Project Report
A Simple Knowledge Organisation Tool

Mark Jordan
Supervisor: Robert Stevens
Program: BSc Computer Science

May 4, 2011

A Third Year Project Report
for the School of Computer Science at the University of Manchester

A Simple Knowledge Organisation Tool
Mark Jordan
Supervisor: Robert Stevens

Abstract

In Knowledge Representation, a field interested in storing and reasoning about information contained in ontologies, there is good support for the final stages of ontology creation, such as adding logical definitions and finding inferences, but little support for the early creation stages, such as deciding on the structure of the ontology.

In this report the creation of a Simple Knowledge Organisation Tool (SKOT) is explained. SKOT allows the user to 'sketch out' ontologies in a way that is as streamlined for the user as possible, without the 'heavyweight' features of more complex tools getting in the way. For future development, SKOT is extensible for taxonomies, terminologies or other knowledge models.

SKOT is currently available from [1].

Acknowledgements

Many thanks to Robert Stevens, my supervisor, whose guidance and advice was invaluable;

Thanks also to my family, for their love and support;

And finally to Bijan Parsia and Sean Bechhofer for giving an excellent KR course.

Contents

1 Introduction **4**
- 1.1 Project Planning . . . 5

2 Background and Literature Survey **6**
- 2.1 Overview of KR methods . . . 6
- 2.2 Semantic Web . . . 7
- 2.3 OWL . . . 7
- 2.4 Existing Methods . . . 9
  - 2.4.1 Protégé . . . 9
  - 2.4.2 Freemind . . . 10
  - 2.4.3 Knowledge Elicitation . . . 11

3 Design and Implementation **12**
- 3.1 Design Overview . . . 12
- 3.2 Design Specifics . . . 15
  - 3.2.1 MessageHandler . . . 15
  - 3.2.2 DiagramUI and Swing custom classes . . . 15
- 3.3 Design Implementation and Changes . . . 16
  - 3.3.1 Design Changes from Testing and User Feedback . . . 16
  - 3.3.2 Future Extensions . . . 17
- 3.4 Design Considerations and Limitations . . . 18

4 Results: An Example Workflow **20**

5 Testing and Evaluation **27**
- 5.1 Structured Testing . . . 27
  - 5.1.1 Performance Evaluation . . . 28
- 5.2 Heuristic Evaluation . . . 28
- 5.3 User Evaluation . . . 31
  - 5.3.1 User Feedback . . . 32

6 Conclusions **34**

A Program Output **35**
- A.1 SKOT output . . . 35
- A.2 Resulting OWL file . . . 35

B Initial Plan **38**

C Initial Use Cases **39**

D Requirements **40**
- D.1 Functional Requirements . . . 40
- D.2 Non-functional Requirements . . . 41
- D.3 Additional Requirements . . . 41

E References **43**

1 Introduction

In this report, the project background is introduced, defining Knowledge Representation (KR), ontologies and OWL in particular; the design and implementation of the project is described, along with various evaluations of the result. In the rest of this chapter, a brief overview is given of the problem and proposed solution, with more detail in chapter 2.

The initial project proposal was to create a generic tool which could "sketch out" a simple diagram of "blobs" and "lines." Further discussion indicated that the main target would be OWL ontologies (logic-based repositories of knowledge, explained in section 2), particularly those currently edited in Protégé (section 2.4.1), another ontology creation tool.

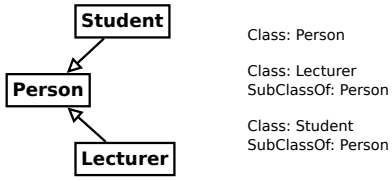

Figure 1: A diagram showing some classes represented as blobs and lines, and the equivalent OWL axioms.

During creation, ontologies proceed through an ontology life-cycle (Figure 2, [19]). After the purpose and scope for the ontology has been defined, and Knowledge Acquisition (section 2.4.3) has been performed, the ontologies go through an iterative cycle of conceptualisation, integration and encoding, where concepts found through KA are: identified and formed into models; integrated with the ontology language or an existing ontology; and defined in some formal representation. After this cycle has completed, ontologies can be documented and evaluated.

The project fits into the "conceptualisation" stage of ontology creation. In this stage, acquired knowledge is used to specify the important concepts within the ontology's scope, as well as relationships between them. These concepts and relations are given labels and formed into a coherent model. After this stage, the generated concepts can be inserted into the resulting ontology and defined using the given formal language (such as frames or logic). SKOT uses a 'blobs and lines' diagram to represent concepts and relations, and provides a tool for the conceptualisation stage of creation.

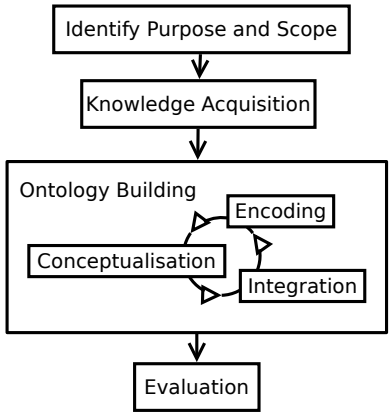

Figure 2: An ontology life-cycle, based on a diagram from [19]

The initial use cases created from this early specification and discussion are listed on page 39 in the appendix. As the program is defined by its user interactions, these use cases also formed the basis of the functional requirements specification for the project. Nielsen's Heuristics [10] were used as the basis non-functional requirements and are discussed later in section 5.2. A created requirements specification for the project is listed in section D.

1.1 Project Planning

The initial project plan can be found on page 38. The basic elements of the plan were to take on new features on a bi-weekly basis (which ended up being weekly to match the supervisor meetings) and have a minimal working product by Christmas, which was the main milestone for the project. This initial product needed to have the ability to input a list of words, turn them into lines and blobs and output some meaningful OWL file from the result (Requirements 1 to 4).

This first objective was met, and after the January exams, additional features were added on an iterative basis, with user feedback influencing the project during the last few weeks. While not all requirements for this second objective were fulfilled, the delivered program was evaluated well by users.

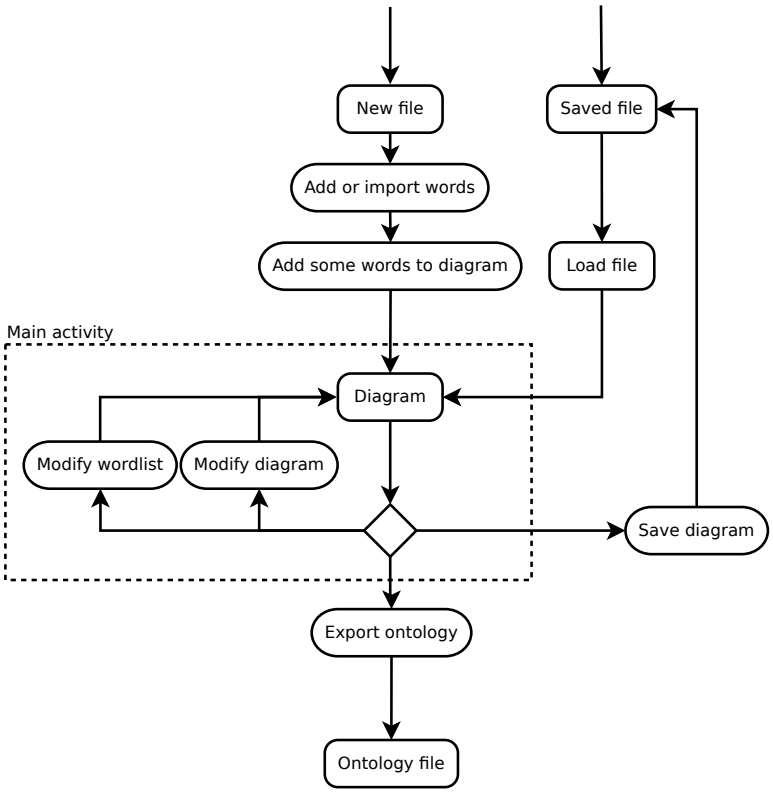

Figure 3: An activity diagram outlining a typical SKOT workflow. More detail is given in sections C (Use Cases) and 4 (Results).

2 Background and Literature Survey

In this section, a brief overview of the field of Knowledge Representation (KR) and ontology engineering is given, along with how the project fits into this field.

Knowledge Representation is the name for a group of methods in Artificial Intelligence (AI) which aim to represent knowledge in ways that a computer or other automated process can reason about and draw inferences from. A 'Knowledge Representation' can also refer to the final result, such as a glossary, taxonomy or ontology.

Ontologies are KR constructs, somewhat analogous to data structures for knowledge. Ontologies contain knowledge, stored in some formally-defined ontology language, usually based on some particular logic system so that the stored knowledge can be reasoned about.

OWL (Web Ontology Language[6]) is a particular ontology language based on a variant of first-order logic. It builds upon RDF[17] and RDFS[18], although it is syntax-independent and does not necessarily need to be stored in XML. Various syntaxes exist, some which are more computer- or human-readable than others.

Protégé is an ontology editor and the main alternative to the project; however, as the project description states, it is limited and inflexible for the purpose of initial ontology creation when the final structure is still partially unknown. Most of Protégé's tools assume that the ontology's structure is final, even while it is being input into the program.

2.1 Overview of KR methods

Knowledge Representation methods evolved from early AI research into expert systems that could store some human experts' knowledge about a particular domain in a format that could be accessed and questioned by other users. Early attempts to mimic human cognition with the use of heuristic-based analysis, neural networks etc were not very successful and around 1970 (with the development of Mycin [20]) research began to focus on rule- and logic-based expert systems as we know them today.

Knowledge representations are usually rated according to three factors, all of which usually involve trade-offs against the other two:

Expressivity What it is possible to state in the representation, and what it is possible to reason about.

Usability How easy it is for the user to create and modify the representation, and how easy it is to understand the processes and results involved.

Computability The computational complexity of common tasks within the representation, such as finding possible inferences or checking for consistency.

Knowledge Representation methods mostly fall into a few general categories:

Terminology A simple collection of words and their definitions, similar to this list. Essentially a glossary, these are generally easy to use, but

not much can be expressed or reasoned about further than a simple searching of terms.

Taxonomy A hierarchical categorisation of terms. Taxonomies often have a defined depth with a name for each level of categorisation. The most obvious example is the practise of biological classification, which has a range of biological types from "Life" to kingdoms, families and species. Taxonomies are generally quite easy to use, though again have limited expressivity.

Ontology A specific, shared representation of some knowledge in a formal ontology language, based on a set of concepts and the relationships between them. 'Ontology' is also a study in philosophy, of the nature of being or existence and the relationships between definitions and objects.

Ontologies are much more expressive, since they are generally based on some reasoning system such as a logic or frame language. On the other hand, they can be computationally expensive and complicated to use.

Different ontology languages can also vary greatly in terms of expressivity etc; depending on the underlying language or logic.

2.2 Semantic Web

Directly related to KR efforts, 'The Semantic Web' is the name for a group of technologies aimed at adding meaning to information and resources on the Web, in much the same way that KR looks at knowledge rather than data. The simplest form of the Semantic Web lies in HTML metadata, both in page descriptions and links. For example, the XHTML Friends Network (XFN) provides a way to describe human relations semantically in (X)HTML by embedding data into the rel attribute of links.

Going one step further, RDF (Resource Description Framework) is one file format that attempts to describe *things* in the way that HTML describes documents. RDF files build RDF graphs, and the hypertext relations between these files create a 'Giant Global Graph', as described by Berners-Lee in [12]. An example of RDF in use is the Friend of a Friend (FOAF) project, which describes people's interests and interconnections in a similar way to XFN. Ontology languages such as OWL are built on top of RDF (and RDFS).

While semantic web technologies are generally agreed to be worth using, the vision of a 'meta-utopia' is far off; Cory Doctorow outlines the common reasons in his 'Metacrap' essay[13]. The popularity of such methods, though, is growing, and the use of semantic methods along with currently-used metadata such as PageRank seems promising.

2.3 OWL

OWL is a current KR ontology language, itself defined as an ontology in RDF(S) [6], although it is largely syntax-independent. It is based on older ontology languages, particularly DAML+OIL. OWL files are defined as lists

of axioms, with different types of axioms each stating relationships between classes. OWL is split up into three variants - Lite, DL (Description Logic) and Full. Lite is a subset of DL, consisting of the more commonly-used and simpler to implement features. DL is basically the same as Full but with some important restrictions that reduce the expressivity of DL, but also significantly reduce its computational complexity.

Classes are the main building block of OWL, and are somewhat analogous to their Object-Oriented Design namesakes. Classes can be assigned "named individuals," analogous to OO objects, and an OWL ontology specifies a set of models, which are possible allocations of individuals to classes such that they satisfy all of the axioms defined in the ontology. If an ontology has no possible models then it is said to be inconsistent.

There are a number of defined relationships within OWL which can be asserted using axioms. Classes can be subclasses of other classes, and individuals can be asserted to be a certain class. These features allow simple taxonomic trees to be built. Importantly, since OWL operates under the Open World Assumption, classes are assumed to overlap unless they are defined or inferred to be disjoint. This can cause some unexpected behaviour when dealing with class membership, since objects can be members of two classes that might be assumed to be disjoint. Classes can also be defined to be absolutely equivalent.

Additionally, 'object properties' can be defined as custom relationships between classes. For example, in an ontology about universities and students, the "`attends`" relationship might exist between the `Student` and the `University`, and could be used to group the students. Extra definition can be added to properties, such as restrictions on cardinality, or definitions of transitivity or symmetry.

An important feature of OWL is the ability to use expressions as well as atomic classes in the definitions of relationships. Classes can be restricted with subclass axioms, or defined totally by making them equivalent to some expression. Expressions can be created by combining or intersecting classes, or using defined properties. For example, in an ontology about animals[1], a `Carnivore` could be defined as a subclass of the *expression* "`Animal AND MeatEater`." Since this uses a subclass relationship, this does not completely define `Carnivore`: some individuals that are both `Animal`s and `MeatEater`s might not be `Carnivore`s, but it does restrict `Carnivore`s to be members of both of those classes. In other words, being both an `Animal` and a `MeatEater` are necessary conditions for being a `Carnivore`, but not sufficient conditions.

As another example, `Carnivore` could be defined as equivalent to the expression "`Animal AND (eats SOME Meat) AND (eats ONLY Meat)`,"[2] where "`eats`" is a previously-defined object property. This definition of `Carnivore` includes all members (and only those members) that are in the intersection

[1] It's interesting to note that the following definitions exclude carnivorous plants. All ontologies are interest-relative and context-sensitive to a certain degree, and it's important to make sure that the ontology's limits are well-defined, as a part of the ontology creation process. In this case, carnivorous plants are discarded for the sake of explanation.

[2] A common point of confusion when creating ontologies comes from not including both `SOME` and `ONLY` here. If we only used the `SOME` clause, then omnivores would be included in our definition. Conversely, omitting the `SOME` clause would include animals that eat nothing at all.

of the class `Animal`, and the class defined as: the set of objects $x \in X$ where some `Meat` y exists such that x "`eats`" y, and no non-`Meat` objects exist that have this relationship with x. Since the equivalence relationship is used, this definition is bi-directional—it is both necessary and sufficient for an object to meet this definition to be classed as a `Carnivore`.

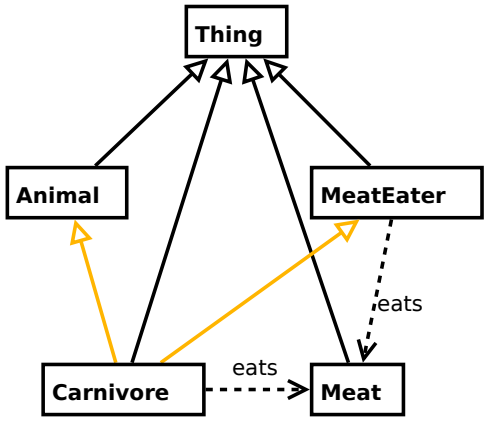

Figure 4: A diagram showing the relationships described above. The relationships in orange are those that the reasoner can infer, given `Carnivore`'s definition. "`Thing`" is an OWL-defined class which includes all objects in the ontology.

For this project, OWL is the main ontology export target, using the already-defined OWL API [7]. For the purposes of the program, "blobs" are equivalent to classes or named individuals (depending on whether the blob is asserted to be an individual or not in the diagram) and lines between blobs are equivalent to different relationships depending on the line type. When they are displayed on the diagram, classes are referred to as "`Concepts`" in the source code to avoid confusion with Java classes. Classes that are only in the word list are stored as "`Word`" objects instead.

2.4 Existing Methods

The two main influences when developing SKOT were Protégé and Freemind.

2.4.1 Protégé

Protégé is a frame- or OWL-ontology creation tool that is widely used. As discussed earlier, while it has comprehensive support for heavyweight ontology features, it does not allow the user to easily build or alter the structure of a new or existing ontology.

Protégé's strengths are its support for advanced reasoning on ontologies. In this screenshot (Figure 5), a simple class hierarchy is shown on the left, with comments and definitions for a particular class shown on the right. The yellow-highlighted definitions under "Equivalent classes" are axioms that Protégé has inferred by itself. The example from section 2.3 is shown here—Protégé has used the axiom "`Carnivore EquivalentTo: Animal and (eats some Meat) and (eats only Meat)`" to deduce that

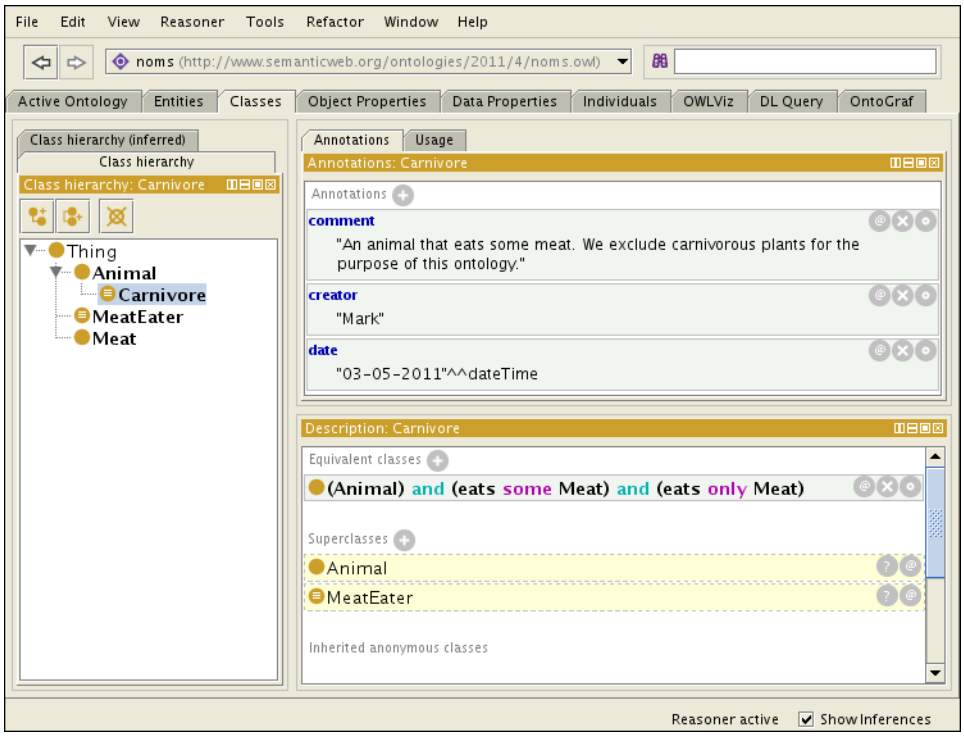

Figure 5: A screenshot of Protégé, showing a class hierarchy on the left, annotations at the top right and class definitions on the bottom right.

`Carnivore` is a subclass of `Animal` and `MeatEater`, which has the definition "`eats some Meat`."

2.4.2 Freemind

Freemind is a mindmap creation tool. Mind maps are hierarchical diagrams, often used for taking notes, brainstorming ideas and collating information in a certain domain. Broad concepts are attached to the root of the diagram, with details or sub-concepts added recursively to those. Mind maps are often distinguished from other types of diagram by high levels of graphical customisation and nonlinearity of information, meant to assist the user in remembering the stored information. A screenshot is shown in Figure 6.

One significant advantage of Freemind over other tools is the keyboard interface. Since the diagram generated is strictly a tree[3] and most of the layout is done automatically, almost all common tasks such as navigating the diagram and adding/editing information can be done through the keyboard. This significantly reduces the difficulty and time taken when performing these tasks. While I was not able to replicate the keyboard interface, reducing this 'interface delay' when performing an action in the program became an important design goal. In fact, a request for a similar editing mode in the program occurred in the user feedback (discussed later in section 5.3.1), and a similar feature could be part of future development.

[3]Freemind supports placing links between otherwise unrelated nodes, but these are mostly cosmetic.

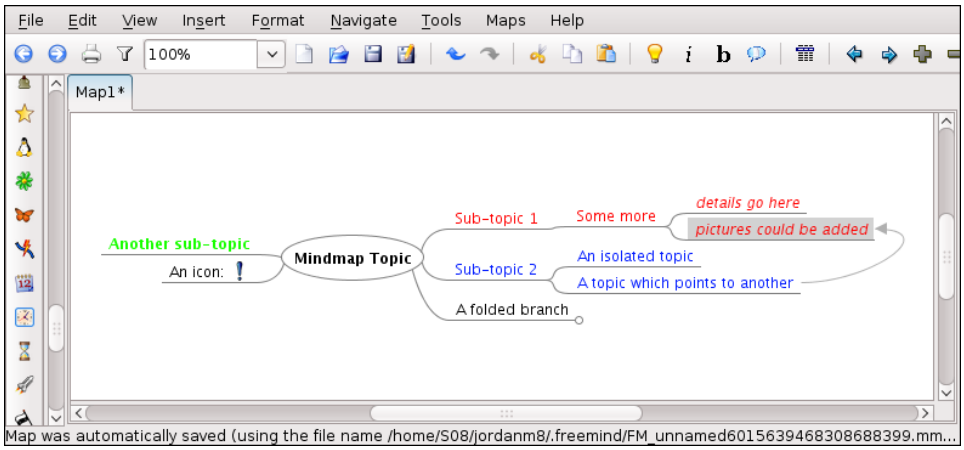

Figure 6: A screenshot of Freemind, showing an example mind map being edited.

2.4.3 Knowledge Elicitation

Knowledge Elicitation (sometimes called Knowledge Acquisition) is the process of extracting knowledge about a domain from human experts so that it can be stored and referenced in some KR. Knowledge Elicitation can be a hard process, since most domains have a large amount of 'implicit' knowledge, which even the domain experts may not be aware of.

Knowledge Elicitation methods often revolve around arranging words, often written onto physical cards or a whiteboard, and finding ideas that link or separate the different concepts that are represented. For example, in one method, called "card sorting", a domain expert may be asked to group some of the cards together, and then is interviewed on the reasons for the particular grouping. Groups may be general collections of similar terms, or fully-qualified taxonomies, depending on the participant's view of the domain. The groups that are generated by this method may be named; these names are often introduced as new concepts and implemented as superclasses. Sorting can also be used to evaluate pre-existing groupings, by getting a participant to sort the cards into the pre-determined groups and observing how neatly the groups work.

A related method, called the "three card trick," works by taking three random cards from the set and presenting them to the participant. The participant then selects two cards which are the most similar, and explains their reasoning as well as what makes the third card different. In this way additional relationships between concepts may be discovered.

Several other elicitation methods include straightforward interviews, where discussions with domain experts about the domain take place; shadowing, where experts are followed about their daily routine, sometimes providing commentary on the actions that they are performing; or other categorisation methods, such as diagram creation or "twenty questions," where the domain expert has to identify an object in the domain, and the types and order of the questions that they ask are noted—these can indicate the features of domain objects that the expert considers most important.

While the project is aimed at the ontology creation stage, after knowl-

edge elicitation has been performed, it is important to take the discussed elicitation techniques—particularly card sorting—into account when developing and evaluating the project, since elicitation and ontology creation are tightly linked processes. However, the project is definitely aimed at ontology creators rather than domain experts.

3 Design and Implementation

In this chapter, the design and some of the implementation issues that arose during the programming effort are discussed. Java was chosen for the programming language, since it was known already and had provisions for creating the required custom UI components needed: using Swing, Java's main UI library.

The design process was loosely based on the Unified Process (UP) [2]. It followed the basic phases of Inception[4], Elaboration, Construction and Transition, and had timeboxed iterations that were defined by the weekly supervisor meetings. These phases can be seen on the initial plan on page 38.

The main design goal was to streamline the process between thinking of ideas and having them instantiated on the diagram. Part of this (requirement 7) was the need for a Low Representational Gap (similar to Larman's thoughts on OO design in [2]) between the UI components in the program and the underlying ontology concepts as well as the ontology being represented. Part of "LRG" is having 'direct manipulation' of ontology objects through the diagram interface—the user should feel as though they are directly modifying the ontology structure; as well as direct visualisation—the user should be able to clearly understand the underlying ontology structure by looking at the diagram.

3.1 Design Overview

The design for the project ended up being simple in overall structure but complicated in implementation. A class diagram is shown on page 13. At the top level, the design has a simple model/view separation with a controller class in the middle that interfaces between the two.

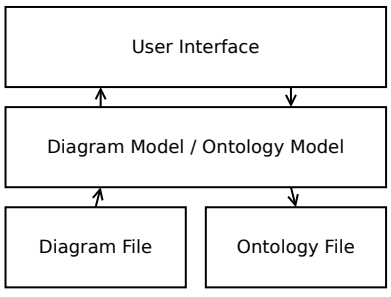

Figure 9: An early architectural diagram, showing the basic model/view separation and data flow.

Model classes are at the top and UI classes are towards the bottom, with `AppController` and `AppModel` interfacing between them. There is a broad parallel between several of the model and view classes - `Concept`s and `Relationship`s are represented in the UI by `Blob` and `Line` classes - similarly with the `WordListUI` class for `Word`s and `AnnotationUI` for `Annotation`s. This parallel in code assists with implementing requirements 2a and 7.

[4]Although Inception is really equivalent to the time before the project was started, when the supervisor created the idea for the project.

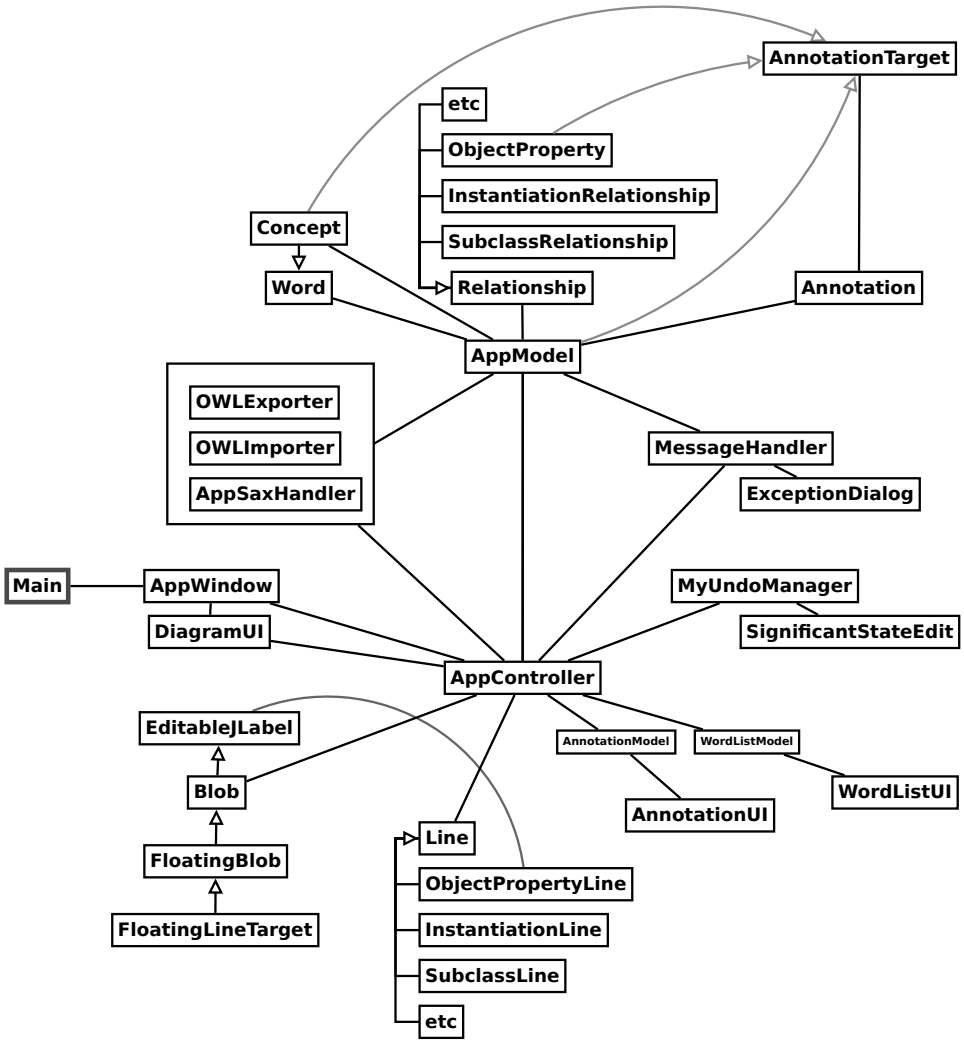

Figure 7: This is a rough class diagram for the program. Arrows show inheritance and lines show collaboration.

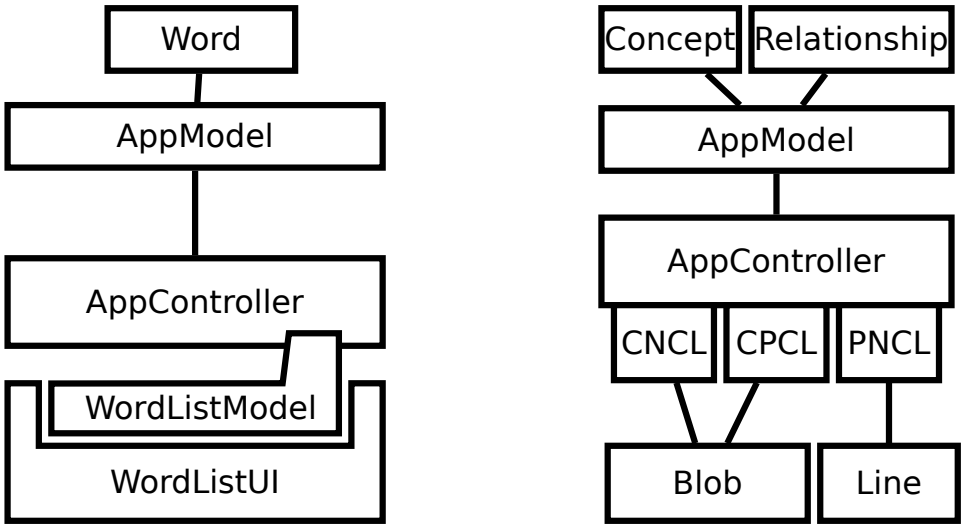

Figure 8: A pair of diagrams showing some of the mechanics of the `AppController` class. Boxes correspond to classes and lines to collaboration.

`Main` is the program entry point, and `AppWindow` and `DiagramUI` are the main UI containers. 'Off to one side' are some of the file I/O classes that interact with both the model and the view. These read files and then set up the program to view them, or collect state from the program and write it out to a file. `OWLExporter/Importer` and `AppSaxHandler` are the main file I/O classes for OWL export, import and SKOT input respectively, while `MessageHandler` handles debugging, error and information messages from the rest of the program.

For both the model classes their equivalent UI components, there are group classes which collect them together and handle creation and deletion. On the model side, this job is mainly handled by `AppModel`, and the UI side has `DiagramUI` for the blobs and lines, which is then collected into `AppWindow` alongside `AnnotationUI` and `WordListUI`.

Figure 8 shows some of the mechanics of the `AppController` class. On the left side, the UI element `WordListUI` has a custom object provided by `AppController` (slightly confusingly called the "model" object by Swing) which specifies the data that the UI component displays. This is implemented by a series of callbacks which are handled by `AppController` and `AppModel`, and inserted into the underlying `JList` with a method call. A similar system is used for the annotation window.

The right diagram shows how `AppController` provides a set of `Listener` objects for Concept Name Change, Concept Position Change and (Object) Property Name Change events respectively. This replaces an older system where the `Word`, `Concept` and `Relationship` objects would be `Listener`s for `Blob`s and `Line`s themselves. The newer design is much more flexible and less error-prone.

3.2 Design Specifics

In this section, several interesting design specifics are discussed, such as MessageHandler and the Swing custom classes with DiagramUI.

3.2.1 MessageHandler

`MessageHandler` is a singleton class that is used by almost all other classes for debugging and user output messages. Several different logging systems were considered, but most of the systems investigated seemed too heavy-weight for the program's needs.

`MessageHandler`'s main advantage is the debug() method. When debugging is enabled (by a static variable within `MessageHandler`), this prints out the given message, along with a timestamp, and the calling method's name and location. This means that the origin of all debugging output is easy to find, and disable when necessary. The timestamps are also very useful when debugging UI methods, as it gives an indication of response time as well as being able to see the results of distinct UI actions by the groupings of the timestamps.

`MessageHandler` also has similar methods for user communication, for putting messages into the status bar, as well as pop-ups of various styles (error, information, etc) and displaying exceptions. When a generic unexpected exception occurs, `MessageHandler` displays information for the user stating that the current action has failed, along with the exception information for debugging. Since the exception has been caught, other UI actions can often continue.

3.2.2 DiagramUI and Swing custom classes

One of the reasons Swing was chosen for the UI code was that it has easily extensible component classes. `DiagramUI` is based on a Swing container, and `Blob`s on (heavily modified) Swing buttons, although `Line`s are wholly custom components. In this way, much of the default event handling and redrawing is handled by Swing, although customisations had to be made in most places.

Other UI components (the word list and annotations window) were more closely based on default Swing components (a `JList` and `JTable` respectively). In these cases, the main source of customisation was providing a custom model[5] and custom objects for rendering each row of the list or table. For parts of the annotations window, a pre-existing component available under the LGPL [14] was used, which gave the option for a bigger editing area than the small default table cell.

For some of the buttons, icons from [15] were used. Disabled "greyed-out" versions were created as necessary. These icons are available under a Creative Commons Attribution license. For other buttons, particularly those representing different relationship lines, it was hard to create buttons that suitably represented the underlying concepts. One idea was to simply recreate the diagram look of each of the lines on the button, but this would

[5] In Swing, 'model' refers to the underlying data structure that a particular component displays, rather than the model in terms of the whole program and model/view separation.

require memorisation of the different line colours, as well as not having much correspondence with the actual underlying relationships that are being created. In the end, it was decided to stay with text for these buttons.

3.3 Design Implementation and Changes

As the design evolved over time during the implementation phase, there were several interesting design decisions:

The overall structure of the program changed over the course of development. Initially, there was not a controller class to interface between the model and view sections. There were still separate groups of model and view classes, but they were able to freely interact with each other. This caused a lot of sloppy programming and some "interesting" behaviour in certain cases. This code was refactored a couple of times, first to insert a controller class (`AppController`) and move UI code into its own package, and then to remove some observers that were left watching the model classes. After this was complete, all the interactions between the model and view classes had to go through the controller class, which became a necessary simplification as the program got more complex.

The addition of a controller class also helped with several other features, such as undo/redo functionality (Requirement 8b). Since all changes to the program state had to go through the controller class, it became an ideal location to detect and store state changes. Undo functionality was implemented by storing the model state in `StateEdit`s, which are basically pairs of sets of objects with one set representing the old state before the edit; and one representing the new state. Redundant objects (those with the same state in both sets) are removed to keep size down. A customised `UndoManager` keeps track of the various edits, provides undo and redo `Action`s for the UI and makes sure that the model state is consistent after certain edits (such as when some of the model objects change classes, a change which can't unfortunately be stored in a `StateEdit`).

3.3.1 Design Changes from Testing and User Feedback

Several features were added as part of user feedback. Some of these features had already been considered for future implementation, but having actual feedback allowed me to prioritise certain features and gave me new ideas for others. A good example is iterating on the diagram UI interactions. As said before, the main goal here was to make it as quick as possible to put ideas onto the diagram after they have been thought of. Part of this goal was to make it easy to quickly move and rearrange groups of blobs (Requirement 17).

One initial idea (Requirement 3e) was to make a 'group' UI object to be something that was created in the diagram, that could then be dragged around and interacted with other diagram objects. This proved to be a problem in several ways. Firstly, it would require a significant error to code and to make sure that it worked in all the right ways with the other UI components. It would have also taken up a lot of space - fitting some rectangle or oval around a group of blobs would also cover a lot of previously empty space. Thirdly, it was hard to decide on what the group object

would represent in an ontology sense, so adding it would violate the least representational gap principle.

Since the group object wouldn't work, the focus shifted to making selections of multiple objects work in an efficient way (Requirement 17). This meant refining the code for what happened for different combinations of clicks, double clicks, drags and modifier keys present (such as shift and control) as well as adding a box-drag selection mode. Other actions such as adding relationships and deleting objects were updated to work with the multiple selections.

Another example of iteration after user testing was the inserting of new words onto a diagram (Requirement 19). From the original use cases, the user would have to go through a long process to insert new words, from typing (or pasting) them into the word entry box, adding them to the word list, then pressing the insert button for each word and placing each one on the diagram. This wasn't too bad for when the user was inserting a precreated list of words at the start of a creation process, but it was particularly unwieldy when adding a new word later on in the creation process. Given this problem, a feature was added where simply double-clicking on an empty part of the diagram would create a new empty blob (with a default name) that would get added to the model if renamed. The user could simply double-click and then begin typing the new name in order to create a new concept, which was a huge speedup.

In a similar vein, added drag-and-drop functionality was added from the word list to the diagram. This meant that the user could select multiple words from the list and drag them onto the diagram, which replaces the 'insert' button functionality (which also only works on one word at a time). However, given some of the feedback (p32, user (2), point 4) it seems that the drag-and-drop functionality was not obvious, and the insert button was confusing in that it seemed to be the only option. In future work, the insert button should probably be removed entirely, since the drag-and-drop feature is significantly better.

3.3.2 Future Extensions

There were a few features that didn't make it into the final program due to lack of time and having to prioritise other features for the demonstration. A couple of the bigger ones were ATR integration and SKOS support (Requirements 1c and 4b. ATR (Automated Term Recognition, sometimes Terminology Extraction[21]) integration would mean an alternative for the usual first step of inserting a list of words into the word list section, and instead starting with some natural-language text and using a web service such as Termine [4] to automatically generate a suitable list of words. This can still easily be done manually, such as in some feedback (p33, user (3), point 2), but integration would be better.

SKOS [5] is a KR ontology built on top of OWL. Instead of focusing on classifying objects within a domain like OWL, SKOS talks about concepts, the different language labels a concept can have as well as its basic relations to other concepts. Adding SKOS support should be fairly simple due to the way that OWL export is implemented. The OWL exporter is simply defined

in a separate class that accesses the diagram models and outputs a file to a given location; it should be easy to write an equivalent SKOS exporter that works in the same way. Unfortunately, due to time constraints and trouble with building the SKOS libraries, this feature had to be skipped.

3.4 Design Considerations and Limitations

Sometimes, design issues were discovered that unfortunately turned out to be limitations of the design rather than issues that were solvable just with coding. Some of these could be worked around, but others had to be accepted as part of the limits of the project.

A first example is annotation display (Requirement 13. The initial idea for displaying annotations was to simply have them as additional blobs on the diagram that are linked to the concepts for which the annotations are relevant. This turned out to be unworkable for a number of reasons: mainly that each blob on the diagram would now have multiple little blobs hanging off of it, which would add a lot of clutter to the diagram, and make accurate group selection much harder. In addition, most blobs would have a similar set of annotations (although with different values) so this would add a lot of redundancy to the diagram.

For these reasons, the annotation display was split off into a separate table that only showed the annotations for the currently selected object. This doesn't make adding or editing annotations any harder, but clears up the potential massive clutter that the original method made. One small disadvantage of this method is that it is harder to see which objects have annotations and which do not; it was planned to add some indication to the objects but this was not implemented due to time constraints.

Another example was showing disjointness relationships. In the program there is a disjointness relationship line which works in the same way as the other relationship lines: it links two blobs and makes them disjoint. However, ontology creators usually want to set more than two classes to be disjoint with each other, and disjointness is not transitive. This means that the number of disjointness lines required rises dramatically as more classes are introduced, which can quickly become unusable. This means that the normal UI is not sufficient to capture this relationship properly.

There are a number of possible solutions to the disjointness problem. One might be to create a 'transitive disjointness' line, which could then be processed during the ontology phase and turned into the required disjointness axioms. This has the disadvantage that it does not match with any existing ontology creation standards, so it could be confusing for the user, although it fits decently into the current UI. Another solution might be to reintroduce the 'groups' idea from above (p16) and have a disjointness group, although this has the disadvantages listed above; it would require a lot of coding, and add to more clutter on the diagram. Finally, another solution might be to add new actions for the most common usages of disjointness, such as setting all subclasses of a particular class to be disjoint. This would mean adding some sort of toggle button for performing this switch and a special UI indication on the particular class. The disadvantages of this would be reduced flexibility and increased UI complexity, although it does

match features available in other programs so most users would probably understand this solution best.

Unfortunately, due to time constraints, none of the solutions above were implemented, particularly as only one disjointness line was needed for the demonstration and so this feature was pushed back in priority. However, given more time, the last solution described would be implemented, since it matches solutions that other programs use and would therefore be easier for most users to use.

4 Results: An Example Workflow

In this section a simple example of SKOT's workflow is shown, using a scenario from the demonstration. The target ontology will be based on the small extract of text in Figure 10.

This text represents the results of some Knowledge Acquisition process, as discussed earlier. The next stage is conceptualisation—converting this knowledge into a set of concepts and relationships that can be stored in the ontology. The first step in conceptualisation is finding a set of terms as a starting point. This can be done manually or though some Automated Term Recognition software such as Termine[4]. In either case, the result is a list of words such as in Figure 11.

Once these words are available, then they can be input into SKOT. Given a comma-separated list of words, they can be pasted directly into the word entry field and entered by pressing the "+" button, shown in Figure 12.

Words can then be dragged out onto the diagram, grouped according to their similarities. At this point we notice that `Information` and `Structure` don't fit into our ontology scope, so they can be discarded. (Figure 13)

At this point, we notice that both `Student` and `Lecturer` have a common superclass `Person`, so we want to add this to the diagram. This can be achieved quickly by double-clicking on the diagram, which gives a new blob where a name can be typed. (Figure 14)

Relationships are now added between concepts. Once `Mark` has an `instanceof` relationship added, it is highlighted as a "named individual" and restricted as to which subsequent relationships can be added to it. We also add some `subclassof` relationships to create a basic subsumption hierarchy, and make `Student` and `Lecturer` disjoint.

Creating relationships is fairly straightforward. First, the beginning of the relationship is selected. This can consist of multiple blobs, which means multiple relationships created. Second, the button for the respective relationship type is pressed, and the target blob selected.

Some object properties have also been added, which work similarly to the other relationships but have a customisable name field. An ontology URI has also been added. This field forms the base of the URIs for each of the concepts and relationships in the ontology, and often points to a location on the Web where a recent version of the ontology can be found. (Figure 15)

Optionally, annotations can be added to the diagram. Annotations are shown in the right window for the most recently-selected object and can be used to store extra information about the concepts involved, such as extended natural-language definitions, alternative names and administration

> We want to store information about a university structure. In our view of the university there are students and lecturers. Lecturers teach lectures, and these lectures are attended by students. There are two types of students: Undergraduates and Postgraduates. One particular Undergraduate is named Mark.

Figure 10: An extract of text which forms the basis for our example ontology

```
University, Information, Structure, Student, Lecture,
Lecturer, Undergraduate, Postgraduate, Mark
```

Figure 11: A list of useful nouns from the source text, generated by hand or some ATR process.

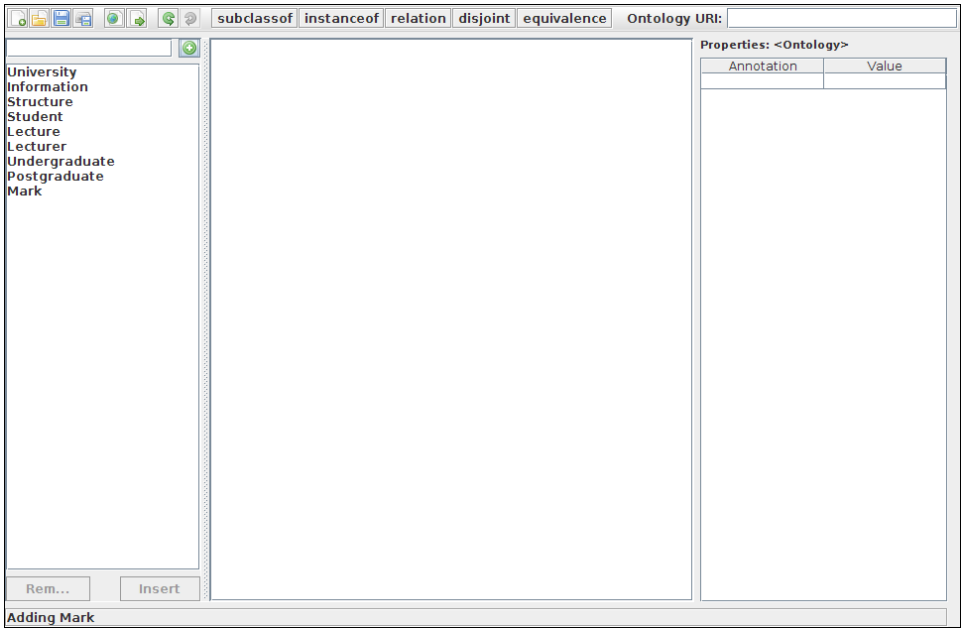

Figure 12: A new instance of SKOT with words inserted into the word list

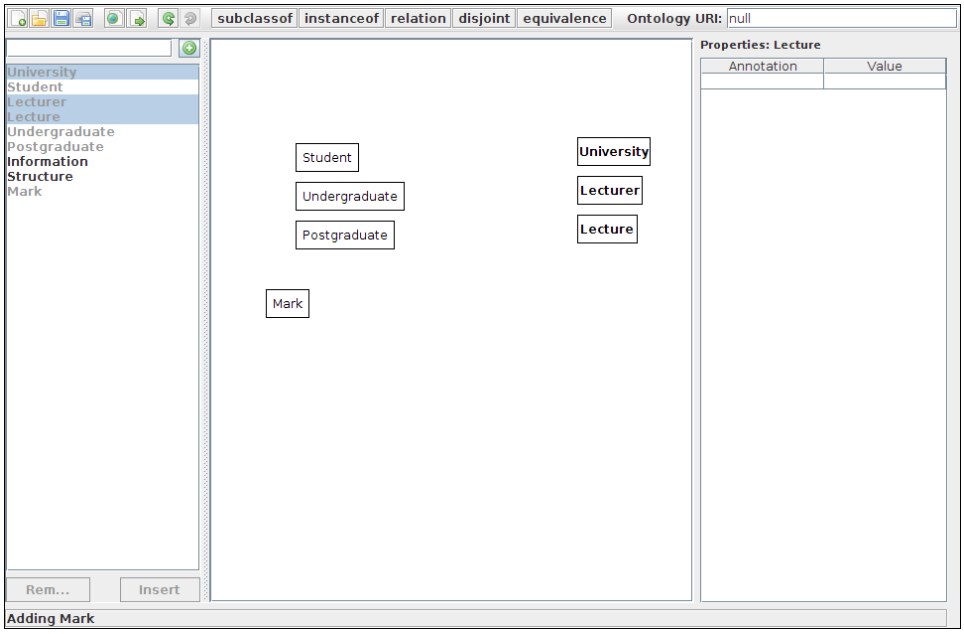

Figure 13: Words have been dragged out onto the diagram, and arranged into rough groups based on their similarities.

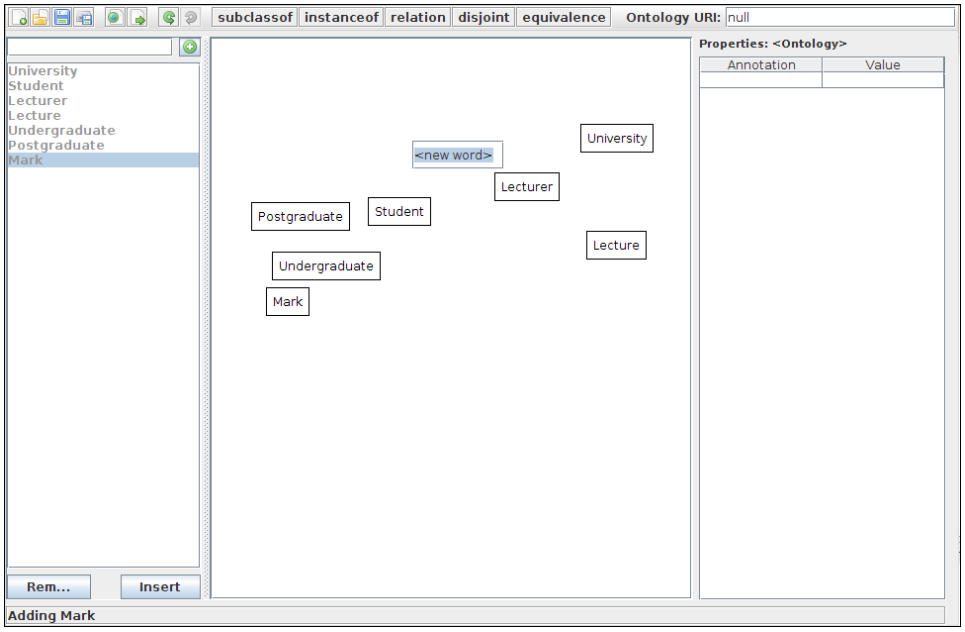

Figure 14: A new blob can quickly be added by double-clicking the diagram.

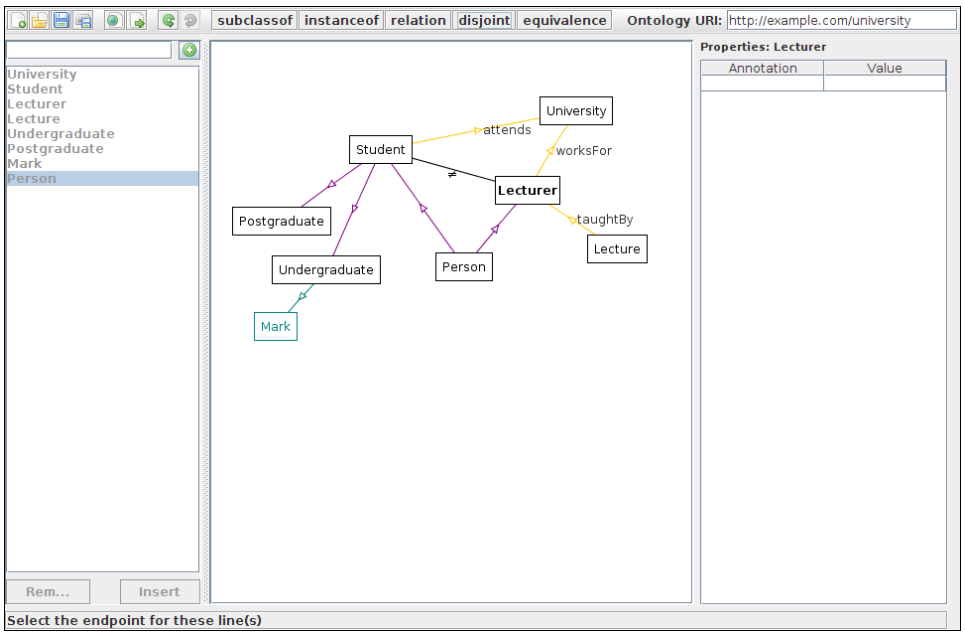

Figure 15: Relationships and an ontology URI have been added

information like contributor names and dates. If no blobs are selected, then annotations are added to the ontology itself. (Figure 16)

At this point there might be some new information available (Figure 17). This can quickly be added, producing the initial result in Figure 18.

We can then export this ontology structure into Protégé using the "Export to OWL" button. This creates an OWL file that Protégé can read. Once our structure is in Protégé, we can add formal definitions to some of the classes. Some examples are given in Figure 19, and an example of the class structure view and details of one of the classes is given in Figure 20.

One benefit of creating these definitions in Protégé is that we can run a reasoner on the ontology and see what additional inferences can be made. In this case (Figure 21), Protégé has spotted a bug in our ontology—the class `TALecture` is equivalent to `Nothing`! This means that `TALecture` is *unsatisfiable*—it can not have any objects within it. If we try to define a named individual as being a member of `TALecture` then the ontology will become inconsistent.

For all inferred results, an explanation can be given by the reasoner as to which axioms it used to find that result (Figure 22). In this case, it boils down to the fact that lectures are taught only by lecturers, but a `TALecture` is taught by a student—and since we have explicitly defined students and lecturers to be disjoint, then this situation is unsatisfiable.

There are a number of workarounds for this problem, including separating people and their roles, allowing `Student` and `Lecturer` to overlap, or having `TeachingAssistant` be a kind-of `Lecturer` and having the actual person be in the ontology twice.

Finally, after the ontology has been modified by Protégé, it is possible to re-import the resulting OWL file back into SKOT. Due to the complexity of the new OWL file, SKOT cannot alter the existing structure of the ontology, since it will most likely break the more heavyweight logic that Protégé has added. However, SKOT can add new axioms. For example, new information might need to be stored about `Room`s; that `Lecture`s take place in `Room`s and that these `Room`s belong to `Universities`. Figure 23 shows these new axioms being added to the the ontology. Now, when the OWL file is exported, the old definitions about `Lecture`s and `TALecture`s will be preserved, and the new class structure will be added.

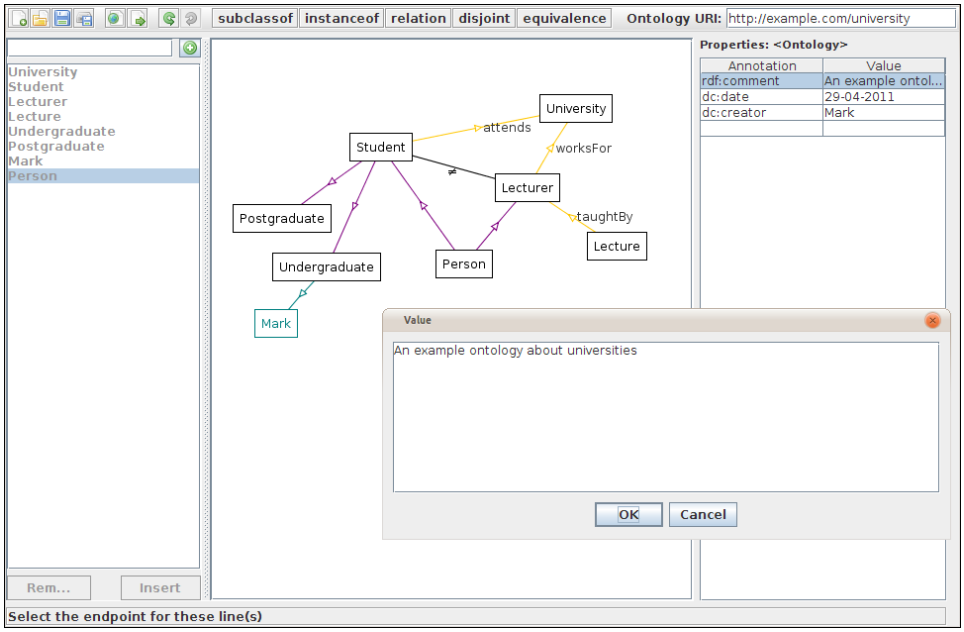

Figure 16: Annotations added to the ontology. A pop-up window is available for editing the values of longer entries.

We also want to store information about Teaching Assistants, or TAs. These are postgraduate students that teach special lectures in place of lecturers.

Figure 17: Some new information that needs to be included in the ontology

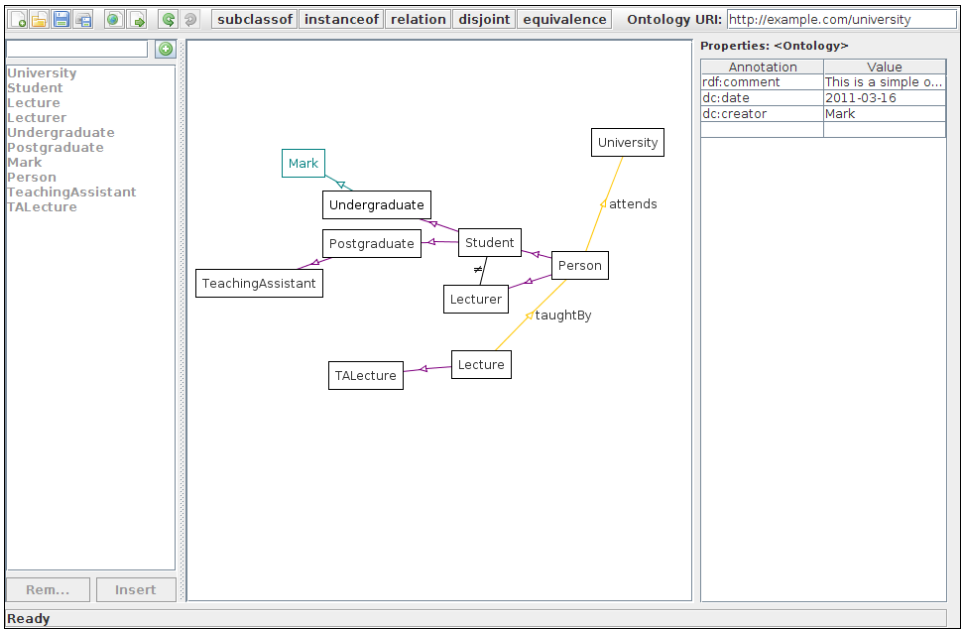

Figure 18: Our initial result, ready to be imported into Protégé

Lecture: `Thing that (taughtBy only Lecturer) and (taughtBy some Lecturer)` TALecture: `Lecture that taughtBy some TeachingAssistant`

Figure 19: Definitions for two of the classes, using Manchester Syntax [16] in Protégé.

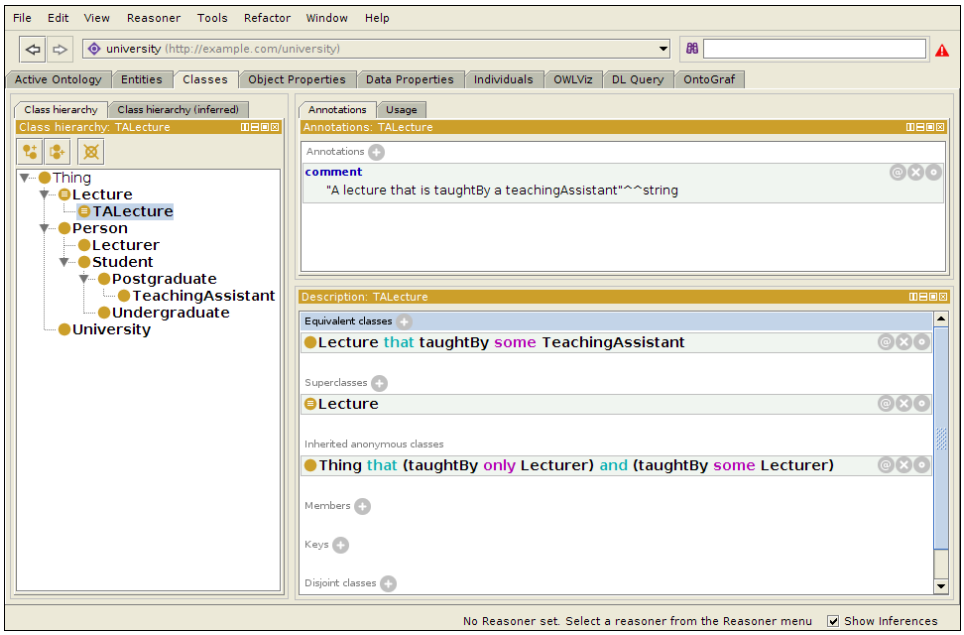

Figure 20: A screenshot of Protégé, showing a class definition that has been added.

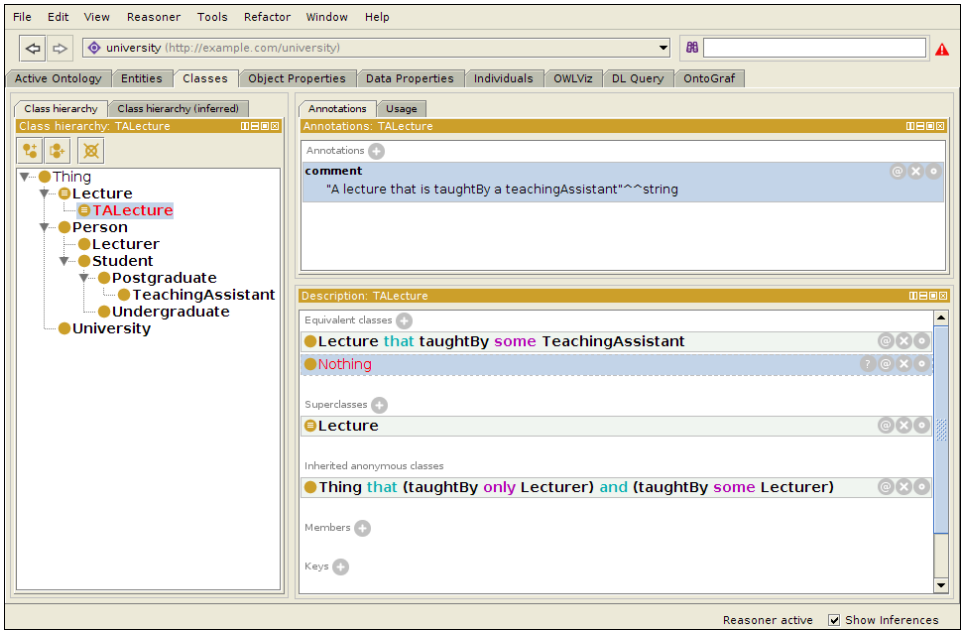

Figure 21: A screenshot of Protégé, with the red text showing that the reasoner has spotted an unsatisfiable class—a potential bug in our ontology.

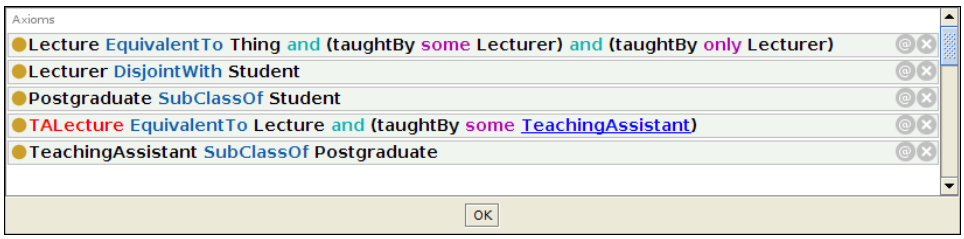

Figure 22: A screenshot of Protégé, showing the reasoner's explanation of the bug.

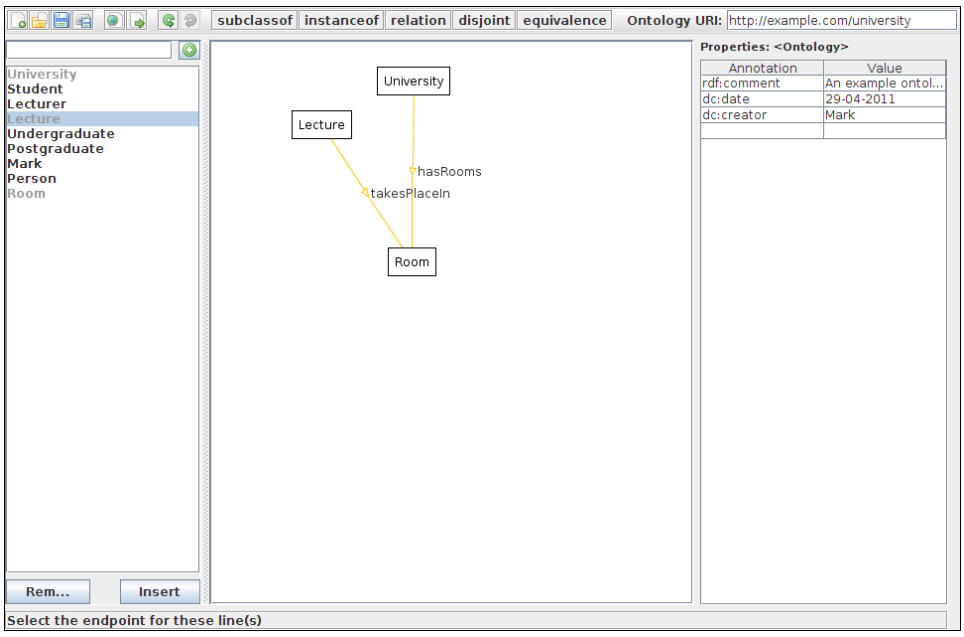

Figure 23: Adding new class structures to an imported OWL file.

5 Testing and Evaluation

During the project, several testing and evaluation methods were investigated. In 'Structured Testing', standard testing methods such as unit tests and performance tests are described. In 'Heuristic Evaluation', the program is assessed using usability heuristics. In 'User Evaluation', results of end-user testing are listed and discussed.

5.1 Structured Testing

During the project implementation, an effort was made to build unit tests in parallel with the implementation classes. Unfortunately, these were not successful for a number of reasons:

- Most of the classes were hard to test. While unit test packages for Java UI objects do exist(such as Abbot[8]), they appear to be mostly limited to pressing buttons or ticking checkboxes–i.e. the basic actions of default components. Other tools (such as Java's Robot class[9]) can be more general, but depend on specific mouse coordinates, which are fragile during testing and may need to be rebuilt when the program layout changes.

- Most classes depended heavily on a context, which made testing classes in isolation very hard. While this may have been indicative of a bad design, most of the classes in the program (especially the UI classes) needed their parent classes and the controller class in order to do meaningful things. In turn, the controller class basically required an instance of a running program in order to work.

- As the program structure evolved, unit tests broke and needed to be re-written. For example, when the controller class was introduced to force model/view separation (and fix some of the sloppy coding), the responsibilities of most of the model classes changed away from providing an interface to the UI and towards simply storing state. This made a lot of the old unit tests worthless, and created the next problem:

- The rest of the (non-hard) classes were *too easy* to test. For example, most of the model classes simply stored state (and sometimes retrieved it for the undo functionality). Beyond a few helper functions, there was nothing to test for these classes apart from accessor and mutator methods.

- Unit tests only test the developer's expectations, not the user's actual experiences. Especially in a UI-driven application, there are many more paths through the program than the developer could write tests for, even assuming that the user was fully knowledgeable about the program and what it could do.

So, for these reasons, unit tests became much more effort than they were worth, and much less useful than testing with real human users, as described below.

A few testing classes were written, although these were mainly prototypes for a couple of complex features that could be partially isolated from the rest of the code–i.e. selection boxes and bezier curves. Writing little test programs to run these classes in allowed easier development because of reduced side effects to worry about.

5.1.1 Performance Evaluation

Performance-wise, the program did not show any slowdown problems on the different computers tested, and none of the users reported performance problems. The main limit on the amount of information that can be stored on the diagram is its size. As some of the users noted, the diagram method begins to fail as the diagram fills up. While scrolling the diagram is available past the limit's of the screen's resolution, losing the ability to see the whole diagram at once severely hinders usability. I believe that this is a limit of the 'blobs-and-lines' solution, although a couple of workarounds could be useful.

One workaround would be to implement a zoom feature. This would allow the user to temporarily see the whole structure of the diagram at the expense of details, which would be slightly better than being limited to scrolling around the diagram. However, it would take significant amounts of coding and personal attempts at implementing a zoom feature were not successful.

Another workaround would be to implement one of the user suggestions (p33, user (3), point 7), of being able to temporarily view the diagram in terms of its class structure, and be able to 'fold away' certain branches of the subclass relationship tree. This would require a significant coding effort, including a layout engine for the class hierarchy view and the ability to quickly switch between views.

Both of these workarounds would significantly improve usability, but still don't solve the scaling problem inherent in the program, in that neither allows viewing the detail of the whole diagram.

5.2 Heuristic Evaluation

One way of evaluating the project is through its adherence to a set of generally-accepted guidelines, or heuristics, which form a set of non-functional requirements. Nielsen's heuristics are one such set, reproduced here from his papers. These heuristics are also available from [10].

Visibility of system status: *The system should always keep users informed about what is going on, through appropriate feedback within reasonable time.*

This is a part of the 'Least Representational Gap principle mentioned earlier. There should be obvious parallels between the UI appearance and and the current program state. In different parts of the program, this is done very well and not at all. For example, most of the UI elements directly correspond to model state, so the user is always aware of "what is going on" when they are editing the diagram. However, I made an oversight in the file I/O parts of the program. These do not

provide any feedback while a file is opening or being saved, apart from a message in the status bar such as "Loading file X." This means that a network problem on a networked file system, or an attempt to load an extremely large file, could result in the program silently hanging. While these problems have never appeared in any personal or user testing, this is a potential usability issue.

Match between system and the real world: *The system should speak the user's language, with words, phrases and concepts familiar to the user, rather than system-oriented terms. Follow real-world conventions, making information appear in a natural and logical order.*
This is also a part of the 'Least Representational Gap' principle. Apart from an issue mentioned in 'Consistency and standards' below, real-world conventions were followed where 'real-world' is in the domain of ontology creation. The application was also designed to have a natural left-to-right flow for information, with the word list on the left, the diagram in the middle and the annotations window on the right matching the usual workflow.

User control and freedom: *Users often choose system functions by mistake and will need a clearly marked "emergency exit" to leave the unwanted state without having to go through an extended dialogue. Support undo and redo.*
Undo and redo: supported. Apart from that, there are not many alternative system states that the user will need to exit, apart from the open/save file dialogues, which have their own cancel buttons built in.

Consistency and standards: *Users should not have to wonder whether different words, situations, or actions mean the same thing. Follow platform conventions.*
This was performed decently, although there was some feedback pointing out some confusion that users had: particularly p33 user (3) point 3, regarding the direction of subclass relationship arrows. This is potentially unresolvable, because UML and Protégé's plugin OWLViz use a subclass-to-superclass direction arrow, while my supervisor and Protégé's plugin OntoGraph (part of the inspiration for the program) use a superclass-to-subclass direction arrow. Perhaps this should be a configurable user preference.

Another example is point 5, which is the result of some internal renaming. Originally, all relationship classes were called properties in the source code, even the subclass and instantiation relationships. This was as a result of early confusion about the meaning of OWL's object properties. Later, properties were renamed to relationships, and the 'relationship property' to object property, but forgot to change the name on the UI button.

Error prevention: *Even better than good error messages is a careful design which prevents a problem from occurring in the first place. Either eliminate error-prone conditions or check for them and present users with a confirmation option before they commit to the action.*

There are a few places in the program where it will attempt to check for common error conditions, stop the UI action if one is found and put a message into the status bar to explain what has happened. Some examples are checking for duplicate names when creating new blobs, or checking for subclass cycles when adding new relationships. The program was limited to a few, very obvious, errors that would be caught, since most of the things that are possible to state in OWL can be reasonable. For example, one user was surprised that the program supported "punning"[6]. Using punning may be an error in some ontologies, but 'naive' OWL API usage and minimal error checking meant that it was available when needed.

Recognition rather than recall: *Minimise the user's memory load by making objects, actions, and options visible. The user should not have to remember information from one part of the dialogue to another. Instructions for use of the system should be visible or easily retrievable whenever appropriate.*

[.. and] Aesthetic and minimalist design: *Dialogues should not contain information which is irrelevant or rarely needed. Every extra unit of information in a dialogue competes with the relevant units of information and diminishes their relative visibility.*
These two goals were achieved easily as a result of the program design, with very few dialogues present, and the majority of the program's state visible at any one time. One area for improvement here is the annotation display; annotations are only displayed for the currently selected object and there is not any indication on the diagram as to which objects are annotated or not, so the user needs to remember their annotation progress.

Flexibility and efficiency of use: *Accelerators—unseen by the novice user —may often speed up the interaction for the expert user such that the system can cater to both inexperienced and experienced users. Allow users to tailor frequent actions.*
Unfortunately, this is one of the features that was often meant to be added to the program but fell to the wayside, due to prioritisation of features for the official demonstration—using accelerators during the demonstration would have been counter-productive since it would be less obvious to the viewer exactly what was happening. However, the program design—with most tasks bundled up into Action objects to be easily distributed across buttons, menus and accelerator keys—should mean that this feature is easy to add in future work.

Help users recognise, diagnose, and recover from errors: *Error messages should be expressed in plain language (no codes), precisely indicate the problem, and constructively suggest a solution.*
In a similar way to "Visibility of system status," this is done both well and not at all well in different circumstances. For error conditions that

[6] Basically, the ability in OWL for a name to refer to two different things, or two different versions of the same thing. See [11] for more details.

are known and handled by the program (for example, an unopenable file or a misplaced subclass line), it will stop the particular action, put a simple message in the status bar or a pop-up window, and continue running. Unexpected or uncaught exceptions, however, are handled by a more detailed dialogue which shows the full exception details for debugging purposes; alongside more generic instructions which state that the current action has failed and that the program may or may not continue working as normal. None of the test users reported seeing this second error dialogue, though.

Help and documentation: *Even though it is better if the system can be used without documentation, it may be necessary to provide help and documentation. Any such information should be easy to search, focused on the user's task, list concrete steps to be carried out, and not be too large.*

The main source of documentation came from a 'Readme' file distributed alongside the program. This was not ideal, as it was produced at the last minute, and should have been available through some menu in the program itself. However, it seems to have been sufficient for most of the test users, apart from a few details here and there. A bigger problem is that the program is fairly incomprehensible without it; feedback suggested labelling the different areas of the diagram to avoid confusion.

5.3 User Evaluation

During the late stages of the project, I had the opportunity to show it to a potential user. In this case, the program was just demonstrated without the user trying it out. This was extremely helpful, since the user was both positive about the project but also had a good idea of what should be worked—on in the last few weeks available—in order to make it better. Most of the points made are described above under design changes. After the demonstration, the project was distributed among a few other users in a 'live' situation, where I was not able to guide the users.

When preparing a packaged version of the project for these readers, I realised that some sort of introduction to the program would be necessary, so I quickly wrote up a small 'Readme' file for them to read. This should really have been included in the program, in the form of a tutorial or help file at least. At best, this Readme file would not be necessary. Some feedback from these users is listed in the next section.

In hindsight, I these user feedback sessions should have been much more frequent. I was initially reluctant (and nervous) to hand out the project in an unfinished state, but the motivation and direction received from the user feedback was extremely valuable.

One issue that came up during the official project demonstration was the line display: for someone with a Software Engineering background, all the lines looked like they were indicating subclass relationships, with the only difference between most of them was the colour. Having greater differences between lines would also aid colour-blind people; and indeed would speed up line recognition even for people with perfect sight.

5.3.1 User Feedback

(1)

The interface feels nicely intuitive and simple to use. It is a useful tool for quickly prototyping ontologies.
It seems to work well, i made a little fish ontology (attached), which is all weird and wrong.
But subclass and instanceof work fine, so does disjoint.
The properties table is a little confusing because you have to click on an invisible first row to define a property assertion.
Save to SKOT and OWL format works fine (i loaded it into protege).
The owl export save window, doesn't remember where you last saved, which is always a little frustrating
enhancements
Remember where you last saved the SKOT output (i.e.when you click "save as", it takes you back to where you last saved as) and the OWL output
Try and make the first row in the properties table visible without clicking on it, I know it says about this in the docs, but it looks like you need to click a button to define a property assertion

(2)

I totally agree with X and Y's comments. Some comments/extensions on the tool:
1. It would be nice when importing an ontology to also create the graph of it (in small and simple ontologies that don't have complex class expressions).
2. As the others mentioned, a problem might arise in case of a bigger ontology and how to represent those terms in the canvas. But for small ontologies it is a really handy tool. It can be used for educational purposes (e.g. demonstrating basic features of ontologies and having a simpler tool for beginners rather than using Protege).
3. In case that the search function in the terms is difficult to implement it would be good to sort the terms alphabetically.
4. Have drag and drop of a term in the canvas or having a keyboard shortcut for the insert button (e.g. ctrl+I).
5. Some basic reasoning functionality would be nice (e.g. just for checking the inconsistency of an ontology and drawing terms that are unsatisfiable with red colour in the graph/terms list). It would be even better if it could create the inferred graph of the ontology (as a future work)
6. I think you don't have the ability to create complex class expressions.
Really nice tool and quite impressed with the functionalities!

(3)

This is really quite nice, I'm impressed! In addition to everything Y said here are some more comments:
1. Would be good to be able to search/filter the imported word list
2. Hooking this up to some term recognition tool would be really great, I was building an ontology about guitars so I ran the wikipedia page on guitars through Termine to generate my word list. Would be nice if I could do this directly in SKOT.
3. The subClassOf arrow points in the wrong direction, the arrow should follow the direction of the relationship (like in UML). Same goes for instanceOf.
4. On export I wanted an option to just export the terms and relationships in my main canvas, not every term in the list.
5. Was confused by the relation link, i eventually realised that it is used to create an object property and add a domain and range. However, I was initially expecting to use it to create relations between individuals or add restrictions on classes. This probably needs a bit more thinking and work to do it properly. Despite this, I tried my best to get it break by creating weird cycles between individuals and classes, but SKOT seemed to handle things ok and was suprised to see that it supported punning!
6. export to SKOS would be nice addition
7. Preview hierarchy - a little popup window that lets me preview my current asserted class hierarchy. Better still would be if the class hierarchies created on the canvas could be viewed as a tree rather than a graph. When working with lots of terms the canvas soon becomes cluttered, if you used trees then users could expand/collapse the trees to free up space. I would then imagine that I could simply drag terms on top of each other to rapidly build my trees.

(4)

On the fly in a meeting we tried to use SKOT to quickly arrange a very large list of words (about 200). straight out of the box, we thought that the open file button was to open a list of words, and not an existing SKOt or OWL file. this should be clarified.
We eventually realised where to paste the words – this should be labelled in some way – similarly for the canvas itself.
It certianly all worked and very impressively didn't fall over. we gave up when the canvas got too full. However, for a simple sketch tool (and simple means not many classes), this worked well for us. It may be that some kind of zoom would work, but thre is a basic limit to these things.

6 Conclusions

Over the course of this project, SKOT—a Simple Knowledge Organisation Tool has been created, which provides an easy-to-use tool for the early conceptualisation stage of ontology creation. Pre-existing tools were too 'heavy-weight' for the task, and did not provide an interface that allowed the user to easily create and modify the structure of the ontology.

In conclusion, the creation of SKOT has been a success for the most part; with most of the major goals met. While some features are missing that would have been desirable, the implementation means that SKOT is easily extensible and additional features are possible for future work. All of the features necessary for the demonstration were produced and polished in time, but these have pushed back a couple of other features (such as accelerator keys or ATR integration) that may have contributed more to the usability of the end product.

Future development would likely focus on two main areas: extending the formats that SKOT can work with—including SKOS, ATR integration and/or extended OWL import functionality, and improving usability across the application through small tweaks, particularly addressing user feedback given in section 5.3.1.

The main insight gained from this project has been the value of real-life user testing and feedback. With hindsight, user testing would have happened earlier and more often, because of the large amount of motivation and direction it provides to the project.

A Program Output

A.1 SKOT output

This is the SKOT file resulting from the process in the Results section, just before the first import into Protégé.

```
<SKOT uri="http://example.com/university" ><WORDLIST>
 <concept name="University" x="470" y="282" />
 <concept name="Student" x="303" y="202" />
 <concept name="Lecture" x="296" y="325" />
 <concept name="Lecturer" x="288" y="259" />
 <concept name="Undergraduate" x="172" y="165" />
 <concept name="Postgraduate" x="172" y="204" />
 <concept name="Mark" x="111" y="114" />
 <concept name="Person" x="397" y="225" />
 <concept name="TeachingAssistant" x="24" y="235" />
 <concept name="TALecture" x="172" y="336" />
</WORDLIST>
<subclassof domain="Student" range="Person" />
<subclassof domain="Lecturer" range="Person" />
<subclassof domain="Undergraduate" range="Student" />
<subclassof domain="Postgraduate" range="Student" />
<instanceof domain="Mark" range="Undergraduate" />
<relation domain="Person" name="attends" range="University" />
<disjoint domain="Lecturer" range="Student" />
<subclassof domain="TeachingAssistant" range="Postgraduate" />
<subclassof domain="TALecture" range="Lecture" />
<relation domain="Lecture" name="taughtBy" range="Person" />
<annotation qname="rdf:comment" target="&lt;Ontology&gt;"
    value="This is a simple ontology about a University" />
<annotation qname="dc:date" target="&lt;Ontology&gt;" value="2011-03-16" />
<annotation qname="dc:creator" target="&lt;Ontology&gt;" value="Mark" />
<annotation qname="rdf:comment" target="Lecture"
    value="Something that is taught by a lecturer" />
<annotation qname="rdf:comment" target="TALecture"
    value="A lecture that is taughtBy a teachingAssistant" />
</SKOT>
```

A.2 Resulting OWL file

This is the OWL file resulting from the process in the Results section, with definitions added from Protégé.

```
<?xml version="1.0"?>
<rdf:RDF xmlns="http://example.com/university#"
     xml:base="http://example.com/university"
     xmlns:dc="http://purl.org/dc/elements/1.1/"
     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
     xmlns:owl="http://www.w3.org/2002/07/owl#"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
    <owl:Ontology rdf:about="http://example.com/university">
        <dc:date rdf:datatype="http://www.w3.org/2001/XMLSchema#string">2011-03-16</dc:date>
        <dc:creator rdf:datatype="http://www.w3.org/2001/XMLSchema#string">Mark</dc:creator>
       <rdf:comment rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
 This is a simple ontology about a University</rdf:comment>
  </owl:Ontology>
  <!--
  ////////////////////////////////////////////////////////////////////////////////
  //
  // Annotation properties
  //
  ////////////////////////////////////////////////////////////////////////////////
   -->
  <owl:AnnotationProperty rdf:about="http://purl.org/dc/elements/1.1/creator"/>
  <owl:AnnotationProperty rdf:about="http://purl.org/dc/elements/1.1/date"/>
  <owl:AnnotationProperty rdf:about="http://www.w3.org/1999/02/22-rdf-syntax-ns#comment"/>
```

```
<!--
///////////////////////////////////////////////////////////////////////////////
//
// Object Properties
//
///////////////////////////////////////////////////////////////////////////////
 -->

<!-- http://example.com/university#attends -->
<owl:ObjectProperty rdf:about="http://example.com/university#attends">
  <rdfs:domain rdf:resource="http://example.com/university#Person"/>
  <rdfs:range rdf:resource="http://example.com/university#University"/>
</owl:ObjectProperty>

<!-- http://example.com/university#hasRooms -->
<owl:ObjectProperty rdf:about="http://example.com/university#hasRooms">
  <rdfs:range rdf:resource="http://example.com/university#Room"/>
  <rdfs:domain rdf:resource="http://example.com/university#University"/>
</owl:ObjectProperty>

<!-- http://example.com/university#takesPlaceIn -->
<owl:ObjectProperty rdf:about="http://example.com/university#takesPlaceIn">
  <rdfs:domain rdf:resource="http://example.com/university#Lecture"/>
  <rdfs:range rdf:resource="http://example.com/university#Room"/>
</owl:ObjectProperty>

<!-- http://example.com/university#taughtBy -->
<owl:ObjectProperty rdf:about="http://example.com/university#taughtBy">
  <rdfs:domain rdf:resource="http://example.com/university#Lecture"/>
  <rdfs:range rdf:resource="http://example.com/university#Person"/>
</owl:ObjectProperty>

<!--
///////////////////////////////////////////////////////////////////////////////
//
// Classes
//
///////////////////////////////////////////////////////////////////////////////
 -->

<!-- http://example.com/university#Lecture -->
<owl:Class rdf:about="http://example.com/university#Lecture">
  <owl:equivalentClass>
    <owl:Class>
      <owl:intersectionOf rdf:parseType="Collection">
        <rdf:Description rdf:about="http://www.w3.org/2002/07/owl#Thing"/>
        <owl:Restriction>
          <owl:onProperty rdf:resource="http://example.com/university#taughtBy"/>
          <owl:someValuesFrom rdf:resource="http://example.com/university#Lecturer"/>
        </owl:Restriction>
        <owl:Restriction>
          <owl:onProperty rdf:resource="http://example.com/university#taughtBy"/>
          <owl:allValuesFrom rdf:resource="http://example.com/university#Lecturer"/>
        </owl:Restriction>
      </owl:intersectionOf>
    </owl:Class>
  </owl:equivalentClass>
  <rdf:comment rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
Something that is taught by a lecturer</rdf:comment>
</owl:Class>

<!-- http://example.com/university#Lecturer -->
<owl:Class rdf:about="http://example.com/university#Lecturer">
  <rdfs:subClassOf rdf:resource="http://example.com/university#Person"/>
  <owl:disjointWith rdf:resource="http://example.com/university#Student"/>
</owl:Class>

<!-- http://example.com/university#Person -->
<owl:Class rdf:about="http://example.com/university#Person"/>
```

```
  <!-- http://example.com/university#Postgraduate -->
  <owl:Class rdf:about="http://example.com/university#Postgraduate">
    <rdfs:subClassOf rdf:resource="http://example.com/university#Student"/>
  </owl:Class>

  <!-- http://example.com/university#Room -->
  <owl:Class rdf:about="http://example.com/university#Room"/>

  <!-- http://example.com/university#Student -->
  <owl:Class rdf:about="http://example.com/university#Student">
    <rdfs:subClassOf rdf:resource="http://example.com/university#Person"/>
  </owl:Class>

  <!-- http://example.com/university#TALecture -->
  <owl:Class rdf:about="http://example.com/university#TALecture">
    <owl:equivalentClass>
      <owl:Class>
        <owl:intersectionOf rdf:parseType="Collection">
          <rdf:Description rdf:about="http://example.com/university#Lecture"/>
          <owl:Restriction>
            <owl:onProperty rdf:resource="http://example.com/university#taughtBy"/>
            <owl:someValuesFrom rdf:resource="http://example.com/university#TeachingAssistant"/>
          </owl:Restriction>
        </owl:intersectionOf>
      </owl:Class>
    </owl:equivalentClass>
    <rdfs:subClassOf rdf:resource="http://example.com/university#Lecture"/>
    <rdf:comment rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
 A lecture that is taughtBy a teachingAssistant</rdf:comment>
  </owl:Class>

  <!-- http://example.com/university#TeachingAssistant -->
  <owl:Class rdf:about="http://example.com/university#TeachingAssistant">
    <rdfs:subClassOf rdf:resource="http://example.com/university#Postgraduate"/>
  </owl:Class>

  <!-- http://example.com/university#Undergraduate -->
  <owl:Class rdf:about="http://example.com/university#Undergraduate">
    <rdfs:subClassOf rdf:resource="http://example.com/university#Student"/>
  </owl:Class>

  <!-- http://example.com/university#University -->
  <owl:Class rdf:about="http://example.com/university#University"/>

  <!-- http://www.w3.org/2002/07/owl#Thing -->
  <owl:Class rdf:about="http://www.w3.org/2002/07/owl#Thing"/>

  <!--
  ///////////////////////////////////////////////////////////////////////////////////
  //
  // Individuals
  //
  ///////////////////////////////////////////////////////////////////////////////////
   -->

  <!-- http://example.com/university#Mark -->
  <owl:NamedIndividual rdf:about="http://example.com/university#Mark">
    <rdf:type rdf:resource="http://example.com/university#Undergraduate"/>
  </owl:NamedIndividual>
</rdf:RDF>

<!-- Generated by the OWL API (version 3.1.0.1592) http://owlapi.sourceforge.net -->
```

B Initial Plan

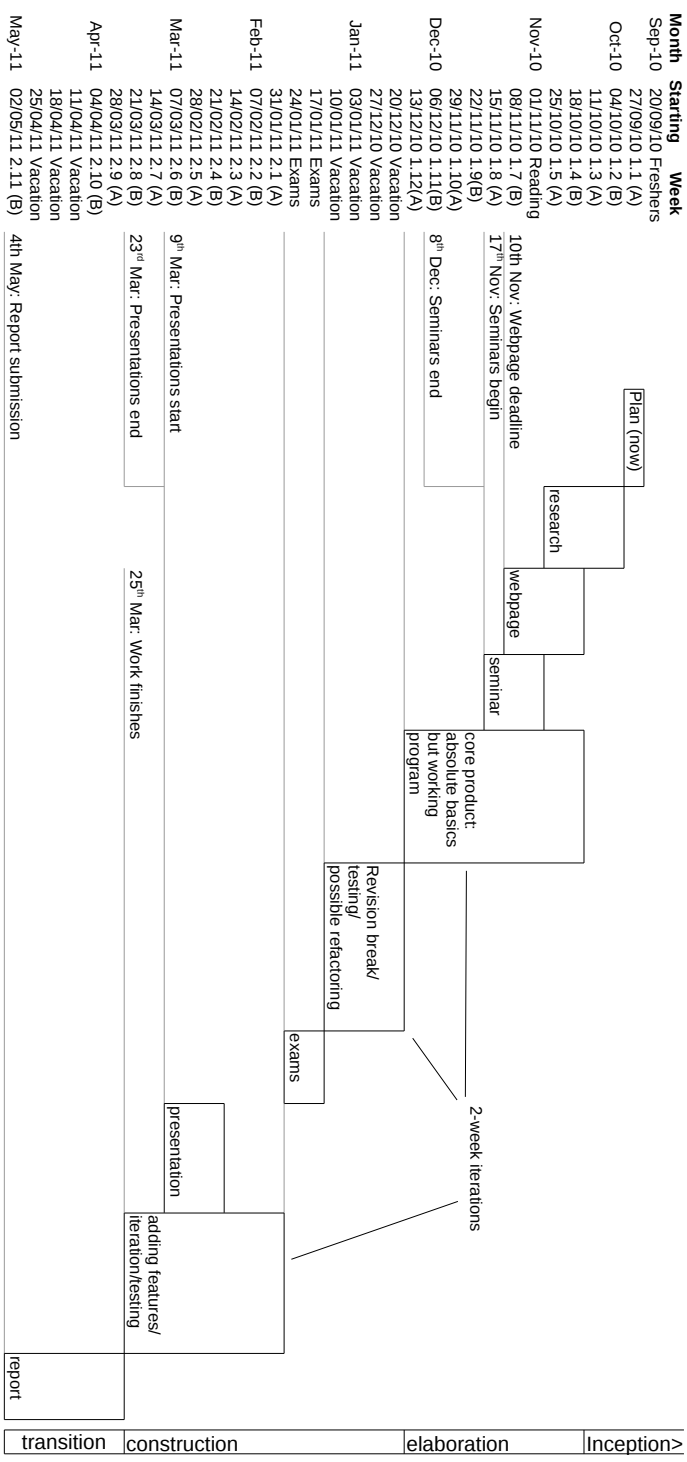

Figure 24: The initial plan, written during the first stages of the project.

C Initial Use Cases

These were the use case descriptions created near the beginning of the project. Since the program is defined by its user interactions, these also formed the basis of the requirements specification.

1. Initially:
   (a) User inputs words into a list: can be one at a time, or comma-separated. User types into text field and presses add.
   (b) User can select one or more words from the list and remove them: they are removed from the diagram if present.
   (c) Ideally each of these can be easily undone rather than have the application ask for confirmation.
2. Once in the list, each word:
   (a) can be removed from the list.
   (b) can be selected; notes or other properties can be added.
   (c) can be moved into the diagram by clicking an "insert" button which means it is put into an available space.
3. Once in the diagram, each word:
   (a) can be linked to other words, and the links labelled.
   (b) can be dragged around.
   (c) (possibly) can be automatically arranged by clicking a button which tries to find a neat arrangement of the words and links.
   (d) can be selected (along with multiple other words) and dragged around.
   (e) can be selected along with multiple other words and grouped together into a meta-entity; such that links within the group are preserved but external links point to the group object.
4. Finally:
   (a) The diagram can be saved for later reuse, which preserves the arrangement.
   (b) The diagram can be exported to some Ontology format.
   (c) (possibly) The program will work as some form of Protégé plugin, and clicking a button will transfer the diagram into Protégé for further editing.

D Requirements

This is a set of requirements for the project based on the initial use cases (section C) as well as those added later.

D.1 Functional Requirements

1. The program must allow the user to import a set of words to be used in the program:
   (a) by typing them manually.
   (b) by pasting a comma-separated list.
   (c) (optionally) by using some integrated ATR service.
2. The program must allow the user to edit, rearrange and link words in a flexible diagram format.
   (a) The words and links must correspond to OWL classes and relationships respectively.
   (b) Customisable links representing object properties must be available.
   (c) Grouping objects should allow the user to collate related words.
3. The program must be able to save diagrams for later use, and load them.
4. The program must allow the user to generate an ontology with the structure described in the diagram.
   (a) The program must support exporting of OWL ontologies.
   (b) The program should support other ontology formats (for example SKOS) in future development and should be easily extensible in terms of output format.
5. The program should allow the user to "link-in" external or imported ontologies, to be able to use the concepts defined therein in the created ontologies.
6. The program should do basic error checking on relationships created. For example:
   (a) Only allowing certain kinds of relationships on "named individual" blobs.
   (b) Checking for redundant or problematic subclass statements such as transitive closures or cycles.
   (c) Checking for duplicate class or relationship names.

D.2 Non-functional Requirements

7. Usability: The program should have a "Low Representational Gap" between the diagram and the underlying ontology structure.
   (a) The program should allow the user to directly manipulate the ontology by modifying the diagram.
   (b) The program should allow the user to easily visualise the structure of the ontology by viewing the diagram.
8. Usability: The program should allow a very non-linear workflow—ontology structure should be able to be modified and refactored on-the-fly.
   (a) The program should allow insertion of new terms throughout the creation process.
   (b) The program should support undo and redo for all operations that change program state.
9. Performance: The program should not limit the user's desired output by way of program slowdown.
   (a) The program should scale well with the amount of complexity on the diagram.
10. Usability: The program should be usable by anyone with knowledge of KR and ontology creation. Specialised knowledge should not be needed.
11. Reliability: The program should not cause user's data or effort to be lost.
12. Usability: The programs should conform to Nielsen's usability heuristics [10] to improve the user experience.

D.3 Additional Requirements

These were added after first user meeting:

13. The program should allow the use to add and edit annotations for objects on the diagram.
   (a) The default RDF(S) and OWL annotations should be available
   (b) These annotations should include the set of Dublin Core Metadata elements as defined in [22].
14. The program should allow the user to import the terms from an already-existing OWL ontology, so that further structure can be added while preserving the existing ontology.
15. The program must allow the user to change the ontology URI.
16. The program should be able to integrate an external web-service such as Termine or Dresden for ATR.

17. Usability: The program should support actions on multiple blobs at once, including:
    (a) Inserting multiple words from the word list.
    (b) Moving multiple blobs around the diagram.
    (c) Adding a relationship from multiple blobs (to one target).
18. Usability: The program should make relationship adding quicker and easier by requiring fewer mouse-clicks.
19. Usability: The program should make adding new blobs much quicker and easier by requiring less mouse movement and clicking.
20. Usability: The word list should be searchable and/or sortable.
21. Supportability: The program should be easily extensible for future development, possibly as a Protégé plugin.

E References

[1] SKOT on assembla.com
URL: https://www.assembla.com/spaces/askot
Last retrieved: 29 Apr 2011

SKOT SVN repository
URL: https://subversion.assembla.com/svn/askot/
Last retrieved: 29 Apr 2011

[2] Craig Larman
Applying UML and Patterns
Prentice Hall PTR, 2004

[3] Gomez-Perez, Fernandez-Lopez and Corcho
Ontological Engineering
Springer, 2004

[4] Termine
URL: http://www.nactem.ac.uk/software/termine/
Last retrieved: 22 Apr 2011

[5] SKOS Reference
URL: http://www.w3.org/TR/skos-reference/
Last retrieved: 22 Apr 2011

[6] Web Ontology Language (OWL)
URL: http://www.w3.org/2004/OWL/
Last retrieved: 25 Apr 2011

[7] The OWL API
URL: http://owlapi.sourceforge.net/
Last retrieved: 25 Apr 2011

[8] Abbot framework for automated testing of Java GUI components and programs
URL: http://abbot.sourceforge.net/doc/overview.shtml
Last retrieved: 23 Apr 2011

[9] Robot (Java Documentation)
URL: http://download.oracle.com/javase/1.4.2/docs/api/java/awt/Robot.html
Last retrieved: 23 Apr 2011

[10] Jacob Nielsen
10 Heuristics for User Interface Design
URL: http://www.useit.com/papers/heuristic/heuristic_list.html
Last retrieved: 24 Apr 2011

[11] Punning - OWL
URL: http://www.w3.org/2007/OWL/wiki/Punning
Last retrieved: 24 Apr 2011

[12] Tim Berners-Lee: Giant Global Graph
URL: http://dig.csail.mit.edu/breadcrumbs/node/215
Last retrieved: 21 Apr 2011

[13] Cory Doctorow: Metacrap
URL: http://www.well.com/~doctorow/metacrap.htm
Last retrieved: 25 Apr 2011

[14] ActionTableCellEditor - a LGPL Swing component
URL: http://jroller.com/santhosh/entry/add_button_to_any_tablecelleditor
Last retrieved: 25 Apr 2011

[15] famfamfam.com: Silk Icons
URL: http://www.famfamfam.com/lab/icons/silk/
Last retrieved: 1 May 2011

[16] OWL Manchester Syntax (w3c Working Group Note)
URL: http://www.w3.org/TR/owl2-manchester-syntax/
Last retrieved: 01 May 2011

[17] Resource Description Framework (RDF)
URL: http://www.w3.org/RDF/
Last retrieved: 3 May 2011

[18] RDF Schema
URL: http://www.w3.org/TR/rdf-schema/
Last retrieved: 03 May 2011

[19] Stevens, Robert and Goble, Carole A. and Bechhofer, Sean (2000)
Ontology-based knowledge representation for bioinformatics
Briefings in Bioinformatics vol 1 no 4, pp398-414

[20] Mycin
URL: http://www.computing.surrey.ac.uk/ai/PROFILE/mycin.html
Last retrieved: 13 Apr 2011

[21] Wikipedia: Terminology Extraction
URL: http://en.wikipedia.org/wiki/Terminology_extraction
Last retrieved: 23 Apr 2011

[22] Dublin Core Metadata Element Set (version 1.1)
URL: http://purl.org/dc/elements/1.1/
Last retrieved: 2 May 2011