. GEF also provides the developer with a series of predefined actions for creating some basic menu items and associating them with their appropriate commands e.g. *Undo*, *Redo*, *Edit*, and *Delete*.

Another useful feature of GEF is the creation of a palette and tools. A palette can contain several toolbars and each toolbar can hold several tools. In an implemented toolbar a user can select a tool to perform an operation on the editor. A palette can contain serveral different types of tools, useful for the selection of existing elements, and creation of new elements and connections, and other needed tools [RWC11].

4.4 Groovy

Groovy[1] is an agile dynamic language for the Java Platform with many features that are inspired by languages such as Python, Ruby and Smalltalk. Groovy is often referred as a scripting language, but it also can be precompiled into Java bytecode and be integrated into Java applications. The language is closely tied to the Java platform, because a part of its basis is implemented in Java whereas the rest is programmed in Groovy itself. Groovy allows harnessing the power of the Java platform with all its available libraries. The integration of Groovy in Java is easy since every Groovy type is a subtype of *java.lang.Object* and every Groovy object is an instance of a type in a normal way. Using Groovy in Java is also easy: If the Groovy class files are in the same classpath then the Groovy classes can be used in a normal fashion [KGK+07].

4.4.1 Features of Groovy

Groovy comes with a set of features that distinguish this language from Java and allows a developer to code at a higher level. This section explains some of these features with examples presented by Dearle [Dea10]. Only those features are considered here that are important in connection with the creation a domain specific language.

Static and optional typing

In statically-typed languages, variables must first be declared with a type before they can have a value assigned to them. In Groovy is assigning a type optional. The type can be left out and will then be determined at the time of value assignment:

```
String str1 = "I'm a String"    //with type definition
str2 = "I'm also a String"      //without type definition
```

Native support for lists and maps

Groovy adds a native support for all of the Java collection types:

```
authors = ['Shakespeare', 'Beckett', 'Joyce', 'Poe']    //List
book = [ fileUnder: "Software Development",title:       //Map
        "Groovy for DSL" , author: "Fergal Dearle"]
```

Closures

One of the most powerful language features of Groovy are closures. Closures are anonymous code fragments that can be assigned to a variable and reused whenever needed. They also can contain as many statements as needed, and optionally may specify a list of identifiers in order to name the parameters passed to it, the "->" arrow marks the end of the identifiers list.

```
//Example 1

biggest = { number1, number2 -> number1<number2?number2:number1 }
// The method can be invoked through the call method of the Closure class
result = biggest.call(7, 1)

// The closure reference can be used as if it was a method
result = biggest(3, 5)

// And with optional parenthesis
result = biggest 13, 1
```

[1]Groovy - http://groovy.codehaus.org/

```
//Example 2

def evenClosure = {element ->        //closure that prints even numbers
    Number number = Integer.parseInt(element)
    if (number%2 == 0){
        print number + " "
    }
}
numbers = ["1", "2", "3", "4", "5", "6", "7", "8"]
print ("Even Numbers-->")
numbers.each(evenClosure) //closure can be passed as arguments
```

Optional syntax

Groovy comes with a couple of syntax simplifications. As already mentioned before, variable type annotations are optional. The type will be determined at runtime:

```
int a = 3
b = 2 //determined at runtime
String t = "hello"
s = 'there' //determined at runtime
```

The trailing semicolon at the end of a statement is also optional. As long as every statement is on a separate code line, then semicolons are not needed:

```
int a = 3; int b = 4;
c = 2
d = 5; e = 6
```

The parentheses in method calls are also optional when the called method has some parameters that are passed on. The same works also for closures, they dont need parentheses when they are called.

```
println( a );
c = 2
print c
printit = { println it }
printit c
```

All the optional syntax creates a much looser programming style and is a big benefit when Groovy is used to build a Domain Specific Language (DSL). Through dropping the semicolons and parentheses the code gets more legible for a non-technical audience.

```
with full syntax:
    Account account = getAccountById( 234 );
    creditAccount( account, 100.00 );

with removed optional syntax:
    account = getAccountById 234
    creditAccount account, 100.00
```

Scripting Support

Groovy provides a *GroovyShell* class for loading and executing Groovy code at runtime. The *GroovyShell* can evaluate any expression or script in Groovy. Through the *Binding* class, variables can be altered from outside the script, or created outside of a script and passed into it.

```
// call groovy expressions from Java code
Binding binding = new Binding();      //Init Binding object
binding.setVariable("foo", new Integer(2));     //'foo' = Integer(2)
GroovyShell shell = new GroovyShell(binding);  //Init Groovy shell

Object value = shell.evaluate(
    "return foo * 10"
);

print value;     //prints 20
```

4.4.2 Groovy Builders

Design patterns are in general reusable solutions to commonly occurring problems in software engineering. One type is called "creational patterns". The creational design patterns hide the instantiation process and therefore help make a system independent of how its objects are created, composed, and represented. One example for the creational pattern is the "Builder" pattern. The builder pattern is central element in Groovy for creating a domain specific language and therefore also used in this thesis for implementing the ONTOX DSL (section 6.4).

Gamma et al. [GHJ94] states the intent of the builder pattern as followed: *"Separate the construction of a complex object from its representation so that the same construction process can create different representations."* The builder pattern can be applied when the building process of a complex object should be independent of the parts that create the object, or when the construction process must allow different representations for the object that is being constructed

The builder pattern has four participants:

- **Builder:** The Builder is an abstract interface which specifies methods that are used to create parts of the of a product object.
- **ConcreteBuilder:** This is the concrete implementation of the builder interface that constructs a specific product.
- **Director:** The Director constructs an object using the builder interface methods.
- **Product:** The Product is the result of the building process. It represents the complex object that is under construction.

These four participations work in the following way: The client instantiates the director object and configures it with a builder object. Next, the director informs the builder whenever a part of the product should be built. The Builder handles the request from the director through the interface methods and adds parts to the product. When the product is ready, the client retrieves the finished product object from the builder ([GHJ94]).

The builder pattern is also implemented in Groovy which is a powerful part of this language. Normally one would expect a director component that is implemented through a director class, but Groovy takes this a step further by providing the *GroovyMarkup*, as a mini DSL, that is directly embedded in the building process, right into the language (Example in Listing 4.1). The *GroovyMarkup* is nothing more than a method call syntax combined with closures and named parameters. Groovy provides various native support for different markup languages from XML, HTML, SAX, W3C DOM, Ant tasks, Swing user interfaces and so forth. The *GroovyMarkup* scripts, have an unusual syntax, but are still just plain Groovy scripts and can therefore combined with regular program logic.

Listing 4.1: Example of MarkupBuilder for building HTML pages.

```
def html = new groovy.xml.MarkupBuilder()
html.html {
   head {
      title "HTML Example Page"
   }
   body {
      h1 "Example heading for HTML page."
   }
}
```

The *GroovyMarkups* are implementations of the *MarkupBuilder*, which is derived from the *BuilderSupport* class. The *BuilderSupport* class is basis for building the own custom builder. According to [Dea10], it is not necessary to understand how the builder is implemented, but it is important to understand how the builder pattern relies on just a few important Groovy language features:

- Closure method calls: Methods in Groovy can accept closures as parameter. Groovy allows the developer to define the body of the closure after the method call and after the other parameters. Therefore the closure block in a builder, looks like a named block of code.
- Closure method resolution. When a method is invoked with the body of a closure and that method does not exists, then Groovy tries to locate the missing method in another object.
- Pretended methods: Groovys Meta Object Protocol (MOP) allows the developer to respond to method calls that do not exist in a class. One can therefore "pretend" that a method exists.
- Named parameters: Individual map elements can be passed alongside other method parameters, giving the effect of a named parameter list.
- Closure delegate: The owner of a closure can be changes through the delegate property, which allows another class to handle its method calls. A builder can therefore manipulate by whom the methods are handled.

To implement a builder, one has to create a new class that extends the *BuilderSupport* class, which provides an interface that implements a node based construction process, the `createNode( Object name, Map attributes, Objects value)` method. This method is called whenever a pretended method is encountered, a method that does not really exist but has been entered in the code block. The *name* parameter contains the name of the pretended method call. The *createNode* method catches all method invocations in the closure code block that are pretended method calls. The *attributes* parameter is a map that contains all passed key-value parameters to the prevented method. The *value* represents a single object, that was also passed to the pretended method. The *createNode* method can return any type of object, which then can be used in the next code block, if it is a nested structure. With this node based construction process, the *BuilderSupport* takes therefore care of pretended methods and also delegation. A developer has just to implement the *createNode* method and can take care of the call hierarchy of all closure code blocks in there. Another important method is the `nodeCompleted( Object parent, Object node)` method, that is called after all children of a node have been created, or in other words: It is called when the last line of a closure code block has been executed. The *nodeCompleted* function can, for example, be used to do some finish or cleanup operation [Dea10].

4.5 Domain Specific Language

A Domain Specific Language (DSL) is a programming language, designed for a particular application domain. The benefit of a DSL is, that it is more concise and therefore quicker to write. It is also easier to maintain and can often be written by non-programmers. The development of a DSL, on the other hand, is not easy because the developer needs both, domain knowledge and language development expertise [MHS05].

A DSLs offer the following advantages over general-purpose programming languages:

- Allow established domain specific notations
- Allow to map domain-specific constructs and abstraction into the DSL
- The DSL offers possibilities for analysis, verification, optimization, parallelization, and transformation
- DSLs need not be executable (e.g. domain-specific data structure representations)

The first phase in DSL development, is the identification of the problem domain and the gathering of domain knowledge. The domain model is the result of the formal domain analysis. There are several domain analysis methodologies available. An overview can be found in Mernik et al. [MHS05]. The next step is to design the DSL. The easiest way to design a DSL is to base it on an existing language, because this makes the development easier. The last phase is the actual implementation of the DSL, if it is executable.

Chapter 5

OntoX Framework

This chapter explains the components of the ONTOX framework. First, the prototype application will be presented with all its main features. In the second part, the implemented DSL is explained with some actual examples that show the capabilities of the framework.

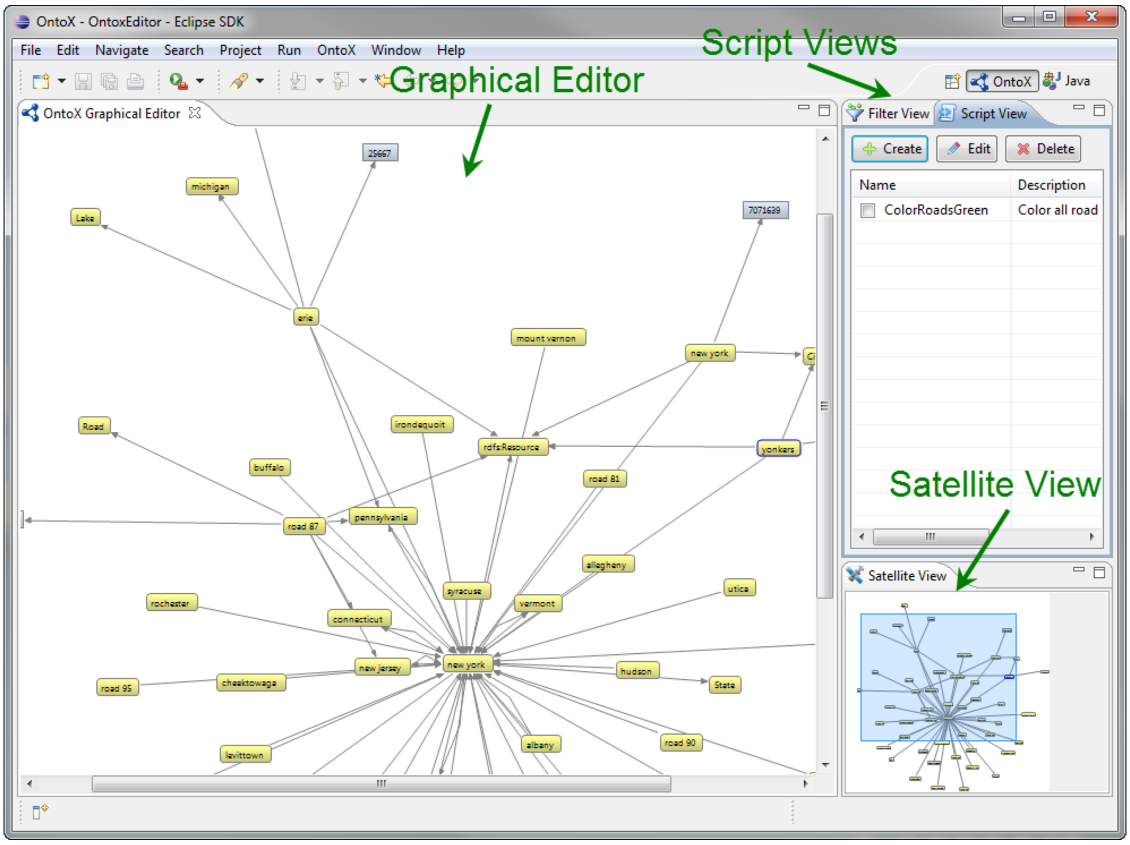
5.1 OntoX Application

The current prototype of the application can be used under Eclipse as plugin or as standalone Rich Client Platform (RCP) application. This section gives a short overview about the functionality and the current available user interface of the prototype. The data set that is used in the in this section, is the Geography ontology presented in [Zel95].

A basic overview of the application is given in Figure 5.1 with its main components of the user interface:

- **Graphical Editor:** The editor area shows the current visible part of the graph. It also allows user interaction, like moving nodes to new positions, scrolling to other parts of the graph and expanding/collapsing of node elements.
- **Script Views:** Currently there are two different views available that show all the implemented scripts. The *Filter View* lists all implemented filter scripts that are used to filter unwanted nodes and edges from the application model. The *Script View* lists all the other scripts that are not used for filtering purposes, but for changing the design and structure of the graph. Both Views have three buttons that are used to *create*, *edit* or *delete* scripts. All scripts can simply be activated and deactivated by clicking on the checkbox that is visible before the script name.
- **Satellite View:** The satellite view is used to give an overview of the entire graph, not just the part that is currently visible in the editor area. The user can also interact with this view and move the viewport to a different location.

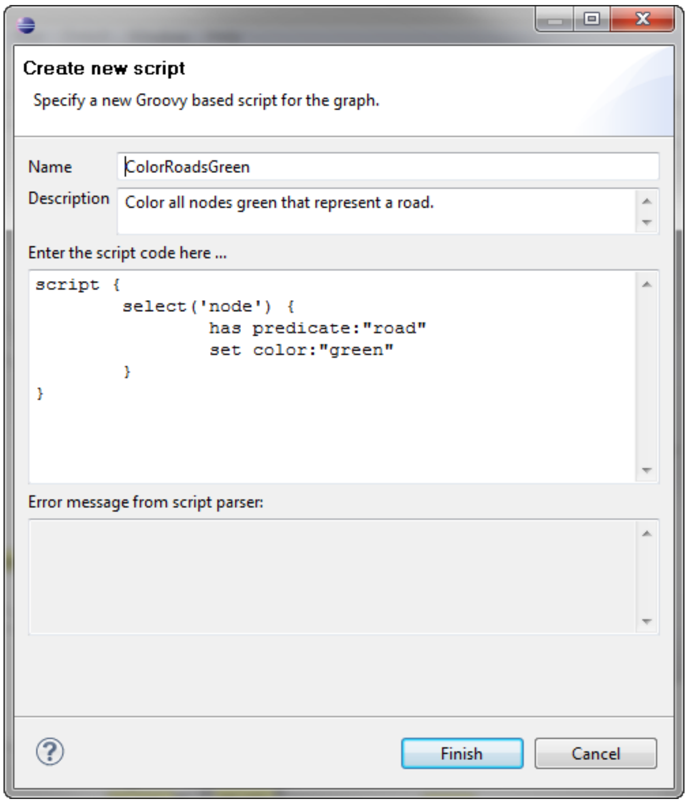
The Filter View has a filter *CoreEdgeFilter* that is always present in this view. It is a hard coded filter into the system that filters out some unneeded edges (RDF statements) that normally would only overload the visualized graph and are most of the time not needed. The following predicates (properties) are filtered by the *CoreEdgeFilter* including its connected nodes (if they are not furher connected to other nodes):



Name	Description
ColorRoadsGreen	Color all road

Figure 5.1: Overview of the application components

- **owl:differentFrom:** An *owl:differentFrom* statement indicates that two URI references refer to different individuals.
- **owl:disjointWith:** Used to express the disjointness of a set of classes. It guarantees that an individual that is a member of one class cannot simultaneously be an instance of a specified other class.
- **owl:topObjectProperty:** The object property connects all possible pairs of individuals.
- **owl:topDataProperty:** The data property connects all possible individuals with all literals.
- **owl:propertyDisjointWith:** Used to specify that two properties are disjoint properties.
- **rdfs:comment:** A description of the subject resource. The comment is already included in the tooltip and therefore it is not necessary to visualize it in the graph.
- **rdfs:label:** This predicate represents human-readable name for the subject. If it is available, then the label is used in the visualized node name. Therefore there is no need to create an extra node and edge for the label statement.



Name: ColorRoadsGreen

Description: Color all nodes green that represent a road.

Enter the script code here ...

```
script {
        select('node') {
                has predicate:"road"
                set color:"green"
        }
}
```

Error message from script parser:

Figure 5.2: Wizard to enter new scripts into the application.

Wizard

To make it as easy as possible for the user to enter new scripts, a wizard dialog box (Figure 5.2) has been implemented into the application. The user can create new scripts or edit the existing ones through the wizard. As soon as a user has entered a new script, the Groovy parser will check the syntax of the script and display a failure message, if the parser throws an error. It there is no problem with the script, the wizard stores the script in the integrated persistent storage system.

Components of the wizard dialog box:

- **Name input field:** In this field the user needs to enter a unique name for the script. The wizard will display an error message, if the field is empty or if the name does not match the regular expression that evaluates the script name. The name is important because it will be used throughout the system to uniquely identify the script by name.
- **Description input field:** A user can enter a description for the script to give a more detailed explanation what the script does or what its purpose is. The field just has an informational purpose to the user.
- **Script input field:** This is multiline input field that allows the user to enter the script code (the ONTOX DSL).
- **Parser error field:** This is a read only text field. Its purpose is to display error message that the Groovy script parser detectes when it examines the script code. The user receives some

information where a syntax failure is and can correct it. This helps to prevent exceptions during runtime.

- **Type** (only filter wizard): The type defines that it is a script for filtering nodes or a script for filtering edges.

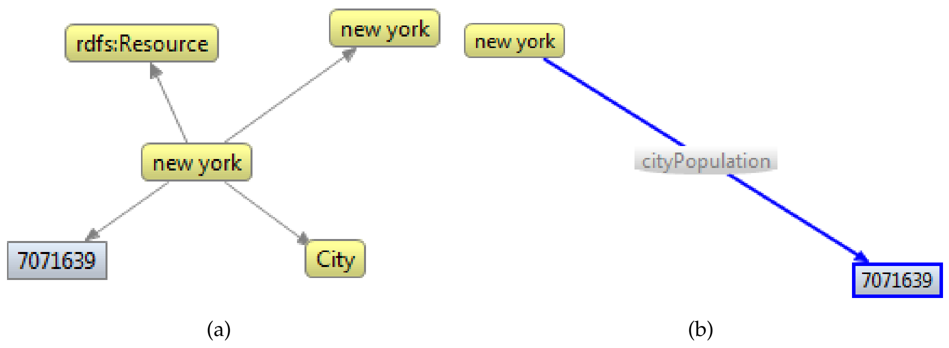


Figure 5.3: Node and edge labels

Label

The labels of nodes and edges are automatically computed according to the following system (Example in Figure 5.3): If the ontology has a *rdfs:label* defined for the nodes, then the *rdfs:label* will be used as the label otherwise the local name (fragment identifier), which is defined in the URI, will be used. Since some nodes have different namespaces but may have the same name (label or local name), the namespace also has to be taken into consideration when creation the node label. For example the URI

```
http://www.w3.org/1999/02/22-rdf-syntax-ns#Resource
```

of the node *Resource* will be shortened to namespace and local name:

```
rdf:Resource
```

If the node has the basic namespace in its URI, then no URI prefix will be added to the label. A similar implementation is used for all edges: The system computes the edge label according to the predicate local name and adds its namespace. To display a literal, the system uses the value of the literal as node label.

Context Menu

Through pressing the right mouse button the user can activate a context menu (Figure 5.4). The context menu presents additional tools that help to explore the graph.

- **Zoom:** A *Zoom In* and a *Zoom Out* action that allows the user to enlarge or to scale down the graph.
- **Collapse:** This action collapses the current selected node. All of its connected edges (incoming and outgoing) will be removed from the graph. If one of the edges is connected to another node that has no further connections, then that node will also be removed. The collapse action is a useful implementation to reduce the amount of edges currently displayed in the graph, and allows the user the keep a better overview of the entire graph.

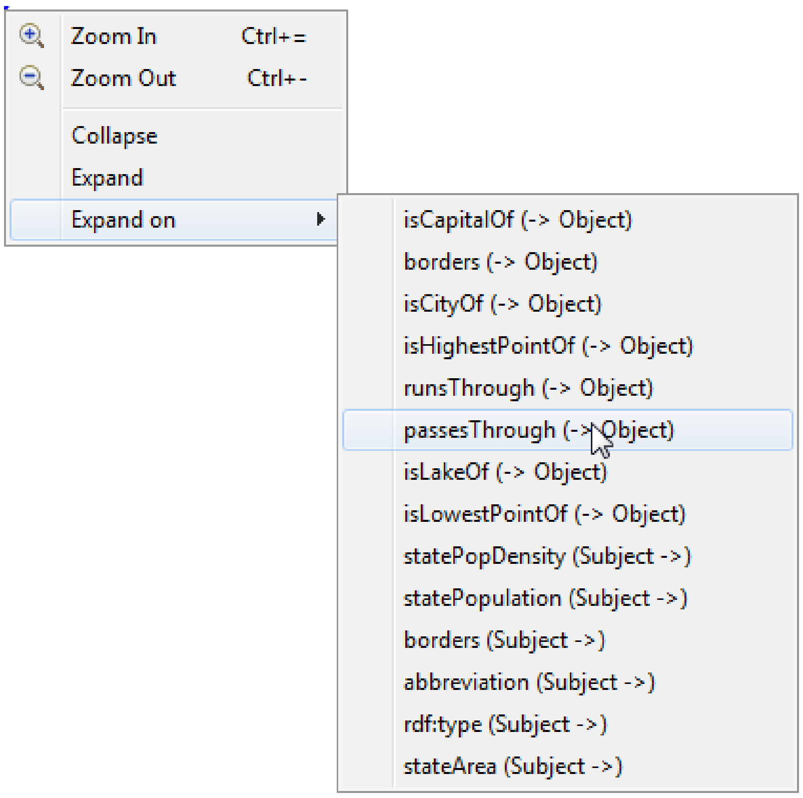


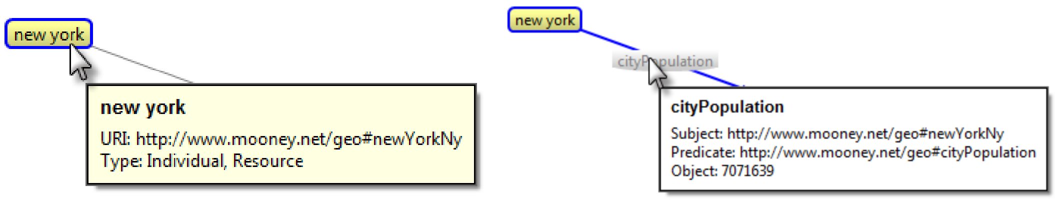
Figure 5.4: Context menu

- **Expand:** This action, fully expands all edges (predicates) for the current selected node. It queries the RDF model and retrieves the additional RDF nodes and statements.

- **Expand on:** The *Expand on* action is similar to the *Expand* action, it adds new edges and nodes to the graph, but it does not expand all edges at once. The user can selected which predicates he likes to see as edge in the editor. The content of this action is dynamically computed every time the user activates the context menu, since different nodes have different edges. Behind the label, in brackets, there is always the indication whether this node is the subject or the object of the expanding statement.

Tooltips

The Tooltips (Figure 5.5) provide additional information about nodes and edges in the system. They appear automatically as soon as the user hovers with the mouse pointer over one graph element (node or edge). When hovering over a node, the tooltip shows additional information like the full URI of the RDF resource or the type of the node. In an ontology, a RDF node can be a *Literal* or a *Resource*, and if it is a resource, then it can additionally be a *Class* or an *Individual* in the context of the ontology. The tooltip also shows a description text, if the RDF node has one. For literal nodes the tooltip displays the datatype, if there is one set in the ontology, otherwise the datatype is set to *plain literal*. The edge tooltip shows the current RDF statement that is represented



new york
URI: http://www.mooney.net/geo#newYorkNy
Type: Individual, Resource

cityPopulation
Subject: http://www.mooney.net/geo#newYorkNy
Predicate: http://www.mooney.net/geo#cityPopulation
Object: 7071639

(a) (b)

Figure 5.5: Node and edge tooltips

with the selected edge. It shows the full URI of the statement elements: subject, predicate and object.

5.2 OntoX DSL

The ONTOX framework comes with its own implemented DSL that is based on Groovy. Groovy is the ideal choice, because it is implemented on top of Java and provides the necessary functionality to create a DSL, as described in section 4.4. The simplified syntax of Groovy, the ability to name parameters, and the implementation of builders, are the basic foundation elements to create a DSL in Groovy [Dea10].

The ONTOX DSL is designed to change the design and structure of the shown graph in the editor part, or to filter out elements that are not needed. The ONTOX framework is not designed to visualize only a specific ontology, it is designed to handle any ontology. Therefore the implemented editor only provides the basis for user interaction and ontology visualization. The editor has no further knowledge about the context where this ontology is used: What information is the user searching for, and how could the framework assist him to reach his goal. This is the part where the ONTOX DSL comes to help. With this simple programming language, the user can give the visualized graph elements more meaning.

ONTOX DSL splits the scripts into two categories, the *Graph Scripts* (used to modify the graph) and the *Filter Scripts*, used to filter elements from the graph. The graph scripts are only used to change the graph and search the model that represents the graph. The filter scripts are used to make elements in the graph invisible by updating the ONTOX model. The ONTOX model is the model that represents the data source of the visualized items. Additionally these filter scripts are also used for the Jena model, preventing the system from adding new nodes or edges to the graph. It is faster to filter elements directly at the data source, then later on, when the elements have been added to the graph. Since the user can decide when to activate a filter, the script needs to filter both models. The filtered elements in the graph model are not deleted, only set to invisible, to make them reappear at the same position when the user decides to deactivate the script.

The following section will give an overview over the complete ONTOX DSL syntax than can be used to interact with the framework.

5.2.1 Graph Scripts

Normally the user uses this script type to change the design or the structure of the graph. Such scripts are initialized with a *script* block statement (see line 1 in Listing 5.1), it instantiates the internal structure that is needed to run the script.

Selection Statement

The *select* statement (see line 2 in Listing 5.1) is used inside the *script* block. Its purpose is to search the graph for specific nodes or edges. The return value is a select object that contains the result of the query. The select code block has one parameter that specifies the search type. The parameter is of type String and can be *node*, *edge*, or *root*. In Listing 5.1 the parameter value is *'node'*, this means the select statement iterates through all nodes in the graph, when performing the search. If the parameter has the value *'edge'*, then the selection process iterates trough all edges. Parameter value *'root'* represents the root model of the graph that is not visible to the user. Every other node, that is visible, is a direct or indirect child of the root node (the model of the graph is implemented as tree and will be explainted in more detail in section 6.2). The parameter *node* is not necessary when looking for nodes, *select* and *select('node')* instantiate the same select object).

Listing 5.1: Basic structure of a select script

```
1   script {
2      select('node') {
3         has predicate:'road'
4         set color:'green'
5      }
6   }
```

The script in Listing 5.1 has the following meaning: *"Find all nodes that have a predicate named 'road' and color these nodes 'green'."* The *has* statements in the select code block, is used to set the search attribute. The user can enter several *has* statements in the select block to refine the search attributes. A full list for all *has* statements when looking for a node can be found in Table 5.1. The *set* statement (see Table 5.2) is used to change the style attributes of the nodes that are part of the current result.

has		
Syntax	Value(s)	Description
has name:<value>	String	Search for the specified name.
has uri:<value>	String	Search for the specified resource URI.
has predicate:<value>	String	Search for a node that matches the entered predicate name.
has predicate:<value>, as:<value>	’subject’ ’object’	Same as “has predicate” statement with addtional parameter “as”, to define if the node is subject or object in the corresponding RDF statement
has type:<value>	’literal’ ’resource’ ’class’ ’individual’	Look for a specific type of node.

Table 5.1: All possible values for “has” in a node selection statement.

set		
Syntax	Value(s)	Description
set name:<value>	String	Sets a new name for the selected node.
set color:<value>	[r,g,b] ’blue’ ’black’ ’cyan’ ’darkblue’ ’darkgrey’ ’darkgreen’ ’grey’ ’green’ ’darkred’ ’darkyellow’ ’red’ ’white’ ’yellow’	Sets a new color for the selected node. It is possible to create a color by specifing an array [red,green,blue] or to enter a textual representation of a color.
set fontColor:<value>	[r,g,b] ’blue’ ...	Sets a new color for the font of the selected node.
set figure:<value>	’standard’ ’circle’ ’roundedRectangle’ ’rect’	Sets a new simple figure for the selected node.
set width:<value>	int	Sets the width of the node.
set height:<value>	int	Sets the height of the node.

Table 5.2: All possible values for “set” in a node selection statement.

As mentioned earlier the select code block returns a *select object* that contains the result of the query. To access this list, the user can use the *result* property of the select object (returns a Groovy List with objects of type Node.groovy or Edge.groovy). The select object has a property *size* than can be used to get the size of the List (see Listing 5.2).

Listing 5.2: Properites of the select object

```
    script {
       s = select('node') { }
       s.result //returns a list of nodes (Node.groovy)
       s.size //returns the size of the result list
    }
```

If the *select* statement is used to query the graph for edges, then a slightly different *has* and *set* statements (with different attributes) need to be used (ses Table 5.3 and 5.4). The options for the root select statement are listed in Table 5.5. The root select only allows the user to change some global graph settings; a *has* statement in a root select has no functionality.

has		
Syntax	Value(s)	Description
has name:<value>	String	Search for the specified name.
has uri:<value>	String	Search for the specified predicate URI.
has predicate:<value>	String	Search for a node that matches the entered predicate name.

Table 5.3: All possible values for "has" in an edge selection statement.

set		
Syntax	Value(s)	Description
set lineStyle:<value>	'solid' 'dot' 'dash' 'dashdot' 'dashdotdot'	Sets a new line style for the selected edge(s).
set lineWidth:<value>	int	Sets the width of a line.
set color:<value>	[r,g,b] 'blue' ...	Sets a new color for the selected edge. It is possible to create a color by specifing an array [red,green,blue] or to enter a textual representation of a color.

Table 5.4: All possible values for "set" in an edge selection statement.

The *select* statement can have an additional *eval* code block (see Listing 5.3). The *eval* statement can be used to enter plain Groovy code inside the *select* code block. This allows the user define new and more precise selection properties. The selection process will take the *eval* code block and iterate through all nodes or edges (depending on the *select* script type) in the system. Every item (node or edge) is then tested against this script. An *eval* code block needs always to evaluate to

set		
Syntax	Value(s)	Description
set algorithm:<value>	'spring' 'radial' 'tree'	Sets an alogrithm for the entire graph.

Table 5.5: All possible values for changing settings of the graph (root node).

true or false. If the result is true, then the item will be added to the result set, otherwise it will be ignored. The *eval* block is necessary to tell the builder that the statements inside the *eval* block should be processed at the end, after all *has* and *set* attributes have been collected and analyzed.

Listing 5.3: Select statement with *eval* code block

```
script {
   select('node') {
      has predicate:'city'
      eval {
         name.contains('new') //or name.contains 'new'
      }
      set color:'green'
   }
}
```

The *eval* block comes with some predefined variables that can be used inside the evaluation script block. In a *select('node')* block are the following variables in the *eval* code block available:

- Variable "name" is a String object that contains the name of the current node under evaluation.
- Variable "uri" is also a String object that contains the URI of the node under evaluation.
- Variable "node" represents the node object of type "Node.groovy". The detailed implementation of the class with all available fields and methods is given in Appendix B.
- If the node is a literal, then the variable "value" is additionally available in the script block.

In a *select('edge')* statement are the following variables in the *eval* code block available:

- Variable "name" is a String object that contains the name of the current edge under evaluation.
- Variable "uri" is also a String object that contains the URI of the edge (URI of the predicate in a RDF statement).
- Variable "statement" is an object of type RDF statement (Jena Statement).
- Variable "edge" represent the edge object of type "Edge.groovy". The detailed implementation of the class with all available fields and methods is given in Appendix C.

Inside the *select* block, it is also possible to define closures for the *set* and *has* values. For example the statement *set width:80* sets the width of the figure to 80 pixels. This is a constant entry that is used for all nodes that have been found by the select statement. *Has* and *set* are methods and *width:80* is the parameter of the method. In more detail: *width:80* is a single map entry, where *width* is the key and *80* the value. Instead of using constant values, it is also possible (only inside the *select* code block) to use a closure as the value of the map entry. The same objects, that were defined previously for the *eval* code block, are also available in the closure blocks of map values.

Subgraph Statement

The *subgraph* statement (see example in Listing 5.4) can be used to create a new subgraph in the graph. A subgraph is represented as a new node in the editor with a specified amount of child nodes. The user specifies the children of the subgraph through the *select* statements that are defined inside the subgraph code block. The *subgraph* statement can have one or several *select* statements in its code block, but can only be used to query nodes and not edges. Nesting of *subgraph* statements is also possible. All available *set* statements for the subgraph are listed in Table 5.6.

Listing 5.4: Example for a 'subgraph' statement

```
script {
   subgraph {
      select {
         has predicate:'road'
      }
      select {
         has predicate:'city'
      }
      set algorithm:'radial'
   }
}
```

set		
Syntax	Value(s)	Description
set name:<value>	String	Sets a new name for the selected subgraph node.
set color:<value>	[r,g,b] 'blue' ...	Sets a new color for the selected subgraph node. It is possible to create a color by specifing an array [red,green,blue] or to enter a textual representation of a color.
set fontColor:<value>	[r,g,b] 'blue' ...	Sets a new color for the font of the selected node.
set figure:<value>	'standard' 'circle' 'roundedRectangle' 'rect'	Sets a new simple figure for the collapsed subgraph node.
set algorithm:<value>	'spring' 'radial' 'tree'	Sets an alogrithm for the subgraph node.
set width:<value>	int	Sets the width of the collapsed subgraph node.
set height:<value>	int	Sets the height of the collapsed subgraph node.
set expWidth:<value>	int	Sets the width of the expanded subgraph node.
set expHeight:<value>	int	Sets the height of the expanded subgraph node.

Table 5.6: All possible values for "set" in a subgraph statement.

5.2.2 Filter Scripts

Filter scripts have their own basic code block (see line 1 in Listing 5.5). The *filter* statement initializes a new filter object. The filter code block allows several *has* statements as seen before in the *select* statement. All the statements for *has* can be found in Table 5.7. The setting, that a filter script should be used to filter nodes or edges of the graph, is not set in the script itself. The user has to select the filter type in the wizard, which is used to enter a script of type filter.

Listing 5.5: Example for a 'filter' code block

```
filter {
   has predicate:'isCityOf'
}
```

has		
Syntax	Value(s)	Description
has name:<value>	String	Search for the specified name.
has uri:<value>	String	Search for the specified resource URI.
has predicate:<value>	String	Search for a node that matches the entered predicate name.
has subject:<value>	String	This statement can be used in the context of an edge search. It specifies the subject name of the RDF statement.
has object:<value>	String	This statement can be used in the context of an edge search. It specifies the object name of the RDF statement.
has type:<value>	'literal' 'resource' 'class' 'individual'	Look for a specific type of node.

Table 5.7: All possible values for "has" in a filter code block.

5.2.3 OntoX DSL in Action

So far an overview of the application components has been given. Also the ONTOX DSL was described in detail. Now it is time to look at some examples. All examples use the *Geography* ontology presented in [Zel95].

Example 1

Figure 5.6 shows a graph that has some expanded nodes. The ontology displayed, just uses the standard layouts and figures of the ONTOX framework, without any modifications via ONTOX DSL scripts. The standard settings uses rounded rectangles nodes for resources and normal rectangles for literals. All edges have the same style and same color. The graph gives enough information to answer a question like *"What are the cities of the state of New York?"* However additional help in form of other figures and colors would be helpful. Therefore a script (Listing 5.6) has been implemented that aids the user in solving such a task.

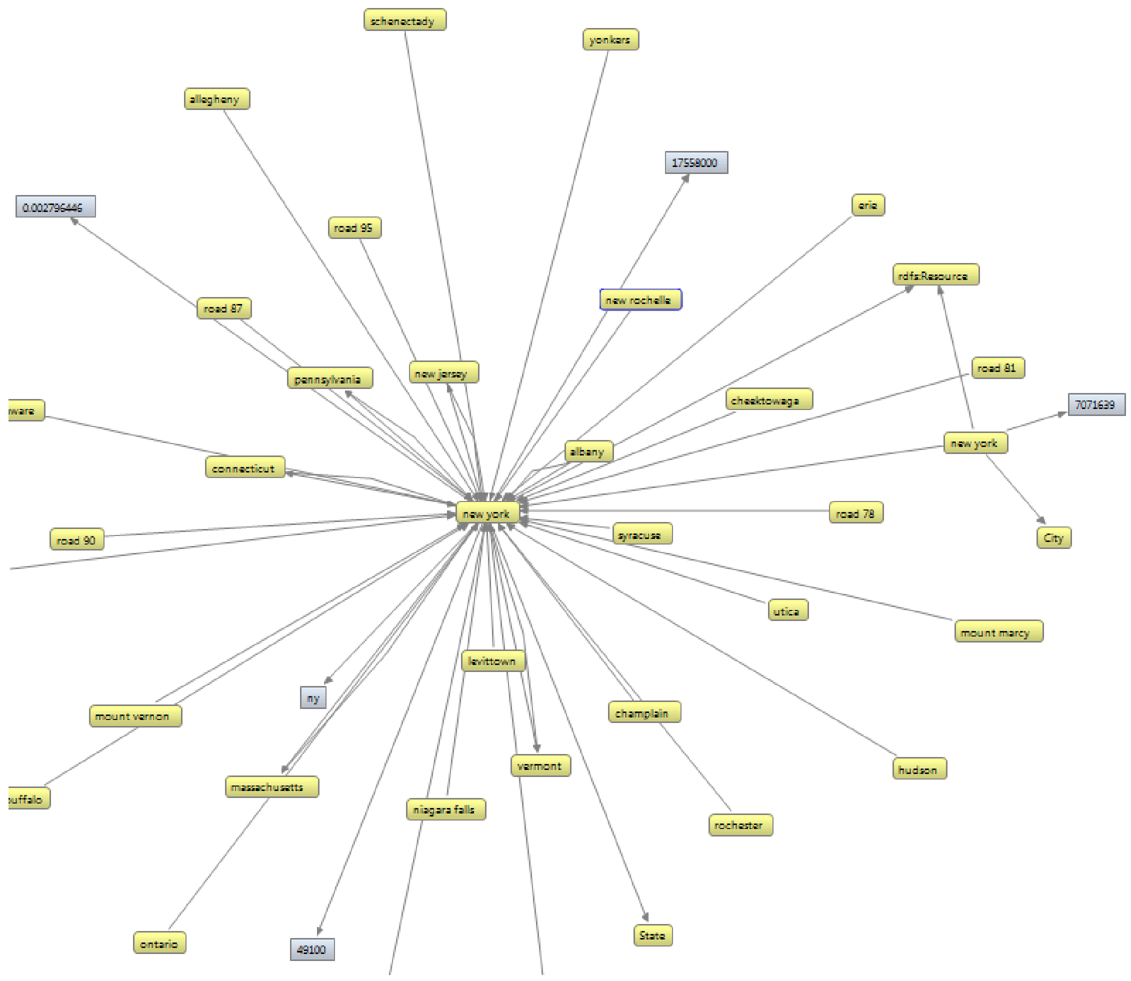


Figure 5.6: Standard visualization example of nodes and edges

Listing 5.6: Example script for changing the design of the graph

```
script {
   select{
      has name:'new york'
      has predicate:'isCityOf', as:'object'
      set color:[135,206,250]
      set fontColor:'white'
      set width:80
      set height:80
   }
   select('edge') {
      has name:'passesThrough'
      set lineStyle:'dot'
      set lineWidth:2
      set color:'red'
   }
   select{
      has predicate:'isCityOf', as:'subject'
      set figure:'circle'
      set color:'green'
   }
}
```

The implemented script makes it easy for the user to spot the correct nodes and edges in the graph (Figure 5.7):

- All cities of the state New York have now a green background color and a circle as figure.
- The state of New York is now colored in light blue with a new increased node size
- All predicates (edges) that define a road, that passes through the state of New York, are now colored in red with a dotted line style.

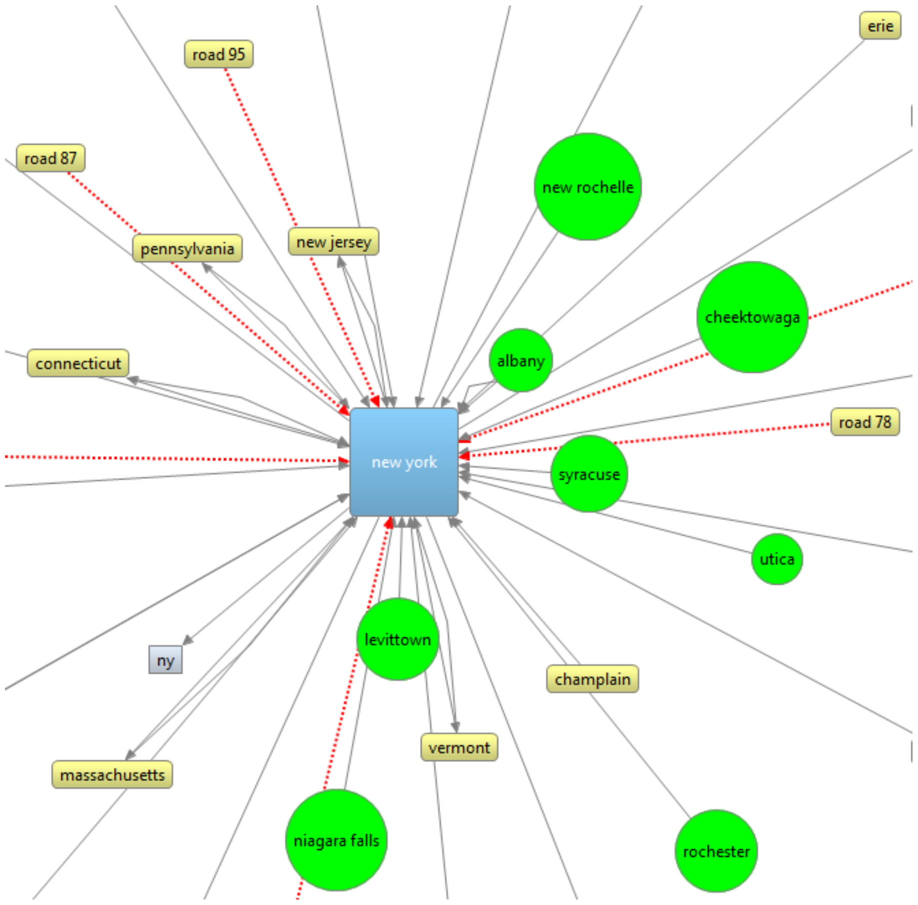


Figure 5.7: Graph with adapted design

Example 2

The previous example just changed the design of the graph by coloring all cities green and replacing their standard node figure with a circle figure. The size of the circle figures was automatically computed by the figure class, so that the label fits into the figure. But it would be better to combine the dimension of a node, representing a city, with a literal that has as value the population of the city. Instead of setting the height and width values of a node by using a constant, it is also possible to use a closure (Listing 5.7), that computes the size according to individual properties of each node. Therefore it is possible to visualize the answer to the question *"Which capital city has the largest population?"* (Figure 5.8).

Listing 5.7: Script that dynamically computes the node sizes

```
script {
   select {
      has predicate:'isCityOf', as:'subject'
      set color:'red'
      set figure:'circle'
      c1 = {
         temp = node.getLiteralValue('cityPopulation')
         temp = temp.toInteger() / 5000
         temp.intValue()
      }
      set width:c1
      set height:c1
   }
}
```

In Listing 5.7 is a closure used as value for the map entries *width* and *height*, instead of an *int*) values. The content of the closure *c1* is explained as follows:

- The object *node* represents a single instance of a node of type *Node.groovy*.
- The *Node* class has a method *getLiteralValue* that takes as parameter the name of a predicate. The method tries to find a predicate with the specified name and returns the literal value of the object in the RDF statement. If the literal has a datatype assigned, then the method returns the correct datatype, otherwise it returns a String value. The used ontology in the example has no datatypes defined; therefore the return value is of type String and needs to be converted to an integer. In Groovy one can simply use the method *toInteger* to turn the String value into an integer. The value is then divided trough 5000, to get a useful size representation. This is necessary because the values are too big and would be directly interpreted as Integer values for width and height. The method *intValue()* is used to turn any type of number into an int, which is expected as parameter by the *set width* and *set height* satements.

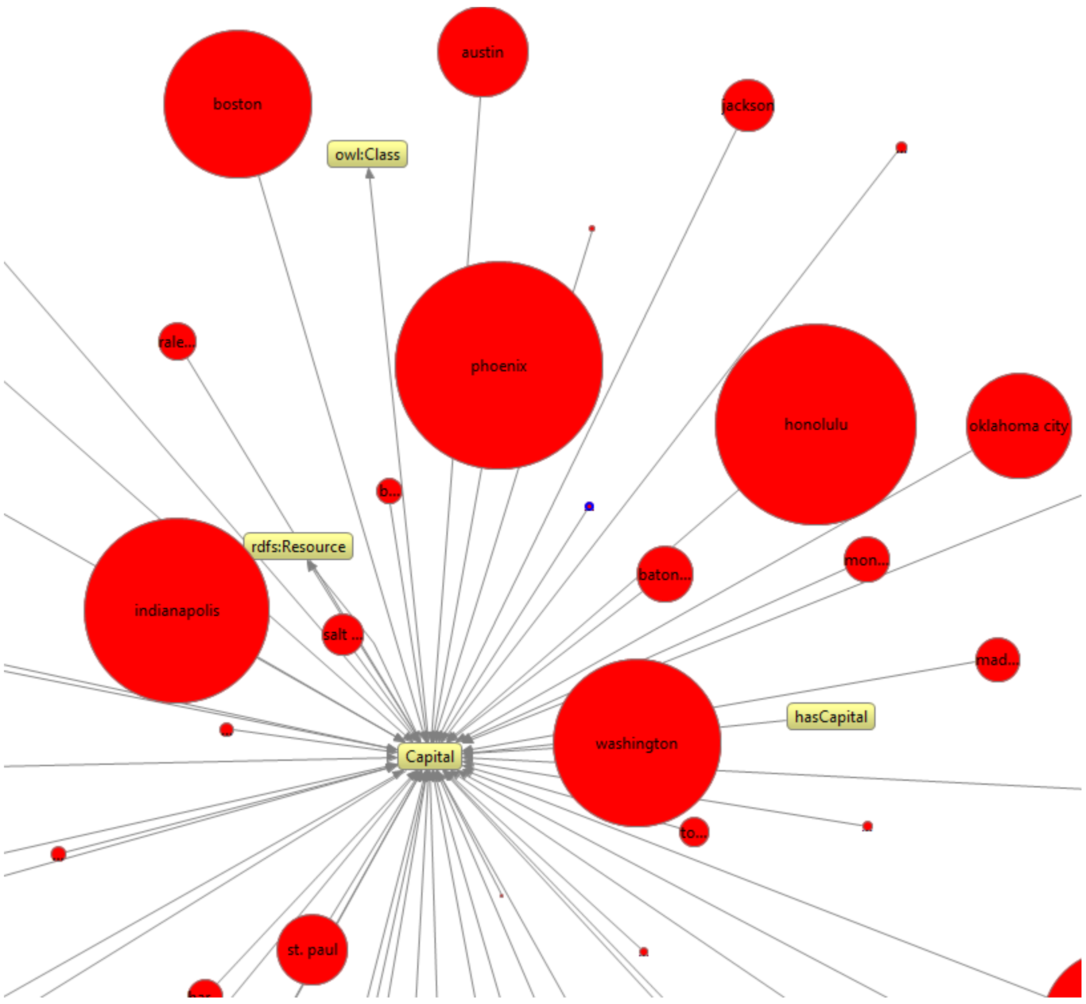


Figure 5.8: Graph with node (city) sizes computed from the population value

Example 3

The last example tries to answer the following question: *"How many roads pass through the state of New York?"* To solve this question, a new script will be created that collects these target nodes and displays them as children in a new subgraph node. The script in Listing 5.8 uses a *select* statement to collect all target nodes. Because the select statement is inside the *subgraph* code block, the nodes will be added to the *subgraph* node. Additionally, the result of the *select* block will be used to the set the name of the new subgraph node, therefore it is not necessary to count the nodes of the result set manually (Result is displayed in Figure 5.9 and the collapsed subgraph state is displayed in Figure 5.10).

Listing 5.8: Script collects the nodes that represent a road

```
script {
   subgraph {
      selection = select {
         has predicate:'passesThrough'
      }
      set name:'Roads (' + selection.size + ')'
      set algorithm:'radial'
   }
}
```

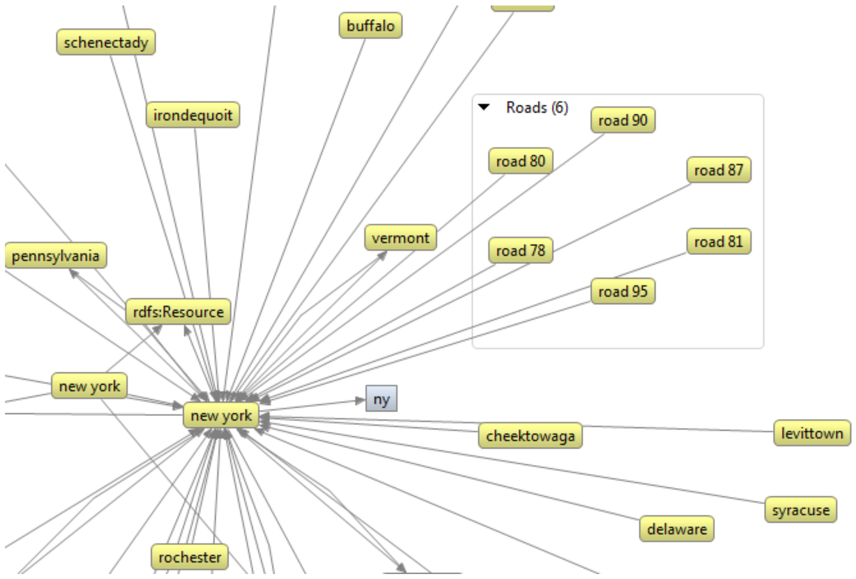


Figure 5.9: Subgraph containing the target nodes

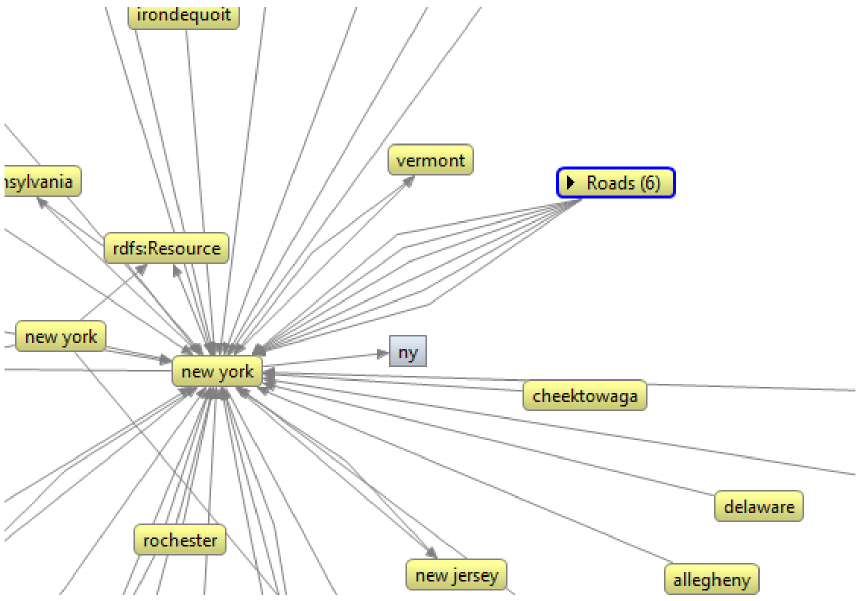


Figure 5.10: Collapsed subgraph node

Chapter 6

Implementation Details

In this chapter, implementation details of the ONTOX framework are described. First, an overview of the application components is presented, then they are described in more detail to show how they are interconnected and to highlight some important features.

6.1 Architecture Overview

The ONTOX framework consists of several main components which are shown in Figure 6.1. With abstraction and programming against interfaces, the system is designed to reduce coupling and to allow the easy exchange of existing components through new implementations.

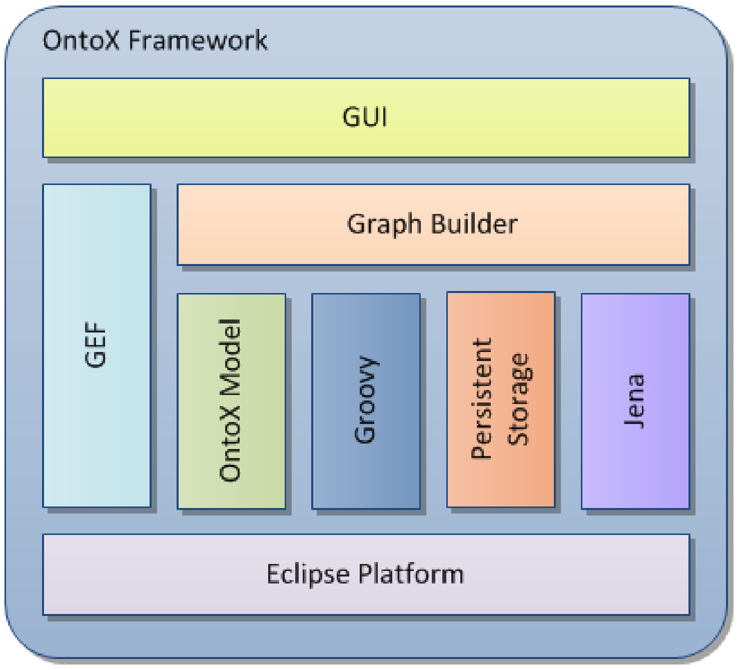


Figure 6.1: Overview of the application components

Main components of the framework:

- **Eclipse Platform:** Eclipse provides the basis for the ONTOX framework. ONTOX can therefore be deployed as Eclipse plug-in or as standalone RCP application.
- **Jena:** Jena framework is used to read and query RDF data sets.
- **GUI:** The top module of the framework that allows user interaction with the underlying components:
  - Eclipse Views are used to present the available ONTOX DSL scripts to the user and also to create a small overview over the entire graph.
  - The Eclipse Editor is the main component for interacting with the graph. The editor is the basic workplace for exploring an RDF data set.
- **OntoX Model:** It is the internal model that defines the structure of the graph.
- **GEF:** The Graphical Editing Framework takes care of drawing the graph, listening to user input, and modifying the ONTOX model if necessary.
- **Groovy:** The Groovy component consists of several parts:
  - Groovy Shell instance which takes care of parsing and running scripts
  - Groovy scripts and classes that provide the basis for the ONTOX DSL.
- **Graph Builder:** As the name already says the Graph Builder takes care of all processes that build and modify the graph.
  - The builder reads data from the Jena model and creates new nodes and edges in the ONTOX model.
  - The building events, the creation of nodes and edges, are triggered by user input or by Groovy scripts.
  - The builder initializes the script managers which handle the user entered scripts.
  - The builder loads and runs filter scripts to filter out unwanted nodes and edges.
- **Persistent Storage System:** The persistent storage system takes care of storing all user entered scripts, so they wont be lost when the user closes the application.

6.2 OntoX Model

GEF is built after the Model-View-Controller software architecture, so GEF needs an internal model to represent the data that the framework should visualize. The data to be displayed comes from a Jena model, but directly using the Jena model is not efficient, because such an model can get very large and the ONTOX framework also needs to be able to add new data fields to the model. Therefore a new internal model for the ONTOX framework was needed.

Node

The implemented ONTOX Model has a tree structured model that represents the parent-child relationship of the nodes (Figure 6.2). Every element in the tree is a child of the abstract base class *AbstractNode*. The *AbstractNode* class contains a list field that contains all child nodes, also of type *AbstractNode*. Another field contains the reference to the parent node, if the node is the highest node in the tree hierarchy then the field would be null. The *AbstractNode* also manages a set of node attributes that define the design (color, font color, name of figure, selection status, name and URI) of the node.

Important fields of *AbstractNode*;

- **layout:** The layout object defines the node's position and size on the screen.
- **backgroundColor:** This field defines the current color of the node.
- **fontColor:** The font color of the node.
- **isVisible:** Boolean attribute to hide or display the node.
- **figureName:** The name of the figure that should be used by GEF to draw the node.
- **isSelected:** Boolean attribute that defines if the node is currently selected node.
- **layoutAlgorithm:** If the node has some children, then it also should have a layout algorithm assigned.
- **name:** The name of the node that is used in the label of the drawn figure.
- **uri:** The unique identifier of the node.

The *AbstractNode* implements two additional lists that contain the connected edges to the node. One list for the source connections, where this node is the source of the directed graph connection, and one list for the target connections, where this node is the target of the directed graph connection. All edges are implementations of the *AbstractEdge* class.

The *AbstractNode* class implements an object of type *PropertyChangeSupport* to add a property change listeners to the node. Listeners are used to signal the controller that the model has changed, and needs to be redrawn on the screen. A repaint event can easily be triggered through the *OntoXNode* interface.

The *AbstractNode* class implements two interfaces, *OntoxNode* and *LayoutEntity*. Every module interacts with node elements trough the *OntoxNode* interface. Since the model is coupled to the GEF implementation part, the remaining components should access model elements trough interfaces to make them independent of the graphics implementation. The *LayoutEntity* interface is used by the layout algorithm. The layout algorithm uses the interface, first, to retrieve the size and some specified constraints for the node, and second, to set the new coordinate values as soon as the layout algorithm has finished its position calculations.

Every node type has to extend the *AbstractNode* class. There are three different types of node implementations available:

- *OntoxGraph:* Only one instance of the OntoxGraph exists at runtime, because it represents the root node of the tree. Every additional node is a direct or indirect child of this root node. Te root node always has a layout algorithm assigned to it and is not visualized in the graph, because it represents the container, where all its children are drawn.
- *OntoxNodeImpl:* The OntoxNodeImpl represents a common node in the graph and is implemented using the decorator pattern as a wrapper around an RDFNode object.
- *OntoxSubgraphImpl:* The subgraph node is an element in the tree that has children. It has two different states implemented, the collapsed and the expanded state, so the node model can be visualized as single node element (collapsed state) or as container object that has some child nodes in its figure boundaries (expanded state). For the two states, the model needs to manage two dimension values of the node.

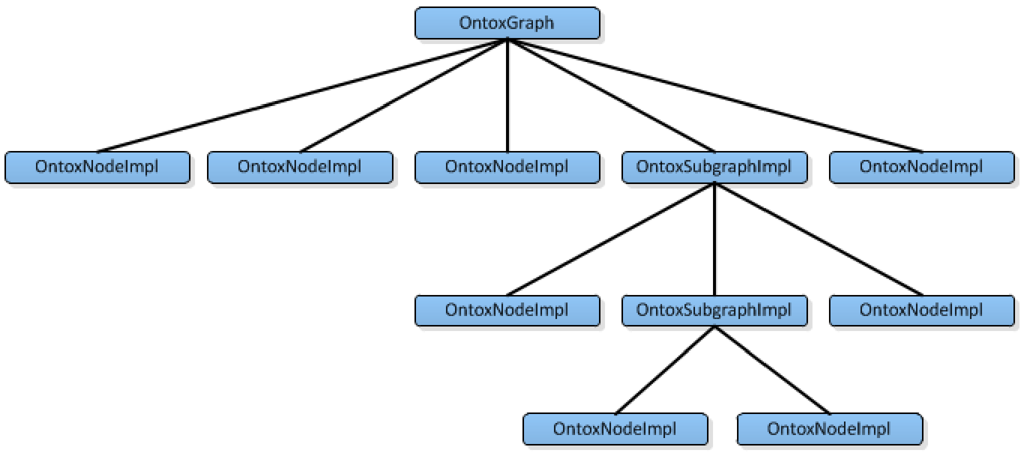


Figure 6.2: OntoX Model

Edge

All edges in the system are child classes of *AbstractEdge*. An edge object is always shared by two *AbstractNode* objects, one represent the source node and the other the target node in a directed graph edge. The *AbstractEdge* is the model implementation of an edge that is visualized in the graph. As the *AbstractNode* class, the *AbstractEdge* class also implements two interfaces: *OntoxEdge* and *LayoutEntity*. *OntoxEdge* is the interface, all other modules use to interact with an edge element. The *LayoutEntity* is the interface that is used by the layout algorithm to include edge information into layout computation.

Important fields of class *AbstractEdge*:

- *sourceNode:* A reference to the source node.
- *targetNode:* A reference to the target node.

- *connectionStyle:* Sets the style that is used when the edge is drawn on the screen (e.g. straight line, dotted line, etc.).
- *color:* Color of the line.
- *lineWidth:* The edge line width.
- *isLabelVisible:* A Boolean value that indicates if the edge label should be painted or not.
- *isVisible:* Boolean value that defines if this element should be rendered or not by Draw2D.

There is currently only one implementation of the AbstractEdge in the system: Class *OntoxEdgeImpl* represents the only available type of edges. It is designed as a wrapper object around a Jena RDF statement.

6.2.1 History Tracker

With the help of the ONTOX DSL the graph can be changed at runtime. The ONTOX scripts allow a user to change the shape, color, size and many other properties of nodes and edges in the graph. But sometimes it is necessary to undo the modification the executed scripts made. For example, if the user enters a wrong script, and instead of coloring some groups of nodes green, they are red. Or the selection formula of the script was not correct and now the wrong nodes are grouped together. To solve this problem, one could enter a new script that fixes these problems, but this would be too complex solution. The framework needs therefore a way to undo all the modifications that have been made with a script. This is the point where the History Tracker comes into place.

The History Tracker records all changes made to a node or edge, so that a change can be undone when needed. To do that, the History Tracker manages a list with all the previous node/edge setting values. For example: When a script changes the color of a node from yellow to green, then the History Tracker stores the old color value in its history list. The history list stores objects of class *HistoryObject*. Every *HistoryObject* has an event name, an object type and the object to be stored. The event name is the name of the event that caused the creation of this history instance. Normally the event name is the name of the script that did run, because the script names are already unique in the system, and therefore we can easily keep track which script changed what in the graph. The object type is an indicator what type of objects is stored in the history element, e.g. color object, etc. The object is the old value that was replaced, e.g. for a color change, this would be the old Java color object. When the user now wants to do an undo operation on a executed script, the History Tracker searches all *HistoryObjects* where their event name matches the script name, and runs the correct undo operation for that every *HistoryObject* type.

All methods, that create a history object, are part of the *OntoxNode* or *OntoxEode* interface and start with the name “update”. They all have as the first parameter the event name. To trigger an undo operation on a node or edge, one needs to use the *undo* method which takes the event name that should be undone as a parameter. Every node and edge has implemented the *undo* method. The *undo* method collects the history element that correspond to the specified event name and passes them on to an instance of the *UndoHandler*, which then performs the undo operations.

When using ONTOX DSL, every *set* operation triggers the creation of a history object. Also the *subgraph* statement has an undo operation, but in contrast to the *set*, it does not set some previous node or edge properties, but deletes the created subgraph element in the graph.

6.3 Graph Builder

The main purpose of the *Graph Builder* is the creation of the displayed graph. It is a mediator class between the Jena model and the ONTOX model and creates new nodes and edges in the ONTOX model. The abstract class called *AbstractGraphBuilder* it the base class, which implements the basic functions that are available to build the graph via the ONTOX DSL. It also acts as the interface for all other modules of the framework when they need to change the graph the structure of the graph. For the current implementation there is a class named *GEFGraphBuilder* which extends the AbstractGraphBuilder. The *GEFGraphBuilder* is graph builder implementation for the GEF framework. This way it is easier to change the graphical system (in this case GEF) with another suitable framework, when desired.

The Graph Builder provides methods to build the graph. For example to add a new Jena *RDFNode* to the graph, there is a method named *addNodeToGraph(RDFNode node)* which creates a graphical representation of the Jena RDFNode, by initializing a new node object (of type *OntoxNodeImpl*) and adding it to the graph.

Other features of the Graph Builder:

- The expand function is used to expand a certain node (to show all connected nodes). It queries the Jena model and adds the result set as new objects to the graph.
- When adding a new node or edge the builder first checks if the node or edge is already in the system. If the element already exists, the process will abort.
- The Graph Builder manages two hash sets: One for the all the nodes and one for all the edges. These sets are used for easy access to all nodes and edges in the system. They are used for the duplicate check of nodes and edges, because the system should not draw the same node or edge more than once.
- On instantiation the Graph Builder also creates an object of class *ModelInterrogator*. The *ModelInterrogator* provides some helper methods to query the Jena Model, e.g. to get a collection of Jena statements for a specific RDFNode object, and other useful methods.
- It also has a method to clear the graph, to delete all edges and nodes. This is necessary if the user chooses to load a different ontology.

The builder also implements the *PropertyChangeListener* interface and listens to the activation of Groovy DSL scripts. As soon as the builder gets an activation or deactivation of a filter script by the user, it initializes the filter process, according to the type of the filter script (node or edge filter). If the filter is deactivated, the builder restores the filtered elements in the graph. A similar functionality is implemented for the graph scripts. The builder receives activation or deactivation events and modifies the design of the graph, or starts an undo operation and restores a previous design state of the graph.

6.4 OntoX DSL

An example script of the ONTOX DSL is shown in Listing 6.1 and has the following meaning: "Locate all nodes in the graph that have a predicate named 'isCityOf' and set their color to green." One advantage of Groovy, that helps to create a more readable and less complex DSL, is all the optional syntax. Syntax elements like parentheses, semicolon, and others are not mandatory. If one had to use the full syntax in Groovy, the script would look as shown in Listing 6.2, which would make the syntax clearer, from a developers point of view, but more complex to use for an application user. All statements *script*, *select*, *has*, and *set* are now recognizable as methods with parameters.

Listing 6.1: OntoX DSL example script

```
script {
   select ('node') {
      has predicate:'isCityOf'
      set color:'green'
   }
}
```

To fully understand the structure of the DSL, a more detailed look is necessary. Let's start at the first line, the *script* statement. The *script* method (see Listing 6.3) is defined in the Groovy script file *CommonBaseScript* which defines basic code collection for all scripts that are entered by users. On runtime an instance of the class *GroovyShell*, which is in charge of parsing and compiling all user entered scripts, adds the *CommonBaseScript* as base class to all user scripts and therefore the script method is available to every entered script. This way there is a simple common start method that starts the entire script building process, but hides the initialization process of the Groovy builder object from the user.

Listing 6.2: Example script with full syntax

```
script ( {
   select ('node') {
      has( predicate:'isCityOf' );
      set( color:'green' );
   }
} )
```

The *script* method (Listing 6.3) takes the inner code of the script (code block in Listing 6.1) as closure parameter, named "content". Then it initializes a Groovy builder, which is specially designed for graph querying and manipulation actions, and sets some initial parameters through methods *getCurrentGraphBuilder* and *getCurrentScriptName*, which are Java objects that are passed on to Groovy trough a binding object at runtime. After initialization, the builder becomes the new owner of the closure code block, assigned through the *delegate* method. The last line, the *build* method, initializes the actual building process of the builder and starts evaluating the content of the closure parameter. The *build* method is a pretended method and has thus not a real method body. The builder object will take care of this method call and pass the closure ont to the *createNode* method, which signals the builder that its initialization is completed, and can now start to process the closure code block.

Listing 6.3: Implementation of the 'script' method

```
def script(Closure content) {
   def builder = new GraphBuilder();
   builder.gBuilder = getCurrentGraphBuilder();
   builder.scriptName = getCurrentScriptName();
   content.delegate = builder;
   builder.build(content);
}
```

The select statement, in the second line (Listing 6.2), is also only a pretended method. It is intended to hide the more complex initialization of a "select" object. The select object can be an object of class *EdgeSelect*, *NodeSelect* or *RootSelect*. The builder takes care of instantiating the correct object according to the String parameter value that has been passed to the *select* method. In the example the parameter value is "node", therefore the builder creates an object of type *NodeSelection*. The *NodeSelect* object is used to query the current nodes in the ONTOX Model and change their attributes. The *EdgeSelect* is for querying edges and changing their attributes. The *RootSelect* is used to target global system settings, changing attributes of the root model, like replacing the current active layout algorithm in the graph.

After the builder has created the correct select object, the builder evaluates the code block that is passed to the select method (lines 3 and 4 in Listing 6.1), parses the *has* and *set* methods, and adds their parameter values to the current active select object. All values of these methods are added as configuration attributes to the select object. Since a user can enter multiple *set* and *has* statements in its script block, it is necessary that the execution of the select process is started at the end, when all parameters have been parsed and added to the configuration attributes list of the select object. Therefore the method *run*, that all select objects implement, is called at the end of the code block, when line 5 is reached.

Listing 6.4: A more complex script example

```
script {
   subgraph {
      selection = select {
         has predicate:'passesThrough'
      }
      set name:'Roads (' + selection.size + ')'
      set algorithm:'radial'
   }
}
```

Another example is shown in Listing 6.4, that has a more complex nested code block structure. Here the *subgraph* method takes a closure as parameter that has two *set* statements and one *select* block. This method initiates a new subgrap object that adds the result of the *select* block as its child nodes (visible to the user through the nested code blocks). Also, the return value of the select statement, which is an object of type *NodeSelect*, is stored in the *selection* variable. Groovy does not need any type declarations; which is a useful feature that helps to facilitate the ONTOX DSL. The *NodeSelect* object implements a field named *size*, where the size of the query result is stored. This size information can then later on used in the *name* method of the *subgraph* code block to display the number of child nodes in the label of the subgraph node.

6.5 GEF Implementation

GEF is an important component of the ONTOX framework, it takes care of drawing the OntoX model and takes care of the user interaction with the graph. This section will explain some important aspects of the GEF implementation in ONTOX.

6.5.1 Controller (EditPart)

As mentioned before the controller elements in the GEF frameworks are called *EditParts*. For every model object GEF creates an EditPart object. This is done automatically through GEF via the *EditPartFactory*. This factory method creates for every type of model a corresponding controller part. GEF takes automatically care of the creation and deletion of EditParts as soon as GEF detects a change in the model. In the presented version of the ONTOX framework are four different types of *EditParts* for four different types of model implementations. On instantiation an *EditPart* initializes its own *EditPolicies*. The *EditParts* also takes care of creating the correct figure for its assigned model. Every *EditPart* implements the *PropertyChangeListener* interface which is used to trigger any kind of model updates to the visual representation of the model. When a update event is received, the method *refreshVisuals* performs the visual update by reading the current state values from the model and updating the figure with the new data. For every type of model in the ONTOX framework, there exits a different *EditPart* (see Table 6.1).

Model	EditPart	Description
OntoxGraph	OntoxGraphEditPart	This EditPart is for managing the primary container that is holding the entire graph. Therefore only one instance of this controller should exist.
OntoxNodeImpl	OntoxNodeEditPart	Controller part for nodes. Every node model is linked with its own instance of OntoxNodeEditPart.
OntoxEdgeImpl	OntoxEdgeEditPart	Controller part for edges. Every edge model is linked with its own instance of OntoxEdgeEditPart.
SubgraphNode	SubgraphEditPart	Controller part for subgraphs. Every subgraph model is linked with its own instance of SubgraphEditPart.

Table 6.1: Overview over all controller (EditPart) classes.

OntoxGraphEditPart

This *OntoxGraphEditPart* is for managing the basic canvas element, on which the entire graph is painted on. Therefore only one instance of this controller part exists. When an update of the graph model ist triggered, this controller initializes the layout computations for the new nodes and edges. As soon as the layout algorithm has completed, it refreshes all elements in the graph, so the user can see the positions of the new elements. The *OntoxGraphEditPart* controller creates a special kind of figure on instantiation. It creates a *FreeformLayer* figure with a free form layout. The *FreeformLayer* is a figure that can expand its current area in any direction and is therefore the basic figure for the dynamic expanding graph of the ONTOX framework. The controller automatically resizes the figure area and computes the new x and y values for its child elements (nodes and edges), and manages the size of the viewport that shows the current visible area to the user. The controller also takes care of repositioning the viewport when a layout computation occurs, since some layout algorithm may reposition all nodes and edges when a user chooses to expand/collapse a certain node in the graph. Therefore the viewport needs to refocus on the current selected node.

OntoxNodeEditPart

The *OntoxNodeEditPart* is the controller for a normal node. On its creation it takes care of creating the node figure for corresponding model object, e.g. if the model specifies a circle as figure for this node, then the controller creates an instance of the correct figure via the *NodeFigureFactory*. When an update occurs the controller refreshes the figure according to the model properties. The controller is triggered by four different types of events (Table 6.2). It also takes care of creating the necessary connection anchors that are always specific to the node figure. When the figure of the node changes, then the controller erases the old figure and creates the new one.

Event name	Description
Node layout changed	Updates the figure values according to the model state. It always repaints the entire figure and its children.
Refresh source connections	Event to repaint all source connections of the node.
Refresh target connections	Event to repaint all target connections of the node.
Selection changed	When the uses selects or deselects a node, then this event is triggered. It updates the selection status of the figure and repaints the figure's border.

Table 6.2: Events to trigger the controller

SubgraphEditPart

The *SubgraphEditPart* is similar to the *OntoxNodeEditPart*. It is a EditPart for managing the nodes figures that have children (the subgraph nodes). A subgraph node model in contrast to the normal node model, has a collapsed and expanded state. Therefore the controller needs to manage two figures: one for the expanded state and one for the collapsed state, but always only one figure is active. The SubgraphEditPart is therefore in charge of updating only the visuals of the current active figure.

OntoxEdgeEditPart

The *OntoxEdgeEditPart* is the controller implementation for the connection figures. The controller listens to changes in the connection model (edges) and updates the visual representation of the connection. For all connections, the controller always uses the same figure class, but changes only the properties of the figure to give the edges a new design.

6.5.2 Figures and their layouts

In Draw2D every visual representation of a model object is called a figure. A figure is a class that implements the interface IFigure or extends the abstract class *Figure*. Every node or edge in the ONTOX framework is also an extended child of the class *Figure*. GEF comes already with a set of basic figures, but for the ONTOX framework some more advanced figures were needed. The current ONTOX implementation comes with one figure for the edges and several different figures for the nodes.

A basic figure in the ONTOX framework, has normally one child figure of class Label for displaying the name of the node or edge, and one simple layout algorithm that defines the positions of all the child figures. Normally, the used layout is the *StackLayout*, which just places each child figure on top of each other; they are stacked inside the bounds of the parent figure. ONTOX uses nested figures for displaying graphical elements: One base figure that defines basic design of the node or edge, and one figure on top (label figure) for displaying the name.

Every figure has a method named *paintFigure(Graphics graphics)*. This method does the actual drawing of the object via the passed on graphics parameter. Every figure has a rectangle bounding box, therefore to make the figure look like, for example a circle, is job of the *paintFigure* method. To make the system more modular and prevent the implementation from having duplicate code, the code of the *paintFigure* was moved to a new painter class. There are several different painter implementations in the framework for drawing circular, rectangle and rounded rectangle figures. The default node and the subgraph node have different functionality on the figure implementation level, but share the same painter implementations.

Default Node

A node in the ONTOX framework is represented normally with a rounded rectangle. For that ONTOX uses the extended figure class named *DefaultNodeFigure*. But there are additional figures to create a ellipse, circle and rectangle figure. One Problem of GEF is, that the controller is thightly coupled to its figure and does not allow the exchange of a figure after it has been initialized. To circumvent this problem and make the system more flexible, a container figure was implemented. The container figure is the wrapper around the actual figure that represents the node, or in other words: the visual node figure is the child of the container figure. With this improvement the controller is always bound to the same figure, but it allows one to easily exchange the child figure on runtime.

Default Edge

Currently there is only one figure (*EdgeFigure*) that is used to represent an edge. The figure represents a straight line in the framework. It uses an additional figure of type *PolygonDecoration* to implement the arrow head to visualize a directed graph.

Subgraph

A subgraph node is a special kind of node. It is a visual container that can have several different default nodes as children, or another subgraph node. Another difference to the default node is that it has two different states, a collapsed and an expanded state. In collapsed form the node just looks like a normal node of the ONTOX framework, but in expanded state it is a rectangle figure that has a couple of default nodes as children. One subgraph node consists of three different figure implementations. The basis is the class *SubgraphFigure*, again this is the wrapper figure that is linked with its controller figure. The *SubgraphFigure* has two child figures the *ExpandedSubgraph-Figure* and the *CollapsedSubgraphFigure*. Only one at a time can be active of these two figures, the other is always set to invisible. The two figures represent the collapsed and expanded state of the node. The expanded figure is a fixed figure in the system that cannot be exchanged with another figure, e.g. turned into a circle or another form. On the other hand the collapsed figure is dynamic and can be replaced at runtime.

When collapsing a subgraph node, the collapse function has to analyze the edges of the subgraph and its children. All edges that are between two child nodes can just set to invisible, but all other edges that are connected to a node outside of the subgraph node, need to be reconnected to the collapsed figure, and no longer being connected to the child nodes. When the state switches back to expanded state, the edges need to be reassigned to the correct child node. Therefore the edge model always has to remember its orginal node connection.

Tooltips

Tooltips are a special type of figures in Draw2D. Every figure can have a Tooltip figure assigned to it which is standard implementation in Draw2D. In the ONTOX framework are two different tooltip figures: one for the nodes and one for the edges. These two tooltip figures are composed in the same way. Both are painted as a rectangle with white background. If a user hovers over a node or edge the tooltip figure is activated, painted on the screen for a couple of seconds, and then deactivated again. Tooltips are used in the ONTOX framework to give some additional information about the edges and nodes that would not fit onto the graph or would give the graph a crowded look.

6.5.3 Layout Algorithms

To draw all the nodes and edges in visual well arranged manner, some layout algorithms are needed. The Zest framework comes with an entire package of layout algorithms that are not bound to Zest and can therefore easily be integrated into the ONTOX framework. From the Zest framework, the following algorithms have been integrated into ONTOX:

- Spring Layout Algorithm
- Tree Layout Algorithm
- Radial Layout Algorithm
- Grid Layout Algorithm

The Zest layout framework also allows the easy creation of custom algorithms, and the combination of two algorithms. By using the two interfaces *LayoutEntity* for the nodes and the *LayoutRelationship* for the edges, it was not so difficult to add the layout algorithms to the ONTOX framework.

Every node in the ONTOX framwork implements the *LayoutEntity* interface and every edge implements the *LayoutRelationship* interface. Through this interfaces the layout algorithms get the current information about all the node and edges in the graph, e.g. position, height and width, etc. The Zest layout framework uses the method *applyLayout()* to start the computation and set the positions of the nodes through the *LayoutEntity* interface. The *applyLayout()* method needs some additional information (passed as parameters to the method) to begin the calculations: An array of all edges and nodes that take part in the layout computation, and additionally the *applyLayout()* needs the height and the width of the available layout area.

To deal with layout computations, the ONTOX framework uses the implemented *LayoutAlgorithmContainer* class that handles all necessary tasks related to the layout computation. The class manages two lists, one for the nodes and one for the edges that should be considered during the layout calculation. It also has some improvements implemented to stabilize the layout algorithms. For example if there is a node in the graph with no connected edges, the Spring algorithm breaks down and clusters all the other remaining nodes in a corner of the graph area that a user no longer can distinguish the individual nodes.

Another problem is that the ONTOX framework is intended to expand the graph dynamically through user interaction in any direction, but the layout area of the graph is not tied to the viewport area. The layout area can expand beyond the visible area the user can see. Therefore it is not possible to use the window size for the height and width parameter for the *applyLayout()* method, that is a necessary parameter of the layout implementation. Therefore the system needs a dynamically calculation graph area, based on the number of nodes and edges shown in the graph. The layout algorithms in such away that they automatically increase the layout space when it is necessary, but unfortunately the graph looked still very compressed. Therefore a simple algorithm needed to be implemented for ONTOX, that computes the necessary width and height of the desired layout area.

The algorithm computes the new area for the graph by counting the number of displayed nodes and edges, and multiplying it with an average node and edge size. This gives the minimum area that is needed to present the graph. The minimum area is then multiplied with a constant graph growth factor to add some extra space. From the new area, one can retrieve the width and height that the layout algorihtms needs. The width and height are always the same, because the layout algorithms compute a more balanced graph layout if it is a square area. This simple algorithm gives an good approximation how the maximal width and height for the layout algorithm should be. As a result the graph looks less compressed and the extra space prevents too much overlapping of nodes and edges.

6.6 Persistent Storage System

Early tests of the ONTOX framework made clear, that some sort of a persistent storage system was needed to store all the scripts. Although the implemented DSL makes the scripting rather easy, but is rather annoying when the program loses all its scripts when the application shuts down. Also some of the scripts are implemented only for the current active ontology, whereas other scripts can be used in multiple ontologies. It makes work much easier, if all the script are stored in a persistent storage unit.

The current version of the ONTOX prototype is only intended to store the user entered scripts, but future versions may also store additional information, like the current state of the graph. This means that it is more suitable, to use a more powerful storage system than just writing the data to a file. Another idea could be to use a remote storage location rather than a local one. So the best approach would be to use a database system that can easily be embedded into the ONTOX framework, and a helper framework that facilitated the storage and retrieval of Java domain objects via object-relational mapping (ORM). For this purpose *Apache Derby* [db.] was chosen as the embedded database and the Hibernate framework as the mediator between ONTOX and Derby.

Derby

Apache Derby [db.] is an open source relational database that is entirely implemented in Java and is based on SQL standards. One advantage is, that Derby comes with a small footprint; only 2.6 megabytes are needed for the base engine and the embedded Java Database Connectivity (JDBC) driver. With this embedded driver, Derby can be easily integrated into any existing Java application. There is also a plug-in available, that allows the easy use of Derby in the Eclipse environment.

Hibernate

Hibernate [hib] is an object-relational mapping library, a framework that provides mapping of an object-oriented domain model to a traditional relational database. Its primary feature is mapping from Java classes to database tables. One of its design goals it to relieve the developer from 95% of common data persistence-related programming task by eliminating the need to implement the own data processing using SQL and JDBC.

Hibernate uses a configuration file to define the driver class for the JDBC connection, to set the user name and password for the database, and other connection properties. Given that every connection information about the database system is in that file, the database system can easily be replaced without the need to change some source code in the Java application. An additional very helpful feature is the automatic generation of database schemes, so there is no need to set up all the tables in the database. The last thing the configuration file needs, is a list of all the classes that need to be mapped to the database.

The configuration file has only basic connection information, so how can Hibernate now map Java objects to the correct table, and fields to the correct column. Through Java Persistence API (JPA) annotations, the necessary metadata for the mapping process ,is added directly to the Java class itself, therefore no additional configuration file is needed (Example in Listing 6.5).

Listing 6.5: JPA entity example.

```
@Entity
@Table( name = "GROOVYSCRIPTS" )
@NamedQuery(
   name="GroovyScriptCode.getGroovyScriptByName",
   query = "FROM GroovyScriptCode code WHERE
         code.scriptName = :targetScriptName"
)
public class GroovyScriptCode {

   @Id
   @GeneratedValue(generator="increment")
   @GenericGenerator(name="increment", strategy = "increment")
```

```
        @Column(name="SCRIPT_ID")
        private Long scriptId;

        @Column(name="SCRIPT_NAME")
        private String scriptName;

        @Lob
        @Column(name="SCRIPT_TEXT")
        private String scriptText;
    }
```

The *@Entity* annotation is used to mark a class as an entity, a class that should be persisted in the database. The *@Table* annotation defines the Table in which an object of this class should be stored. To map the fields to the column in the table the annotation *@Column* is used and the *@Id* defines the identifier of the entity. Sometimes it is necessary to use additional queries to retrieve stored Java objects. Predefined queries are called *named queries* and are added through the annotation *@NamedQuery*. These queries can then be used throughout the framework by using their specified names.

The ONTOX framework has currently three entities that need to be stored in a database. The first is the *GroovyScriptCode* entity, which stores all the Groovy scripts that a user can enter via the user interface. Every entity of *GroovyScriptCode* has a unique identifier of type *Long*, a script name and the actual script text. The script text can be a large String, that's why the variable is marked as a large object (*@Lob*).

The next entity is the *FilterEntity*. This entity is used the store a ONTOX DSL based filter. The *FilterEntity* has a foreign key that references the filter object with a *GroovyScriptCode* object (Listing 6.6. The *@OneToOne* annotation defines a single-value association to another entity (in the example in Listing Listing 6.6 to an object of class *GroovyScriptCode*). Another annotation *@JoinColumn* defines the mapping for the composite foreign keys.

Listing 6.6: One to one reference example

```
    @OneToOne(cascade = CascadeType.ALL)
    @JoinColumn(name="SCRIPT_CODE_ID")
    private GroovyScriptCode scriptCode;
```

The last entity *GraphScriptEntity* is used to store all the scripts related to the manipulation of the displayed graph. Like the filter entity, it has a single-value association to a *GroovyScriptCode* instance.

The entire storage module can be accessed through the *OntoxPersistenceManager* interface. This allows the developer to implement any other storage system. Some of the methods may throw a *ScriptNameNotUniqueException* this is because all the script names need to be unique, so the framework can keep track of the active scripts and easily identify them by name.

Chapter 7

Evaluation

This chapter describes the evaluation that was performed to assess the implemented application. First, an overview is given about the realized study, and then the results are presented and discussed.

7.1 Introduction

In order to evaluate serveral aspects of the implemented ONTOX application, an user study was performed in the form of an online survey. The goal was to find answers to the following questions:

1. Is the application easy to install and to run?
2. Is the user interface understandable and easy to use?
3. Is the OntoX DSL understandable?
4. Can users easily create new scripts with the OntoX DSL?
5. Does the application provide useful help for the exploration of an ontology?

In order to answer these questions the participants had to solve some tasks and then rate then the application.

7.2 Setup

The survey consisted out of six parts. The first part was the *Information/Setup* part. This part was designed to give the participant some background information about RDF and what an ontology is, so that participants had the basic knowledge to understand the purpose of the ONTOX application. Also the necessary links for the software download and an install guide was included in the first part. The second part was the *Introduction* part, where some basic information about the participants were collected and to check their existing knowledge of this research area. In the third part, the *Tasks* section, participants had to use the application to solve small tasks that tested the functionality of the ONTOX application. The fourth part *General Questions* did consist of several questions concerning the general impression of the application. In the fifth part *Features*, participants had to share their opinion about the features and functionalities of the application. The last part, the *Remarks* part, offered the participants to give any other remarks about the prototype, e.g. likes and dislikes, and ideas for future improvements. The survey can be found in Appendix A.

7.3 Results

This section will summarize the results of the survey. The last part of the survey was composed of open questions that allowed the participants to enter some comments about the application. Whereas the rest of the questions had to be rated with a Likert scale [Lik32]:

1. Strongly disagree
2. Disagree
3. Neither agree nor disagree
4. Agree
5. Strongly agree

7.3.1 Tasks

The *Tasks* section confronted participants with simple assignments that allow the evaluation of the functional aspects of the application. Also the participants learned to interact with the system, so they were able answer the questions in the next sections of the survey.

Task 1

The purpose of the first task was to see, if the participants were able to load a data set into the application and to select a specified start node. As the result in Table 7.1 shows, no one had any noteworthy problems solving the first task.

Task: *Load the Ontology "geography.owl" (file) into the application via menu entry "OntoX" ->"Load File" and choose "newYorkNy" (the city) as the start node.*

Question	Mean	Median	STDV
Was the task easy to solve?	4.86	5	0.38

Table 7.1: Results of task 1

Task 2

The second task was designed to give some information about how easy it is for a user to interact with the system, and to explore the loaded ontology (from task 1). The result (Table 7.2) is similar to the result in task 1, participants didnt find it difficult to interact with the system and to retrieve the correct answer to the question in this task.

Task: *What is the population of New York (City)?*

Question	Mean	Median	STDV
Was the missing information easy to find?	4.86	5	0.38

Table 7.2: Results of task 2

Task 3

A simple task designed to test the *full expand* functionality with the participants, and to locate another specified node in the graph. As it is visible in Table 7.3, since the user already had experience in the process of exploring the graph from task 2, it is not difficult for the participant to solve this task.

Task: *Fully expand node "new york" (City) and node "new york"(State).*

Question	Mean	Median	STDV
Was this task easy to accomplish?	5	5	0

Table 7.3: Results of task 3

Task 4

In Task 4, participants had to create a filter that removes some unwanted nodes in the graph. The purpose of this task was to see, how difficult it is for the participants to create a filter script with the wizard, and subsequently to run the script. The script code, that the participants needed to enter, was already included in the task description. The results are presented in Table 7.4, but will be discussed in the next task, since they are related.

Task: *Create a new filter named "FilterCities" for filtering all nodes from the graph that represent a city.*

Question	Mean	Median	STDV
Was the filter easy to create?	4.72	5	0.49

Table 7.4: Results of task 4

Task 5

In the previous task, users had to create a filter script; this was used to test the second type of scripts, the scripts for modifying the graph design. Again the participants were given a script that should help solving the question, posed in this task. At the end the participant had to comment the usefulness of the script. The result is visible in Table 7.5.

The ONTOX application has two views for entering scripts, one *Filter View* for entering filter scripts and a *Script View* for entering the rest of the scripts. Although all participants were able to enter and to run the scripts from this task and task 4, for some participants it was not always so obvious in which view to enter the script. One problem could be that the name *Script View* is misleading, since a filter is also a script. Or two different views and two different wizards for entering scripts are confusing. It would be better, if the user would have only one wizard for entering their scripts to prevent confusion.

Task: *Try to answer the question: "How many cities has the state of New York in the current ontology?" Create a new script named "ColorAllCitiesGreen" for coloring all cities in the current graph green.*

Question	Mean	Median	STDV
Was this script helpful in answering this question?	4.86	5	0.38

Table 7.5: Results of task 5

Task 6

So far the scripts were given in the description. Therefore task 6 tries to evaluate how easy it is for users to implement their own script by looking at the examples in the previous.

The results in Table 7.6 indicate that it was a bit more difficult to create a script, but everyone was able to implement it with the given information. This shows that the ONTOX DSL is understandable and not to difficult for the user to handle. Afterwards no one had any problems finding the answer to the question posed in the task.

Task: *Try to locate the Hudson River and then determine its length? Create a new script named "HudsonRiver" that paints the node (representing the Hudson River) red, or changes the figure of the node in such a way that it can be spotted more easily.*

Statement	Mean	Median	STDV
The script was easy to create	4.43	5	0.79
The information was easy to find	5	5	5

Table 7.6: Results of task 6

Task 7

For the last task, the participant was given a script that provided the answer to this task's question. So again, the participants hat to use the code and then rate the usefulness of the script and how understandable it is, since the script is a little bit more complex.

The conclusion of this task is, that all participants thought that the script was extremely helpful, providing the answer to the question. Therefore the ONTOX DSL really brings some advantages that enrich the functionality of the application. The second statement in Table 7.7 shows that the complex script is good understandable by most participants.

Task: *Try to answer the question: How many roads pass through the state of New York?*

Statement	Mean	Median	STDV
The script helped a lot to solve this task	5	5	0
The above script is easy to unterstand	4.86	5	0.38

Table 7.7: Results of task 7

7.3.2 General Questions

The *General Questions* section consists of questions about the interaction with the ONTOX application. All questions and the results are listed in Table 7.8.

#	Statement	Mean	Median	STDV
1	The application provides useful assistance for the task of exploring an ontology.	5	5	0
2	The application is easy to operate with.	5	5	0
3	The domain specific language (DSL) to enter the scripts, is easy to understand.	4.29	4	0.49
4	The domain specific language (DSL) to enter the scripts, has a flat learning curve.	4.14	4	0.69
5	The graphical editor shows all nodes and edges in a clear arranged layout.	4.57	5	0.53
6	The Satellite View provides help to keep a general view of the entire graph.	4.57	5	0.53
7	The Filter View gives a detailed overview over all filter scripts.	5	5	0
8	The Script View gives a detailed overview over all scripts available to manipulate the graph.	5	5	0
9	The wizard windows for entering new scripts, clearly describes what a user has to enter into the input fields.	3.14	3	0.69

Table 7.8: General Questions

Statements 3 and 4 indicate that the ONTOX DSL could use some minor improvements in making it more understandable and easier to use. This is probably also related to statement 9: Most participants find that the wizard does not provide enough help for entering the scripts.

7.3.3 Features

The fifth part *Features* consists of questions about the different features that the application offers. The questions and the results can be found in Table 7.9.

#	Statement	Mean	Median	STDV
1	Using the Groovy DSL script to filter and change the design of the graph, is a useful way to interact with the application.	4.57	5	0.54
2	The coloring of nodes/edges is a useful feature to give more meaning to the elements in the graph.	5	5	0
3	The edge tooltip provides helpful information about the selected edge.	5	5	0
4	The functionality to expand and collapse a single node is a useful feature that helps to manage the amount of nodes that are displayed in the graph.	5	5	0

Table 7.9: Features

Almost all features were rated with a strong agreement, so the participants liked these implementations. Only statement 1 falls a little bit behind the average rating. One can assume that this has something to do with the fact that user are normally confronted with a complex user interface with buttons for a every action the user can make. The ONTOX application now comes with another approach that is using a DSL to interact with the application. The DSL allows the creation of ontology specific, complex actions, but in order to do that the user first has to learn the language and understand how it works.

7.3.4 Remarks

The last part of the survey was a set of open questions, where the participants could comment the application. This section gives a summary about the opinions/problems of the participants.

What did you like the most?

- Use of filters for hiding elements
- The "Expand on" feature
- The possibility to expand and collapse nodes
- Use of colors and different shapes to change the design
- The feature to enter custom scripts
- Creating a visual collection of nodes

What didn't you like?

- Installation process is extremely slow
- Missing help for the script language
- The use of the "Expand" action repositions all elements in the graph
- The distinction between scripts and filters is confusing at the beginning
- Entering a script (filter or script) in the wrong view generates an error

Please share your suggestions how to improve the prototype

- Help window that explains the script language
- Using the built-in help system that is bound to the question mark button available in the wizard pages for giving a quick overview of the language in the form of a cheat sheet
- New "Collapse on" function, similar to the "Expand on" function
- One wizard for entering the scripts, instead of two different versions

7.4 Discussion

The idea behind the evaluation was to find answers to the questions listed in the *Introduction* section of this chapter. The first question was related to the installation and running of the application. No participant mentioned any errors during the installation of the plug-in in Eclipse or any failures that prevented the application from running. But a problem was that the plug-in took a long time to install. The long installation is caused by the dependency check that Eclipse performs at the beginning of the installation. The ONTOX plug-in needs SWT, GEF, Zest, Jena, Groovy and other libraries to run. All these libraries exists as separate plug-ins and are not integrated into to the ONTOX plug-in, therefore Eclipse needs to perform a dependency check on all libraries. Some of these libraries could also be directly included into the framework to speed up the installation process.

The next question, that the evaluation needed to answer, was associated with the evaluation of the interface considering usage and understandability. Some participants were a little bit confused at the beginning, between the distinction of the two script types (filter and graph) and in which wizard they had to enter them. But it turned out not to be an obstacle that prevented the participants from solving all tasks. However, it would better to improve the script wizard in a future version, to make it clearer for the user.

The next two questions were related to the ONTOX DSL and finding answers to how easy can users understand and create scripts. The results of the tasks showed that users do not find it to hard to understand the DSL. Also they can easily create new simple scripts out of some given examples. But for more complex scripting, a help for the user is needed.

Although everything was given in the Survey description, including some script examples, the participants suggested as main improvement to include help for the DSL. After the evaluation, the prototype was extended with a simple help documentation that gives an overview over the entire language. Also some further ideas for improving the use of the ONTOX DSL are mentioned in the *Future Work* (section 8.2) of this thesis.

An additional improvement, that was suggested by the percipients, such as the *Collapse on* function, which allows user to remove a single edge without the need to write a filter script, was also included into the framework. Also the wizards inform the user now, if he enters the wrong script in one of the wizards.

Statement	**Mean**	**Median**	**STDV**
I have advanced knowledge about Eclipse	3.71	4	0.76
I have advanced knowledge about the Semantic Web and the Resource Description Framework	3.14	3	1.57

Table 7.10: Results of the participants experience

All participants were between the age of 25 and 30, male, with a background in computer science. The survey was performed with seven participants. As Table 7.10 shows all participants were familiar with Eclipse but the range of knowledge in Semantic Web/RDF goes from no knowledge to professionals. Nevertheless all were able to solve the tasks. Also the evaluation shows that the prototype is not too difficult for users to understand and to control, and the prototype brings some helpful features for the exploration of ontologies. But there are still some points that need to improve in future versions.

Chapter 8

Final Remarks

The last chapter recapitulates the work and draws some conclusions about the implemented ONTOX framework. In addition, some suggestions are presented how the ONTOX framework could be improved.

8.1 Conclusion

In this thesis a new visualization framework for the Semantic Web was presented. A framework that provides help in the exploration of Semantic Web data sets with a simple user interface. In addition to the visualization part, a new domain specific language was developed that allows users to interact with the visualized data in a new and advanced way.

First, the basic technologies were presented, that were used as the basis for the framework implementation, including some background knowledge about DSL development on the basis of Groovy. Next, the application prototype was presented with all its features and the user interface. Also the developed ONTOX DSL was described in full detail with some user examples that showed the capabilities of the implemented language and how the language can assist the user. In a following chapter further details about the implementation of the framework were explained, which are necessary to understand how certain components work that are not visible to the user, e.g. the History Tracker to undo script operations.

Finally, an evaluation was performed to see how the application prototype is accepted by users. With simple tasks the user learned to interact with the application and to write scripts that would assist users to look for specific information in the data set. In respect to the application it can be said that the participants were generally positive towards the ONTOX prototype. There are still improvements open, especially some assistance for the user that he can enter the ONTOX DSL more easily. But the application shows potential for further development and some of these new features are presented in the next chapter.

8.2 Future Work

The first version of the ONTOX framework worked quite well and was able to reach the goal of this thesis. But there is still room for improvements and new features, which are presented in this section.

8.2.1 SPARQL Query Language for RDF

SPARQL [PS08] is an RDF query language that is used for querying RDF graphs via pattern matching. It is a syntactically SQL-like language. A *SELECT* query is used to extract value from a SPARQL endpoint, and the result is returned in a table format. Listing 8.1 shows an example of SPARQL query that is used to find the title of a book from a given data graph. The query consists of two parts: The first part, the *(SELECT* clause, identifies the variables to appear in the query results, and the second part, the *WHERE* clause, provides the basic search query. In this example the query consists of a single triple pattern with a single variable *(?title)* in the object position.

Listing 8.1: SPARQL query example

```
SELECT ?title
WHERE {
  <http://example.org/book/b1> <http://purl.org/elements/title> ?title.
}
```

SPARQL is a powerful language to query an RDF graph. Therefore it should be included into the OntoX framework: SPARQL could be used to specify the start node, instead of selecting the start node from a list. The user could enter a query and use the result as the initial start node set. Also dynamic highlighting or the selection of nodes via SPARQL query could integrated into the framework. Another possibility would be to use the SPARQL statements directly in the OntoX DSL language, to query the Jena Model for certain elements (used in OntoX *select* or *subgraph* statements).

8.2.2 Layout Algorithms

This section will give some ideas about how to improve the layout algorithms used in the framework.

Improved Layout Algorithms

The implemented layout algorithms are limited. When the number of nodes or edges changes in the graph, then the algorithm has to recalculate the entire graph. The graph gets a complete new structure, because every node gets a new position depending on their amount of connected nodes through edges. Therefore a new algorithm should be implemented that can layout new nodes and edges without changing the structure of the already existing nodes. Another idea would be that the user has to choose the target position where the node should be expanded to, and the algorithm tries to layout the nodes in the area that the user selected.

Another approach would be to include the layout algorithms from the Jung framework. Jung comes with its own set of layout algorithms. One important aspect that all the algorithms in Jung offer, is that the positions of the vertices can be locked. This means the algorithm takes existing nodes into account when computing a new layout but does not change their position. Through this approach the graph would become more stable, when expanding and collapsing nodes, then it is in the current implementation. But it could also lead to new problems, since some graph structures would now be rigid and do not reposition when their number of connected nodes change.

Extension Point

At the moment the layout algorithms are all included as direct implementations in the ONTOX framework. Since these layout algorithms are a vital point when it comes to displaying the graph elements in a well-arranged manner, it would be best if they could be easily exchanged in the source code. Therefore the layout algorithms could be implemented in separate plug-in and integrated via extension point into the ONTOX plug-in, which would make the system more modular with respect to layout algorithms.

8.2.3 Persistent Graph State

The ONTOX framework stores only the user entered script in the persistent storage system. An improvement would be, if the user also could store its current exploration state of the graph with all the modifications he made. So the user could continue his work another time or send the graph to someone else. Here is a short list of the important parts that should be stored:

- **Nodes and edges:** All elements in the graph need to be stored including their status, design, and position information.
- **History data:** : The history data of all objects need to be stored to allow undo operations.
- **Active status of scripts:** Which ones where active when the user stored the state of the graph.

8.2.4 Enhancing OntoX DSL

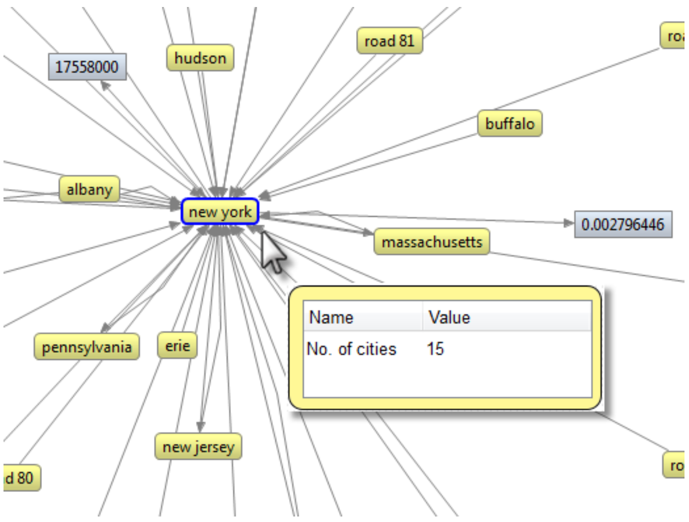
This section provides some ideas how to enhance the current ONTOX DSL by adding some graphical and analytical features.

Analytical DSL

So far the user has the ability to enter scripts for filtering elements in the graph and changing the design and structure of the graph. To add some analytical functionality to the ONTOX framework, one could add a new script type that performs a statistical/analytical computation and then displays the result value in the user interface.

In the Geography ontology [Gru93], such a computation could directly help to solve a question like *"How many cities has the state of New York?"* To do this, the user would enter a new script, and in that script he first defines the target node(s) for which this calcualtion is (node, edge, or a global value). Then he enters the script for the actual calculation. When the user now hovers with the mouse over the element (see Figure 8.1), for which the script was designed, then the computed value appears in the toolip as a name-value entry. Instead of using a tooltip, the value could also appear in a new view or the Eclipse *property* view. These calculations could also be used in other scripts, e.g.connecting the result of the calculation with the size of the node.

Another example would be when using a code ontology, like the source code ontology SEON [WGRG10], the user could compute some metric values: Aggregate the lines of codes, count the number of methods a class has, etc. Or to create a textual representation of the answer to the question *"What are the callers of method add?"*.



Name	Value
No. of cities	15

Figure 8.1: Tooltip example for an analytical script

Graphics DSL

The ONTOX framework used hard-coded figures to display nodes as circles, rectangles, etc. An additional improvement of the ONTOX DSL could be the creation of an ONTOX Graphics DLS, that extends the current DSL by allowing users to define new graphic elements. Groovy has already a package named *GraphicsBuilder* [Alm] that can be used to create Java2D graphic objects (see Listing 8.2 and Figure 8.2 [Alm] for an example). Maybe it is possible to integrate this builder into the framework. If not the builder could be used as template for creating a new builder that is based on SWT and Draw2D for drawing new node shapes.

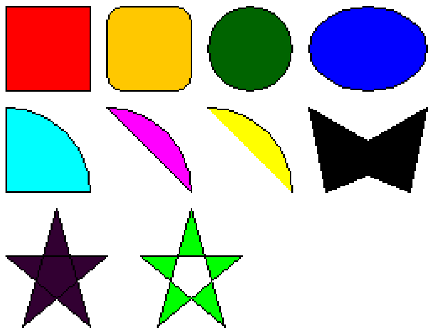


Figure 8.2: GraphicsBuilder example shapes

Listing 8.2: Example scripts from GraphicsBuilder

```
rect( x: 10, y: 10, width: 50, height: 50, borderColor: 'black', fill:'red' )
rect( x: 70, y: 10, width: 50, height: 50, arcWidth: 20, arcHeight: 20,
   borderColor: 'black', fill:'orange' )
circle( cx: 155, cy: 35, radius: 25, borderColor: 'black', fill: 'darkGreen' )
ellipse( cx: 225, cy: 35, radiusx: 35, radiusy: 25, borderColor: 'black',
   fill: 'blue' )
```

```
arc( x: -40, y: 70, width: 100, height: 100, start: 0, extent: 90,
   borderColor: 'black', fill: 'cyan', close: 'pie' )
arc( x: 20, y: 70, width: 100, height: 100, start: 0, extent: 90,
   borderColor: 'black', fill: 'magenta', close: 'chord' )
arc( x: 80, y: 70, width: 100, height: 100, start: 0, extent: 90,
   borderColor: 'black', fill: 'yellow', close: 'open' )
polygon( points: [190,70,225,90,260,70,250,120,225,110,200,120],
   borderColor: 'black', fill: 'black' )
path( borderColor: 'black', fill: 'purple', winding: 'nonzero' ){
   moveTo( x: 40, y: 130 )
   lineTo( x: 20, y: 200 )
   lineTo( x: 70, y: 158 )
   lineTo( x: 10, y: 158 )
   lineTo( x: 60, y: 200 )
   close()
}
path( borderColor: 'black', fill: 'lime', winding: 'evenodd' ){
   moveTo( x: 120, y: 130 )
   lineTo( x: 100, y: 200 )
   lineTo( x: 150, y: 158 )
   lineTo( x: 90, y: 158 )
   lineTo( x: 140, y: 200 )
   close()
}
```

8.2.5 Enhanced Script Editor

When a user enters an incorrect script into the *Script Wizard Window*, he receives the error messages from the Groovy parser. This is enough to detect Groovy syntax errors in the script code, but does not help if the user enters a wrong combination of the ONTOX DSL. Also the error message from the Groovy parser can sometimes be cryptic to understand, when for example the parsers reports the error on the wrong position in the code. Therefore an advanced text parser would be helpful that understands the ONTOX DSL and can give helpful error information to the user. Also a code completion function would be useful to make the input of the DSL easier, like Eclipse posses when entering Java code.

8.2.6 Design

Another way of improving the framework can be done through enhancing the design of the framework.

Additional Figures

ONTOX allows the user to create a rectangle, a rounded rectangle, a circle, a square, or an ellipse figure for the nodes. This is a small set of figures that is included into the system. Therefore a future improvement would be, to add more figures to the system and increase the set of figures the user can choose from. The new figure don't need to be simple shapes, they can also be more advanced implementations to represent, for example, a set of nodes as a tree or a treemap. Another possibility would be, that the user can use images instead of computed figures.

Animation

The ONTOX framework can be used to incrementally explore any ontology. When new nodes are added to the graph, the layout algorithm recalculates the positions for all nodes and therefore changes the entire layout of the graph. This means that the user is suddenly confronted with a complete new structured graph, what makes it is difficult for the user to keep track of the current work context. At the moment, the ONTOX framework tries to refocus on the selected element, so that the user doesn't get completely lost. An additional help to the user would be, if the user could follow the transition from one graph state to the other. An animation would visualize the repositioning of the elements to the user, so he could keep track of the change. For example: When the user decides to expand a selected node, the new nodes would emerge from the selected node am move to their new positions in the graph, visualized with an animated sequence. Another example would be the change of the layout algorithm, when the nodes slowly move away from their old position to the new one, that is computed by the layout algorithm.

Friedrich et al. [FE02] presents some criteria for a good animation:

- The movement of nodes and edges should be easy to follow.
- The movements of the graph should be structured (uniform and symmetrical movement).
- The transition from source to destination should be smooth.
- The change should occur in an adequate speed.

Appendix A

Survey

OntoX Survey

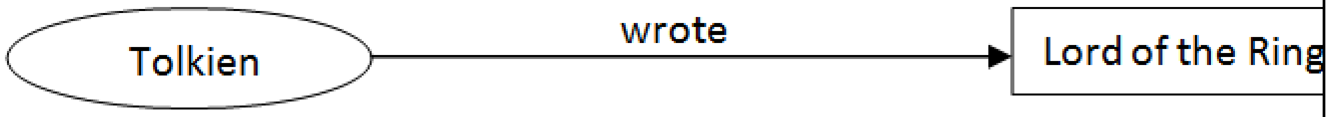
1. Information / Setup

Thank you for participating in this survey. With your help, I will be able to improve the current prototype of my visualization framework for the Semantic Web.

Resource Description Framework (RDF)
This is a short introduction to the Resource Description Framework (RDF). If you are already familiar with RDF, you can skip this section.

RDF is a language for representing information about resources in the World Wide Web. It is based on the idea of identifying resources using Uniform Resource Identifiers (URI). Resources are described in triples (also called statements). A RDF triple consists of “Subject –Predicate – Object”:
Subject: The Subject is the thing (the resource) we want to make a statement about.
Predicate: The predicate defines the kind of information we want to express about the subject.
Object: The object defines the value of the predicate. The object can be a literal or another resource.

Example: The sentence "Tolkien wrote Lord of the Rings." can be transformed into an RDF statement: Subject "Tolkien", predicate "wrote", and object "Lord of the Rings".



Ontology
Ontologies are the structural frameworks for organizing information and are used in the Semantic Web and other areas as a form of knowledge representation about the world or some part of it.

Common components of ontologies include:
- Individuals: instances or objects (the basic objects)
- Classes: sets, collections, concepts, classes in programming, types of objects, or kinds of things
- Attributes: aspects, properties, features, characteristics, or parameters that objects (and classes) can have
- Relations: ways in which classes and individuals can be related to one another

OWL (Web Ontology Language) is a language for making ontological statements, developed as a follow-on from RDF. It is intended to be used over the World Wide Web, and all its elements (classes, properties and individuals) are defined as RDF resources, and identified by URIs.

Installation
Before we can begin with the survey, you need to set up the prototype. Please download the needed software and the necessary files, and use the Installation Guide to install the software.

Download links:
- Eclipse Helios 3.6.2
- OntoX Eclipse Plugin
- Geography Ontology (geography.owl)
- Installation Guide

OntoX Survey

2. Introduction

At the beginning some statistical questions.

1. Age?

2. Gender?

○ male	○ female

3. I have advanced knowledge about Eclipse (software development environment)?

○ --	○ -	○ -/+	○ +	○ ++

4. I have advanced knowledge about the Semantic Web and the Resource Description Framework?

○ --	○ -	○ -/+	○ +	○ ++

3. Tasks

On this page are some simple tasks that will show the features of the current prototype. Some tasks are dependent on each other, therefore they should be solved in the specified order.

1. Load the Ontology "geography.owl" (file) into the application via menu entry "OntoX" -> "Load File" and choose "newYorkNy" (the city) as the start node.

Was the task easy to solve?

○ --	○ -	○ -/+	○ +	○ ++

Problems/Comment

2. Try to answer the following question: "What is the population of New York (City)?" (Hint: The context menu (right mouse button on selected nodes) provides the necessary tools to expand/collapse nodes in the graph. The answer can be entered in the comment field.)

Was the missing information easy to find?

○ --	○ -	○ -/+	○ +	○ ++

Problems/Comment

3. Fully expand node “new york” (City) and node “new york”(State).

Was this task easy to accomplish?

○ --	○ -	○ -/+	○ +	○ ++

Problems/Comment

4. Create a new filter named "FilterCities" for filtering all nodes from the graph that represent a city. You can use the following script code (based on a implemented Groovy domain specific language -> DSL) to solve this task:

```
filter {
    has predicate:'isCityOf'
}
```

(Hint: Enter the script in the "Filter View". Scripts can be activated and deactivated by clicking on their checkbox in the list.
The code block has the following meaning: Filter all nodes that have a predicate named "isCityOf".)

Was the filter easy to create?

○ -- ○ - ○ -/+ ○ + ○ ++

Problems/Comment

5. Try to answer the question: "How many cities has the state of New York in the current ontology?"
Create a new script named "ColorAllCitiesGreen" for coloring all cities in the current graph green. Use the given script code to solve this task (don't forget do deactivate the filter created in step 4):

```
script {
    select ('node') {
        has predicate:'isCityOf', as:'subject'
        set color:'green'
    }
}
```

(Information: Enter all scripts, that are not filter scripts, in the "Script View".
The above script block has the following meaning: Select all nodes that have an predicate named 'isCityOf' with the target node as the subject of the RDF statement.
Then change the color of all found nodes to green.
The keyword "script" defines the basic script block. The statement "select('node')" defines a selection code block with a "has" and a "set" property.)

Was this script helpful in answering this question?

○ --	○ -	○ -/+	○ +	○ ++

Problems/Comment

6. Try to locate the Hudson River and then determine its length?
Create a new script named "HudsonRiver" that paints the node (representing the Hudson River) red, or changes the figure of the node in such a way that it can be spotted more easily (to solve this task look at the script used in task 5).

- **- To look for a certain name use: has name:'hudson'**
- **- To set a new figure use: set figure:'circle'**
- **- To set a certain color use: set color:'red'**
- **- or use: set color:[250,172,191]**

	--	-	-/+	+	++
The script was easy to create	○	○	○	○	○
The information was easy to find	○	○	○	○	○

Problems/Comment

7. Try to answer the question: How many roads pass through the state of New York?
Create a new script named "GroupAllRoads" that groups all roads together:

```
script {
    create {
        selection = select {
            has predicate:'passesThrough'
        }
        set name:'Roads (' + selection.size + ')'
        set algorithm:'radial'
    }
}
```

(Hint: This script will create a new subgraph node.)

	--	-	-/+	+	++
The script helped a lot to solve this task	○	○	○	○	○
The above script is easy to unterstand	○	○	○	○	○

Problems/Comment

4. General Questions

1. General Questions

	--	-	-/+	+	++
The application provides useful assistance for the task of exploring an ontology.	○	○	○	○	○
The application is easy to operate with.	○	○	○	○	○
The domain specific language (DSL) to enter the scripts, is easy to understand.	○	○	○	○	○
The domain specific language (DSL) to enter the scripts, has a flat learning curve.	○	○	○	○	○
The graphical editor shows all nodes and edges in a clear arranged layout.	○	○	○	○	○
The Satellite View provides help to keep a general view of the entire graph.	○	○	○	○	○
The Filter View gives a detailed overview over all filter scripts.	○	○	○	○	○
The Script View gives a detailed overview over all scripts available to manipulate the graph.	○	○	○	○	○
The wizard windows for entering new scripts, clearly describes what a user has to enter into the input fields.	○	○	○	○	○

5. Features

1. Features

	--	-	-/+	+	++
Using the Groovy DSL script to filter and change the design of the graph, is a useful way to interact with the application.	○	○	○	○	○
The coloring of nodes/edges is a useful feature to give more meaning to the elements in the graph.	○	○	○	○	○
The edge tooltip provides helpful information about the selected edge.	○	○	○	○	○
The functionality to expand and collapse a single node is a useful feature that helps to manage the amount of nodes that are displayed in the graph.	○	○	○	○	○

OntoX Survey

6. Remarks

1. What did you like the most?

2. What didn't you like?

3. Please share your suggestions how to improve the prototype.

4. Please share any other remarks.

Appendix B

Class Node.groovy

```
/**
 * This class represents a node in the OntoX DSL language and
 * acts as a wrapper class for an OntoxNode object.
 */
class Node {

   protected AbstractGraphBuilder graphBuilder = null; //graph builder
   protected OntoxNode ontoxNode = null; //Node object

   /**
    * Fields that can be used from the OntoX DSL.
    * They provide an easy access for important information.
    */
   String fullURI = null; //full URI of the RDF resource
   String uri = null; //URI with namespace prefix
   String label = null; //Label of the node
   String name = null;  //Same value as label
   boolean isLiteral = false; //Is this node a literal?
   boolean isResource = false; //Is this node a resource?
   boolean isAnon = false; //Is this node an anonymous node?
   boolean isOntClass = false; //Represents this node an ontology class?
   boolean isIndividual = false; //Represents this node an indiviudal class?
   Object value = null; //If this node is a literal than it has a value.
   RDFNode rdfNode = null; //The RDFNode that this object represents
   def edges = { return this.getEdges(); }  //returns all connected edges
   def outEdges = { return this.getOutEdges(); }; //returns all outgoing edges
   def inEdges = { return this.getInEdges(); }; //returns all incoming edges

   /**
    * Constructor
    */
   public Node() {

   }

   /**
    * Set the OntoxNode object and compute the fields values.
 * @param ontoxNode  Node object
 */
public setOntoxNode(OntoxNode ontoxNode) {
   this.ontoxNode = ontoxNode;
   this.graphBuilder = ontoxNode.getGraphBuilder();
   this.setFieldValues();
}

/**
 * This method computes the field
 * values for this object.
 */
private setFieldValues() {
   this.fullURI = this.ontoxNode.getFullURI();
   this.uri = this.ontoxNode.getURI();
   this.label = this.ontoxNode.getNodeLabel();
   this.name = this.label;
   this.isLiteral = this.ontoxNode.getNode().isLiteral();
   if (this.isLiteral) {
      this.value = this.ontoxNode.getValue();
   }
   this.isResource = this.ontoxNode.getNode().isResource();
   this.isAnon = this.ontoxNode.getNode().isAnon();
   this.isIndividual = this.ontoxNode.isIndividual();
   this.isOntClass = this.ontoxNode.isOntClass();
   this.rdfNode = this.ontoxNode.getNode();
}

/**
 * Returns the literal value of given edge that is connected to this node.
 * @param predicateName  Name of edge (= predicate name)
 * @return     Value of the literal as String, Integer, etc., or null on error.
 */
def getLiteralValue(predicateName) {
   if (predicateName == null || predicateName == "") {
      return null;
   }
   if (graphBuilder == null || ontoxNode == null) {
      return null;
   }

   if ( ontoxNode.isResource() ) {
      ModelInterrogator modelInterrogator = this.ontoxNode.getGraphBuilder().getModelInterrogator();
      Collection <Statement> statements = modelInterrogator.getProperties(ontoxNode.getNode().asResource())
      for (Iterator iterator = statements.iterator(); iterator.hasNext();) {
         Statement statement = (Statement) iterator.next();
         if ( ModelHelper.computeEdgeLabel(statement).equals(predicateName) ) {
            RDFNode tempNode = statement.getObject();
            if (tempNode.isLiteral()) {
               Literal literal = (Literal) tempNode;
               return literal.getValue();
            }
         }
      }
   }
```

```
      else {
         return null;
      }
   }

   /**
    * Returns a list with all connected edges.
    */
   def getEdges() {
      return this.convertToEdgeType(this.ontoxNode.getConnections());
   }

   /**
   * Returns a list with all outgoing edges.
   */
   def getOutEdges() {
      return this.convertToEdgeType(this.ontoxNode.getSourceConnections())
   }

   /**
   * Returns a list with all incoming edges.
   */
   def getInEdges() {
      return this.convertToEdgeType(this.ontoxNode.getTargetConnections())
   }

   /**
    * Helper method that converts a list of OntoxEdge objects to Edge objects.
    * @param listToConvert  OntoxEdge list to convert
    * @return     New list with Edge objects
    */
   def convertToEdgeType (List<OntoxEdge> listToConvert) {
      ArrayList<Edge> resultList = new ArrayList<Edge>();
      for(ontoxEdge in listToConvert) {
         Edge edge = new Edge();
         edge.setOntoxEdge(ontoxEdge);
         resultList.add(edge);
      }
      return resultList;
   }
}
```

Appendix C

Class Edge.groovy

```
/**
* This class represents an edge in the OntoX DSL language and
* acts as a wrapper class for an OntoxEdge object.
*/
class Edge {

   protected AbstractGraphBuilder graphBuilder = null; //graph builder
   protected OntoxEdge ontoxEdge = null; //Edge object

   /**
    * Fields that can be used from the OntoX DSL.
    * They provide an easy access for important information.
    */
   String fullURI = null; //full URI of the RDF Statement (predicate)
   String uri = null; //URI with namespace prefix
   String label = null; //Label of the node
   String name = null; //Same value as label
   Node sourceNode = null; //Source node of edge
   Node targetNode = null; //Target node of edge
   Statement statement = null; //The Statement that this object represents


   /**
   * Constructor
   */
   public Edge() {

   }

   /**
   * Set the OntoxEdge object and compute the fields values.
   * @param ontoxEdge  Edge object
   */
   public setOntoxEdge(OntoxEdge ontoxEdge) {
      this.ontoxEdge = ontoxEdge;
      this.graphBuilder = ontoxEdge.getGraphBuilder();
      this.setFieldValues();
    }

    /**
    * This method computes the field
    * values for this object.
    */
    private setFieldValues() {
       this.fullURI = this.ontoxEdge.getFullURI();
       this.uri = this.ontoxEdge.getURI();
       this.label = this.ontoxEdge.getEdgeLabel();
       this.name = this.label;
       this.statement = ontoxEdge.getStatement();
       OntoxNode sourceOntoxNode = this.ontoxEdge.getSourceNode();
       OntoxNode targetOntoxNode = this.ontoxEdge.getTargetNode();
       this.sourceNode = new Node();
       this.sourceNode.setOntoxNode(sourceOntoxNode);
       this.targetNode = new Node();
       this.targetNode.setOntoxNode(targetOntoxNode);
    }
 }
```

References

[Alm] Andres Almiray. GraphicsBuilder graphicsbuilder is a groovy builder for java 2d, http://groovy.codehaus.org/graphicsbuilder.

[BdR06] W. Beaton and J. d. Rivieres. Eclipse platform technical overview. Technical report, The Eclipse Foundation, 2006.

[BLHL01] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific American*, 284(5):34–43, May 2001.

[Car06] Jorge Cardoso. Programming the semantic web. In Jorge Cardoso and Amit P. Sheth, editors, *Semantic Web Services, Processes and Applications*, volume 3 of *Semantic Web And Beyond Computing for Human Experience*, pages 351–380. Springer, 2006.

[Cui07] W. Cui. A survey on graph visualization. Technical report, Hong Kong University of Science and Technology, 2007.

[db.] Apache derby: Java relational database, http://db.apache.org/derby/.

[Dea10] F. Dearle. *Groovy for Domain-Specific Languages*. PACKT PUB, 2010.

[FE02] Carsten Friedrich and Peter Eades. Graph drawing in motion. *J. Graph Algorithms Appl.*, 6(3):353–370, 2002.

[GHJ94] Erich Gamma, Richard Helm, and Ralph E. Johnson. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley Longman, Amsterdam, 1st ed., reprint. edition, 1994.

[Gru93] Thomas R. Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2):199–220, 1993.

[GW04] Sunil Goyal and Rupert Westenthaler. Rdf gravity (rdf graph visualization tool), 2004.

[HCL05] Jeffrey Heer, Stuart K. Card, and James A. Landay. Prefuse: a toolkit for interactive information visualization. http://prefuse.org/. In *Proc. CHI 2005, Human Factors in Computing Systems*, 2005.

[hib] Hibernate relational persistence for java, http://www.hibernate.org/.

[jen] Jena semantic web framework, http://jena.sourceforge.net/.

[KGK$^+$07] Dierk Knig, Andrew Glover, Paul King, Guillaume Laforge, and Jon Skeet. *Groovy in Action*. Manning, 2007.

[KWV07] S. Krivov, R. Williams, and F. Villa. GrOWL: a tool for visualization and editing of OWL ontologies. *Journal of Web Semantics*, 5(2):54–57, 2007.

[Lik32] R. Likert. A technique for the measurement of attitudes. *Archives of Psychology*, 22(140):1–55, 1932.

[MFN03] J. Madadhain, D. Fisher, and T. Nelson. JUNG java universal network/graph framework, http://jung.sourceforge.net/, 2003.

[MFS$^+$05] J. Madadhain, D. Fisher, P. Smyth, S. White, and Y.B. Boey. Analysis and visualization of network data using jung. *Journal of Statistical Software*, 10:1–35, 2005.

[MG03] Paul Mutton and Jennifer Golbeck. Visualization of semantic metadata and ontologies. In *Proc. Information Visualization*, 2003.

[MGL06] Michael Meyer, Tudor Grba, and Mircea Lungu. Mondrian: an agile information visualization framework. In Eileen Kraemer, Margaret M. Burnett, and Stephan Diehl, editors, *SOFTVIS*, pages 135–144. ACM, 2006.

[MHS05] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys (CSUR)*, 37(4):316–344, 2005.

[Nov02] Ondrej Novak. Visualization of large graphs. Postgraduate study report, Faculty of Electrical Engineering, Czech Technical University in Prague, 2002.

[OWL04] Owl web ontology language overview. W3c recommendation, World Wide Web Consortium, February 2004.

[Pow03] Shelley Powers. *Practical RDF - solving problems with the resource description framework.* O'Reilly, 2003.

[pre06] Prefuse information visualization toolkit, http://prefuse.org/, 2006.

[pro] Ontograf - protege wiki, http://protegewiki.stanford.edu/wiki/ontograf.

[PS08] Eric Prud'hommeaux and Andy Seaborne. SPARQL query language for RDF. W3C Recommendation, 15 Jan. 2008. Available at .

[RWC11] D. Rubel, J. Wren, and E. Clayberg. *The Eclipse Graphical Editing Framework (Gef)*. Eclipse Series. Pearson Education, Limited, 2011.

[Shn96] Ben Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *VL '96: Proceedings of the 1996 IEEE Symposium on Visual Languages*, page 336. IEEE Computer Society, 1996.

[SNM$^+$02] Margaret-Anne D. Storey, Natasha F. Noy, Mark A. Musen, Casey Best, Ray W. Fergerson, and Neil A. Ernst. Jambalaya: an interactive environment for exploring ontologies. In *IUI*, pages 239–239, 2002.

[Tuf01] Edward R. Tufte. *The Visual Display of Quantitative Information*. Graphics Press, Cheshire, CT, 2. edition, 2001.

[wc3a] Ontology vocabularies, http://www.w3.org/standards/semanticweb/ontology.

[wc3b] Semantic Web, http://www.w3.org/standards/semanticweb/.

[WGRG10] Michael Wuersch, Giacomo Ghezzi, Gerald Reif, and Harald Gall. Supporting developers with natural language queries. In Jeff Kramer, Judith Bishop, Premkumar T. Devanbu, and Sebastin Uchitel, editors, *ICSE (1)*, pages 165–174. ACM, 2010.

[Zel95] John M. Zelle. *Using Inductive Logic Programming to Automate the Construction of Natural Language Parsers*. PhD thesis, Department of Computer Sciences, The University of Texas at Austin, Austin, TX, 1995.