OC

AN INTEGRATED ONTOLOGY DEVELOPMENT ENVIRONMENT FOR DATA EXTRACTION

by

Kimball A. Hewett

A thesis submitted to the faculty of

Brigham Young University

In partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science

Brigham Young University

April 2000

BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

Of a thesis submitted by

Kimball A. Hewett

This thesis has been read by each member of the following graduate committee and by majority vote has been found to be satisfactory.

25 Apr 2000
Date

David W. Embley, Committee Chairman

25 Apr. 2000
Date

Stephen W. Liddle, Committee Member

25 Apr 2000
Date

Scott N. Woodfield, Committee Member

BRIGHAM YOUNG UNIVERSITY

As chair of the candidate's graduate committee, I have read the thesis of Kimball A. Hewett in its final form and have found that (1) its format, citations, and bibliographical style are consistent and acceptable and fulfill university and department style requirements; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the graduate committee and is ready for submission to the university library.

April 25, 2000
Date

David W. Embley
Chair, Graduate Committee

Accepted for the Department

J. Kelly Flanagan
Graduate Coordinator

Accepted for the College

Nolan F. Mangelson
Associate Dean
College of Physical and Mathematical Sciences

ABSTRACT

AN INTEGRATED ONTOLOGY DEVELOPMENT ENVIRONMENT

FOR DATA EXTRACTION

Kimball A. Hewett

Department of Computer Science

Master of Science

There is an enormous amount of readily accessible data on the World Wide Web today. Unfortunately, this data is mainly unstructured making it extremely difficult to search and impossible to conduct traditional database queries. The Data Extraction Group has developed a system to address this opportunity [E+99]. This system extracts and structures data from unstructured sources, based on an ontology that describes the data with its relationships, identifying keywords, and lexical appearance. The ontology is parsed to produce a database scheme, and the identifying keywords are used in a keyword recognizer that is fed unstructured Web documents. Matches are organized into records according to the ontology. These records can then be used for structured queries in a traditional database. The Data Extraction Group has created a number of separate automated processes to assist in the extraction of data from the web. However, a tool for managing the complexity of an ontology has not been created. We also desired a portable solution to allow our processes to be run on multiple platforms.

The purpose of this thesis is to create a portable integrated ontology development environment as a tool to facilitate the creation of application ontologies. This tool provides a method of editing an Object-Relationship Model (ORM) and its associated data frames. It provides debugging functionality for editing data frames by displaying sample text with highlighting on identified structure. Furthermore, it provides the ability to export the application ontology for use in the data extraction process.

ACKNOWLEDGMENTS

I would like to thank the many people who supported and assisted me in the completion of this thesis project. I want to thank Earl Boyce, my supervisor at *epixtech, inc.*, who allowed me to take the time I needed away from work to attend classes or visit my professors. I want to thank my graduate committee: Dr. Embley, Dr. Liddle and Dr. Woodfield for their valuable input and insight into the project. I especially want to thank Dr. Liddle for the great deal of effort that he made to meet with me each week and work through design and programming issues with me. Thanks to David Lewis who assisted me in some of the final stages of the programming. To my children: Sarah, Heather and Joseph for their unfailing love. Most of all I would like to show my appreciation for my wife, Rochelle, for enduring the many long hours that I spent on the computer rather than with her and for her love and support throughout this project.

TABLE OF CONTENTS

LIST OF FIGURES

1 INTRODUCTION

1.1 Background of Data Extraction

The advent of the World Wide Web has brought with it an enormous amount of readily accessible data. Unfortunately, this data is mainly unstructured making it difficult to search and impossible to conduct traditional database queries. The paper A Conceptual-Modeling Approach to Extracting Data from the Web [E+99] describes a method for extracting this data from unstructured sources. Figure 1 gives a graphical view of this process. The approach is to define an ontology that describes the data with its relationships, identifying keywords, and lexical appearance. The ontology is then parsed to produce a database scheme and the identifying keywords are sent to a keyword recognizer that is fed unstructured Web documents for possible matches to the ontology. The result is data that can be inserted into a database for querying.

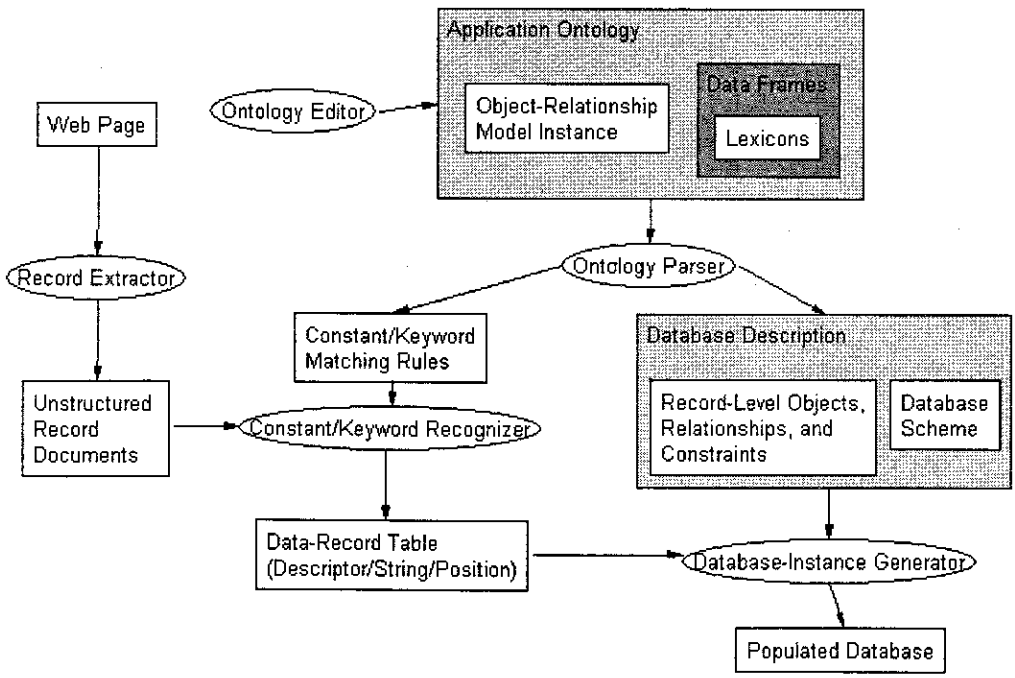


Figure 1: Data Extraction Process

1.2 Overview of Ontology Editor's Role

The shaded box for the application ontology shown in Fig. 1 represents an ontology with its associated structural model and identifying keywords. Up to this point, the only manual step in the data extraction process was the creation of the ontologies. The ontologies have been written manually, using a formal language. This manual creation of ontologies is a tedious process prone to errors that can be difficult to debug. The purpose of this thesis project is to create an integrated ontology development environment in order to facilitate this process. This graphical tool consists of an Object-Relationship Model (ORM) editor, a data frame editor, a text viewer, and a method for exporting the application ontology.

An Object-Relationship Model (ORM) instance is a conceptual model of the structure of a set of objects and relationships [EKW92, Embley98]. Other ORM editors exist but they do not provide a mechansim for editing and debugging data frames, nor are they portable to multiple platforms [OSM]. The ORM editor of this project provides the ORM editing capabilities of the previous editors with enhancements but is implemented such that it is portable as well as being integrated with a data frame editor.

Data frames describe extraction patterns and identifying keywords for object sets defined in the ORM. Each lexical object set in the ORM may have a data frame associated with it. The patterns within a data frame mentioned above consist of regular expressions. One purpose of the data frame editor of this project is to reduce the possibility of syntactical errors. As an example of the complexity of the data frame syntax, the following text is an excerpt from a relatively simple data frame on student course work that consists of the definition of the "Course Number" data frame:

```
Data frame Course Number [4]
Value " (9\d|[1-7]\d\d)R?"
   " (\.|Courses|advanced undergraduates\))\s*"  left adjoining and
   " \.\s*(\(([^)])*\))?([^(])*\(\d"  right adjoining;
end;
```

The text viewer is a debugging tool that assists the ontology creator in defining correct regular expressions within a data frame. The user associates a sample text file with the text viewer, which in turn highlights the matches from the regular expressions found in a data frame. A pattern editor [L98] was previously created to assist in debugging regular expressions. This project integrates and extends that capability to allow debugging to take place while editing a data frame.

2 ANALYSIS OF ONTOLOGY EDITOR

2.1 General Goals

1. **Portability** – The project is portable to Unix, Windows, and Macintosh operating systems. This requirement allows multi-platform usage. In order to achieve this end the project uses Java as its programming language. A prototype with some of the required functionality was written during the analysis stage to ensure that Java could provide the necessary capabilities.
2. **Extensibility** – The current project provides a graphical Object Relationship Model (ORM) editor. The project lays the ground work for extending the functionality to include Object Behavior Models (OBM) and Object Interaction Models (OIM) so that subsequent projects can benefit from the work already completed.
3. **Maintainability** – As the basis for future projects, the project code must be created in a manner that it can be easily maintained. This is done through the use of good programming practices. Specifically, the project is documented through the analysis and design as well as the implementation. The implementation documentation has been created through the use of `javadoc` comments. Furthermore, code reviews ensure that the code is well written, understandable and suitably commented.
4. **Features**
    a. Graphical ORM Editor – the ORM editor provides an interface for defining the structural relationships within an ontology.
    b. Graphical Data Frame Editor – the data frame editor provides an interface for defining the regular expressions that identify objects defined in the ORM.
    c. Graphical Text Viewer – the text viewer allows for graphical highlighting of identified objects within a sample of text. This tool facilitates debugging of regular expressions defined in the data frames.

2.2 Graphical and Structural Analysis

2.2.1 ORM Editor

The ORM editor supports the ORM editing features contained in Allegro [C98]. Specifically it supports the following features:

1. Editing of structures contained in an ORM (see Fig. 2)
2. A Multiple Document Interface (MDI) environment
    a. Multiple views
    b. Toolbar support
    c. Support for tiling and cascading windows
3. Drawing preferences
    a. Font
    b. Color
    c. Line width
4. Editing support
    a. Cut, Copy and Paste
    b. Selection of objects
    c. Alignment
    d. Move to front (move to back)
5. Persistence support
    a. Reading to file
    b. Writing to file

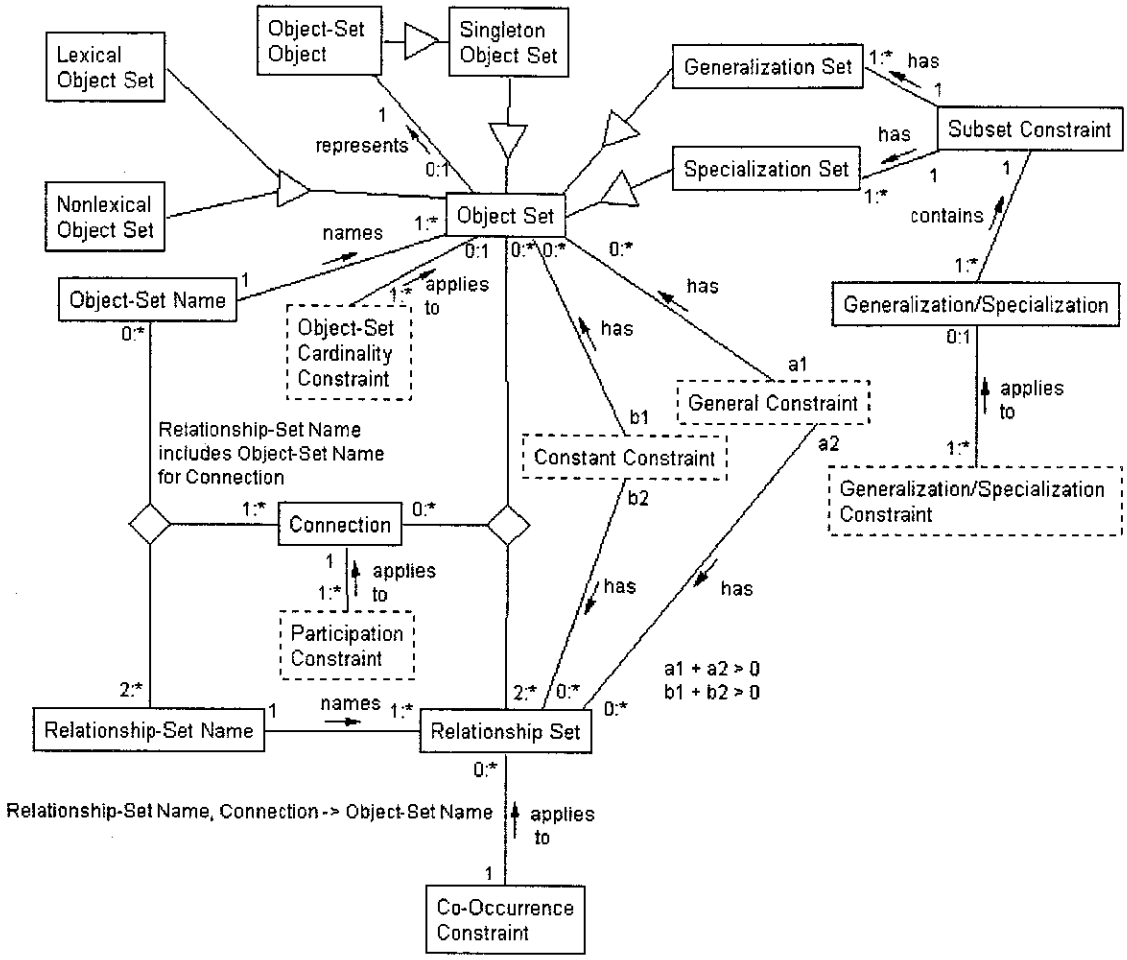


Figure 2: Object Relationship Model

2.2.2 Data Frame Editor

A data frame includes the value and context expressions needed to extract objects (defined in an ORM) from an unstructured text source. The graphical data frame editor needs to provide a graphical interface for entering value and context expressions. The interface must provide for entry of the information shown in Fig. 3.

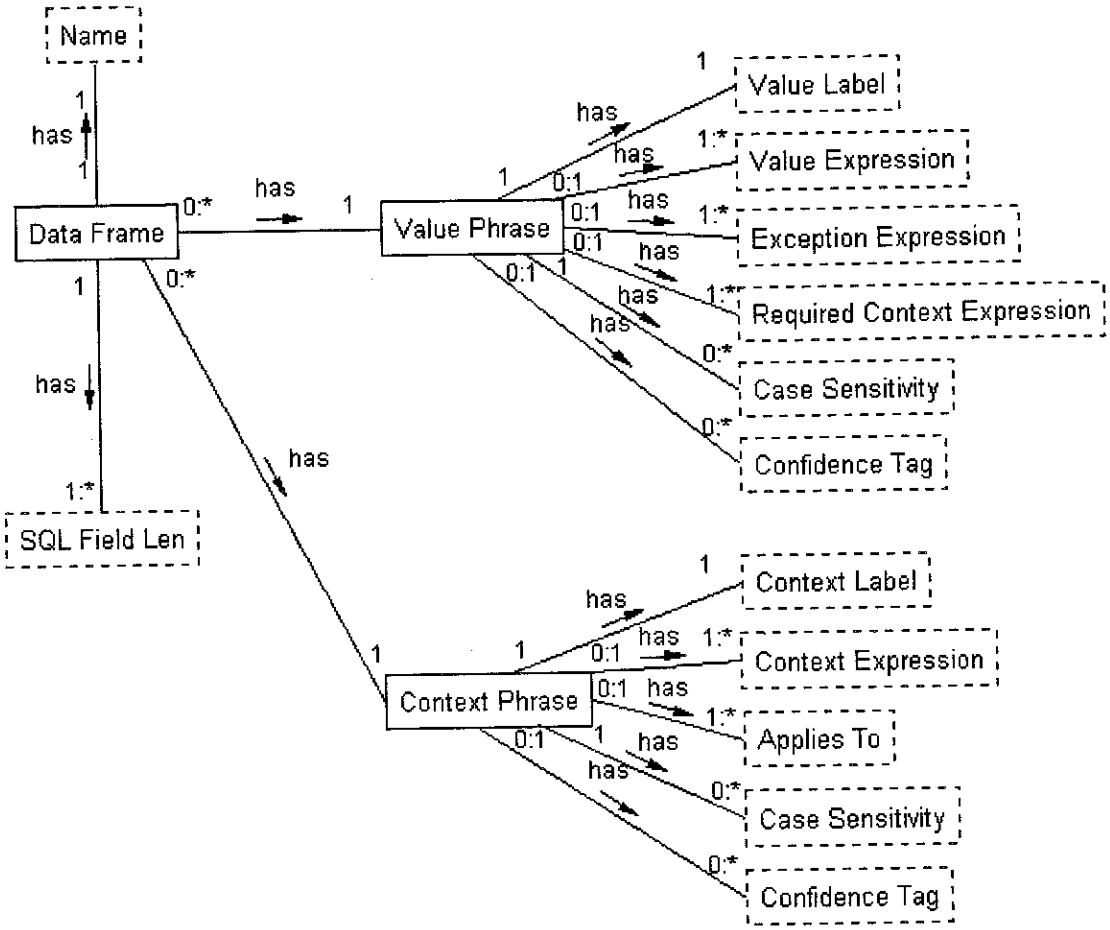


Figure 3: Data Frame ORM

2.2.3 Text Viewer

The graphical text viewer is used for debugging data frames. It takes advantage of the pattern matching conducted by the Pattern Editor utility [L98] to display text matches from the data frames. The graphical text viewer allows the user to associate a sample text file with the viewer. When the user selects a data frame the graphical text viewer highlights the text matching the regular expressions in the data frame. Fig. 4 is a screen shot of the text viewer integrated with the data frame editor.

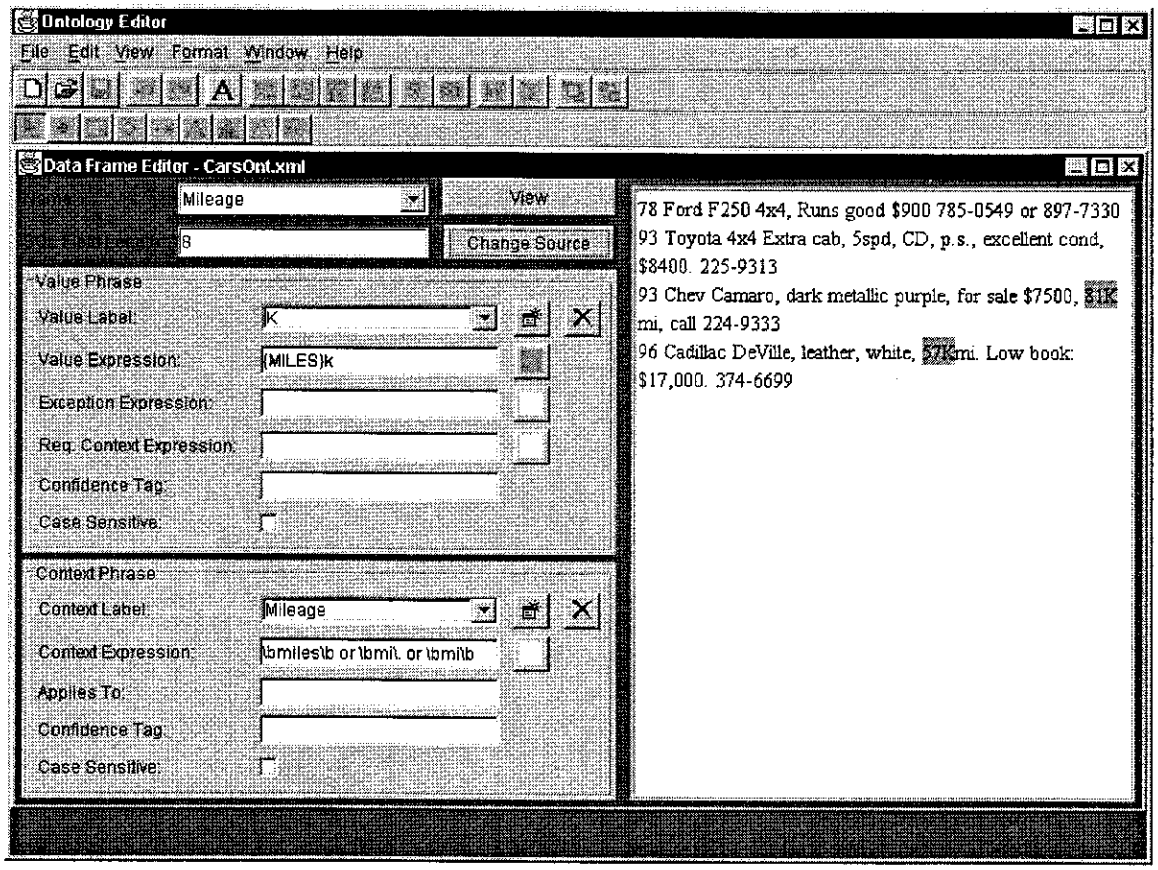


Figure 4: Text Viewer

3 DESIGN OF ONTOLOGY EDITOR

The description of the design is divided into two main sections: the graphical design dealing with the user interface and the class design dealing with the internal objects of the Ontology Editor. Both play an integral role in the complete design of the Ontology Editor. We first look at the graphical design followed by the class design.

3.1 Graphical Design

This section describes the user interface that has been designed and gives screen shots of the implementation of the design. There are three main areas of concern in the graphical design: the ORM editor, the data frame editor and the text viewer that is used for debugging data frames. Together they constitute the main editing functionality that the user sees, and each is discussed in turn.

3.1.1 ORM Editor

The ORM editor is similar to Allegro's GUI interface [C98]. As such, many of the menu options and toolbar buttons will be familiar to those who have worked with Allegro. The goal of this portion of the project was not to redesign the ORM editor found in Allegro but rather implement it in Java so that it would be portable in the future. Therefore the Ontology Editor's look and feel is similar to Allegro (see Fig. 5).

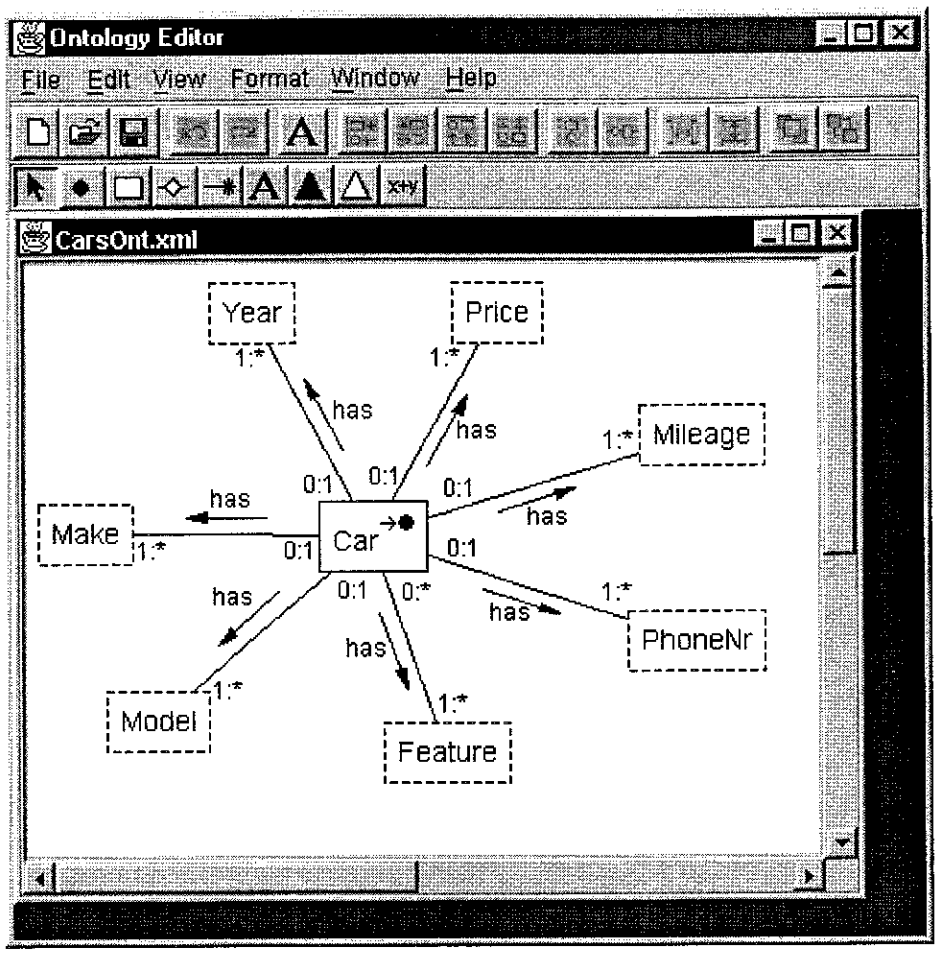


Figure 5: Ontology Editor Screen Shot

The Ontology Editor has the concept of a drawing state. The user can create new objects or object sets by clicking on the object or object set state buttons respectively. This sets the "state" of the editor, at which point any click in the drawing canvas creates an object or object set depending upon the state. The user can edit the name of an object or object set by double clicking on the text field of the name. Indeed, any text field can be edited in the same fashion. Right-clicking on an object set displays a popup menu with options available for modifying the object set such as making it lexical, read-only, an object-set object, or adding a cardinality constraint (Fig. 6).

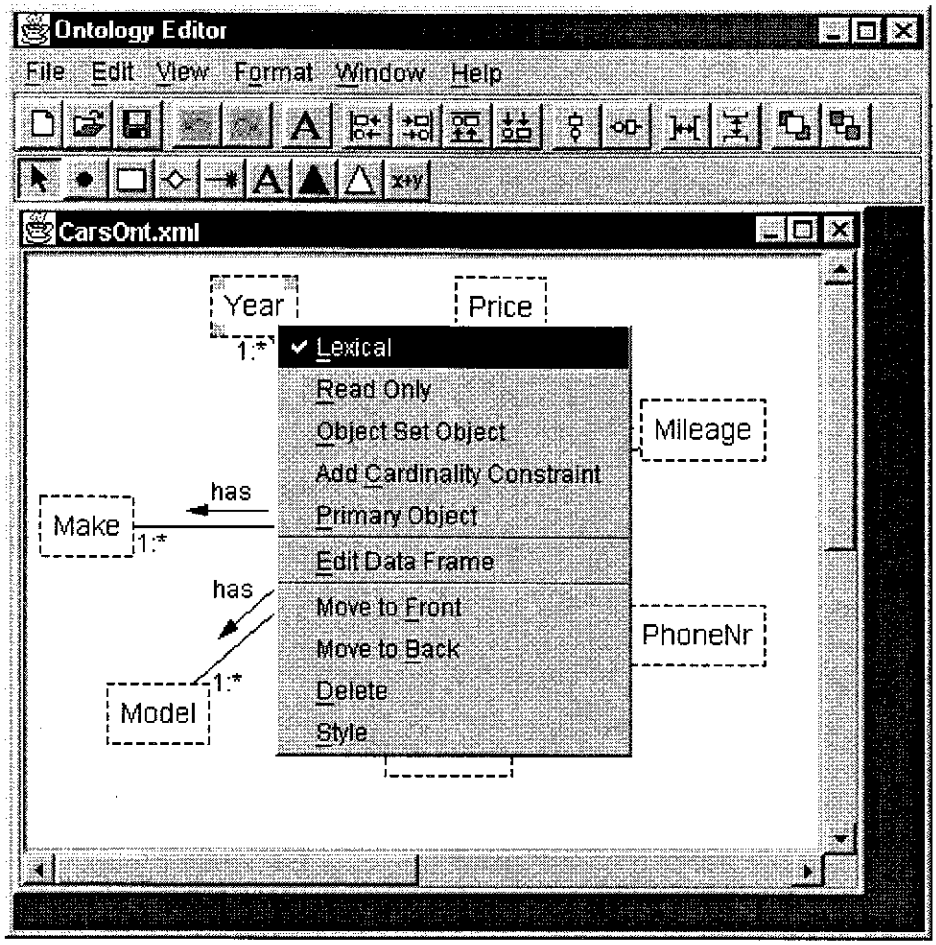


Figure 6: Object Set Popup Menu

The user can create relationship sets by clicking on the relationship set button, clicking on one object set, and then dragging the mouse cursor to another object set where he releases the mouse button. This action creates a default relationship set between the two object sets. The user can then customize the relationship set by editing the participation constraints and relationship set name. Right-clicking on a relationship set displays a popup menu with options to show left and/or right arrow heads, show a diamond, or add a co-occurrence constraint (Fig. 7).

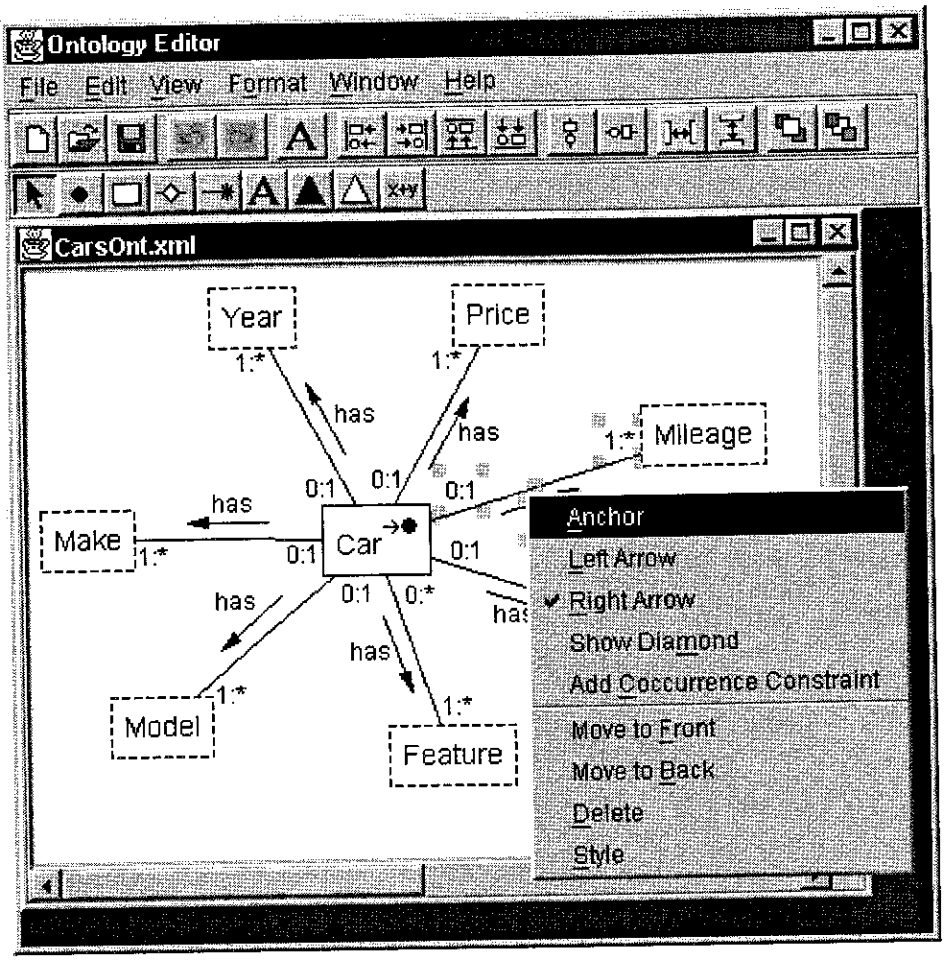


Figure 7: Relationship Set Popup Menu

The toolbar also provides state buttons for adding notes, creating specializations and adding general constraints. The selection state button allows the user to select and move object sets, notes, text fields, and other items that have been added to the canvas. Above the state button toolbar is a toolbar with action buttons. Clicking an action button performs the corresponding manipulation, such as aligning objects, bringing objects to the front or sending them to the rear of other objects, and changing fonts, line widths and coloring. All toolbar actions are also available through the pull down menus.

3.1.2 Data Frame Editor

Prior to the Ontology Editor, data frames were created by writing textual descriptions. Fig. 3 showed the structure of the data required in data frames. Fig. 8 shows the graphical design of the data frame editor dialog. The "Name" field is a combo box from which the user can select any of the object sets that exist on the canvas. Once a name is selected, all fields are updated to show the value and context phrases associated with the object set. Initially an object set does not have any value or context phrases, but they can be added by clicking the new button to the right of the "Value Label" combo box or "Context Label" combo box. After adding a new phrase, the user can edit its label in the combo box. The user can remove unwanted phrases by selecting the phrase in the combo box and clicking the delete button.

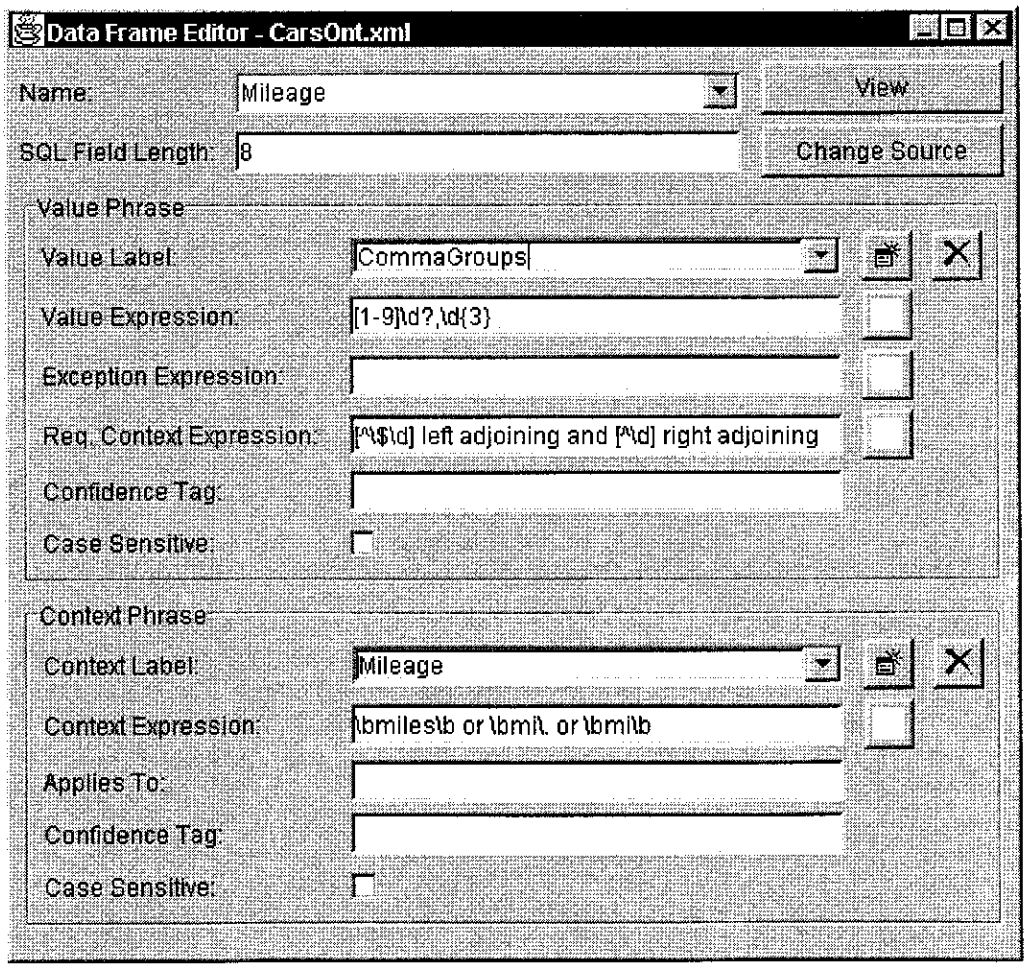


Figure 8: Data Frame Editor

The expressions below the Value and Context labels are extended regular expressions that can be entered in the corresponding edit control. To the right of each expression edit control is a colored button. The color of the button is the color used by the text viewer to highlight matches when the regular expression is found in the scanned text. If the color is pure white, highlighting is disabled for the corresponding expression.

In order to facilitate regular expression creation, macros and lexicons can be defined and subsequently used in data frames. Like the data frame editor, the macro and lexicon editor is needed to edit regular expressions. Therefore it has a similar design; due to the similarities it is not further mentioned here. Macros are regular expressions that we expect to reuse and lexicons reference a file of common strings such as a list of surnames.

3.1.3 Text Viewer

The text viewer uses a Java class library called ORO Matcher [ORO99] for manipulating Perl 5 style regular expressions. The Ontology Editor integrates the text viewer with the data frame editor and the macro frame editor. In Fig. 8 shows a View button and a Change Source button. The Change Source button is used to select a file to be used as the text in which to search for the regular expressions defined in the data frame or macro.

The View button expands the data frame editor dialog to show the testing text, which is highlighted with the colors associated with the regular expressions from the data frames or macros. (Fig. 4) Note: JDK 1.2's JTextArea does not support the coloring. JDK 1.3 or higher is required in order to see the highlighting colors.

3.2 Class Design

Many of the classes that are discussed shortly follow a model-view-controller architecture. Particularly when the classes represent persistent objects. The basic idea of the model-view-controller architecture is to separate out underlying data structure from the user interface controls. All classes that represent data structures are called "models" and the classes that represent the user interface controls are called "views". The concept is that different views can be used to display a model.

3.2.1 Framework Classes

In order to support the Multi-Document Interface the Ontology Editor allows for several application ontology documents to be opened at one time. Fig. 9 graphically describes this design. Each document has a frame associated with it. A frame is a child frame main window that delegates all knowledge of ontology editing to an editing canvas. When a user interacts with the Ontology Editor, it passes the messages on to either the document or frame as appropriate. The frame delegates ontology editing messages to the canvas. The canvas also receives editing interaction directly from the user. The canvas handles all editing interaction, and specifically all keyboard and mouse interactions within the canvas area. The classes `OntologyDocument`, `OntologyFrame`, and `OntologyCanvas` represent the document, frame, and canvas respectively.

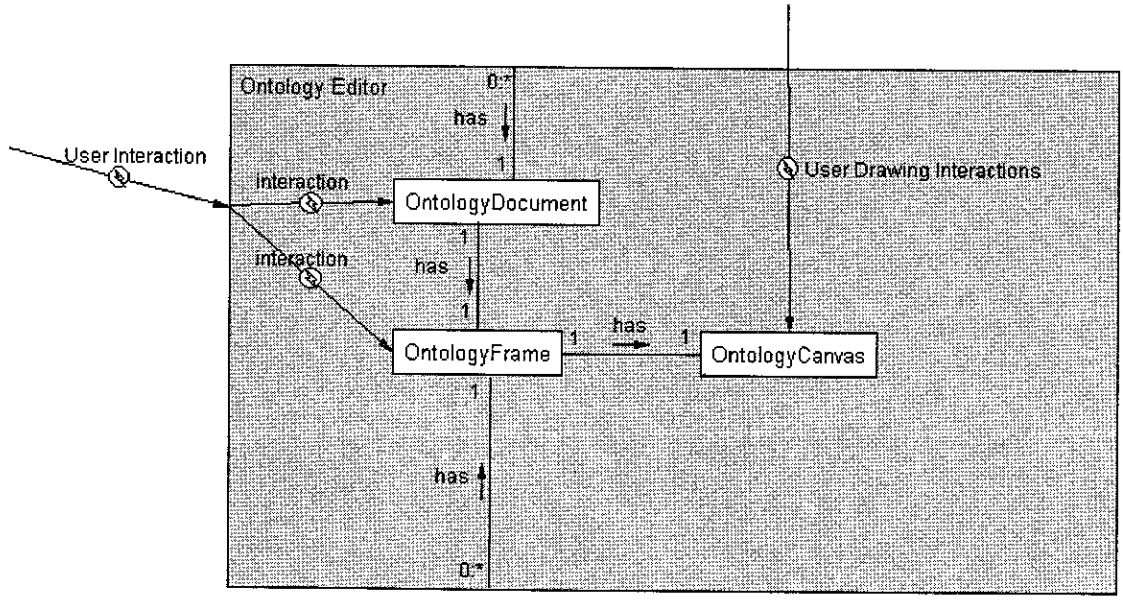


Figure 9: General Design

3.2.2 ORM Classes

All ORM objects derive from `DrawObj`. `OntologyCanvas` is the container that displays all `DrawObj`'s in the ORM. It also instantiates all derivations of `DrawObj` as required by the menu/button

options selected by the user. The `DrawObj` class encapsulates all of the common functionality of objects, object sets, relationship sets, etc.

In order to separate the user interface from actual data, each `DrawObj` has an associated model. The model itself has no knowledge of the `DrawObj`, nor how it is to be displayed or used. `DrawObj` contains the rules for drawing a model. Fig. 10 shows the relationships between the various `DrawObj` derivations and their associated models.

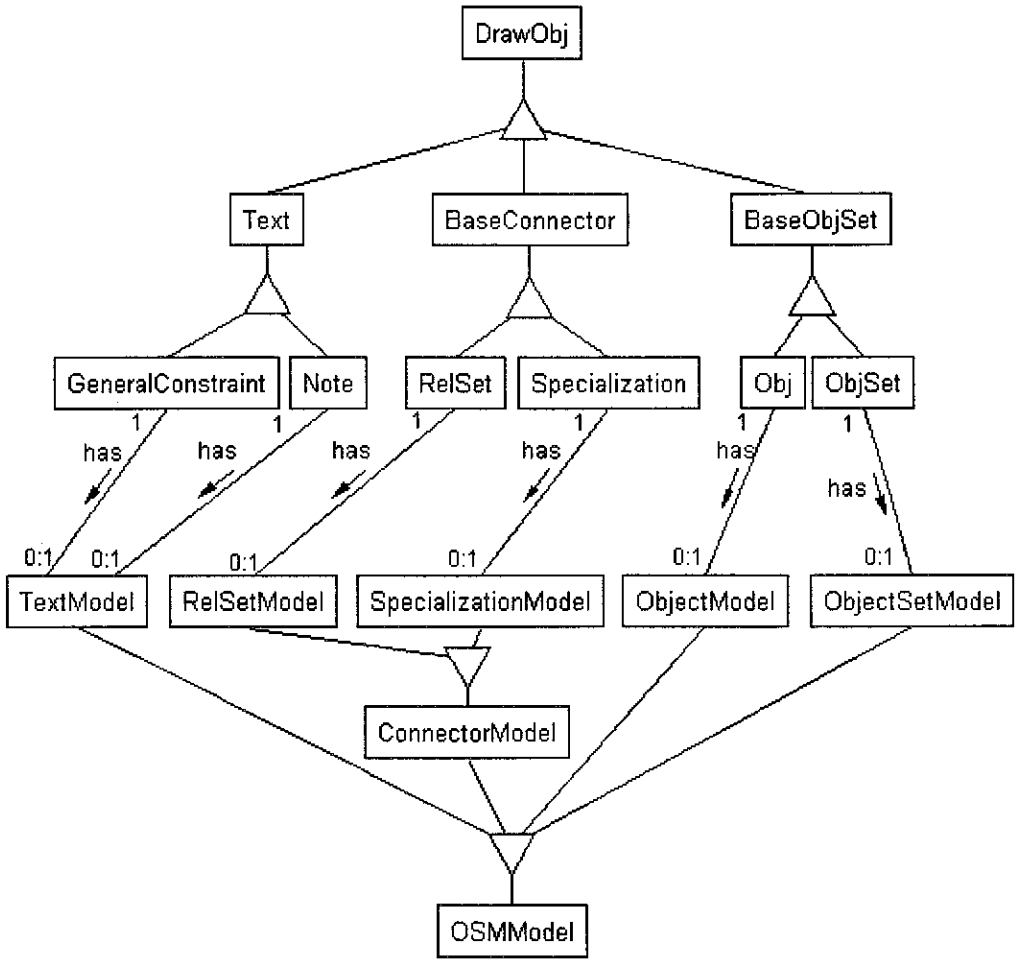


Figure 10: ORM Object Design

3.2.3 Data Frame Classes

The graphical design of the data frame editor basically provides a dialog with multiple combo boxes and edit controls. Unlike the ORM editor, which required specialized "view" components for drawing to the canvas, the data frame editor need only tie in the data contained in models to the user interface controls of the dialog. Therefore the design of the data frame classes concentrates only upon the models used to contain that data. In the analysis section, Fig. 3 showed the relationships that comprise a data frame. We simply need to convert the nonlexical object sets Data Frame, Value Phrase and Context Phrase from Fig. 3 to the models `DataFrameModel`, `ValuePhraseModel` and `ContextPhraseModel` as shown in Fig. 11.

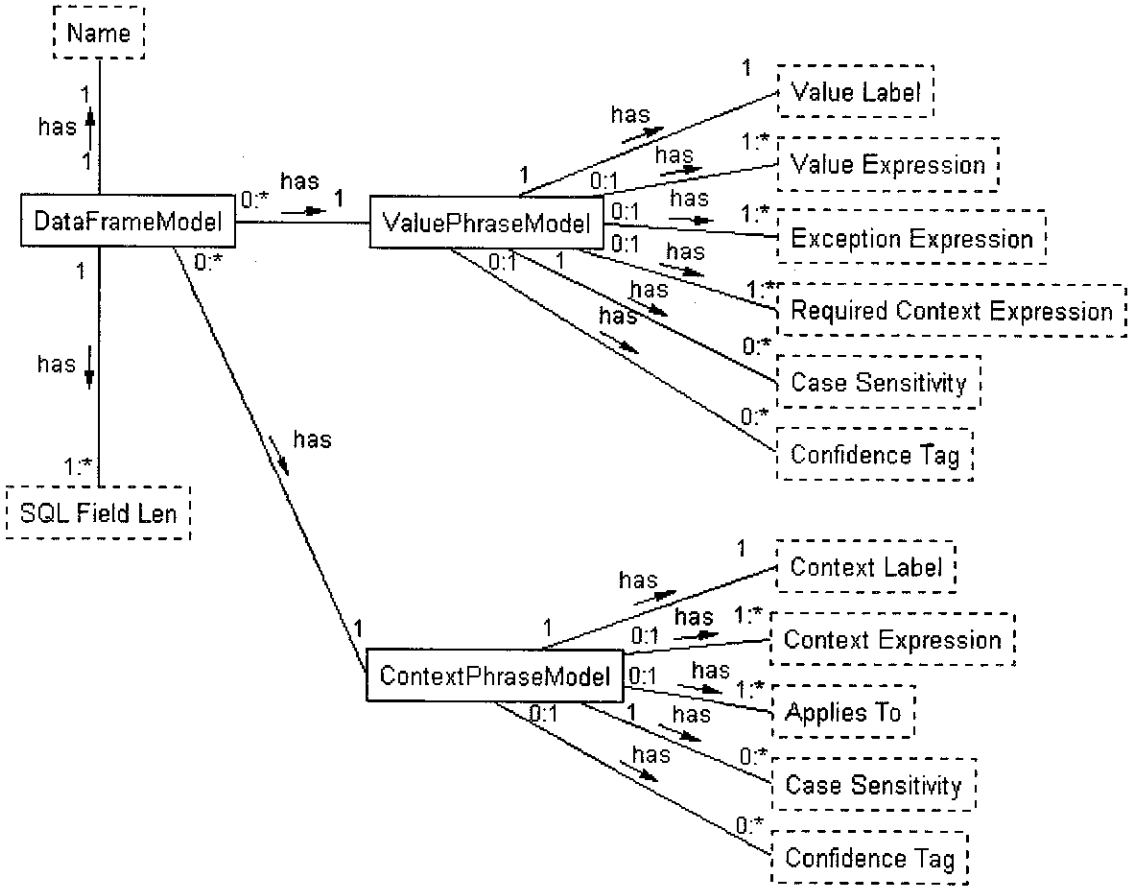


Figure 11: Data Frame Design

`DataFrameModel`, `ValuePhraseModel` and `ContextPhraseModel` each provide methods for accessing their associated lexical data. The `DataFrameModel` in addition provides methods

for accessing the `ValuePhraseModel` and `ContextPhraseModel` instances that participate in relationships with it.

3.2.4 Persistence

We chose XML as the foundation for persistence. The file storage method used by Allegro was MFC specific and did not allow for data frame information. The previous hand-created ontologies and data frames were strictly textual and did not specify graphical user interface information. Either way a new format was required for our project. Instead of continuing to use a proprietary format we chose to use an open format that would allow for other products to visualize the stored data. Furthermore, we found numerous existing XML implementations that we could leverage. We chose to use Sun's Java Project X: Technology Release 2. It is a Java class libraries that implements the XML Java interfaces defined by the World Wide Web Consortium (W3C). We conducted some preliminary testing of the toolkit and found it to be up to the task and easy to use.

The following is the Document Type Definition (DTD) for the Ontology Editor. It allows the storage of user interface information such as the layout of the objects, but does not require it. The DTD was designed with the idea of allowing it to be used by follow on tools that require the data but not the user interface information.

```
<!ELEMENT OSM (Style?,(ObjectSet | Object | GeneralConstraint | Note |
            RelationshipSet | GenSpec | Association | Macro |
            Lexicon)*)>
<!ATTLIST OSM
    x       CDATA   #IMPLIED
    y       CDATA   #IMPLIED
    width   CDATA   #IMPLIED
    height  CDATA   #IMPLIED
    ID      NMTOKEN "1"
>

<!ELEMENT ObjectSet (DataFrame?, Style?,(ObjectSet |
RelationshipSet)*)>
<!ATTLIST ObjectSet
    ID              NMTOKEN #REQUIRED
    Name            CDATA   "ObjSet"
    x               CDATA   "0"
    y               CDATA   "0"
    Lexical             (Y|N) "N"
    ReadOnly            (Y|N) "N"
    HighLevel           (Y|N) "N"
    Primary       (Y|N) "N"
    ObjectSetObject     NMTOKEN     #IMPLIED
```

```
    CardinalityConstraint CDATA #IMPLIED
>

<!ELEMENT Object (ObjectName)>
<!ATTLIST Object
    ID              NMTOKEN #REQUIRED
    x               CDATA   "0"
    y               CDATA   "0"
    ObjectSet       NMTOKEN #IMPLIED
>

<!ELEMENT ObjectName (Text)>
<!ELEMENT Note (Text)>
<!ELEMENT GeneralConstraint (Text)>

<!ELEMENT Text (Style?)>
<!ATTLIST Text
    ID              NMTOKEN #IMPLIED
    x               CDATA   #IMPLIED
    y               CDATA   #IMPLIED
    text            CDATA   #IMPLIED
>

<!ELEMENT Style (Line?,Font?)>
<!ATTLIST Style
    FillColor       CDATA   #IMPLIED
>

<!ELEMENT Font EMPTY>
<!ATTLIST Font
    Family          CDATA   #IMPLIED
    Style           CDATA   #IMPLIED
    Size            CDATA   #IMPLIED
    Color           CDATA   #IMPLIED
>

<!ELEMENT Line EMPTY>
<!ATTLIST Line
    Width       CDATA   #IMPLIED
    Color       CDATA   #IMPLIED
>

<!ELEMENT RelationshipSet (Style?,RelationshipSetName, Connection,
            Connection+, CoOccurrenceConstraint*, (ObjectSet |
            RelationshipSet)*)>
<!ATTLIST RelationshipSet
    ID              NMTOKEN #REQUIRED
    x               CDATA   #IMPLIED
    y               CDATA   #IMPLIED
    LeftArrow       (Y|N)   "N"
    RightArrow      (Y|N)   "Y"
    ShowDiamond     (Y|N)   "N"
>

<!ELEMENT RelationshipSetName (Text)>
```

```
<!ELEMENT Connection (Style?, ParticipationConstraint*)>
<!ATTLIST Connection
    ObjectSet           NMTOKEN         #REQUIRED
>
<!ELEMENT ParticipationConstraint (Text)>

<!ELEMENT CoOccurrenceConstraint EMPTY>
<!ATTLIST   CoOccurrenceConstraint
    x                       CDATA   #IMPLIED
    y                       CDATA   #IMPLIED
    LeftSet                 CDATA   #IMPLIED
    RightSet                CDATA   #IMPLIED
    CardinalityConstraint   CDATA   #IMPLIED
>


<!ELEMENT GenSpec (Style?, GenConnection+, SpecConnection+)>
<!ATTLIST GenSpec
    ID              NMTOKEN #REQUIRED
    x               CDATA   #IMPLIED
    y               CDATA   #IMPLIED
    GenSpecConstraint (UNION|MUTEX|PARTITION|INTERSECTION|NONE)   "NONE"
>
<!ELEMENT SpecConnection (Style?)>
<!ATTLIST SpecConnection
    ObjectSet           NMTOKEN         #REQUIRED
>
<!ELEMENT GenConnection (Style?)>
<!ATTLIST GenConnection
    ObjectSet           NMTOKEN         #REQUIRED
>

<!ELEMENT Association (Connection, Connection)>
<!ATTLIST Association
    ID              NMTOKEN #REQUIRED
    x               CDATA   #IMPLIED
    y               CDATA   #IMPLIED
>

<!ELEMENT DataFrame ((ValuePhrase | ContextPhrase)*)>
<!ATTLIST DataFrame
    SQLFieldLen     CDATA   #IMPLIED
>

<!ELEMENT ValuePhrase EMPTY>
<!ATTLIST ValuePhrase
    Label                   CDATA   #IMPLIED
    ValueExpression         CDATA   #IMPLIED
    ValueExpColor           CDATA   #IMPLIED
    ExceptionExpression     CDATA   #IMPLIED
    ExceptionExpColor       CDATA   #IMPLIED
    ReqContextExpression    CDATA   #IMPLIED
    ReqContextExpColor      CDATA   #IMPLIED
    ConfidenceTag           CDATA   #IMPLIED
    CaseSensitive           (Y|N)   "N"
>
```

```
<!ELEMENT   ContextPhrase   EMPTY>
<!ATTLIST   ContextPhrase
    Label                   CDATA   #IMPLIED
    ContextExpression       CDATA   #IMPLIED
    ContextExpColor         CDATA   #IMPLIED
    AppliesTo               CDATA   #IMPLIED
    ConfidenceTag           CDATA   #IMPLIED
    CaseSensitive           (Y|N)   "N"
>

<!ELEMENT   Lexicon         EMPTY>
<!ATTLIST   Lexicon
    Label                   CDATA   #REQUIRED
    FileName                CDATA   #REQUIRED
    CaseSensitive           (Y|N)   #IMPLIED
    Separators              CDATA   #IMPLIED
>

<!ELEMENT   Macro           EMPTY>
<!ATTLIST   Macro
    Label                   CDATA   #REQUIRED
    Body                    CDATA   #REQUIRED
    Color                   CDATA   #IMPLIED
>
```

4 IMPLEMENTATION OF ONTOLOGY EDITOR

In order to achieve the general goal of portability, we chose Java as the implementation language. In researching Java, we discovered that the Abstract Windowing Toolkit (AWT) was not as powerful as we would have liked. However, the Java Foundation Classes (JFC) provided many of the controls that were lacking in AWT. It also provided a framework to assist development of an MDI application. Furthermore, JFC is written such that it does not use native operating system GUI controls. Rather, it creates each control itself, which allows for the development of an application that will look the same on any platform. JFC allows the "look and feel" to be chosen at run time from among Windows, Motif, or native Java "Metal" look and feel.

Initially, we started development with JDK 1.1.7. This version of the JDK allowed for JFC to be used as a separate package. The release of JDK 1.2 integrated JFC into the JDK. Version 1.1.7 though useful still had some limitations. The graphics toolkit was rather immature and did not provide some useful functionality. In particular, it did not provide for drawing lines greater than a single pixel in width. JDK 1.2 greatly enhanced the graphics toolkit. It not only allowed for lines of any width, but also provided for drawing end caps at the end of each line. Another weakness of JDK 1.1.7 was that it only allowed for the translation of drawings within the coordinate space. JDK 1.2 allows translation, rotation, scaling, and more. All of these are very powerful drawing capabilities that are truly needed in a graphical editing application such as the Ontology Editor. Therefore, the Ontology Editor application took advantage of the capabilities of JDK 1.2 and requires JRE 1.2 or higher to run.

We first discuss the code implementing our design, also providing some statistics about the code. We then describe the use of Javadoc for creating code documentation. The code and instructions for installation of this project can be found at http://www.deg.byu.edu/ontologyeditor.

4.1 Coding

The coding section is organized into the following sections: Framework, ORM Editor, Data Frame Editor, Persistence, and Coding Statistics. The Framework section describes the classes used for the implementation of the frame windows, menus, and toolbars that support the Multi-Document Interface (MDI). The ORM Editor section describes the implementation of the class design discussed earlier.

Likewise the Data Frame Editor section explains the classes used for implementing data frames, macros and lexicons. The Persistence section covers how data captured in classes from the framework, ORM editor, and data frame editor are saved and restored using XML. Finally, the Statistics section gives statistical information about the code such as the number of lines of code written in the project.

4.1.1 Framework

The framework was designed to make extensions straightforward. With that goal in mind, we have worked to support extensions with few changes to the existing code. For example, there is an external properties file that can be modified to specify new menu options and classes to load that introduce new functionality. We first give a brief overview of the framework and then discuss Internal Frames, Resources, Menus and Toolbars, and finally Actions.

Every Java application contains a `main()` method. Like C++, the `main()` method is the primary entry point for the program. The `OntologyEditor` class contains the `main()` method in this application. This class is also the main window of the application and as such derives from JFrame. JFrame is a JFC class that provides a frame window with the ability to create child windows such as internal frames, menus, and toolbars. Besides being a container for other windows, this class also manages the loading of appropriate resources for creating menus and toolbars. It also loads and prepares "`Action`" classes, which are the primary means of redirecting user input to various parts of the program.

4.1.1.1 Internal Frames

In order to create an MDI environment, we created the `OntologyEditor` class from a `JFrame` and added to it the menus and toolbars. The client or desktop area is created from a `JDesktopPane`, which provides support for multiple child frame windows. All child frames are derivations of `OntologyInternalFrame`, which extends `JInternalFrame`. Each child frame is added to the instance of `JDesktopPane` assigned to the `OntologyEditor`. The `JDesktopPane` class handles many of the MDI requirements in regards to child frames, though we did have to implement our own cascading and tiling.

The primary internal frames of concern in this application are `OntologyFrame`, `DataFrame`, and `MacroFrame`. `OntologyFrame` is the child frame for the ORM Editor. `DataFrame` and `MacroFrame` are the child frames for the Data Frame editor and Macro and Lexicon editor respectively. We discuss the implementation for each later.

4.1.1.2 Resources

The `OntologyEditor` class uses a `ResourceBundle` to store all information needed to create menus and toolbars. The `ResourceBundle` in this case is data loaded from a file called `OntologyEditor.properties`. The following is the first part of the file as an example:

```
#
# Resource strings for OntologyEditor

Title=Ontology Editor

# list all actions
Actions=new open close save saveAs exit select object objectSet
      relationshipSet association note aggregation specialization
      constraint editDataFrame editMacro cascade tileHor tileVert
      delete selectAll about font alignLeft alignRight alignTop
      alignBottom alignHorCenter alignVertCenter moveFront moveBack
      spaceAcross spaceDown

MainToolbar=new open save - undo redo - font fillColor lineColor
      lineWidth - alignLeft alignRight alignTop alignBottom -
      alignHorCenter alignVertCenter - spaceAcross spaceDown -
      moveFront moveBack
ObjectToolbar=GROUP select object objectSet relationshipSet association
      note aggregation specialization constraint

menubar=file edit view format window help

file=new open close save saveAs - exit
fileLabel=File
fileHotKey=F

newLabel=New
newHotKey=N
newImage=images/new.gif
newAction=NewAction
newTooltip=New

openLabel=Open
openHotKey=O
openImage=images/open.gif
openAction=OpenAction
openTooltip=Open
...
```

Resource bundles work by assigning particular labels to data. For instance, the first line is "Title=Ontology Editor" which associates the keyword "Title" to the value "Ontology Editor". The `OntologyEditor` class knows to look for the keyword "Title" in the resource bundle to determine what to place as the caption on the main window. We could have hard coded this title, but by placing the data in a resource bundle we can easily change it without recompiling the program. In addition a resource file gives great flexibility for translating resources to other languages. Furthermore, Java supports the idea of locale specific resources that can be loaded by specifying the locale either at startup or while running. Using resource bundles provides the possibility of translating the resources for the OntologyEditor and providing locale support if it is ever desired.

4.1.1.3 Menus and Toolbars

Besides providing a location for storing resource strings, the `OntologyEditor.properties` file also controls what menu options are available and visible as well as specifying the class to invoke when a menu option is selected or a toolbar button is clicked. Notice the "Actions" keyword. After this keyword are listed all user interface actions that are supported. Following this list are the "MainToolbar", "ObjectToolbar", and "menubar" keywords. These keywords refer to two toolbars and the main menu respectively. The two toolbars show the action buttons and the menu shows the menu items to display. For example, the menu bar shows "file edit view format window help" which correspond to the menu items seen in the Ontology Editor menu of "File, Edit, View, Format, Window, and Help".

As an example we will look at the "file" action to gain an understanding of the information available:

```
file=new open close save saveAs - exit
fileLabel=File
fileHotKey=F
```

The "file" keyword shows "new open close save saveAs – exit". Recall that the "file" action was listed as being a part of the "menubar". The "file" keyword is likewise showing the child menu items which are displayed when the "File" menu option is selected as shown in Fig. 12.

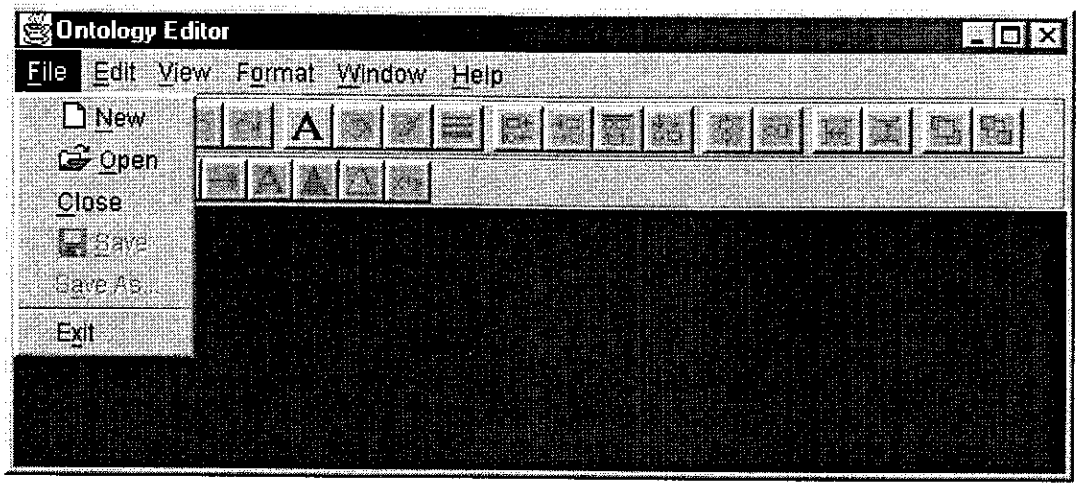


Figure 12: File Menu

Notice that the pull down menu contains menu items for "New, Open, Close, Save, Save As and Exit." The hyphen in the action list refers to a spacer in the case of toolbars and a separator line as seen here in the case of menus. Every action that has a label is designated by its keyword with the "Label" extension. In this example the keyword "fileLabel" associates the word "File" which will be the string displayed for that action. If a hotkey is desired it can be associated by appending "HotKey" to the action keyword. Notice that Fig. 12 shows "F" as being a hotkey for the "File" menu.

In order to see the other associations that are available we will now inspect the "new"action:

```
newLabel=New
newHotKey=N
newImage=images/new.gif
newAction=NewAction
newTooltip=New
```

The "new" action has a label and a hotkey associated with it like the "file" action did. Since there are not further submenus this action does not show a list of actions. An image can be associated with an action by appending "Image" to the keyword. In this case, it provides the path to the image to use. The image is displayed wherever such display is appropriate. In this case it is seen in the drop down "File" menu as well as on the toolbar. Appending "Tooltip" to the keyword can associate a tooltip that displays whenever the mouse cursor hovers over the button on a toolbar. "Action" is appended to a keyword to associate the Java class that will be used to handle all user interaction that causes this action to fire. If an

action class is not specified, then by default `OntosAction` is used. This leads us to our next discussion: Actions.

4.1.1.4 Actions

During initialization, the Ontology Editor instantiates an action handler for each action found in the "Actions" list. Every such class must be an instance of the class `OntosAction` or a derivation thereof. The `OntosAction` class extends `AbstractAction` overriding the `actionPerformed()` method. By default, every action is forwarded to the active child frame as shown in this code snippet:

```
public void actionPerformed(ActionEvent evt)
{
   JInternalFrame frame = editor.getActiveFrame();
   if (frame instanceof ActionListener)
      ((ActionListener)frame).actionPerformed(evt);
}
```

The new, save, saveAs, close, and exit actions as well as others all have specific derivations of `OntosAction` to handle their special requirements. The following shows the code for the "new" action's Action class:

```
class NewAction extends OntosAction
{
   public NewAction()
   {
   }

   public void actionPerformed(ActionEvent evt)
   {
      OntologyDocument doc = new OntologyDocument();
      if (doc.newDocument())
      {
         OntologyFrame frame = new OntologyFrame(doc, editor);
         frame.addInternalFrameListener(
            new FrameWindowListener(editor,frame));
         frame.setState(
            OntologyFrame.getStateByString(editor.objectState));
         editor.addFrame(frame);
         frame.toFront();
         frame.setVisible(true);

         try
         {
            frame.setSelected(true);
         }
```

```
            catch (PropertyVetoException e)
            {
            }
        }
    }
}
```

As can be seen, the derivations of `OntosAction` have a primary concern of overriding the `actionPerformed()` method to supply special functionality. In this case a new `OntologyDocument` is instantiated, and it is then associated with a new instance of an `OntologyFrame`. After creation the internal frame is added to the desktop and configured to be the topmost window.

The `OntosAction` class is a key to easing the integration of future extensions. An independent addition need only add the appropriate menu options to the `OntologyEditor.properties` file and create a derivation of `OntosAction` to handle creation of any new internal frame instances that are needed. Obviously, any new functionality that requires that a preexisting action be overloaded would require modifications to that action class. For example, we may want the new button to pull up a dialog to allow the user to choose what type of new document to create. Currently there is only knowledge of an `OntologyDocument`, but this could be extended in the future to support other document types. In this case, we could modify NewAction to display a dialog allowing the user to choose from document types that are supported.

4.1.2 ORM Editor

`OntologyFrame` is the internal frame that contains the ORM editor. It loads its data from the `OntologyDocument` that is assigned to it upon its creation. `OntologyDocument` handles persistence issues discussed later. `OntologyFrame` is actually only a container for the `OntologyCanvas`. The canvas is the drawing window where the majority of the work occurs in the ORM editor. `OntologyCanvas` extends `JComponent`. In JFC, every window control such as an edit control or combo box derives from `JComponent`. `JComponent` also extends the `Container` class and as such is

allowed to have child windows that are typically other JComponents. OntologyCanvas is a JComponent so that it can be the parent window to all ORM elements that are added to it.

All drawable objects in the ORM (see Fig. 8) derive from DrawObj. DrawObj itself derives from JComponent. As the base class for all drawable objects it defines key behavioral methods, including:

```
                DrawObj(OSMModel model)
int             getID()
void            setID(int id)
void            addNotify()
void            delete()
void            deleteSelected()
boolean         isSelected()
void            setSelected(boolean bVal)
void            selectAll(Rectangle rect)
void            unselectAll()
void            offset(int x, int y)
void            offsetSelected(int x, int y)
Point           getCenterPoint()
Point           getIntersectionPoint(Point pt)
OSMModel        getModel()
void            setModel(OSMModel model)
Rectangle       normalizeRect(Point p1, Point p2)
Canvas          getCanvas()
```

These methods are used by OntologyCanvas on all DrawObj's that it contains. Derivations of DrawObj sometimes need to override some of these methods to get behavior other than the default. For example, the offset() method is overridden in RelSet in order to call the offset() method of the Relation instances that participate in the relationship with it.

In order to expound upon the implementation of the design, we must now revisit the design in more detail. Fig. 11 shows DrawObj and its derived classes. Notice the EditableText class that derives from JTextArea, which is a multi-line edit control with some functionality for handling markup text such as HTML. When text is required within the drawing canvas an EditableText object is used. OntologyCanvas handles all drawing interactions from the user, therefore all DrawObj and EditableText instances forward interactions that they receive to the OntologyCanvas. In this way any requests for selecting or dragging objects can be uniformly handled. The one exception is when the

user is actually editing text within `EditableText`. In this case its base class `JTextArea` handles the editing.

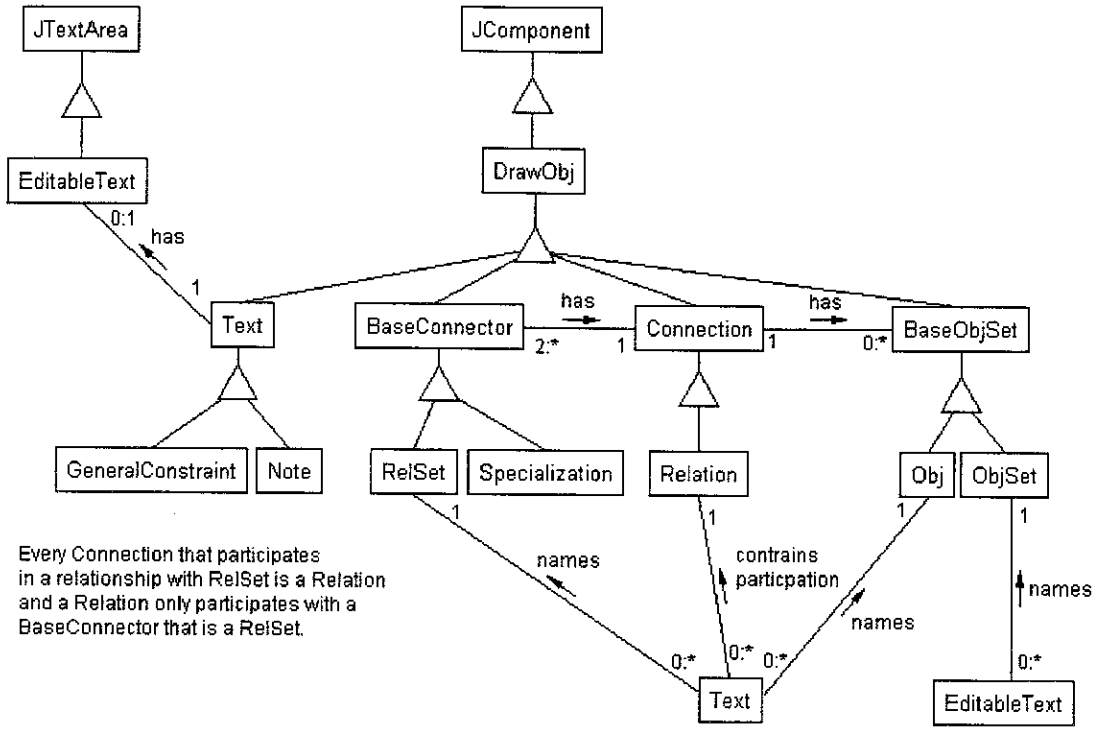


Figure 13: ORM Implementation

All `DrawObj` classes support common user interactions (via the `OntologyCanvas`). These interactions include selection, dragging, deletion, etc. The Text class is used whenever there is a requirement for draggable text. The Text class simple wraps the `EditableText` as an embedded component. The one case that the Text class does not wrap `EditableText` is with the name of the `ObjSet`. In this case the `ObjSet` itself wraps the `EditableText`.

Although `OntologyCanvas` handles all interaction with the user, the individual `DrawObj` instances handle their own particular functionality. For example, each derivation of `DrawObj` overrides the `paint()` method to allow specialized drawing.

Many derivations of `DrawObj` require some type of border. In the case of `ObjSet`, it is a rectangular border. Borders may need to be solid lines, dashed lines, or double lines. Instead of

implementing the necessary borders for each derivation, we created the `ObjBorder` class. The `ObjBorder` class extends the `AbstractBorder` class from JFC. It supports borders of various widths and colors as well as supporting a common way of drawing a border showing the object to be selected. Every `DrawObj` has an `ObjBorder`, and every derivation can configure the border with customized drawing options.

JDK 1.2 provided a graphics library much more powerful then previous JDKs. In particular it supports drawing lines with multiple widths and endcaps. It also supports the idea of creating drawable shapes that can then be translated, rotated or scaled. These are powerful graphical concepts, which came in very useful in the ORM editor. Besides drawing rectangles, the Ontology Editor requires the ability to draw more complex shapes such as diamonds, triangles, and arrows. While the diamonds and triangles are not too difficult to draw, the ability to draw and arrow and then rotate and translate was very useful for drawing relationship sets.

4.1.3 Data Frame Editor

The implementation of the Data Frame Editor required some modification to the original analysis and design. Fig. 12 shows an ORM with the major classes used in the implementation of data frames.

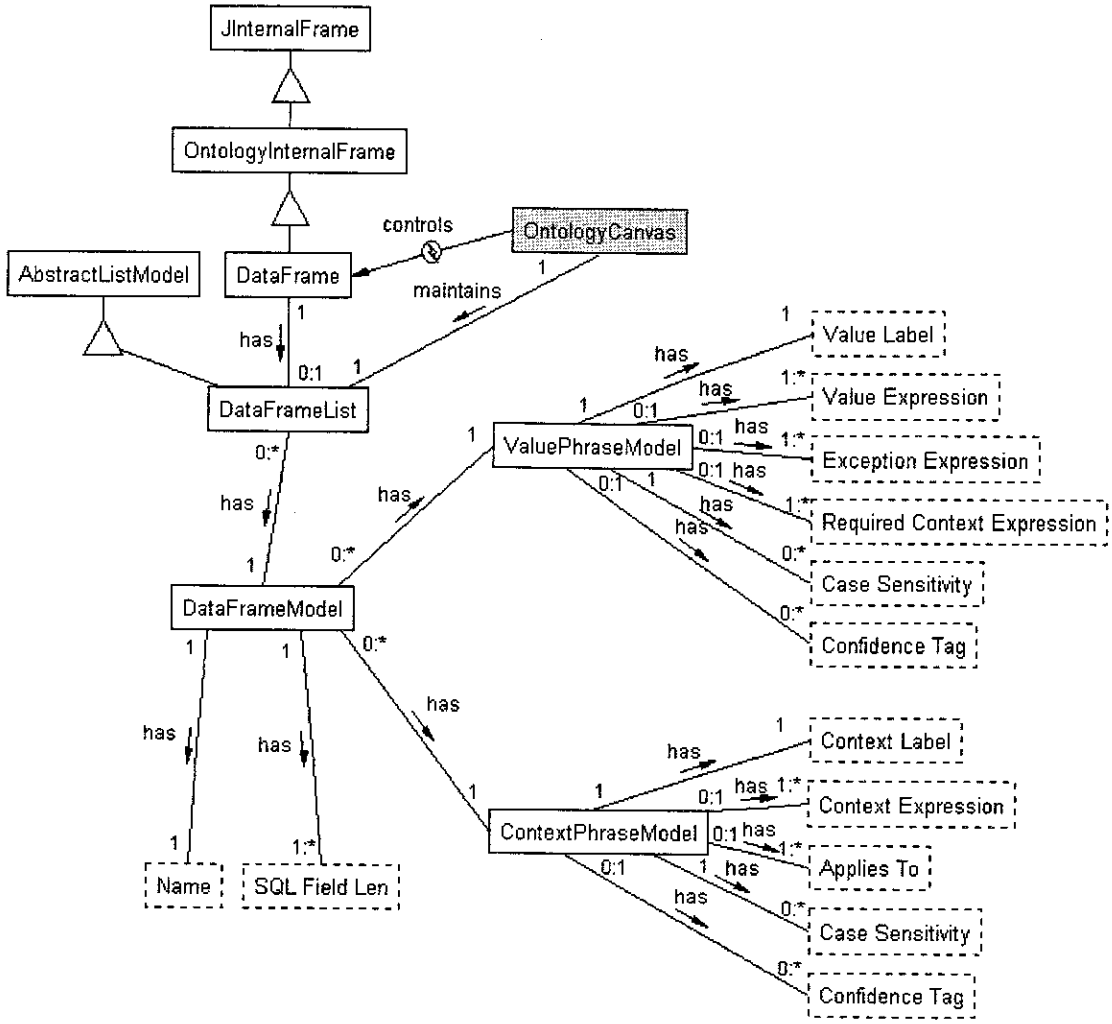


Figure 14: Data Frame Implementation

The `DataFrame` class derives from `OntologyInternalFrame`, which in turn derives from `JInternalFrame`. As a `JInternalFrame`, it is the child frame window inside the primary frame window of the Ontology Editor. Not shown in Fig. 12 are all the user interface controls that are used by `DataFrame`, but the layout is exactly as shown in Fig. 6 from the graphical design.

Java supports the concept of layout managers. In this case, we used a `GridBagLayout` to position all of the controls on the frame. The `GridBagLayout` is a powerful tool that allows for individual controls to be configured within a grid. Each control has preferred size, minimum size, and growth parameters that `GridBagLayout` uses to determine actual layout positions and sizes.

In the graphical design of the data frame editor, there is a drop down combobox that allows the user to select which `ObjSet` to edit. Rather than merely copying a list into the combobox, we created the `DataFrameList` class. This class derives from `AbstractListModel` and can be used by a combobox to specify items within the combobox. The `DataFrameList` is a collection of all of the `DataFrameModel` instances associated with `ObjSets` existing in the ORM. The combobox displays the name of each `DataFrameModel` in its drop down list. Using the `DataFrameList` also provides another advantage; if a user left the data frame editor open and returns to the ORM editor to add another `ObjSet`, then the combobox is immediately aware of the new addition.

The `DataFrameModel` class now becomes what was the Data Frame in the analysis (Fig. 3). As mentioned before, we use models to contain only data and not user interface controls. The only significant differences between Fig. 3 and Fig. 12 are that java classes were created for both `ValuePhraseModel` (previously Value Phrase) and the `ContextPhraseModel` (previously Context Phrase). Each of these classes provides accessor methods for the data they control. Not allowing direct access to member variables is a good programming practice and in this case very necessary, because the member variables actually do not exist and the accessor methods access an XML document to determine the information.

Before continuing it is worth noting here some of the underlying control that occurs within the `DataFrame` class. Many of the controls need to be disabled until their parent control has an appropriate entry created. Any changes made to the data displayed in the controls are immediately recognized and propagated to the appropriate model. In this manner the XML document is always kept up to date in memory and ready to be saved to disk.

4.1.4 Persistence

We chose to implement persistence through the use of XML. In the design phase we created a Document Type Definition (DTD). The DTD defines the valid format for defined XML documents. In order to implement XML, we searched for pre-existing tools and found a toolkit created by Sun Microsystems called Java Project X, Technology Release 2. This toolkit conforms to an API defined by the World Wide Web Consortium (W3C) for accessing XML documents.

Every instance of an `OntologyDocument` is associated with an `XmlDocument` from the XML toolkit. This `XmlDocument` provides access to all elements contained therein. Any changes made during editing of an ontology are immediately made to the `XmlDocument` held in memory which can then be saved upon request.

Many of the elements defined in the DTD are directly represented in code by java classes. In fact, the toolkit provides a class called an `ElementNode` that represents an XML element. All model classes used in the `OntologyEditor` either derive from `ElementNode` directly or from one of its derivations. In this manner, we were able to tie in the model classes directly with XML for persistence. But the classes that use the models only access methods specific to the individual model and not `ElementNode` with the exception of the `OntologyCanvas`. The `OntologyCanvas` is responsible for associating each model with its view class and therefore requires use of the `XmlDocument` and its `ElementNode` instances as it creates the associations. If the underlying persistence mechanism ever needs to change then those changes will be isolated to the `OntologyDocument`, `OntologyCanvas` and the model classes' implementations. Fig. 15 shows all of the models used in the OntologyEditor.

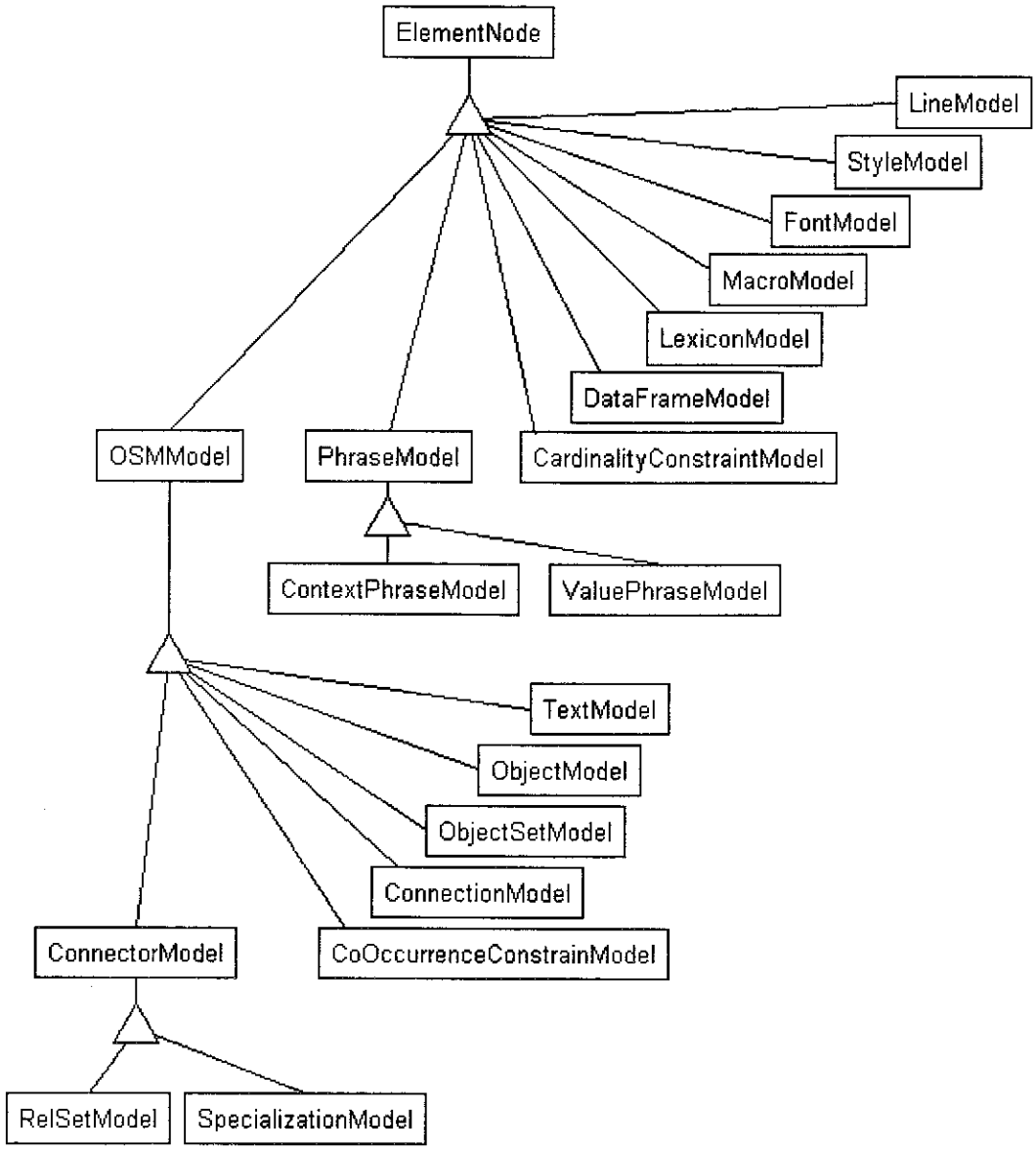


Figure 15: Model Implementation

4.1.5 Coding Statistics

There has been quite a bit of work involved in coding this project. Wherever possible we have tried to use existing code libraries such as the XML class libraries and OROMatcher. In addition we have strived to design our classes with reuse and maintenance in mind. The following are some of the coding statistics:

Lines of code: 11,470

Java Class Files: 70

Java Class Count: 110 (includes internal classes)

4.2 Javadoc

In order to enhance the extensibility and maintainability of the code it is useful to have code documentation. However as code changes and evolves outside documentation easily becomes out of date. Code that is documented within the source files can be updated as changes are made to the code. Of course this still takes some discipline on the programmer's part but a programmer is much more apt to maintain documentation if it is found with the code then otherwise.

The `javadoc` utility that comes with the JDK provides a very convenient way of creating html documentation for existing java source files. Without any effort on a programmer's part, `javadoc` can create a complete listing of all classes within a project and their API. In addition it provides a hierarchy tree of all classes within a project and their derivations.

Any documentation added to a class, method or member variable will be placed in the HTML documentation created by `javadoc` provided he follows the `javadoc` syntax. The following is the documented source code of the `EditableText` class.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.JComponent;
import javax.swing.JTextArea;
import javax.swing.JLabel;
import javax.swing.text.Highlighter;


/**
 * <code>EditableText</code> is a simple edit control derived
```

```
 * from <code>JTextArea</code>. It is used by all derivations
 * of <code>DrawObj</code> that require text to be displayed
 * and edited.
 * <BR><BR>
 * This control modifies the behavior of JTextArea by disabling
 * the control unless it is double-clicked.
 *
 * @author Kimball Hewett
 * @see DrawObj
 * @see ObjSet
 * @see Text
 */
public class EditableText
    extends       JTextArea
    implements    FocusListener, MouseListener, MouseMotionListener
{

      /**
       * This boolean member variable maintains the state of
       * whether or not the control is in edit mode.
       *
       * @see #isEditingText()
       */
      private boolean m_bEditing = false;
      private Color   m_selectColor;
      private Color   m_selectTextColor;

      /**
       * When the control loses focus this method will take
       * the control out of edit mode by calling editText
       * (false).
       *
       * @param evt
       * @see #editText(boolean)
       */
      public EditableText()
      {
          initialize();
      }

      /**
       * EditableText constructor
       *
       * This method like the default constructor calls
       * <code>initialize()</code> to
       * prepare the control for display.
       *
       * @param text The edit control is initialized with the String
       * <code>text</code>
       * @see #initialize()
       */
      public EditableText(String text)
      {
            super(text);
            initialize();
      }
```

```
/**
 * This method initializes the control for display.
 * It sets up the control as a <code>FocusListener</code>,
 * <code>MouseListener</code> and
 * <code>MouseMotionListener</code>.
 * <BR><BR>
 * By default the <code>JTextArea</code> control would
 * change the colors of the text when it is disabled.
 * This is inappropriate for the use in the
 * <code>OntologyEditor</code> so the initialize method
 * also sets the disabled colors to be the same as the
 * enabled colors.
 */
private void initialize()
{
      addFocusListener(this);
      addMouseListener(this);
      addMouseMotionListener(this);
      setDisabledTextColor(getForeground());
      m_selectColor = getSelectionColor();
      m_selectTextColor = getSelectedTextColor();
      setSelectionColor(getBackground());
      setSelectedTextColor(getForeground());
      setEnabled(false);
      setEditable(false);
}

/**
 * This method may be called to determine if the control is
 * currently in editing mode.
 *
 * @return true if editing
 */
public boolean isEditingText()
{
    return m_bEditing;
}

/**
 * This method is called to place the control in either
 * edit or non-edit mode. If in edit mode it will enable
 * editing and will display a caret in the control.
 * <BR><BR>
 * <code>editText(boolean)</code> is typically called when
 * double-clicked or when the control loses focus.
 *
 * @param bEdit bEdit is true if the control is to be placed
 * in edit mode.<BR>
 * bEdit is false if the control is to be placed in non-edit
 * mode.
 * @see #focusLost
 * @see #mouseClicked
 */
public void editText(boolean bEdit)
{
      m_bEditing = bEdit;
      setEnabled(m_bEditing);
```

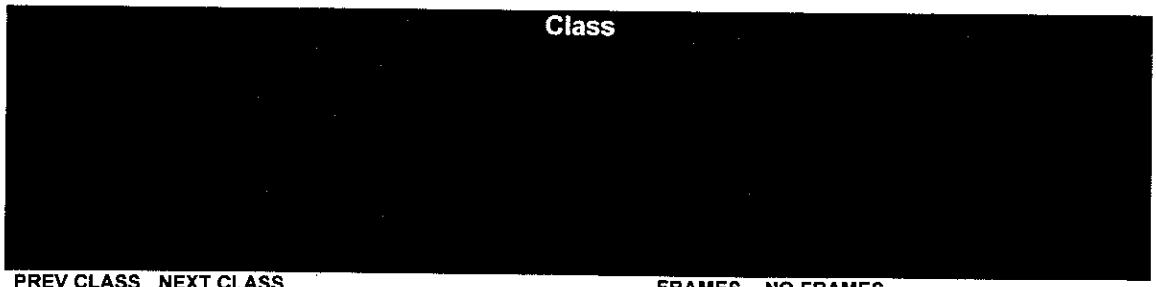
```
        setEditable(m_bEditing);
        if (m_bEditing)
        {
              requestFocus();
              setSelectionColor(m_selectColor);
              setSelectedTextColor(m_selectTextColor);
        }
        else  // text selected upon lost focus? unselect it
        {
            setSelectionColor(getBackground());
            setSelectedTextColor(getForeground());
            select(0,0);
        }
}


public void focusGained(FocusEvent evt)
{
}

/**
 * When the control loses focus this method will take the
 * control out of edit mode by calling
 * <code>editText(false)</code>.
 *
 * @see #editText
 */
public void focusLost(FocusEvent evt)
{
      editText(false);

      // Since containers will resize, let's leave something in
      // the control if it is empty
      if (getText().length() == 0)
          setText(" ");
}


/**
 * This method checks to see if the control has been
 * double-clicked. If so and the control is not in edit mode,
 * then the control is placed in edit mode by calling
 * <code>editText(true)</code>.
 *
 * @param evt evt provides the number of mouse clicks
 * @see #editText
 */
public void mouseClicked(MouseEvent evt)
{
      if (!m_bEditing && evt.getClickCount() > 1)
      {
            editText(true);
      }
}
public void mouseEntered(MouseEvent evt)
{
}
```

```
        public void mouseExited(MouseEvent evt)
        {
        }
        public void mousePressed(MouseEvent evt)
        {
        }
        public void mouseReleased(MouseEvent evt)
        {
        }
        public void mouseDragged(MouseEvent evt)
        {
        }
        public void mouseMoved(MouseEvent evt)
        {
        }
        public void setForeground(Color c)
        {
              super.setForeground(c);
              setDisabledTextColor(c);
        }
}
```

All code documentation that is entered by the programmer must adhere to the syntax required by `javadoc`. Basically it must precede the documented class, method or variable and be inside comments as shown above. Programmer documentation may use any valid HTML tags such as `<BOLD>` to cause bolding, but the use of header tags such as `<H1>` is discouraged because header tags are used by `javadoc` itself. Links within the documentation can be created through the use of special keywords such as `@see` as done in several places within the `EditableText` code.

Having looked at one of the source files to visualize the documented code, it is now useful to see the HTML documentation that can be created by javadoc. The HTML document corresponding to the EditableText class was created by running javadoc with the following options:

```
javadoc -private -use -version -author -splitindex -windowtitle
" Ontology Editor"  EditableText.java
```



Class EditableText

java.lang.Object
|
+-java.awt.Component
|
+-java.awt.Container
|
+-javax.swing.JComponent
|
+-javax.swing.text.JTextComponent
|
+-javax.swing.JTextArea
|
+-EditableText

public class **EditableText**
extends javax.swing.JTextArea
implements java.awt.event.FocusListener, java.awt.event.MouseListener, java.awt.event.MouseMotionListener

EditableText is a simple edit control derived from JTextArea. It is used by all derivations of DrawObj that require text to be displayed and edited.

This control modifies the behavior of JTextArea by disabling the control unless it is double-clicked.

Author:
Kimball Hewett

See Also:
DrawObj, ObjSet, Text, Serialized Form

javax.swing.JTextArea.AccessibleJTextArea

javax.swing.text.JTextComponent.AccessibleJTextComponent, javax.swing.text.JTextComponent.ClipboardObserver, javax.swing.text.JTextComponent.ComposedTextCaret, javax.swing.text.JTextComponent.DefaultKeymap, javax.swing.text.JTextComponent.FocusAction, javax.swing.text.JTextComponent.InputMethodRequestsHandler, javax.swing.text.JTextComponent.KeyBinding, javax.swing.text.JTextComponent.MutableCaretEvent

javax.swing.JComponent.AccessibleJComponent, javax.swing.JComponent.IntVector, javax.swing.JComponent.KeyboardBinding, javax.swing.JComponent.KeyboardState

java.awt.Component.AWTTreeLock

private boolean	**m_bEditing** This boolean member variable maintains the state of whether or not the control is in edit mode.
private java.awt.Color	**m_selectColor**
private java.awt.Color	**m_selectTextColor**

columns, columnWidth, rowHeight, rows, uiClassID, word, wrap

caret, caretColor, caretEvent, composedText, composedTextCaret, composedTextContent, composedTextEnd, composedTextStart, DEFAULT_KEYMAP, defaultBindings, defaultClipboardOwner, disabledTextColor, editable, editor, FOCUS_ACCELERATOR_KEY, focusAccelerator, focusAction, focusedComponent, highlighter, inputMethodRequestsHandler, keymap, keymapTable, margin, model, needToSendKeyTypedEvent, opaque, originalCaret, selectedTextColor, selectionColor

_bounds, accessibleContext, alignmentX, alignmentY, ANCESTOR_USING_BUFFER, ancestorNotifier, autoscroller, border, changeSupport, clientProperties, flags, HAS_FOCUS, IS_DOUBLE_BUFFERED, IS_OPAQUE, IS_PAINTING_TILE, KEYBOARD_BINDINGS_KEY, listenerList, maximumSize,

minimumSize, NEXT_FOCUS, paintImmediatelyClip, paintingChild, preferredSize, readObjectCallbacks, REQUEST_FOCUS_DISABLED, tmpRect, TOOL_TIP_TEXT_KEY, ui, uiClassID, UNDEFINED_CONDITION, vetoableChangeSupport, WHEN_ANCESTOR_OF_FOCUSED_COMPONENT, WHEN_FOCUSED, WHEN_IN_FOCUSED_WINDOW

component, containerListener, containerSerializedDataVersion, dispatcher, layoutMgr, maxSize, ncomponents, serialVersionUID

actionListenerK, adjustmentListenerK, appContext, assert, background, BOTTOM_ALIGNMENT, CENTER_ALIGNMENT, changeSupport, componentListener, componentListenerK, componentOrientation, componentSerializedDataVersion, containerListenerK, cursor, dropTarget, enabled, eventMask, focusListener, focusListenerK, font, foreground, hasFocus, height, incRate, inputMethodListener, inputMethodListenerK, isInc, isPacked, itemListenerK, keyListener, keyListenerK, LEFT_ALIGNMENT, locale, LOCK, minSize, mouseListener, mouseListenerK, mouseMotionListener, mouseMotionListenerK, name, nameExplicitlySet, newEventsOnly, ownedWindowK, parent, peer, peerFont, popups, prefSize, RIGHT_ALIGNMENT, serialVersionUID, textListenerK, TOP_ALIGNMENT, valid, visible, width, windowListenerK, x, y

EditableText() When the control loses focus this method will take the control out of edit mode by calling editText (false).
EditableText(java.lang.String text) EditableText constructor This method like the default constructor calls initialize() to prepare the control for display.

void	**editText**(boolean bEdit) This method is called to place the control in either edit or non-edit mode.
void	**focusGained**(java.awt.event.FocusEvent evt)
void	**focusLost**(java.awt.event.FocusEvent evt) When the control loses focus this method will take the control out of edit mode by calling editText(false).
private void	**initialize**() This method initializes the control for display.
boolean	**isEditingText**() This method may be called to determine if the control is currently in editing mode.

void	**mouseClicked**(java.awt.event.MouseEvent evt) This method checks to see if the control has been double-clicked.
void	**mouseDragged**(java.awt.event.MouseEvent evt)
void	**mouseEntered**(java.awt.event.MouseEvent evt)
void	**mouseExited**(java.awt.event.MouseEvent evt)
void	**mouseMoved**(java.awt.event.MouseEvent evt)
void	**mousePressed**(java.awt.event.MouseEvent evt)
void	**mouseReleased**(java.awt.event.MouseEvent evt)
void	**setForeground**(java.awt.Color c)

append, createDefaultModel, getAccessibleContext, getColumns, getColumnWidth, getLineCount, getLineEndOffset, getLineOfOffset, getLineStartOffset, getLineWrap, getPreferredScrollableViewportSize, getPreferredSize, getRowHeight, getRows, getScrollableTracksViewportWidth, getScrollableUnitIncrement, getTabSize, getUIClassID, getWrapStyleWord, insert, isManagingFocus, paramString, processComponentKeyEvent, replaceRange, setColumns, setFont, setLineWrap, setRows, setTabSize, setWrapStyleWord, writeObject

, addCaretListener, addInputMethodListener, addKeymap, copy, createComposedString, cut, exchangeCaret, fireCaretUpdate, getActions, getCaret, getCaretColor, getCaretPosition, getDisabledTextColor, getDocument, getFocusAccelerator, getFocusedComponent, getHighlighter, getInputMethodRequests, getKeymap, getKeymap, getMargin, getScrollableBlockIncrement, getScrollableTracksViewportHeight, getSelectedText, getSelectedTextColor, getSelectionColor, getSelectionEnd, getSelectionEnd, getSelectionStart, getSelectionStart, getText, getText, getUI, isEditable, isFocusTraversable, isOpaque, isProcessInputMethodEventOverridden, loadKeymap, mapCommittedTextToAction, mapEventToAction, modelToView, moveCaretPosition, paste, processInputMethodEvent, read, readObject, removeCaretListener, removeKeymap, removeNotify, replaceInputMethodText, replaceSelection, select, selectAll, setCaret, setCaretColor, setCaretPosition, setDisabledTextColor, setDocument, setEditable, setEnabled, setFocusAccelerator, setHighlighter, setInputMethodCaretPosition, setKeymap, setMargin, setOpaque, setSelectedTextColor, setSelectionColor, setSelectionEnd, setSelectionStart, setText, setUI, updateUI, viewToModel, write

_paintImmediately, addAncestorListener, addNotify, addPropertyChangeListener, addPropertyChangeListener, addVetoableChangeListener, adjustPaintFlags, alwaysOnTop, bindingForKeyStroke, checkIfChildObscuredBySibling, computeVisibleRect, computeVisibleRect, contains, createToolTip, enableSerialization, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, fireVetoableChange, getActionForKeyStroke, getAlignmentX, getAlignmentY, getAutoscrolls, getBorder, getBounds, getClientProperties, getClientProperty, getComponentGraphics, getConditionForKeyStroke, getDebugGraphicsOptions, getFlag, getGraphics, getHeight, getInsets, getInsets, getLocation, getMaximumSize, getMinimumSize, getNextFocusableComponent, getRegisteredKeyStrokes, getRootPane, getSize, getToolTipLocation, getToolTipText, getToolTipText, getTopLevelAncestor, getVisibleRect, getWidth, getX, getY, grabFocus, hasFocus, isDoubleBuffered, isFocusCycleRoot, isLightweightComponent, isOptimizedDrawingEnabled, isPaintingTile, isRequestFocusEnabled, isValidateRoot, keyboardBindings, paint, paintBorder, paintChildren, paintComponent, paintImmediately, paintImmediately, paintWithBuffer, processFocusEvent, processKeyBinding, processKeyBindings, processKeyBindingsForAllComponents, processKeyEvent, processMouseMotionEvent, putClientProperty, rectangleIsObscured, rectangleIsObscuredBySibling, registerKeyboardAction, registerKeyboardAction, registerWithKeyboardManager, removeAncestorListener, removePropertyChangeListener, removePropertyChangeListener, removeVetoableChangeListener, repaint, repaint, requestDefaultFocus, requestFocus, resetKeyboardActions, reshape, revalidate, scrollRectToVisible, setAlignmentX, setAlignmentY, setAutoscrolls, setBackground, setBorder, setDebugGraphicsOptions, setDoubleBuffered, setFlag, setMaximumSize, setMinimumSize, setNextFocusableComponent, setPaintingChild, setPreferredSize, setRequestFocusEnabled, setToolTipText, setUI, setVisible, shouldDebugGraphics, superProcessMouseMotionEvent, unregisterKeyboardAction, unregisterWithKeyboardManager, update

add, add, add, add, add, addContainerListener, addImpl, applyOrientation, countComponents, deliverEvent, dispatchEventImpl, dispatchEventToSelf, doLayout, eventEnabled, findComponentAt, findComponentAt, getComponent, getComponentAt, getComponentAt, getComponentCount, getComponents_NoClientCode, getComponents, getCursorTarget, getLayout, getMouseEventTarget, getWindow, initIDs, insets, invalidate, invalidateTree, isAncestorOf, layout, lightweightPrint, list, list, locate, minimumSize, nextFocus, paintComponents, postProcessKeyEvent, postsOldMouseEvents, preferredSize, preProcessKeyEvent, print, printComponents, printOneComponent, processContainerEvent, processEvent, proxyEnableEvents, proxyRequestFocus, remove, remove, removeAll, removeContainerListener, setCursor, setFocusOwner, setLayout, transferFocus, updateCursor, validate, validateTree

action, add, addComponentListener, addFocusListener, addKeyListener, addMouseListener, addMouseMotionListener, areInputMethodsEnabled, bounds, checkImage, checkImage, coalesceEvents, constructComponentName, contains, createImage, createImage, disable, disableEvents, dispatchEvent, enable, enable, enableEvents, enableInputMethods, getBackground, getBounds, getColorModel, getComponentOrientation, getCursor, getDropTarget, getFont_NoClientCode, getFont, getFontMetrics, getForeground, getInputContext, getIntrinsicCursor, getLocale, getLocation, getLocationOnScreen, getName, getNativeContainer, getParent_NoClientCode, getParent, getPeer, getSize, getToolkit, getToolkitImpl, getTreeLock, getWindowForObject, gotFocus, handleEvent, hide, imageUpdate, inside, isDisplayable, isEnabled, isEnabledImpl, isLightweight, isShowing, isValid, isVisible, keyDown, keyUp, list, list, list, location, lostFocus, mouseDown, mouseDrag, mouseEnter, mouseExit, mouseMove, mouseUp, move, nextFocus, paintAll, postEvent, prepareImage, prepareImage, printAll, processComponentEvent, processMouseEvent, remove, removeComponentListener, removeFocusListener, removeInputMethodListener, removeKeyListener, removeMouseListener, removeMouseMotionListener, repaint, repaint, repaint, resize, resize, setBounds, setBounds, setComponentOrientation, setDropTarget, setLocale, setLocation, setLocation, setName, setSize, setSize, show, show, size, toString, transferFocus

clone, equals, finalize, getClass, hashCode, notify, notifyAll, registerNatives, wait, wait, wait

m_bEditing

private boolean m_bEditing

This boolean member variable maintains the state of whether or not the control is in edit mode.

See Also:

isEditingText()

---

m_selectColor

private java.awt.Color m_selectColor

---

m_selectTextColor

private java.awt.Color m_selectTextColor

EditableText

public EditableText()

When the control loses focus this method will take the control out of edit mode by calling editText (false).

Parameters:

evt -

See Also:

editText(boolean)

EditableText

public EditableText(java.lang.String text)

EditableText constructor This method like the default constructor calls initialize() to prepare the control for display.

Parameters:

text - The edit control is initialized with the String text

See Also:

initialize()

---

initialize

private void initialize()

This method initializes the control for display. It sets up the control as a FocusListener, MouseListener and MouseMotionListener.

By default the JTextArea control would change the colors of the text when it is disabled. This is inappropriate for the use in the OntologyEditor so the initialize method also sets the disabled colors to be the same as the enabled colors.

---

isEditingText

public boolean isEditingText()

This method may be called to determine if the control is currently in editing mode.

Returns:

true if editing

---

editText

public void editText(boolean bEdit)

This method is called to place the control in either edit or non-edit mode. If in edit mode it will enable editing and will display a caret in the control.

editText(boolean) is typically called when double-clicked or when the control loses focus.

Parameters:

bEdit - bEdit is true if the control is to be placed in edit mode.
bEdit is false if the control is to be placed in non-edit mode.
See Also:
focusLost(java.awt.event.FocusEvent), mouseClicked(java.awt.event.MouseEvent)

---

focusGained

public void focusGained(java.awt.event.FocusEvent evt)

Specified by:
focusGained in interface java.awt.event.FocusListener

---

focusLost

public void focusLost(java.awt.event.FocusEvent evt)

When the control loses focus this method will take the control out of edit mode by calling editText(false).
Specified by:
focusLost in interface java.awt.event.FocusListener
See Also:
editText(boolean)

---

mouseClicked

public void mouseClicked(java.awt.event.MouseEvent evt)

This method checks to see if the control has been double-clicked. If so and the control is not in edit mode, then the control is placed in edit mode by calling editText(true).
Specified by:
mouseClicked in interface java.awt.event.MouseListener
Parameters:
evt - evt provides the number of mouse clicks
See Also:
editText(boolean)

---

mouseEntered

public void mouseEntered(java.awt.event.MouseEvent evt)

Specified by:
mouseEntered in interface java.awt.event.MouseListener

---

mouseExited

public void mouseExited(java.awt.event.MouseEvent evt)

Specified by:
mouseExited in interface java.awt.event.MouseListener

---

mousePressed

public void mousePressed(java.awt.event.MouseEvent evt)

Specified by:
mousePressed in interface java.awt.event.MouseListener

---

mouseReleased

public void mouseReleased(java.awt.event.MouseEvent evt)

Specified by:
mouseReleased in interface java.awt.event.MouseListener

---

mouseDragged

public void mouseDragged(java.awt.event.MouseEvent evt)

Specified by:
mouseDragged in interface java.awt.event.MouseMotionListener

---

mouseMoved

public void mouseMoved(java.awt.event.MouseEvent evt)

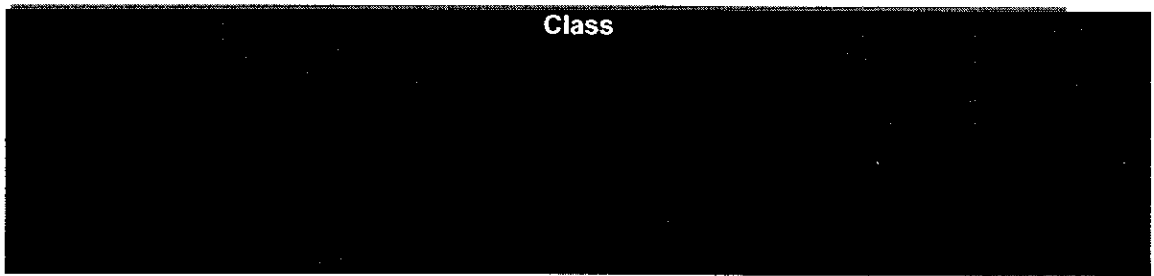
Specified by:
mouseMoved in interface java.awt.event.MouseMotionListener

---

setForeground

public void setForeground(java.awt.Color c)

Overrides:
setForeground in class javax.swing.JComponent



As can be readily seen, the documentation that can be created by `javadoc` is quite extensive. Even without programmer comments in the code the tool provides quick access for viewing the API of each class. With the programmer comments it becomes even more powerful. The major classes of the Ontology Editor have had programmer comments added to their methods in order to facilitate understanding the code by future programmers who will extend and maintain the code.

Prior to leaving this section it will be useful to show the tree of classes that javadoc produces. We have already found this HTML page very useful when explaining the hierarchy of classes within the Ontology Editor. Its benefits lie in not only listing each class but especially in the derivation hierarchy.

PREV NEXT FRAMES NO FRAMES

Hierarchy For All Packages

Class Hierarchy

- class java.lang.Object
  - class **AboutDialog.Hyperactive** (implements javax.swing.event.HyperlinkListener)
  - class **AboutDialog.SymAction** (implements java.awt.event.ActionListener)
  - class javax.swing.AbstractAction (implements javax.swing.Action, java.lang.Cloneable, java.io.Serializable)
    - class **OntosAction**
      - class **AlignBottomAction**
      - class **AlignHorCenterAction**
      - class **AlignLeftAction**
      - class **AlignRightAction**
      - class **AlignTopAction**
      - class **AlignVertCenterAction**

        - class **CloseAction**
        - class **DataFrameAction**
        - class **ExitAction**
        - class **FontAction**
        - class **HelpAboutAction**
        - class **MacroEditorAction**
        - class **MoveBackAction**
        - class **MoveFrontAction**
        - class **NewAction**
        - class **OpenAction**
        - class **SaveAction**
        - class **SaveAsAction**
        - class **SpaceAcrossAction**
        - class **SpaceDownAction**
        - class **StateAction**
        - class **WindowCascadeAction**
        - class **WindowFrameAction**
        - class **WindowTileHorAction**
        - class **WindowTileVertAction**
- class javax.swing.border.AbstractBorder (implements javax.swing.border.Border, java.io.Serializable)
    - class **ObjBorder**
- class java.util.AbstractCollection (implements java.util.Collection)
    - class java.util.AbstractList (implements java.util.List)
        - class java.util.Vector (implements java.lang.Cloneable, java.util.List, java.io.Serializable)
            - class **ExpressionVector**
- class javax.swing.AbstractListModel (implements javax.swing.ListModel, java.io.Serializable)
    - class **DataFrameList** (implements javax.swing.ComboBoxModel)
- class **Arrow** (implements java.awt.Shape)
- class **ColorIcon** (implements javax.swing.Icon)

- class java.awt.Component (implements java.awt.image.ImageObserver, java.awt.MenuContainer, java.io.Serializable)
    - class java.awt.Container
        - class javax.swing.JComponent (implements java.io.Serializable)
            - class javax.swing.AbstractButton (implements java.awt.ItemSelectable, javax.swing.SwingConstants)
                - class javax.swing.JButton (implements javax.accessibility.Accessible)
                    - class **ColorButton**
            - class **CoOccurrenceConstraint.ArrowComponent**
            - class **DrawObj**
                - class **AlignableObj**
                - class **BaseConnector**
                    - class **Aggregation**
                    - class **Association**
                    - class **Conjunction**
                    - class **Interaction**
                    - class **RelSet**
                    - class **Specialization**
                - class **BaseObjSet**
                    - class **Obj**
                    - class **ObjSet** (implements javax.swing.event.DocumentListener)
                    - class **PlaceHolder**
                    - class **State**
                    - class **Transition**
                - class **Connection**
                    - class **Relation**
                - class **CoOccurrenceConstraint** (implements javax.swing.event.DocumentListener)
                - class **Text** (implements javax.swing.event.DocumentListener)
                    - class **GeneralConstraint**

- class **Note**

- class **GridSpacer**

- class javax.swing.JFileChooser (implements javax.accessibility.Accessible)

  - class **ExtFileChooser**

- class javax.swing.JInternalFrame (implements javax.accessibility.Accessible, javax.swing.RootPaneContainer, javax.swing.WindowConstants)

  - class **OntologyInternalFrame** (implements javax.swing.event.InternalFrameListener)

    - class **DataFrame** (implements java.awt.event.ActionListener, java.awt.event.FocusListener, javax.swing.event.InternalFrameListener, java.awt.event.ItemListener)

    - class **MacroFrame** (implements java.awt.event.ActionListener, java.awt.event.FocusListener, javax.swing.event.InternalFrameListener, java.awt.event.ItemListener)

    - class **OntologyFrame** (implements java.awt.event.ActionListener)

- class javax.swing.text.JTextComponent (implements javax.accessibility.Accessible, javax.swing.Scrollable)

  - class javax.swing.JTextArea

    - class **EditableText** (implements java.awt.event.FocusListener, java.awt.event.MouseListener, java.awt.event.MouseMotionListener)

- class **MacroFrame.Spacer**

- class **OntologyCanvas** (implements java.awt.event.ActionListener, Canvas, java.awt.event.KeyListener, java.awt.event.MouseListener, java.awt.event.MouseMotionListener)

- class **PrimaryMarker**

- class java.awt.Window

  - class java.awt.Dialog

    - class javax.swing.JDialog (implements javax.accessibility.Accessible,

javax.swing.RootPaneContainer,
javax.swing.WindowConstants)

- class **AboutDialog**
- class **FontDialog**

- class java.awt.Frame (implements java.awt.MenuContainer)
  - class javax.swing.JFrame (implements javax.accessibility.Accessible, javax.swing.RootPaneContainer, javax.swing.WindowConstants)
    - class **OntologyEditor**

- class **ExpressionVector.Expression**
- class javax.swing.filechooser.FileFilter
  - class **ExampleFileFilter**
    - class **OSMFileFilter**
- class **FontDialog.SymAction** (implements java.awt.event.ActionListener)
- class **FontDialog.SymItem** (implements java.awt.event.ItemListener)
- class **FrameInfo**
- class **IDGenerator**
- class javax.swing.event.InternalFrameAdapter (implements javax.swing.event.InternalFrameListener)
  - class **FrameWindowListener**
- class java.awt.event.KeyAdapter (implements java.awt.event.KeyListener)
  - class **FontDialog.SymKey**
- class **MatchMarker**
- class com.sun.xml.tree.NodeBase (implements org.w3c.dom.Node, com.sun.xml.tree.NodeEx, org.w3c.dom.NodeList, com.sun.xml.tree.XmlWritable)
  - class com.sun.xml.tree.ParentNode (implements com.sun.xml.tree.XmlReadable)
    - class com.sun.xml.tree.ElementNode (implements com.sun.xml.tree.ElementEx)
      - class **CardinalityConstraintModel**
      - class **DataFrameModel**
      - class **FontModel**
      - class **LexiconModel** (implements LabelModel)

- class **LineModel**
- class **MacroModel** (implements LabelModel)
- class **OSMModel**
  - class **ConnectionModel**
  - class **ConnectorModel**
    - class **AssociationModel**
    - class **RelSetModel**
    - class **SpecializationModel**
  - class **CoOccurrenceConstraintModel**
  - class **ObjectModel**
  - class **ObjectSetModel**
  - class **TextModel**
- class **PhraseModel** (implements LabelModel)
  - class **ContextPhraseModel**
  - class **ValuePhraseModel**
- class **StyleModel**

- class **OntologyDocument**
- class **OntologyDocument.OntologyDocError** (implements org.xml.sax.ErrorHandler)
- class **OntologyEditor.ActionChangedListener** (implements java.beans.PropertyChangeListener)
- class **OntologyPopupMenu**
- class **PhraseEditor** (implements javax.swing.ComboBoxEditor, javax.swing.event.DocumentListener)
- class **Style**
- class java.awt.event.WindowAdapter (implements java.awt.event.WindowListener)
  - class **FontDialog.ThisDialogAdapter**
  - class **WindowCloser**

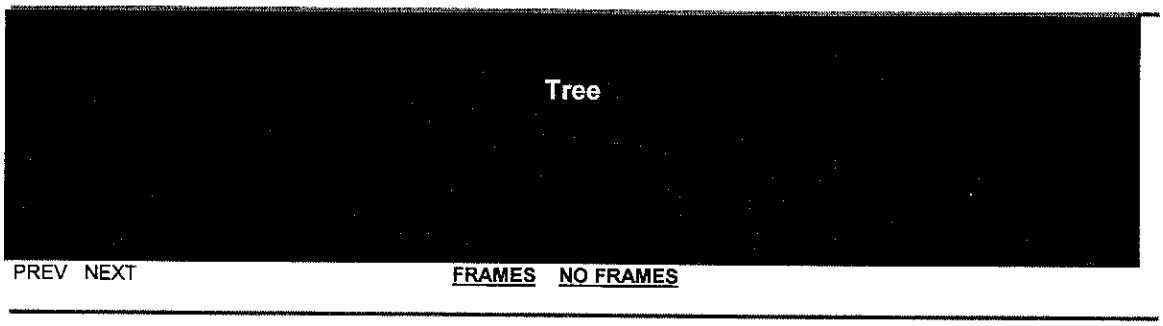
Interface Hierarchy

- interface java.util.EventListener
  - interface java.awt.event.KeyListener

- interface **Canvas**(also extends java.awt.event.MouseListener, java.awt.event.MouseMotionListener)

- interface java.awt.event.MouseListener
  - interface **Canvas**(also extends java.awt.event.KeyListener, java.awt.event.MouseMotionListener)
- interface java.awt.event.MouseMotionListener
  - interface **Canvas**(also extends java.awt.event.KeyListener, java.awt.event.MouseListener)

- interface **LabelModel**



The hierarchy tree is a convenient way of browsing the classes. Every local class has a link that can be clicked to pull up the documentation for that particular class.

The use of `javadoc` provides an extremely valuable tool for understanding the API and hierarchy of the classes in Ontology Editor. Those tasked with extending or maintaining the Ontology Editor in the future should find their job of understanding the existing code easier with the help of the documentation created by `javadoc`.

5 CONCLUSION

In conclusion, we should revisit the goals set forth for the Ontology Editor to evaluate our success. Those goals were portability, extensibility, maintainability, and the features of the ORM editor, data frame editor, and text viewer.

5.1 Portability

We chose to write the Ontology Editor in Java so that we could leverage the ability of Java to run on multiple platforms. The Ontology Editor has been successfully run on Microsoft Windows 95, Windows NT, Linux and Solaris operating systems. By implementing the project in a programming language that provides portable user interface controls, we were able to meet the goal of portability. The Ontology Editor should run fine on any operating system for which JDK 1.2 or higher has been provided.

5.2 Extensibility

Throughout the project we have strived to design and implement in such a manner as to ease the job of extending the project in the future. Specifically, several areas that enhance the extensibility of the Ontology Editor are:

- The method used for creating menu and toolbar items which allows for new menus items and toolbar buttons with a simple entry into the OntologyEditor.properties file. The entry specifies the Action class that can be implemented to handle the new functionality.
- The DrawObj class and its immediate specializations provide a framework for creating new drawable objects.
- The Model/View method of separating data structures from their user interface will allow the reuse of the data structures when integrating other data extraction processes into the Ontology Editor.

Two items from the Object Relationship Model were not implemented initially because they are not specifically needed for ontologies. They are association and aggregation. In order to see how easily the project could be extended, we had David Lewis implement the association relationship. He was able to

create a new derivation of DrawObj and integrate it into the ORM editor within a few hours. Most of which time he said was spent on creating the `paint()` method for drawing the new drawable object.

All of the above helps substantiate the claim that the Ontology Editor has been made extensible and that future development work on this project will benefit from the groundwork that has been laid.

5.3 Maintainability

Maintainability has been a primary goal of this project. We have accomplished this goal by the following actions:

- Striving for readability in the code
    - Descriptive naming of classes, methods and variables
    - Commenting code where appropriate
    - Resisting overly long methods
- Creation of an elegant design that can be readily followed
- Providing coding documentation through the use of `javadoc`

Towards the end of the project, David Lewis was assigned to help with the final coding stages. He was able to come into the project on the tail end and quickly come up to speed on the design and structure of the classes. His ability to become productive so quickly is evidence of the maintainability of the project.

5.4 Features

The project was composed of three main areas: An ORM Editor, a data frame editor and a text viewer for debugging. Each area has been completed according to the requirements analysis.

In conclusion, this project represents a key tool in the data extraction process. The Integrated Ontology Development Environment now provides support for editing ontologies and dataframes, and the ability to debug the data frames. As an integrated tool, this project now allows for the graphical creation of application ontologies without a detailed knowledge of its underlying textual syntax. The Integrated Ontology Development Environment, together with the other projects that are providing tools for the data extraction model can be a great asset for accessing the unstructured data residing on the WWW today.

BIBLIOGRAPHY

[C98] Carter, Eric. Allegro. http://osm7.cs.byu.edu/~eric/allegro.html.

[CJB99] Chandrasekaran, B., John R. Josephson, and Richard Benjamins. "What Are Ontologies, and Why Do We Need Them?" IEEE Intelligent Systems (January/February 1999): 20-26.

[E+98] Embley, David W., et al. "A Conceptual-Modeling Approach to Extracting Data from the Web," Proceedings of the 17th International Conference on Conceptual Modeling (November 1998): 78-91.

[E+99] Embley, David W. et al. "Conceptual-Model-Based Data Extraction from Multiple-Record Web Pages," Data and Knowledge Engineering (November 1999).

[ECLS98] Embley, David W., Douglas M. Campbell, Stephen W. Liddle, and Randy D. Smith. "Ontology-Based Extraction and Structuring of Information from Data-Rich Unstructured Documents," Proceedings of the Conference on Information and Knowledge Management (November 1998): 52-59.

[EJN99] Embley, David W. S. Jiang and Y.K. Ng. "Record-boundary discovery in Web documents," Proceedings 1999 ACM SIGMOD International Conference on Management of Data (1999).

[EKW92] Embley, David W., Barry D. Kurtz, Scott N. Woodfield. Object-Oriented Systems Analysis: A Model-Driven Approach. New Jersey: Yourdon Press, 1992.

[Embley80] Embley, David W. "Programming with data frames for everyday data items," National Computer Conference (1980): 301-305.

[Embley98] Embley, David W. Object Database Development: Concepts and Principles. Massachusetts: Addison Wesley Longman Inc., 1998.

[FF99] Fikes, Richard, and Adam Farquhar. "Distributed Repositories of Highly Expressive Reusable Ontologies," IEEE Intelligent Systems (March/April 1999): 73-79.

[FFF99] Frank, Gleb, Adam Farquhar, and Richard Fikes. "Building a Large Knowledge Base from a Structured Source," IEEE Intelligent Systems (January/February 1999): 47-54.

[JOE99] Java Ontology Editor. http://www.engr.sc.edu/research/CIT/demos/java/joe/joeBeta.html.

[KPE82] Khan, Shams A., Michael R. Paige, and David W. Embley. "Reading Data Items without Constraints on Form, Format or Completeness," Proceedings: Advances in Information Technology (May 27, 1982): 74-82.

[L98] Liddle, Stephen W. Pattern Editor. http://www.deg.byu.edu.

[LGSS99] López, Mariano F., Asunción Gómez-Pérez, Juan Pazos Sierra, and Alejandro Pazos Sierra. "Building a Chemical Ontology Using Methontology and the Ontology Design Environment," IEEE Intelligent Systems (January/February 1999): 37-46.

[ORO99] ORO Matcher. http://www.oroinc.com.

[OSM] Object-Oriented Systems Modeling Laboratory. http://osm7.cs.byu.edu.

[ST99] Studer, Rudi, V. Richard Benjamins, and Dieter Fensel. "Knowledge Engineering: Principles and Methods," Data & Knowledge Engineering 25 (1998): 161-197.

[VRMS99] Valente, Andre, Thomas Russ, Robert MacGregor and William Swartout. "Building and (Re)Using an Ontology of Air Campaign Planning," IEEE Intelligent Systems (January/February 1999): 27-36.