Ontology Development*

Kenneth BACLAWSKI
College of Computer Science, Northeastern University
Boston, Massachusetts 02115 USA
`Ken@Baclawski.com`

Abstract

Ontologies are emerging as an important tool for dealing with very large, complex and diverse sources of information. It has also been recognized for some time that ontologies are advantageous for software development. Lyee is a successful software methodology due, in part, because of its incorporation of ontological notions. However, given that ontologies are themselves a form of software, it is important that appropriate attention be paid to developing them. In particular, ontologies must be consistent, provide sufficient coverage of the target domain and be developed at the appropriate level of detail. While there are many ontology development tools, there are very few ontology development methodologies, and those that have been proposed do not address these issues adequately. This is part of a general tendency in computer science to develop programs without designs or even requirements. Because ontologies are closely related to modern object-oriented software design, it is natural to adapt existing object-oriented software development methodologies for the task of ontology development. In this paper, ontology development is compared with software development in general, and with Lyee software development in particular. While a number of features of object-oriented software development methodologies are especially well suited to ontology development, it is also the case that they sometimes conflict with modern ontology languages. However, these conflicts can be reconciled, and the conclusion is that object-oriented software development methodologies such as Lyee show promise as a basis for ontology development methodologies.

Keywords: ontology, Lyee, ontology development, software development methodologies

*© IOS Press. Frontiers in Artificial Intelligence and Applications

1 Introduction

Ontologies are emerging as an important tool for dealing with some of the most important issues facing information processing:

- Ever larger amounts of data are available at much faster rates.
- Data structures are increasing in their complexity.
- Much more diverse sources of information are becoming relevant to each activity.

These trends have also increased interest in automating many activities that were traditionally performed manually. Web-enabled agents represent one technology for addressing this need [40]. These agents can reason about knowledge and can dynamically integrate services at run-time. Formal ontologies are the basis for such agents. In addition, it has been recognized for some time that ontologies are advantageous specifically for software development. One of the papers in the previous workshop in this series addressed this issue [51].

Given the large amount of interest in ontologies, it is surprising that ontology development today is in such a poor state. This is partly due to the large number of logic and ontology languages [19]. But this proliferation is just a symptom of a more serious problem: the rush to produce tools and languages without any clear purpose. Rather than tools being constructed to support methodologies and process models, the methodologies are being created for the sake of understanding how to use the tools. Because of the lack of any generally accepted processes and methodologies, the tools exist independently and have little support for or concern with interoperability. This is surprising since one of the primary purposes of ontologies is to foster sharing and interoperability.

It is easy to be lured into a false sense of security by ad hoc development techniques that work well with small ontologies. Ontologies today can be enormous. The Unified Medical Language System (UMLS) [28] currently contains a taxonomy of over 800,000 concepts, and its taxonomy is only one feature of an ontology that is both large and complex. Systematic development techniques, including good support for modularity and reuse, are essential for developing ontologies of this size and complexity.

Lyee is a successful software methodology due, in part, because of its incorporation of ontological notions. However, given that ontologies are themselves a form of software, it is important that appropriate attention be paid to developing them. In particular, ontologies must be consistent, accurate, provide sufficient coverage of the target domain and be developed at the appropriate level of detail. Ontology development methodologies, such as they are, do not address these issues adequately.

Ontologies are closely related to modern object-oriented software design, although the analogy between the two is not perfect. Nevertheless, they share enough in common to propose to use existing object-oriented software development methodologies for the task of ontology development. In [8] and [9], there is a detailed comparison of the Unified Modeling Language (UML) with the DARPA Agent Markup Language (DAML). These papers consider issues of language and semantics, but they do not consider process and methodology.

In this paper, ontology languages (OLs) and ontology development are compared with object-oriented programming languages (OOPLs), object-oriented modeling languages (OOMLs) and software development, in general, and with Lyee software development in particular. The conclusion is that object-oriented software development methodologies such as Lyee show promise as a basis for ontology development methodologies.

The paper begins in Section 2 with some background material about software, ontologies and the relationship between them. It then considers the various phases of software and ontology development in Section 3. This is followed by three sections dealing with the major features of ontologies. Classes are discussed in Section 4. Relationships are discussed in Section 5. Logic is discussed in Section 6. The paper ends with some concluding remarks.

The individuals and organizations that are sponsoring the software or ontology development are referred to by various names, such as *users* and *customers*. Neither of these are very descriptive of the role that they play in ontology development. In this paper, these individuals and organizations will be called the *stakeholders* of the development process.

2 Background

Representing knowledge is an important part of any system that purports to be knowledge-based. In particular, all artificial intelligence systems must support some kind of knowledge representation (KR). Expressing knowledge in machine-readable form requires that it be represented as data. So it is not surprising that KR languages and data languages have much in common, and both kinds of language have borrowed concepts from each other. As is the case with data languages, most KR languages have the ability to express schemata that define the structure and constraints of data (instances or objects) conforming to the schema. A schema in a KR language is called an *ontology* [25] [26] [41] [46]. KR languages in general, and ontologies in particular, were derived from corresponding notions in Philosophy. See the classical work by Bunge [15] [16] as well as more recent work by Wand [63], and by Guarino, Uschold and their colleagues [24] [60]. Ontologies are fundamental for communication between individuals in a community. They make it possible for individuals to share information in a meaningful way. Formal ontologies adapt this idea to automated entities (such as programs, agents or databases). Formal ontologies are useful even for people, because informal and implicit assumptions often result in misunderstandings. Furthermore, "meaningful names" do not, in themselves, convey meaning because their meaning may be substantially different for different members of the community. The relationships between concepts must also be explicit. As stated by Wittgenstein (Proposition 3.3 in [66]), "Only the proposition has sense; only in the context of a proposition has a name meaning." Sharing of information between disparate entities (whether people or programs) is a fundamental purpose of an ontology.

KR languages can be given a rough classification into three categories:

- Logical languages. These languages express knowledge as logical statements. One of the best-known examples of such a KR language is the Knowledge Interchange Format (KIF) [22].
- Frame-based languages. These languages are similar to object-oriented database languages.
- Graph-based languages. These include semantic networks and conceptual graphs. Knowledge is represented using nodes and links between the nodes. Sowa's conceptual graph language is a good example of this [57].

Perhaps because of the strong analogy between hypertext and semantic networks, most recent KR languages have been graph-based.

If one identifies semantic network nodes with Web resources (specified by Universal Resource Identifiers or URIs) and semantic network links with hypertext links, then the result appears to form a basis for expressing knowledge representations that could span the entire World Wide Web. However, links in hypertext, in much the same manner as goto's in programming languages, are at much too low a level of abstraction for ontologies. The Resource Description Framework (RDF) [38] was introduced under the auspices of the World Wide Web Consortium as a means of remedying this inadequacy. RDF is a recommendation within the XML suite of standards. It introduces relationships with formally defined semantics, that can link Web resources. RDF is developing quickly [18]. There is now an RDF Schema language, and there are many tools and products that can process RDF and RDF Schema.

The DARPA Agent Markup Language (DAML) [17] [27] has introduced an OL called DAML+OIL that extends RDF and RDF Schema and that expresses a much richer variety of constraints. DAML+OIL is now being superseded by the Web Ontology Language (OWL) [56], under the auspices of the World Wide Web Consortium. OWL has three versions (or levels), called OWL Lite, OWL-DL and OWL Full. The distinction between these three is explained in [61]. Figure 1 shows an example of an OWL Lite ontology in diagrammatic form. The following is part of what this ontology would look like when specified in the XML format for OWL:

```
<owl:Class rdf:ID="Invoice"/>
<owl:Class rdf:ID="Item"/>
```

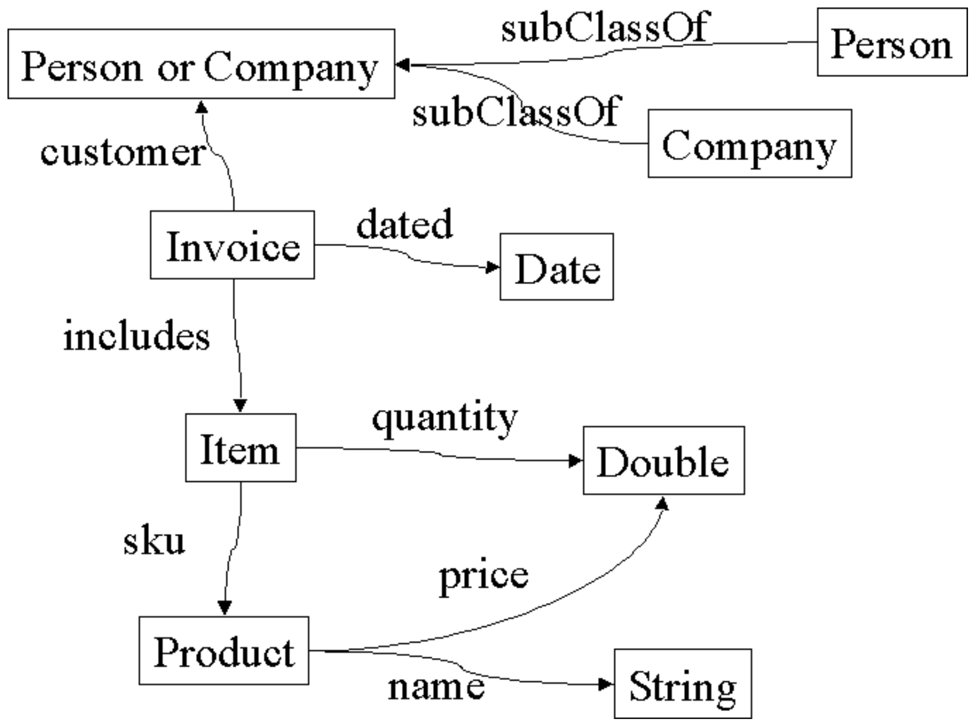

Figure 1: Diagram of an OWL ontology for invoices. The boxes are classes, and the arrows linking them are properties. The PersonOrCompany class is the union of the Person and Company classes.

```
<owl:Class rdf:ID="Product"/>
<owl:Class rdf:ID="Person"/>
<owl:Class rdf:ID="Company"/>
<owl:FunctionalProperty rdf:ID="purchasedBy">
  <rdfs:domain rdf:resource="#Invoice"/>
  <rdfs:range>
    <owl:Class rdf:ID="PersonOrCompany">
      <owl:unionOf parseType="Collection">
        <owl:Class rdf:about="#Person"/>
        <owl:Class rdf:about="#Company"/>
      </owl:unionOf>
    </owl:Class>
  </rdfs:range>
</owl:FunctionalProperty>
<owl:InverseFunctionalProperty rdf:ID="includes">
  <rdfs:domain rdf:resource="#Invoice"/>
  <rdfs:range rdf:resource="#Item"/>
</owl:InverseFunctionalProperty>
<owl:FunctionalProperty rdf:ID="sku">
  <rdfs:domain rdf:resource="#Item"/>
  <rdfs:range rdf:resource="#Product"/>
</owl:FunctionalProperty>
```

A functional property is one that is mathematically a (partial) function, i.e., an object can be related to at most one other object. An inverse functional property is a property whose inverse (i.e., the property that interchanges the roles of the domain and range) is functional. The `includes` property is more than just an inverse functional property. An invoice is a "composite" or "aggregation" of its items. The notion of a composite can be specified in OOMLs, but is not currently available in Web-based ontology languages.

One can regard RDF, RDF Schema, DAML+OIL and the OWL levels as being a series of progressively more expressive OL languages. The XTM Topic Maps (XTM) language is another XML-based OL that has a very different history and semantics [67]. XTM is a graph-based OL similar to RDF. It differs in the following respects:

- XTM relationships (called *associations*) can have any number of roles. By contrast, the RDF languages only support typed binary relationships (i.e., ternary relationships in which one role, called the *predicate*, is distinguished from the other two).
- XTM has a notion of scope or context that the RDF languages lack.
- The RDF languages have a formal semantics. XTM only has a formal meta-model.

Data conforming to an ontology is often referred to as an *annotation* or as *markup*, since it typically abstracts or annotates some natural language text (or more generally a hypertext document). The following is an invoice expressed as an annotation for the ontology in Figure 1 as expressed in the OWL language:

```
<Person rdf:ID="C4365"/>
<Product rdf:ID="P284"/>
<Product rdf:ID="P683"/>
<Invoice dated="2003-06-21">
  <purchasedBy rdf:resource="#C4365"/>
  <includes>
    <Item quantity="2.0">
      <sku rdf:resource="#P284"/>
    </Item>
    <Item quantity="1.0">
      <sku rdf:resource="#P683"/>
    </Item>
  </includes>
</Invoice>
```

An ontology may include vocabulary terms, taxonomies, relations, rules and assertions, all of which can be represented in a machine-readable form. Accordingly, ontologies may be computer artifacts, but they are not programs, at least not in the usual sense of the term.

When computers were large, slow and very expensive, there was never an issue about what constituted computer software: Software consists of computer programs. With the rapid expansion of computers into everyday life, it is no longer so easy to characterize software. It is now possible for a company to create a large, sophisticated software system without any programming at all in the traditional sense. LyeeAll is a good example of a tool that helps make this possible. Conversely, artifacts that one might not think of as software can involve sophisticated programming and software engineering. This paper, for example, was developed using a number of XML-based programming languages as well as design, configuration control, version control and testing techniques that one would normally associate with traditional software development rather than document preparation. Clearly, software is increasingly much more than just computer programs.

Unfortunately, the fiction that software consists of computer programs is still deeply embedded in most Computer Science curricula. Students still have a strong tendency to "program first and design later." In many cases, the requirements and analysis are completely omitted. Furthermore there is a significant bias in favor of programs over other artifacts in the software development process. One exception is Bjørner who includes business modeling as an important part of software engineering [11].

Software process models such as the Model Driven Architecture (MDA) [42] and Lyee [43] attempt to counter this tendency. MDA development focuses first on the functionality and behavior of a distributed application or system, undistorted by idiosyncrasies of the technology or technologies in which it will be implemented. MDA separates implementation details from business functions. I have been involved in the introduction of formal methods (specifically using category theory from mathematics) to the MDA [54] [55]. I have also been very active in running conferences and workshops that foster precise, explicit, and elegant specifications of business and system semantics [4] [5] [6] [7] [32].

Lyee has an underlying theoretical axiomatic framework [44] that is founded upon (or at least inspired by) cognitive models of human behavior, including human intention and consciousness.

The MDA emphasizes the creation of many artifacts (called *models*) from which programs are derived. It is reasonable to consider these artifacts as being a form of software. Formal ontologies are another example of artifacts that are not programs but which may reasonably be classified as being software.

Given the bias mentioned above, it is not so surprising that the Computer Science community has approached ontology development from a tools point of view. It is yet another example of the tendency to write programs before they are designed or even have requirements. I have been working to counter this tendency by adapting existing software development methodologies for ontology development [8] [9] [36]. This paper continues this effort.

3 The Development Process

According to Pressman [52], the foundation of software engineering is the process layer. Software engineering methods are built upon the process layer, and software engineering tools provide support for the methods and process. In the case of ontology development, there are a great number of tools, some methods, and only a few explicit process models. For a survey of the many tools, see [20]. A good source for methods is "Ontology 101" by Noy and McGuinness [45]. The only complete process model is the Knowledge Analysis and Documentation System (KADS) [59] which is intended primarily for developing expert systems, not ontologies. KADS has been mostly ignored by current ontology efforts. In fact, support for any kind of knowledge engineering methodology is rare [19].

The following table gives an approximate correspondence between the typical activities of ontology development with the main activities that occur in software engineering.

Ontology Development	**Software Engineering**
Acquire domain knowledge from experts	Requirements analysis
Organize the ontology	Software design
Elaborate the ontology	Implementation
Check consistency	Test
Verify by domain experts	Test

In this section we compare the activities in the rows of the table above.

3.1 Requirements

Modern software development process models include a requirements phase, and there is a substantial literature on this subject, including a journal exclusively devoted to it [39]. This activity involves domain experts (also known as *subject matter experts* or SMEs). The corresponding activity in ontology development is the acquisition of the domain knowledge that will be formalized in the ontology. SMEs play a crucial role in this activity.

While SMEs are involved in both requirements analysis and knowledge acquisition, their role is very different in the two cases. Requirements analysis not only involves acquiring an understanding of the domain, it also determines the required function, behavior, performance and interfaces. Ontology development typically omits these latter requirements. In most cases, ontology development projects have no explicitly stated purpose beyond the acquisition of the domain knowledge. When there is a stated purpose, it is usually too generic to be useful in the ontology development process. Having a detailed stated purpose would not entirely determine the required function, behavior, performance and interfaces, but it would certainly help.

This situation is unfortunate because it has been known at least since the middle of the nineteenth century [65] that the design of an ontology depends on its purpose and viewpoint. Yet this fact has been forgotten and painfully rediscovered frequently since then. It is important for any software development project to have precisely stated requirements, and ontology development is no different. However, it is common (for example, [19]) to view ontology development as beginning with knowledge acquisition and to mention purpose only during the elaboration step, if at all. The viewpoint is seldom considered in ontology development. By contrast, OOMLs such as the Reference Model for Open Distributed Processing (RM-ODP) have an explicit notion of viewpoint [31].

By directly involving the stakeholders in the design process, Lyee offers the possibility of capturing the purpose of the development effort more effectively than processes that separate the stakeholders from the developers. The latter processes are especially ineffective when the purpose is left unstated, as is typical of ontology development efforts today.

Because the amount of knowledge to be acquired may be very large, it is important for the knowledge to be structured. It is also important to track the source of knowledge so that one can determine its trustworthiness. Both of these issues can be handled by introducing a suitable notion of scope. Most OOMLs have a well developed notion of scoping, which is not the case for all modern ontology languages.

3.2 Design and Implementation

There are some methodologies for organizing and elaborating ontologies. See for example, Ontology 101 [45]. These methodologies are generally tool-specific. The Ontology 101 paper is specific to the Protégé-2000 tool from Stanford. What seems to be lacking in this phase of ontology development is a systematic and tool-independent methodology that is then supported by tools. In this section, some possibilities are introduced that could help achieve this goal.

3.2.1 Design Patterns and Ontology Reuse

In accordance with the general theme of this paper, techniques are proposed by analogy with software engineering. The first of these is Design Patterns. The notion of business patterns for modeling languages was introduced about 10 years ago by Kilov and

Ross [34] [33]. A similar notion was introduced to OOPL at about the same time, and it has been a very successful and popular methodology [21]. This approach can be applied to ontology development. Indeed, many of the design patterns in [21] and later books could be adapted to the needs of ontology development. In fact, virtually any well developed (and preferably relatively small) ontology can serve as a design pattern. All that is necessary is a systematic methodology for mapping (transforming) one ontology into another.

The process whereby one ontology is transformed to another is therefore the key to design patterns, and the reuse of ontologies in general. The mathematical basis for ontology reuse is the concept of a *colimit* from category theory. Some tools have been developed that support colimits, such as OBJ [23] and Specware$^{\mathrm{TM}}$ [58] [62]. Unfortunately, these tools require a great deal of mathematical sophistication. These tools are difficult to use even for an experienced, well-educated software engineer. A tool that would be usable by an SME who is not a software engineer seems far out of reach.

To create an ontology reuse methodology that is logically well founded as well as easy to use, it is necessary to consider the cognitive aspects of this process. Ontology transformation has a cognitive interpretation in terms of *metaphor*. This was first developed by Indurkhya [29]. The close connection of ontology reuse with cognition as developed by Indurkhya has yet to be exploited by any ontology development methodology.

Ontology reuse is also very useful for ontology management. As ontologies become larger, they become progressively more difficult to manage. Some kind of modularity technique is necessary, not only for developing the ontology but for maintaining it after it is developed. The colimit is very well suited for supporting ontology modularity, at least theoretically. What remains is the problem of making this technique practical as well.

3.2.2 Refactoring

The second major technique is called Refactoring. While this technique is not nearly as well understood as Design Patterns, some theoretical models have been developed that show promise, as for example in [47] and [1]. Refactoring is concerned with adjusting a software design when it has been found to be inadequate. The Lyee methodology achieves this by executing the program as it is being designed, and the design can be modified to fit the stakeholders' intentions.

These same issues often arise in ontology development as well. These issues are generally grouped under the term *ontological commitment*, but little has been done to formalize the processes that take place. Some examples of these processes include:

- Modifying property domains and ranges. This is the process that gives refactoring its name. A set of properties on subclasses can be changed to a single property on a superclass, or a property on one class can be shifted to become a property on a related class.

- Reification. This is the process whereby concepts that are not classes are given class status. For example, a relationship can be reified to become a class. Reifying a binary relationship will replace the relationship with a class and two relationships.
- Unreification. This is the reverse process to reification.
- Metalevel shifting. This process is very powerful, yet seldom mentioned. In this process, metalevel notions, such as that of a class or an attribute, are instantiated as ordinary notions. Most OOPLs allow the structure of the program to be accessed by the program. This is known as *reflection*, and it is one example of the process of metalevel shifting.

As an example of refactoring, consider the Invoice ontology in Figure 1. It actually may have a subtle error, and a knowledge engineer who was not an SME might easily miss it. In this design the `price` of an `Item` is that of the `Product` that was `purchased by` the customer. This may be operationally incorrect. While the `price` of an `Item` is related to that of the `Product`, there are any number of ways that it may differ. For example, there may be a discount. Furthermore, if the `price` of a `Product` subsequently changes, that should not affect any previously existing invoices. To deal with these possibilities, the design should be refactored as shown in Figure 2.

As another example of refactoring, consider the `price` of a `Product`. The stakeholders may require that this property have multiple values each with its own attributes, such as the date it was introduced, the date when it expires, who authorized the price, and so on. When a property is functional, one can attach such additional attributes to the domain class. However, this breaks down when the property is multi-valued. The most effective technique for dealing with these requirements is to reify the property. The `price` property is replaced by a `Price` class and two new properties, as in Figure 3.

Examples of the metalevel shifting technique are given in Section 4.1 and Section 4.2 below.

One interesting aspect of the invoice refactoring is that problems are often most easily found by viewing the ontology operationally. Ontologies do not specify operations (i.e., behavior); indeed, one of their strengths is that they are not bound by operational considerations. However, even if one does grant this, it is still the case that the use of operational language assists in finding errors.

Lyee, in theory, seems to be especially well suited to this process. However, it certainly cannot do so in its present form. The problem is exactly the issue mentioned above: ontologies are not operational even though their suitability for a purpose generally involves some kind of operational reasoning. Where can Lyee (or any development process) obtain the operations that are necessary for the process? This issue arises again in testing and validation in Section 3.3 below, and at least a partial answer is proposed.

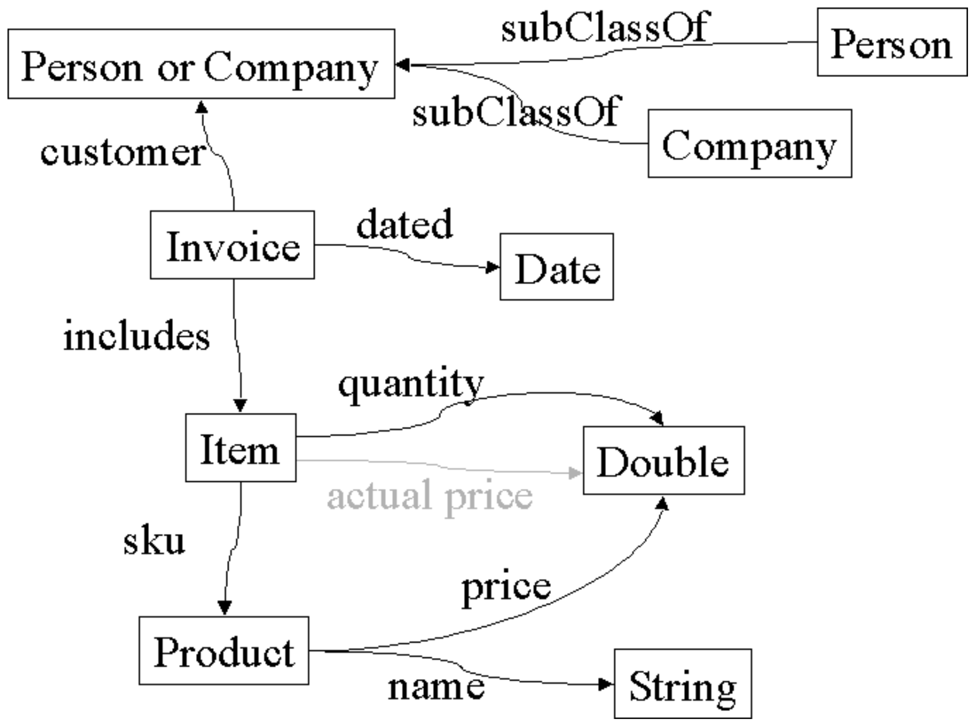

Figure 2: Refactored ontology for invoices. The new property is shown in gray.

3.3 Testing and Validation

Testing and validation have always been accepted as an important part of ontology development. However, these activities are casual and informal in most cases. This is remarkable because ontology development is much more formal than software development. The most likely reason for this seeming contradiction is the fact that ontology development usually has no precisely stated purpose. As a result, one cannot formally test that the purpose has been achieved.

One activity (and perhaps the only one) that can be performed without explicitly specifying requirements is consistency checking. Obviously, consistency is fundamental for any formal system that supports inference. If a formal system is inconsistent, then every statement can be proven true (and also proven false, since true = false in an inconsistent formal system). Accordingly, inference is useless when a formal system is inconsistent.

Despite the importance of consistency for ontology languages, both the languages themselves and ontologies expressed in those languages are often checked manually. Substantial inconsistencies have been found in KIF, RDF and DAML+OIL long after these languages were proposed (and even accepted) as standards [10]. The inconsistency of KIF is especially noteworthy given that KIF has been a draft standard for many years. Furthermore,

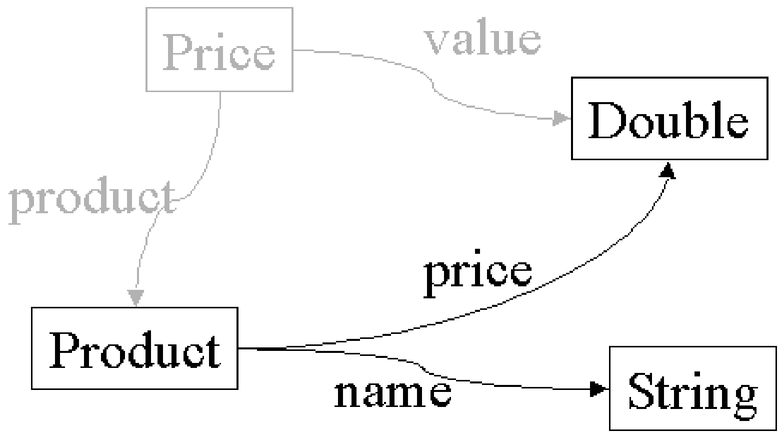

Figure 3: Reification of the price of a product. The reification is shown in gray.

the inconsistency was not due to some obscure axioms, but occurred in the axioms for lists, arguably the most fundamental axioms of all. Specifically, the inconsistency was in the axioms for the *first* element of a list and for the *rest* of a list (cf. [10]).

Consistency checking is one characteristic of ontology development that distinguishes it from general software development. While it would certainly be desirable to prove that software is formally correct, it is not common to do so, largely because most programming languages are not formally defined.

On the other hand, it is also desirable to test ontologies as one would test programs. This is certainly possible, but it is not common to do so. The reason is that performing tests requires that the requirements be explicitly stated. As mentioned above in Section 3.1, it is not common to specify requirements, so it is not possible to perform meaningful acceptance tests and validations.

In the absence of explicitly stated requirements, the only alternative is to involve the stakeholders directly in the testing process. Testing would proceed by presenting a stakeholder or group of stakeholders with randomly generated scenarios designed to determine the accuracy of the ontology. Lyee uses a similar technique for its testing and validation process. For example, if a property was not constrained to be functional, then the system would generate a scenario in which the property links a single object with two provably different objects. The stakeholder(s) would then be asked if this is acceptable. It is challenging to generate a sufficient number of scenarios so that one can feel confident in the ontology. The error in the Invoice ontology in Figure 1 is an example of a flaw that one would not find using scenarios based on violations of static constraints such as cardinality constraints. However, it is reasonable to predict that a solid theoretical framework is possible and that this challenge can be met.

4 Classes

The notion of class is fundamental both to object-oriented software and to modern ontologies. While both concepts arose from Philosophy, they have diverged from one another and are now substantially different. OLs define a class to be a set (i.e., a one-place predicate) while object-oriented classes have considerably more structure. In this section we discuss a number of features of classes that distinguish OLs from OOPLs.

4.1 Sets or Templates

In OOPLs a *class* is a template for constructing objects that have a specified data structure and associated functions. Objects are instances of a class because they have been explicitly constructed to be instances. An object never changes its class throughout its lifetime.

OLs and many OOMLs have a very different point of view. For such a language, a class is a set of objects. Membership of an object in a set can be explicitly declared, but it can also be inferred. For example, one can define a class `Teenager` to be the subclass of `Person` such that the `age` attribute is between 13 and 19. See Figure 4. Thus an object can change from being not in the class, to being in the class, and back to being not in the class again. Objects can also be instances of several classes that are not otherwise related to one another. By contrast, an object in an OOPL is constructed as an instance of exactly one class, and the object will only be an instance of superclasses of this class.

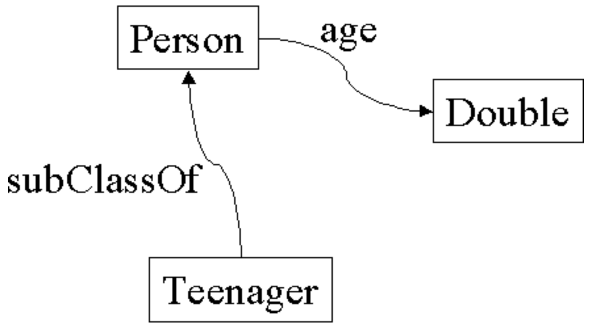

Figure 4: An example of a class such that an instance can subsequently cease to be an instance. A teenager can cease to be a teenager without ceasing to be a person. Object-oriented programming languages do not allow the type of an object to change.

In general, an OL class can be defined in terms of other classes using the set-theoretic operations of union, intersection and complement. An OL class can also be restricted to take particular attribute values (e.g., the `Teenager` class defined above). The number of values of an attribute can also be restricted (using a cardinality restriction). These are called *class constructors*. Note that this is very different from the constructors used in OOPLs. The former constructs classes while the latter constructs objects.

Since OL classes are sets, the subclass concept has a simple interpretation as the subset relationship. One immediate consequence of this interpretation is that the subclass relationship is transitive. To some degree one can also view an OOPL class as a set; namely the set of instances of the class. However, this is not always accurate, and the failure of this interpretation becomes clear when one considers the subclass relationship. This relationship cannot, in general, be interpreted as the subset relationship. It satisfies the weaker notion of *substitutability*: an instance of a subclass can be used as if it is also an instance of an immediate superclass. While substitutability appears to be the same as the subset relationship, it does not imply transitivity.

Reconciling the points of view of OLs and OOPLs is not easy. However, the issue arises only if OL classes are intended to be realized as OOPL classes. Even if an ontology is to be realized using an OOPL, it is not necessary for there to be an exact correspondence between OL classes and OOPL classes. The ontology should not be restricted by technological considerations. Thus it is important to understand the intentions of the stakeholders. If such a realization is part of the purpose of the ontology, then the design should not use classes for those parts of the ontology that would violate the constraints of the OOPL class concept. The design can be modified by using the metalevel shifting technique that was discussed in Section 3.2.2 above.

4.2 Behavioral versus Set-Theoretic Semantics

OOPL classes are much more than just sets of objects. They also endow objects with behavior as defined by functions called *methods*. This has led to frequent confusion concerning the meaning of a class hierarchy. As explained by [3], the difficulty in modeling a hierarchy of concepts using software environments is best exemplified by the failure of attempts to find an epistemological foundation of the subclass concept. Brachman [12] [13] [14] points out that there are six different kinds of generic-generic relations and four different kinds of generic-individual relations all of which are grouped together under the IS-A label.

The classic example of the resulting confusion is the question of whether square is a subclass of rectangle. From a set-theoretical (logical) point of view, it seems obvious that squares are a proper subset (and therefore subclass) of rectangles as in design A of Figure 5. However, according to one view of cognition and concepts, objects can only be defined by specifying the possible ways of acting on them [29] [48]. For instance, Piaget [48] showed that the child *constructs* the notion of object permanence in terms of his or her own actions.

Accordingly, the concept square, when defined to be the set of all squares without any actions on them, is not the same as the concept square in which the objects are allowed to

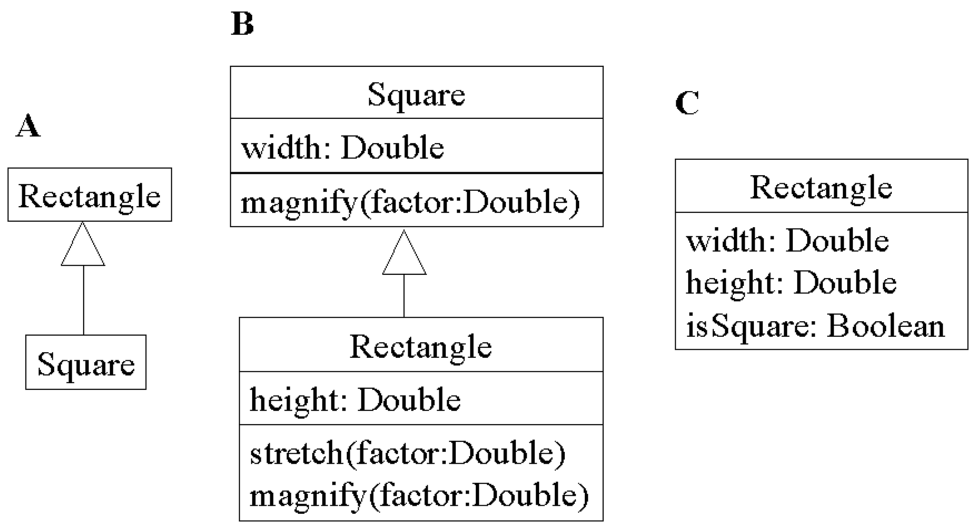

Figure 5: Three designs for modeling squares and rectangles. In design A, the Square class is a subset of the Rectangle class. The attributes and behavior are defined separately. In design B, the Rectangle class is an extension of the Square class. All attributes and behavior are shown (including the overloading of the magnify method). In design C, an attribute

be shrunk or stretched. In fact, it has been found that children's concepts of square and rectangle undergo several transformations as the child's repertoire of operations increases [49] [50].

OOPLs take a behavioral point of view, and so design B in Figure 5 seems to be more appropriate. This is a design that can be fully instantiated using C++ or Java classes. However, even that design may not be entirely adequate. If a figure is constructed to be a rectangle, it cannot become a square no matter how one stretches it. In design C of Figure 5, the design has used metalevel shifting (cf. Section 3.2.2) to refactor the classes in the first two designs into an attribute. There being just two classes, the attribute is Boolean. For larger class hierarchies, one would need a more complex property.

Unfortunately, concepts in the real world, which ontologies attempt to model, do not come in neatly packaged hierarchies. This mistaken view is the result of restricting oneself to simple examples such as persons and elephants [64]. Wegner, in fact, assumes that these "intuitive" hierarchies are mind-independent, preexisting, real-world concept hierarchies. We need to look only a little further to see how complex our everyday concept hierarchies can be, as discussed in [2] [30] [37] [53].

Lyee explicitly recognizes the role of cognitive agent in creating objects and structures. This aligns it very well with OOPLs, but not with OLs. In spite of the agent emphasis in modern OLs, ontologies tend to take the set-theoretic rather than the behavioral point of view toward classes. On the other hand, as discussed in Section 3.2.2, operational reasoning is important even for set-theoretic classes.

5 Properties

Ontologies are defined in Philosophy as the study of what exists and the relationships between the existents. The two fundamental notions in formal ontologies are classes and relationships. There are two notions of relationship: attribute and association. An attribute is a feature of an object that has a concrete interpretation (e.g., a text string or a number). Associations, on the other hand, relate objects to other objects. The two notions are often treated as subclasses of a single notion, variously called a relationship or a property. For definiteness, we will use the term *property*.

One important feature of most OLs is that properties have the same status as classes. Properties are defined independently of classes and are linked with them only by means of restrictions. One says that the properties are “first-class” because they can be specified at the highest modeling level. By contrast, OOPLs encapsulate properties within classes. Consequently, an OOPL property is always subsidiary to the class that “owns” it. Constraints on OOPL properties are local to the class that owns the property, but constraints on OL properties can be unrelated to any classes. For example, an OL property can be constrained to be transitive and/or symmetric.

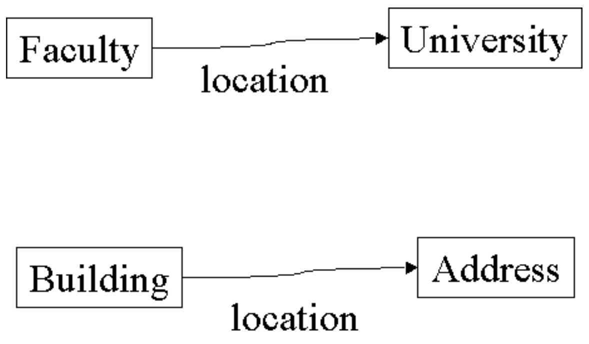

Figure 6: An example of the same property being used in two different contexts.

To understand this distinction, consider the notion of `location`. In an OOPL this would be specified separately for every class, while it would be a single property in an ontology. For instance, the `location` property could associate `Faculty` with `University`. Each link of this association would give the University affiliation of a faculty member. The same property might also be used for associating a `Building` with its `Address`. See Figure 6. In an OOPL these two location notions would be different attributes in spite of having the same name.

The fact that OL properties are independent modeling notions is inherited from the origins of OLs in logic languages where properties are called "predicates." In logic-based OLs, it is the predicates that are the primary modeling notion and classes (if they exist at all) are secondary. One consequence is that OLs do not support encapsulation. Since encapsulation is an important modularity mechanism, this has consequences for ontology development. The loss of encapsulation is to some degree compensated by the added flexibility obtained by having first-class properties. In the OOPL community a first-class function is called an *aspect*, so it is reasonable to assert that OLs support aspect-oriented modeling. Aspect-oriented programming is a popular research topic in the Object-oriented programming community. A notion of aspect-oriented modeling (although that name was not used) was introduced in 1996 in [35]. The first proposal for adding aspects to current OOMLs was made in [8], and further elaborated in [9].

Modern OLs are primarily based on the mathematical notion of a graph or network (consisting of a set of vertices and edges). Natural Language Processing (NLP) systems are well suited to being represented with an OL because an edge from one vertex to another corresponds to a predicate linking a subject to an object. The fact that verbs are defined independently of nouns (and hence are first-class concepts) is another reason why NLP is more compatible with OLs than with OOPLs. Of course, both verbs and nouns must be disambiguated, but this is a very different issue. Both OLs and OOPLs presume that all notions are unambiguous. Lyee is sensitive to natural language constructs, and this makes it rather more appropriate for NLP than most object-oriented development methodologies.

Introducing aspect-oriented modeling to object-oriented modeling is not as easy as it first appears. One common suggestion is to reify the property, as shown in Figure 3 of Section 3.2.2. Since classes are first-class modeling elements, this appears to solve the problem. For example, instead of modeling `location` as an OOPL property, one could model it as a class `Location`. This is certainly appropriate when location is multi-valued and can have its own attributes. However, in the absence of such requirements, reification has several disadvantages. It produces complex, unnatural ontologies. It also puts the burden on the ontology developer to deal with this incompatibility issue. The most subtle problem with this technique is that it produces an unbounded mapping, in the sense defined in [8].

An alternative that can achieve the spirit of aspect-oriented modeling is to use ontology reuse techniques based on the colimit operation, as suggested in Section 3.2.1. This technique is theoretically well founded. However, as discussed in that section, it is difficult to use in currently available tools.

6 Logic

OLs differ from OOPLs in their explicit support for logical inference. Although OOPLs do not claim to support logical inference, they do, in fact, have a number of inference mechanisms. The best known is inheritance, whereby an object of one class can be inferred to belong to other classes. This inference mechanism takes place at run-time when polymorphic methods are invoked. In addition to inheritance, OOMLs have support for various constraints such as cardinality constraints. This section discusses the ways in which the inference mechanisms of OOPLs differ from the explicit inference mechanisms or OLs.

6.1 Open versus Closed Worlds

An important distinction between OLs and OOMLs is the notion of *monotonicity*. A logical system is monotonic if adding new facts can never cause previous facts to be falsified. Of course, one must be careful to define which facts are being considered in this process so that it makes sense. Modern OLs are generally monotonic: asserting a new fact can never cause a previously known fact to become false.

By contrast, OOMLs are typically not monotonic. There are many forms of non-monotonic logic, but the one that is closest to OOMLs is a logic that assumes a *closed world*. A simple example can illustrate how monotonicity affects inference. Suppose that one specifies that every person must have a name. Consider what would happen if a particular person object does not have a name. In an OOML this situation would be considered to be a violation of the requirement that every person must have a name, and a suitable error message would be generated. In a monotonic logic, on the other hand, one cannot make any such conclusion. The person who appears not to have a name really does have one, it just isn't known yet.

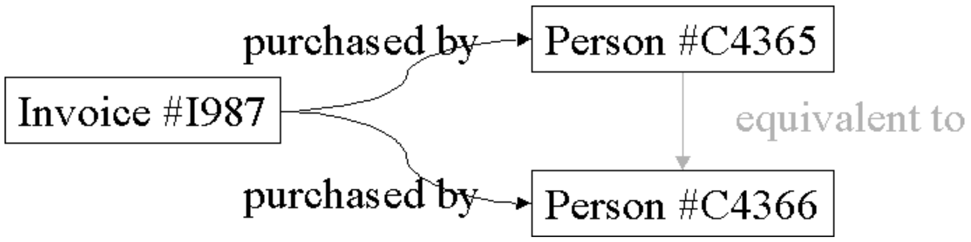

Figure 7: An example of the distinction between open and closed worlds. An invoice is intended for exactly one customer. This particular invoice was specified to have two customers. In a closed world, this would be a violation of the constraint. In an open world, one can infer that the two persons are the same. In the diagram the inferred information is shown in gray.

As another example, consider the Invoice ontology of Figure 1. In this ontology a invoice can have at most one purchaser. This is specified by asserting that the `purchasedBy` property is functional. Now suppose that one has specified that a particular invoice has two purchasers. In OWL this would be stated as follows:

```
<Invoice dated="2003-06-23" id="I987">
  <purchasedBy rdf:resource="#C4365"/>
  <purchasedBy rdf:resource="#C4366"/> </Invoice>
```

In an OOML this would violate the cardinality constraint, and that would be the end of it. In a monotonic logic, on the other hand, one cannot make such a conclusion, at least not directly. It is possible that `C4365` and `C4365` are the same customer. It is unlikely that the annotator intended this, but it is consistent nonetheless, unless there are other constraints that might be violated by it. For example, suppose that the `PersonOrCompany` class had a `name` functional property whose range was of type `String`. Then `C4365` and `C4365` can be equivalent only if their names were the same.

As another example, suppose that it is required that an invoice have exactly one purchaser. In an OOML an invoice that had no purchaser at all would clearly violate this requirement. In a monotonic logic, on the other hand, there is no inconsistency at all. There is a purchaser, one just does not know who it is, as shown in Figure 8.

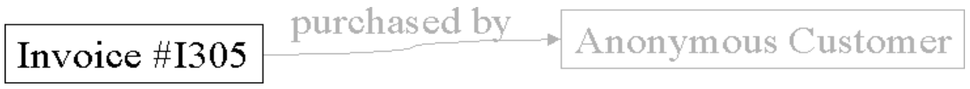

Figure 8: An example of an unspecified but mandatory relationship. This is not allowed in a closed world. In an open world, one can infer that there is an anonymous object that fulfills the requirement. Since the invoice did not have a purchaser, an anonymous customer is introduced for this purpose. The inferred customer and relationship are shown in gray.

Virtually all of the consistency constraints that one is accustomed to impose with OOMLs (including domain constraints, range constraints and cardinality constraints) have very different consequences in monotonic and non-monotonic logics. This distinction between OOMLs and OLs would therefore seem to be insurmountable.

Fortunately, in practice there are many ways to limit the scope of an ontological inference. If two individuals (which could be people or programs) are interacting, then there must have been a prior agreement that specifies the ontologies that will be used for communication. By limiting the scope, one is effectively replacing the “open world” with a closed one. The conclusion is that while OLs are defined using open, monotonic logics, they are operationally very similar to OOPLs which used closed world assumptions.

7 Conclusion

This paper has presented a case for employing more rigorous process models and methodologies in ontology development than the ad hoc, informal methodologies currently in use. It has been proposed that object-oriented software development in general (and Lyee in particular) is a good starting point for such process models and methodologies. While there are many ways in which ontologies and ontology development differ from programs and program development, the differences are not insurmountable. A number of suggestions were made in this regard, but they are only the first steps toward the goal of creating viable process models and methodologies. It is hoped that this paper will encourage further work in this area.

8 Acknowledgments

This material is based upon work supported by the Air Force Research Laboratory, Contract Number F30602-00-C-0188. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Unites States Air Force. I also wish to acknowledge Haim Kilov, who carefully read the paper and made many useful suggestions.

References

[1] O. Arai and H. Fujita. A word-unit-based program: Its mathematical structure modeal and actual application. In *New Trends in Software Methodologies, Tools and Techniques*, pages 63–74. IOS Press, Amsterdam, 2002.

[2] K. Baclawski. Long time, no see: Categorization in information science. In S. Hecker and G.C. Rota, editors, *Essays on the Future. In Honor of the 80th Birthday of Nick Metropolis.* Birkhauser Boston, Cambridge, MA, 1997.

[3] K. Baclawski and B. Indurkhya. The notion of inheritance in object-oriented programming. *Comm. ACM*, 37(9):118–119, September 1994.

[4] K. Baclawski and H. Kilov, editors. *Ninth OOPSLA Workshop on Behavioral Semantics.* Northeastern University, College of Computer Science, Boston, MA, November 2000.

[5] K. Baclawski and H. Kilov, editors. *Tenth OOPSLA Workshop on Behavioral Semantics.* Northeastern University, College of Computer Science, Boston, MA, October 2001.

[6] K. Baclawski and H. Kilov, editors. *Eleventh OOPSLA Workshop on Behavioral Semantics.* Northeastern University, College of Computer Science, Boston, MA, November 2002.

[7] K. Baclawski, H. Kilov, A. Thalassinidis, and K. Tyson, editors. *Eighth OOPSLA Workshop on Behavioral Semantics*. Northeastern University, College of Computer Science, Boston, MA, November 1999.

[8] K. Baclawski, M. Kokar, P. Kogut, L. Hart, J. Smith, W. Holmes, J. Letkowski, and M. Aronson. Extending UML to support ontology engineering for the Semantic Web. In M. Gogolla and C. Kobryn, editors, *Fourth International Conference on The Unified Modeling Language*, volume 2185, pages 342–360. Springer-Verlag, Berlin, October 2001.

[9] K. Baclawski, M. Kokar, P. Kogut, L. Hart, J. Smith, J. Letkowski, and P. Emery. Extending the Unified Modeling Language for ontology development. *Software and System Modeling*, 1(2):142–156, 2002.

[10] K. Baclawski, M.M. Kokar, and R. Waldinger. Consistency checking of Semantic Web ontologies. In *Proc. of the International Semantic Web Conference*, June 2002.

[11] D. Bjørner. Domain models of "the market" – in preparation for *e*-transaction systems. In H. Kilov and K. Baclawski, editors, *Practical Foundations of Business and System Specifications*, pages 111–144. Kluwer Academic Publishers, 2003.

[12] R. Brachman. On the epistemological status of semantic networks. In N. Findler, editor, *Associative Networks: Representation and Use of Knowledge by Computers*, pages 3–50. Academic Press, New York, NY, 1979.

[13] R. Brachman. What IS-A is and isn't: An analysis of taxonomic link in semantic networks. *Computer*, 16(10):30–36, 1983.

[14] R. Brachman. "I lied about the trees" or, defaults and definitions in knowledge representation. *AI Magazine*, pages 80–93, Fall 1985.

[15] M. Bunge. *Treatise on basic philosophy. III: Ontology: The furniture of the world.* Reidel, Dordrecht, 1977.

[16] M. Bunge. *Treatise on basic philosophy. IV: Ontology: A world of systems.* Reidel, Dordrecht, 1979.

[17] DARPA Agent Markup Language Web Site, 2001. `www.daml.org`.

[18] S. Decker, D. Brickley, J. Saarela, and J. Angele. A query and inference service for RDF. In *QL'98 - The Query Language Workshop*, 1998.

[19] M. Denny. Ontology building: A survey of editing tools, November 2002. `www.xml.com/-pub/a/2002/11/06/ontologies.html`.

[20] M. Denny. Ontology editor survey results, November 2002. `www.xml.com/2002/11/06/-Ontology_Editor_Survey.html`.

[21] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[22] M. Genesereth. Knowledge Interchange Format draft proposed American National Standard (dpANS) NCITS.T2/98-004, 1998. Available at `logic.stanford.edu/kif/dpans.html`.

[23] J. Goguen and G. Malcolm, editors. *Software Engineering with OBJ: Algebraic specification in action.* Kluwer Academic Publishers, Dordrecht, The Netherlands, 2000.

[24] N. Guarino and P. Giaretta. Ontologies and knowledge bases: Towards a terminological clarification. In N. Mars, editor, *Towards Very Large Knowledge Bases.* IOS Press, 1995.

[25] J. Heflin, J. Hendler, and S. Luke. Coping with changing ontologies in a distributed environment. In *AAAI-99 Workshop on Ontology Management.* MIT Press, 1999.

[26] J. Heflin, J. Hendler, and S. Luke. SHOE: A knowledge representation language for Internet applications. Technical Report `www.cs.umd.edu/projects/plus/SHOE`, Institute for Advanced Studies, University of Maryland, 2000.

[27] J. Hendler and D. McGuinness. The DARPA Agent Markup Language. *IEEE Intelligent Systems*, 15(6):67–73, 2000.

[28] B. Humphreys and D. Lindberg. The UMLS project: making the conceptual connection between users and the information they need. *Bulletin of the Medical Library Association*, 81(2):170–177, 1993.

[29] B. Indurkhya. *Metaphor and Cognition.* Kluwer Academic Publishers, Dordrecht, The Netherlands, 1992.

[30] B. Indurkhya. On the philosophical foundation of Lyee: Interaction theories and Lyee. In *New Trends in Software Methodologies, Tools and Techniques*, pages 45–51. IOS Press, Amsterdam, 2002.

[31] *Basic Reference Model for Open Distributed Processing. Use of formal specification techniques for ODP (ISO/IEC JTC1/SC21/WG7 N 753)*, November 1992.

[32] H. Kilov and K. Baclawski, editors. *Practical Foundations of Business and System Specifications.* Kluwer Academic Publishers, Dordrecht, The Netherlands, 2003.

[33] H. Kilov and W. Harvey, editors. *Object-Oriented Behavioral Specifications.* Kluwer Academic Publishers, 1996.

[34] H. Kilov and J. Ross. *Information Modeling: An Object-Oriented Approach.* Prentice-Hall, Englewood Cliffs, NJ, 1994.

[35] H. Kilov and I. Simmonds. Business patterns: reusable abstract constructs for business specification. In P. Humphreys et al., editors, *Implementing Systems for Supporting Management Decisions: Concepts, methods and experiences*, pages 225–248. Chapman and Hall, 1996.

[36] P. Kogut, S. Cranefield, L. Hart, M. Dutra, K. Baclawski, M. Kokar, and J. Smith. UML for ontology development. *Knowledge Engineering Review*, 2001.

[37] G. Lakoff. *Women, Fire, and Dangerous Things: What Categories Reveal about the Mind.* The University of Chicago Press, Chicago, 1987.

[38] O. Lassila and R. Swick. Resource description framework (RDF) model and syntax specification, Feburary 1999. `www.w3.org/TR/REC-rdf-syntax`.

[39] P. Loucopoulos and J. Mylopoulos, editors. *Requirements Engineering.* Springer-Verlag, Limited, London, 1996.

[40] D. McGuinness. Ontologies and online commerce. *IEEE Intelligent Systems*, 16(1):8–14, 2001.

[41] D. McGuinness, R. Fikes, J. Rice, and S. Wilder. An environment for merging and testing large ontologies. In *Proceedings of the Seventh International Conference on Principles of Knowledge Representation and Reasoning (KR2000)*, Breckenridge, Colorado, USA, April 12–15 2000.

[42] The Model Driven Architecture, 2003. `www.omg.org/mda`.

[43] F. Negoro. Principle of Lyee software. In *Proceedings of the 2000 International Conference on Information Society in the 21st Century (IS2000)*, pages 441–446, November 2000.

[44] F. Negoro. Lyee's hypothetical world. In *New Trends in Software Methodologies, Tools and Techniques*, pages 3–22. IOS Press, Amsterdam, 2002.

[45] N. Noy and D. McGuinness. Ontology 101, 2001. Available at `protege.stanford.edu/-publications/ontology_development/ontology101.html`.

[46] B. Opdahl, A. Henderson-Sellers and F. Barbier. An ontological evaluation of the OML metamodel. In E. Falkenberg, K. Lyytinen, and A. Verrijn-Stuart, editors, *Information System Concepts: An Integrated Discipline Emerging*, volume 164, pages 217–232. IFIP/Kluwer, 1999.

[47] J. Philipps and B. Rumpe. Refactoring of programs and specifications. In H. Kilov and K. Baclawski, editors, *Practical Foundations of Business and System Specifications*, pages 281–297. Kluwer Academic Publishers, 2003.

[48] J. Piaget. *The Construction of Reality in the Child.* Ballantine Books, New York, NY, 1971.

[49] J. Piaget and B. Inhelder. *The Child's Conception of Space.* W.W. Norton and Company, New York, NY, 1967.

[50] J. Piaget, B. Inhelder, and A. Szeminska. *The Child's Conception of Geometry.* W.W. Norton and Company, New York, NY, 1981.

[51] D. Pisanelli, A. Gangemi, and G. Steve. Ontologies and information systems: the marriage of the century? In *New Trends in Software Methodologies, Tools and Techniques*, pages 125–133. IOS Press, Amsterdam, 2002.

[52] R. Pressman. *Software Engineering: A Practitioner's Approach, Fifth Edition.* McGraw-Hill, New York, 2001.

[53] E. Rosch and B. Lloyd, editors. *Cognition and Categorization.* Lawrence Erlbaum Associates, Hillsdale, NJ, 1978.

[54] J. Smith, M. Kokar, and K. Baclawski. Formal verification of UML diagrams: A first step towards code generation. In A. Evans, R. France, A. Moreira, and B. Rumpe, editors, *Practical UML-Based Rigorous Development Methods – Countering or Integrating the eXtremists*, volume P-7, pages 224–240. Gesellshaft für Informatik, October 2001.

[55] J. Smith, M. Kokar, K. Baclawski, and S. DeLoach. Category theoretic approaches of representing precise UML semantics. In *Proceedings of the Precise UML Workshop at ECOOP 2000*, Sophia Antipolis, France, 2000.

[56] M. Smith, D. McGuinness, R. Volz, and C. Welty. OWL web site, November 2002. `www.-w3.org/TR/owl-guide/`.

[57] J. Sowa, editor. *Knowledge Representation: Logical, Philosophical, and Computational Foundations.* PWS Publishing, 2000.

[58] *Specware*$^{\text{TM}}$ *User Manual: Specware*$^{\text{TM}}$ *Version Core4*, October 1994.

[59] D. Tansley and C. Hayball. *Knowledge-based systems analysis and design.* Prentice Hall International (UK) Ltd., Hertfordshire, England, 1993.

[60] M. Uschold and M. Gruninger. Ontologies: Principles, methods and applications. *Knowledge Engineering Review*, 11(2):93–155, June 1996.

[61] F. van Harmelen, J. Hendler, I. Horrocks, D. McGuinness, P. Patel-Schneider, and L. Stein. OWL Web ontology language reference, March 2003. `www.w3.org/TR/owl-ref/`.

[62] R. Waldinger et al. *Specware*$^{\text{TM}}$ *Language Manual: Specware*$^{\text{TM}}$ *2.0.3*, March 1998.

[63] Y. Wand. A proposal for a formal model of objects. In W. Kim and F. Lochovsky, editors, *Object-oriented concepts, databases and applications*, pages 537–559. Addison-Wesley, 1989.

[64] P. Wegner. The object-oriented classification paradigm. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 479–560. MIT Press (Computer Systems Series), Cambridge, MA, 1987.

[65] W. Whewell. *The Philosophy of the Inductive Sciences, Second Edition.* Parker, London, 1847.

[66] L. Wittgenstein. *Tractatus Logico-Philosophicus.* Dover Publications, Incorporated, 2001.

[67] The XTM Web Site, 2000. `topicmaps.org`.