

# Ontology-Oriented Design and Programming

Mark A. Musen  
*Stanford Medical Informatics*  
*Stanford University*  
*Stanford, California USA*  
*musen@Stanford.EDU*

**Abstract:** In the construction of both conventional software and intelligent systems, developers continue to seek higher level abstractions that both can aid in conceptual modeling and can assist in implementation and maintenance. In recent years, the artificial intelligence community has placed considerable attention on the notion of explicit ontologies—shared conceptualizations of application areas that define the salient concepts and relationships among concepts. Such ontologies, when joined with well defined problem-solving methods, provide convenient formalisms for modeling and for implementing solutions to application tasks. This chapter reviews the motivation for seeking such high-level abstractions, and summarizes recent successes in building systems from reusable domain ontologies and problem-solving methods. As the environment for software execution moves from individual workstations to the Internet at large, casting new software applications in terms of these high-level abstractions may make complex systems both easier to build and easier to maintain.

## 1. The Search for Reusable Abstractions

The field of artificial intelligence (AI) has contributed a great deal to conventional software engineering. For example, work in the 1970s that was couched in the framework of “knowledge acquisition” for expert systems [1] has led to interviewing strategies, methods for systems analysis, and modeling techniques that have been influential more broadly in the area of requirements engineering. The conceptual modeling techniques pioneered by early workers in AI have become standard elements of industrial software design [2]. Object-oriented programming made its first major strides forward in the 1970s as researchers developed a host of practical frame-based knowledge-representation systems, and entered the mainstream of software engineering in the 1980s as AI researchers created extensions to the LISP language to support data encapsulation and message passing [3]. The myriad tools for computer-assisted software engineering (CASE) based on the unified modeling language (UML) and pervasive

object-oriented programming languages such as C++ and Java owe much of their heritage to early work in AI. At the same time, there are countless algorithms such as techniques for search and optimization, probabilistic reasoning, and signal understanding that have been translated from AI research laboratories to the software libraries that developers use every day to build thousands of mundane computer programs.

Despite this great tradition of translation of ideas from AI to conventional software engineering, workers in AI also have had a history of getting themselves into trouble by not recognizing the central importance of *software* to the AI enterprise itself. The developers of the first knowledge-based systems, who contributed so many techniques to the requirements-engineering community, faltered as they attempted to scale up their work. These AI developers claimed that construction of electronic knowledge bases was *knowledge engineering*, not software engineering. For many years, they ignored the rapid advances in software engineering that were taking place in parallel to their fledgling work to build large-scale knowledge-based systems. While workers in software engineering struggled with the complexities of large-scale systems, many developers in AI believed that large knowledge-based systems could be constructed simply by adding more and more “modular” if-then statements to the disordered collection of production rules that characterized most knowledge-based systems at the time. The failure of many AI systems developed in the 1980s to meet their requirements and to allow maintainability over time was a function of this view of rather naïve view of systems engineering. The eventual emergence of more formal methodologies for construction of knowledge-based systems such as KADS [4] occurred as a direct response to these problems.

Contemporaneously, not all was well in the software-engineering community, however. Indeed, from the time of the very first computer programs, scores of authors began to write of a “software crisis” that prevents reliable and maintainable software from being developed on time and within budget [5]. Conventional software frequently fails to meet requirements? and software that meets current requirements often is insufficiently malleable to adapt to changing organizational demands. Development of new software applications routinely requires the programming from scratch of large amounts of error-prone and unmaintainable source code. Brooks [6] has argued that there is no single “silver bullet” that can fix the software crisis technologically; he suggests that intrinsically creative processes such as software engineering always are critically dependent on the unique abilities of the creative people who perform those processes.

Nevertheless, it is clear that an essential element of software engineering that can be addressed from a technological perspective involves the management of complexity. From the earliest days of structured programming, the recurrent theme in software engineering has concerned the appropriate abstraction and encapsulation of procedures and data. To that end, developers now turn to the use—and reuse—of software components as a potential way out of our perpetual software crisis [7]. The belief is that libraries of software components can offer modular and composable building blocks for use within large application programs. These building blocks can allow developers to view their software more abstractly. These building blocks ostensibly are already tested and debugged. The goal becomes not to construct software *de novo*, but rather to select the appropriate components from libraries and to “glue” them together in some fashion.

The vision of component-based software development is compelling, but leaves us with uncertainty regarding what exactly are the right kinds of components to reuse.

Krueger's thoughtful review of software reuse identifies ways in which developers can reapply different kinds of software abstractions [8]. These alternatives include the use of high-level and very-high-level languages, design and code scavenging, subroutine libraries, design patterns, application generators, and automatic programming. In each case, the goal is to overcome complexity by use of appropriate abstractions, and to minimize the "cognitive distance" between the way in which a developer thinks about a problem to be solved and the language available for specifying a solution [9].

Although Krueger [8] enumerates a variety of forms of software reuse at different levels of abstraction, by far and away the most familiar examples of software reuse by applications programmers have involved predefined libraries of explicit subroutines. Since the 1960s, developers have turned to subroutine libraries to fill well defined gaps in their program code. With such libraries, the semantics of each subroutine and of each parameter are clear. The meaning of each function in the library is common background knowledge shared by both the developers and the consumers of the software elements. The assumptions made by each subroutine can be readily enumerated. Despite the clear success of these reuse libraries, however, the level of abstraction provided by the individual software components is obviously modest. Software engineers continue to look for reusable components that are more encompassing in scope than are routines to calculate the *sine* and *cosine* of angles, or to present user-interface widgets to end users.

In the past decade, the notion of *design patterns* has become wildly popular among software developers [10]. Design patterns provide abstract descriptions of program solutions that are suitable for particular problems within particular contexts. They provide frameworks by which software engineers can structure their program code, offering standard structures that can ease both design and maintenance of large software systems. Thus, unlike subroutine libraries, design patterns can address large-scale issues in program design. Unfortunately, however, design patterns can only inspire implementations. They do not provide reusable program code, and do not obviate the need for careful testing and debugging.

The original KADS methodology identified something akin to design patterns for the modeling of expertise within knowledge-based systems. In the 1980s, the authors of KADS suggested that *interpretation models* for common tasks such as diagnosis and planning could guide the conceptual modeling of the domain knowledge and frame the implementation of the knowledge base [11]. KADS provided a library of rather abstract interpretation models, and assumed that developers would be able to apply these inference patterns when designing conceptual models for nascent knowledge-based systems. Like the design patterns that software engineers now use to build conventional computer programs, the interpretation models themselves were not executable in any way, and provided only a scaffolding on which to implement a working system.

The theme in the knowledge-based systems community during the past decade has been to develop the means for reuse of large-scale software components when building intelligent systems [12, 13]. Unlike interpretation models and design patterns, these reusable components have an operational dimension that provides working code. Unlike elements of mathematical subroutine libraries, these components are of rather large scale and can provide algorithms for automating complete tasks. These components

are *domain ontologies* that provide a characterization of the concepts and relationships in an application area [14], and *problem-solving methods* that offer abstract algorithms for achieving solutions to stereotypical tasks [15, 16]. By turning to these large-scale reusable components, many workers in AI believe that they have mitigated much of the “software crisis” that plagues the construction of intelligent systems. This chapter examines the use of these abstractions to build knowledge-based systems, and speculates how similar abstractions may be useful in the development of more conventional software applications.

## 2. Moving Beyond Rules

Before discussing current component-based architectures for knowledge-based systems, it is important to acknowledge that these architectures are only beginning to be appreciated outside of academic circles. Despite the range of modern development technologies and tools surveyed in this volume, many computer scientists still think of knowledge-based systems in terms of the rule-based approaches that were popularized during the 1970s [17]. The continued widespread availability of generic “shells” for constructing rule-based systems (such as OPS5 and CLIPS) reinforces this notion. Although the majority of knowledge-based systems continue to be built using rule-based frameworks, there are well-known limitations to the scalability and maintainability of such systems that researchers began to identify nearly as soon as the first rule-based systems had been developed [18]. The practice of using explicit ontologies and problem-solving methods to build modern knowledge-based systems overcomes many of the limitations of traditional rule-based architectures.

Clancey [19] is credited for demonstrating early on the limitations of constructing knowledge-based systems simply by amassing collections of production rules. His careful analyses of the MYCIN knowledge base showed how the developers on that system’s rule base had to construct production rules that would interact with one another in rather arcane ways in order to coerce the system to demonstrate desired problem-solving behavior. For example, the sequencing of the clauses in the left-hand side of a rule often needed to be considered very carefully; changing the order of the conditions might radically change the way in which the program would gather and process information about the case under consideration. Although members of the MYCIN project never documented such programming practices explicitly, knowledge-base builders would carefully tinker with the ordering of rule clauses in order to achieve necessary system performance [20]. When subsequent developers made seemingly innocent changes to the rule base, surprising changes in the system’s program-solving behavior could result. Although an official claim was often made that rules such as those in MYCIN are “independent and modular,” it became clear that developers needed to view production rules as elements of a very high-level programming language. Developers intentionally (although perhaps subconsciously) created dependencies among the various rules in a knowledge base in order to effect the desired problem-solving behavior. Because the rules in most rule-based systems generally are not annotated, classified, or organized, the purpose of individual rules and the relationships among them can be difficult to determine by direct inspection of a knowledge base.

Experience in industry confirms that construction of large software systems by amassing unorganized collections of production rules is a problematic enterprise. The large, commercial rule-based systems such as XCON, which provided some of the first convincing demonstrations of knowledge-based technology, became unmanageable without the imposition of considerable additional structure on the knowledge base [21]. In the subsequent two decades, there has been substantial work to develop improved design methodologies for knowledge-based systems that specifically has emphasized techniques to manage complexity and to clarify the way in which knowledge is used during problem solving.

### 3. Reusable Ontologies

By the end of the 1980s, workers in AI came to accept that very general concepts about an application domain could be represented independently from whatever problem solvers might ultimately automate the tasks to which the domain knowledge could be applied. Although Chandrasekaran and his colleagues cautioned that the distinctions that developers make about any application domain are necessarily determined by problem-solving requirements [22], there often seemed to be certain foundational concepts that would emerge as salient in even a cursory analysis of a new application domain. If these foundational concepts can be represented within separate, editable data structures, then the corresponding enumeration of the concepts might be reused to build different kinds of knowledge bases. This point of view was fundamental to the original KADS methodology for knowledge engineering [4], which encouraged developers to build models that made a clear distinction between foundational domain concepts and the inferences and problem-solving procedures that might be applied to those concepts. In current parlance, these enumerations of domain concepts—and of relationships among the concepts—are referred to as domain *ontologies* [23]. An ontology provides a domain of discourse that is understandable by both developers and computers, and that can be used to build knowledge bases containing detailed descriptions of particular application areas.

A good example of a well-understood ontology is the categorization that Yahoo! provides users for searching the Internet. The Yahoo! ontology defines broad categories of entries on the World Wide Web. Users understand the ontology, and apply it to locate the concepts that define their interests; the Yahoo! search engine also can process the ontology, and uses it to locate corresponding Web pages. The relationships among the concepts in the Yahoo! ontology generally are taxonomic (i.e., they are primarily class–subclass relationships). Although the ontology does include some part–whole relationships, in general the goal is simply to provide an enumeration of searchable concept descriptions. It is clear that the engineering of machine-processable ontologies has become a major business for companies promoting access to complex information sources, and is at the heart of many electronic-commerce applications. Indeed, as Studer and his colleagues suggest [16], the expanding use of ontologies to drive Web-based applications soon will overshadow their use in developing standalone software systems.

The notion of reusable ontologies has become increasingly important to developers of intelligent systems. Just as modern database systems are driven by

*conceptual schemas* that define the classes of entities about which the database stores specific data, modern knowledge-based systems incorporate as a central component of their knowledge bases a model of the classes of domain-related entities about which problem solving takes place [2]. This ontology defines the relevant concepts and the attributes that concepts may have, but generally is silent regarding the specific values of the attributes that are assumed by particular instances of the concepts. Descriptions of instances of concepts generally appear in the contents of the *knowledge bases* that are derived from a particular ontology. Of course, developers cannot construct an ontology without considerable forethought regarding what instances ultimately will need to be represented. Consideration of potential instances informs the developer's conceptualization of the classes that need to be included in an ontology; once an ontology has been defined, developers encode the instances in terms of the classes and relationships that the ontology dictates.

Because ontologies are models, there is not a single, correct way to define an ontology. There are many different perspectives that a modeler can take on a domain, and sometimes alternative perspectives may need to be captured simultaneously. For example, in medical ontologies, drugs can be categorized both on the basis of their physiological function and on the basis of their chemical structure. Some medical ontologies may wish to include a relationship between diseases and their causative agents, and some ontologies may not; some ontologies may wish to include a relationship between diseases and their clinical manifestations, and some ontologies may not. Ontologies represent convenient ways of characterizing a set of concepts and relationships in an application area. They do not, and cannot, capture absolute Platonic truths about what might exist in the world. The merits of a particular ontology can be measured only in terms of how well that ontology supports development of the application programs for which it was designed, and of how easy it is for developers to reuse that ontology to build new applications.

As Shadbolt and colleagues discuss in detail [14], there has been explosive interest in reusable ontologies in recent years. Developers are seeking to create libraries of ontologies that can capture a set of "core" concepts needed to model different application areas, thus paving the way for reuse of previously developed ontologies within new systems [24, 25]. Developers also see reference ontologies as a means for providing a canonical description of concepts that can allow integration of multiple information sources, particularly when individual information sources adopt idiosyncratic ways of referring to the same concept [26].

For builders of intelligent systems, ontologies provide the structure for domain knowledge bases that allows for coherent views of the relevant concepts and relationships among those concepts. Each knowledge base is thus an extension of some applicable ontology, where the ontology provides a roadmap for the classes of concepts that will comprise that knowledge base. Just as a *schema* provides the organizing framework for a database, an ontology provides the framework for a domain knowledge base. Although specific ontologies rarely are reusable *in toto* from one application to the next, they often provide considerable guidance when developers wish to create new systems in the same domain.

#### 4. Problem-Solving Methods

Concomitant with the increasing interest in domain ontologies, it was becoming apparent to many investigators that there were recurring problem-solving strategies that seemed to emerge from the behaviors of many rule-based systems. Clancey [27] proposed *heuristic classification* as a clearly defined inference pattern that could be identified in well known systems such as MYCIN and PUFF [20, 28]. When performing heuristic classification, a problem solver would start with primitive case data that the end-user would input into the system (e.g., a patient's white blood cell count is less than 2500 cells per cubic millimeter) and abstracted those data into general descriptors (e.g., that the patient is a "compromised host"). Heuristics would link those abstractions to general candidate solutions (e.g., that bacteremia in compromised hosts often is caused by Gram-negative organisms), and additional knowledge would be used to refine general solutions into specific classifications (e.g., the patient most likely is infected with *E. coli*). MYCIN used *feature abstraction* to create generalizations of its input data, *heuristic match* to identify classes of possible organisms suggested by abstract patient characteristics, and *solution refinement* to reduce the set of possible organisms to a small number of likely pathogens. PUFF [28], on the other hand, used *feature abstraction* to translate numerical measurements of pulmonary physiological parameters into conclusions about the patient's lung function, and *heuristic match* to identify possible pulmonary diseases associated with those functional states. *Solution refinement* could then allow the program to narrow the set of possible diseases by considering additional information about the patient. Although the two systems had rather different purposes, both MYCIN and PUFF used heuristic classification as a common problem-solving paradigm. Clancey's analysis showed that heuristic classification is a well-defined problem-solving approach that several developers happened to have used to create a variety of disparate decision-support systems [27]. Many modern knowledge-based systems continue to use heuristic classification as the basis for their intelligent behavior.

At around the same time that Clancey was studying heuristic classification, Chandrasekaran's group at Ohio State University had identified several recurring problem-solving strategies in the knowledge-based systems that they had been building. Chandrasekaran referred to these stereotypical problem-solving behaviors as *generic tasks* [29]. Meanwhile, John McDermott's group at Carnegie-Mellon University was noting a set of *problem-solving methods* that provided the control structure for a number of other knowledge-based systems built in non-medical areas [15]. All these investigators demonstrated that many intelligent systems had highly regular mechanisms for sequencing certain classes of inferences. These domain-independent problem-solving strategies provided standard ways of addressing certain kinds of application tasks. Even though the original developers of these knowledge-based systems might never have thought about these regularities explicitly, there were nevertheless a number of well-defined, generic strategies that were emerging from analysis of how diverse automated problem solvers addressed their associated application tasks. These generic strategies are now referred to as *problem-solving methods* by nearly all workers in the knowledge-based-systems community [30].

Problem-solving methods can provide a structure for building intelligent systems. When a designer can come to understand the domain knowledge needed to solve an

application task in terms of a predefined problem-solving method, it becomes clear how each element of the domain knowledge might ultimately contribute to the problem-solving behavior of the system. When designing a heuristic classifier [27], for example, the developer can identify readily whether a primitive inference (e.g., a given production rule) is used to perform *feature abstraction* of case data, *heuristic matching* of case descriptions to a possible solution, or *solution refinement* into more specific classifications. The heuristic-classification model thus becomes a unifying framework by which to relate all the elements of domain knowledge that the developer might acquire. Use of the problem-solving method as the basis for conceptual modeling at design time limits the roles that domain knowledge can play in problem solving to those particular knowledge roles (e.g., *feature abstraction*, *heuristic match*) that are defined by the method. The knowledge roles of problem-solving methods make it clear what domain knowledge is needed to solve a task that can be automated with a given method (all the method's roles need to be filled for the method to work), and thus clarify the purpose of each piece of elicited knowledge [15]. Thus, when system builders examine a knowledge base, knowing the role that some proposition plays in problem solving helps them to understand the assumptions associated with that piece of knowledge. When the developers enter new knowledge into a knowledge base, the added propositions explicitly must fill some role (or roles) in the problem-solving method; if the entered knowledge does not fill some role, then the propositions necessarily must be superfluous to the task being automated.

So far, we have described problem-solving methods rather conceptually as stereotypical procedures. When associated with a piece of program code that implements the relevant algorithm, however, the problem-solving method becomes much more than an abstraction useful for conceptual modeling; the method becomes a building block that a programmer can use to implement a working system [31]. Developers can use the method conceptually to help them to model the domain knowledge that they will need to acquire to build the decision-support system. In this conceptual analysis, system builders use the *knowledge roles* of the problem-solving method to categorize the kinds of domain knowledge that they need to elicit when building an initial ontology. They then can use the operational form of the problem-solving method to implement the particular knowledge-based system. Ultimately, the problem-solving method functions like an element from a mathematical subroutine library: It provides a reusable piece of software that facilitates implementation of the required computer program.

The literature now describes dozens of reusable problem-solving methods for well-defined tasks such as fault diagnosis [32], constraint satisfaction [33], skeletal planning [34], and Bayesian classification. Some methodologies for building intelligent systems, such as Common KADS [35], emphasize the use of abstract descriptions of problem-solving methods primarily for conceptual modeling [36]; other approaches, such as Protégé [37, 38] and EXPECT [39], actually provide libraries of implemented problem-solving methods that can contribute program code to a system under development.

Problem-solving methods provide an enormous degree of procedural abstraction, allowing developers to treat complex algorithms as “black boxes.” Architectures such as those provided by Protégé [40] and KSM [41] allow developers to compose problem-solving methods so that aggregations of methods can solve particularly complex tasks.

The interactions among these problem-solving methods can be reminiscent of the communication that takes among agents in agent-oriented systems [42]. Autonomous agents, however, encapsulate both problem-solving knowledge and domain knowledge, whereas problem-solving methods are purely procedural, processing knowledge bases that are external to the procedural code. In developing intelligent systems that include separate domain ontologies and problem-solving methods, a primary concern involves linking the concepts represented in the domain ontology to the corresponding referents in the problem-solving method that ultimately will allow the method to operate on the relevant domain concepts.

## 5. Putting the Pieces Together

Because an ontology typically does not contain *instances* of concepts, we can view a knowledge base as an instantiation (or an extension) of an ontology. Thus, a knowledge base comprises “filled in” concept descriptions, enumerating the details of the particular application being built. Given a domain ontology, knowledge-acquisition systems such as Protégé [37, 38] allow straightforward entry of the corresponding knowledge base. The Protégé system permits developers to create a domain ontology using a simple editing system. Protégé then uses the domain ontology to create programmatically a user interface through which subject-matter experts can enter the detailed content knowledge required to populate a knowledge base [43]. The tools generated by Protégé also can be used to browse and to update the knowledge base as necessary—provided that the overarching domain ontology remains constant. If the ontology should change, then it may be necessary to generate new knowledge-entry tools that capture the corresponding changes to the concepts and relationships in the ontology. (Of course, generating such tools automatically using Protégé is considerably more convenient than having to reprogram the tools by hand.)

Construction of knowledge bases for intelligent systems using tools such as Protégé is considerably easier than entering production rules or other data structures into a traditional knowledge-base editor. The domain ontology establishes the terms used for entry of all the content knowledge. The structure of the ontology translates directly into the windows, forms, and widgets of a graphical user interface that organizes the domain knowledge—and that acquires and presents that knowledge—in a coherent manner that is understandable to any user who can understand the distinctions that are made in the original ontology. In our laboratory, we have demonstrated the utility of the Protégé approach in building and maintaining a wide variety of complex knowledge bases [44].

In modern knowledge engineering, building intelligent systems is not construed as the encoding of production rules or the creation of specific knowledge representations. Rather, the process is viewed as the design and assembly of domain ontologies, the knowledge bases that instantiate those ontologies, and domain-independent problem-solving methods. Given a task that must be automated, the challenge is to identify—or to construct—an appropriate problem-solving method, and to link that problem solver to an ontology that defines the relevant concepts in the application area. The emphasis has shifted from the writing isolated production rules to the creation of libraries of potentially reusable ontologies and problem-solving methods. The goal is to transform system

development into a matter of selecting, modifying, and assembling previously tested and debugged components, rather than to require programming of each new application from scratch [13]. The hope is that, because these components are at a quite high level of abstraction, the assembly of the components can take place more easily than in other cases of software reuse [8].

In our own laboratory, we have taken a general-purpose constraint-satisfaction problem-solving method known as propose-and-revise [33] and associated that method with a number of different domain ontologies to build a variety of intelligent systems. When the propose-and-revise method operates on an ontology of elevator parts, building codes, and engineering constraints, it automates the task of designing elevators for new buildings [45]. When propose-and-revise operates on an ontology that describes the molecular components of the *E. coli* ribosome and the kinds of constraints that experimental data place on the location of those molecular components in three-dimensional space, it automates the task of determining plausible conformations for the ribosome [46]. Similarly, we have developed appropriate ontologies to use propose-and-revise to address the tasks of monitoring patients receiving mechanical ventilation [47] and the planning of therapy for patients who have AIDS [48]. In building all these application programs, the propose-and-revise problem-solving method was completely reusable. The challenge in each case was in identifying how the generic data on which the propose-and-revise method operates can be related to the specific concepts in the different domain ontologies.

In general, as the knowledge-based-systems community has turned to building systems from reusable problem-solving methods and domain ontologies, the principal challenge has been to define the best way to glue the components together. In the approach that we have taken in the Protégé project, we have defined different types of *mappings* that provide the necessary relations between concepts described in a domain ontology and the input-output requirements of a generic problem-solving method [49]. Thus, in systems built using Protégé, we create explicit objects that link abstract data elements on which the problem-solving method operates to particular concepts in the domain ontology. For example, mappings can relate the concept of “restrictions on the use of antiretroviral drugs” as defined in the domain ontology for HIV therapy to the concept of *constraints* as used in the propose-and-revise method. Current work concentrates on refining an ontology of these kinds of mappings [50]. This *mappings ontology* provides a structure for relating domain ontologies to problem-solving methods, thus guiding the process by which the two principal kinds of components are brought together within the same software system. The ontology allows developers to classify mappings according to whether they involve either complete class definitions or some combination of class attributes; whether the mappings apply to only the specified class or also include subclasses in an inheritance tree; whether the mappings are determined statically at the time of system construction or dynamically at run time; and the degree to which the mappings might involve some functional transformation of the source values. By establishing how these characteristics apply to each mapping, system builders can completely specify the information needed to relate domain ontologies to the problem-solving methods that operate on those ontologies.

Workers on the Protégé project believe that maximal flexibility can be achieved by using declarative mappings to relate problem-solving methods to domain ontologies.

The developers of the Common KADS methodology, however, make no commitment to the manner in which ontologies and problem-solving methods should be brought together in an actual implementation [35]. Other researchers, such as Fensel and Motta [51], suggest that it may be more efficient to modify the problem-solving method directly by means of an *adapter* that can facilitate its interoperation with a domain ontology. Although there is considerable unanimity that domain ontologies and generic problem-solving methods provide the right kinds of abstractions for building intelligent component-based systems, there is less agreement on the optimal way for these individual components to communicate with one another. The problem of linking domain ontologies to reusable problem-solving methods remains a very active area of research within the knowledge-based–systems community. The principal challenge is to permit developers to combine domain ontologies and problem-solving methods into a coherent architecture without having to perform considerable programming to glue the pieces together. Any apparent advantages to the component-based approach are lost if there is a cost to reusing problem-solving methods and domain ontologies that suddenly appears during the assembly stage.

## 6. Discussion

For many years, workers in the field of conventional software engineering have anticipated a time when complex systems could be built rapidly by bringing together reusable software components. Although the general goals of software reuse have remained elusive [8], the knowledge-based–systems community has achieved increasing success in reusing domain ontologies and problem-solving methods to craft new applications. In our own laboratory, the reuse of the propose-and-revise method to construct four different end-user systems provides a clear demonstration of how modular software components can be reapplied to construct programs for a variety of applications.

Whereas the actual reuse of program code remains a difficult problem in conventional software engineering, there has been considerable interest in the identification and reuse of *design patterns* that can help developers to structure their software solutions [10]. Design patterns in object-oriented programming provide exemplars to facilitate the conceptual modeling and implementation of software systems based on commonly recurring programming constructs and abstract data types. The reusable domain ontologies and problem-solving methods being investigated by the knowledge-based systems community, however, allow much more direct reuse of previously tested solutions in the construction of new software. Domain ontologies and problem-solving methods provide not only a framework for conceptual analysis and design, but also actual program code that can be incorporated wholesale into new applications.

The use of distinct domain ontologies and problem-solving methods contrasts sharply with traditional object-oriented architectures, in which program code (in the form of “methods”) is interleaved with the representation of the domain model (in the form of class and instance objects). Standard object systems facilitate specialization of program code on the basis of distinctions in the domain model (i.e., method polymorphism), and can clearly identify the data structures that are most closely related to the control

structures that operate on those data. Such properties make traditional object-oriented systems rather malleable whenever programmers must adapt their software to evolving requirements. There thus can be a degree of fuzziness regarding the semantics of encoded objects. Object-oriented languages have the advantage of making it relatively straightforward for developers to graft new functionality onto existing program code. The tight linkage between the class hierarchies that constitute the domain model and the program code embodied in the objects' methods makes it difficult for an analyst to view either element independently, however. Thus, in a traditional C++ program, it is impossible to consider control-flow relationships separately from the data structures that comprise the object hierarchy. It also is impossible to view the data model without being distracted by associated program code.

The development of intelligent systems using separate domain ontologies and reusable problem-solving methods is particularly justified when the ontologies or the methods are likely to be reused to build new or derivative applications. The additional up-front engineering costs required to design the components for reuse then can be amortized over subsequent development efforts. This approach also is particularly attractive when developers can anticipate unusual requirements for software maintenance. The need for significant program revisions may result from domain models or domain-specific facts that are likely to evolve over time. In this situation, the incorporation of declarative domain ontologies can make domain-dependent information explicit, accessible, and easily editable, thus minimizing the difficulty of updating the system. A high degree of software maintenance may also be necessary when developers design new algorithms to achieve better performance; encapsulating an algorithm as a problem-solving method allows systems engineers to "plug in" new control strategies without having to alter either the domain ontology or the knowledge base [13]. (Of course, the mappings between the domain ontology and the new problem-solving method will need to be changed.)

There are many compelling examples of successful reuse of domain ontologies and problem-solving methods. In practice, however, there often are limitations to the reapplication of these components. It is quite clear that the distinctions that developers make about any application area depend heavily on how those developers imagine domain knowledge being used in problem solving. If a diagnostic task is to be solved using heuristic classification, for example, different domain knowledge is required than if that task is to be solved using Bayesian reasoning. Chandrasekaran's group has identified this interdependency as "the interaction problem" [22]. It is not uncommon that previously defined domain ontologies require modification before they can be used for new purposes. Similarly, complex domain tasks may be difficult to construe in terms of simple, stereotypical problem-solving procedures. In the case of our own work to automate different aspects of medical therapy [34], for example, we often have found it necessary either to enhance the capabilities of our standard reusable problem-solving method by invoking additional domain-specific problem solvers, or to make changes to the standard problem solver itself [52]. Such fine tuning of problem-solving methods is far easier than is programming a method from scratch, but it is clear that developers still have not reached the stage where they can routinely construct complex software systems simply by assembling "plug and play" components.

Despite these difficulties, the use of components such as ontologies and reusable problem-solving methods offer the promise of making our software artifacts easier to build, easier to understand, and easier to maintain over time. These advantages can occur because domain ontologies and problem-solving methods provide a means to view an intelligent system at a high level of abstraction—one that separates out the enumeration of domain concepts from the way in which those concepts might be used during problem-solving. Of course, the developers of rule-based systems in the 1970s contended that they had separated out “declarative knowledge” (in the form of rules) from “procedural knowledge” (in the form of *inference engines* that process those rules). Unfortunately, the inference engines of rule-based systems are special-purpose programs that operate on specific data structures (i.e., production rules). The rules, on the other hand, implicitly may encode considerable control-flow information [19]. In modern architectures, problem-solving methods are not procedures that operate on predefined data structures, but rather procedures that operate on *ontologies*. Maintaining the perspective of relating domain ontologies to well defined problem solvers offers the basis both for effective conceptual modeling and for system implementation using components at an appropriate level of abstraction.

This chapter has highlighted the use of reusable ontologies and problem-solving methods to build intelligent systems of a conventional nature. As evidenced throughout this book, however, the demands of Internet-based programming only will increase the importance of the reusable abstractions that we have described. The World Wide Web Consortium is now promoting the Resource Description Framework (RDF) as a standard means to annotate Web pages within machine-processable domain knowledge [53]. Knowledge bases and ontologies encoded in RDF and RDF-schema promise to interact with Internet-based problem solvers in ways that will greatly expand the scope of what we commonly view as a “knowledge-based system.” Intelligent systems promise to insinuate themselves ubiquitously into the very fabric of the World Wide Web. As the software-engineering community seeks scalable solutions to the problem of developing systems for this highly dynamic, distributed environment, large-grained abstractions such as domain ontologies and reusable problem-solving methods will inevitably be important components of Web-based software systems. There may always be a “software crisis” of some sort, but researchers are now responding with the kinds of abstractions that may make intelligent computer systems of all kinds easier to build and, more important, easier to maintain as requirements and domain knowledge evolve.

## **Acknowledgments**

This work has been supported by grant LM05708 from the United States National Library of Medicine, and by contracts supported by the Defense Advanced Research Projects Agency. The Protégé-2000 system for ontology editing and knowledge acquisition may be freely downloaded from our Web site under an open-source license agreement (<http://www.smi.stanford.edu/projects/protege>).

## References

- [1] A.L. Kidd (ed.), *Knowledge Acquisition for Expert Systems*. Plenum, New York, 1987.
- [2] M.L. Brodie, J. Mylopoulos, and J.W. Schmidt (eds.), *On Conceptual Modeling: Perspectives from Artificial Intelligence, Databases and Programming Languages*. Springer-Verlag, New York, 1984.
- [3] M. Stefik and D.G. Bobrow, Object-Oriented Programming: Themes and Variations. *AI Magazine* **6**(4) (1986) 40–62.
- [4] G. Schreiber, B. Wielinga, and J. Breuker (eds.) *KADS: A Principled Approach to Knowledge-Based System Development*, Academic Press, London, 1993.
- [5] F.P. Brooks, *The Mythical Man-Month*. Addison-Wesley, Reading MA, 1975.
- [6] F.P. Brooks, No Silver Bullet: Essence and Accidents of Software Engineering. *IEEE Computer* **20** (1987) 10–19.
- [7] J. Sametinger, *Software Engineering with Reusable Components*. Springer-Verlag, Berlin, 1997.
- [8] C.W. Krueger, Software Reuse. *ACM Computing Surveys* **24** (1992) 131–183.
- [9] D.A. Norman, Cognitive Engineering. In: D.A. Norman and S.W. Draper (eds), *User-Centered System Design*, pp. 31–61, Laurence Erlbaum, Hillsdale NJ, 1986.
- [10] E. Gamma, et al., *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading MA, 1995.
- [11] J. Breuker, et al., *Model Driven Knowledge Acquisition: Interpretation Models (Deliverable AI, ESPRIT Project 1098)*, 1987.
- [12] M.A. Musen, Dimensions of Knowledge Sharing and Reuse. *Computers and Biomedical Research*, **25** (1992) 435–67.
- [13] M.A. Musen and A.T. Schreiber, Architectures for Intelligent Systems Based on Reusable Components. *Artificial Intelligence in Medicine* **7** (1995) 189–99.
- [14] N. Shadbolt, K. O’Hara, and H. Cottam, The Use of Ontologies for Knowledge Acquisition. In: J. Cuenca, et al., (eds.) *Knowledge Engineering and Agent Technology*. IOS Press, Amsterdam, 2000, this volume.
- [15] J. McDermott, Preliminary Steps Toward a Taxonomy of Problem-Solving Methods. In: S. Marcus (ed), *Automatic Knowledge for Acquisition for Expert Systems*. Kluwer Academic Publishers, Boston, 1988, pp. 225–54.
- [16] R. Studer et al., Situation and Perspective of Knowledge Engineering. In: J. Cuenca, et al. (eds.), *Knowledge Engineering and Agent Technology*. IOS Press, Amsterdam, 2000, this volume.
- [17] R.O. Duda and E.H. Shortliffe, Expert Systems Research. *Science*, **220** (1983) 261–268.
- [18] J. Bachant and J. McDermott, R1 Revisited: Four Years in the Trenches. *AI Magazine* **5**(3) (1985) 21–32.

- [19] W.J. Clancey, The Epistemology of a Rule-Based Expert System: A Framework for Explanation. *Artificial Intelligence* **20** (1983) 215–51.
- [20] B.G. Buchanan and E.H. Shortliffe, Uncertainty and Evidential Support. In: Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project. Addison-Wesley, Reading, MA, 1984.
- [21] J. Bachant, RIME: Preliminary Work Toward a Knowledge-Acquisition Tool. In: S. Marcus (ed), Automatic knowledge for acquisition for expert systems. Kluwer Academic Publishers, Boston, 1988, pp. 201–24.
- [22] T. Bylander and B. Chandrasekaran, Generic Tasks for Knowledge-Based Reasoning: The "Right" Level of Abstraction for Knowledge Acquisition. In: B.R. Gaines and J.H. Boose (eds), Knowledge Acquisition for Knowledge-Based Systems. Academic Press, London, 1988, pp. 65–77.
- [23] N. Guarino, Formal Ontology, Conceptual Analysis and Knowledge Representation. *International Journal of Human-Computer Studies* **43** (1995) 625–40.
- [24] N. Guarino, Understanding, Building and Using Ontologies. *International Journal Of Human-Computer Studies* **42**(2/3) (1997) 293–310.
- [25] T.R. Gruber, A Translation Approach to Portable Ontology Specifications. *Knowledge Acquisition* **5** (1992) 199–220.
- [26] Y. Arens, C.A. Knoblock, and W.M. Shen, Query Reformation for Dynamic Information Integration. *Journal of Intelligent Information Systems: Integrating Artificial Intelligence and Database Technologies* **6**(2–3) (1996) 99–130.
- [27] W.J. Clancey, Heuristic Classification. *Artificial Intelligence* **27** (1985) 289–350.
- [28] J. S. Aikins, et al., PUFF: An Expert System for Interpretation of Pulmonary Function Data. *Computers and Biomedical Research* **16** (1983) 199–208.
- [29] B. Chandrasekaran, et al., Task-Structure Analysis for Knowledge Modeling. *Communications of the ACM* **35**(9) (1992) 124–137.
- [30] D. Fensel and R. Straatman, The Essence of Problem-Solving Methods: Making Assumptions to Gain Efficiency. *International Journal of Human-Computer Studies* **48**(2) (1998) 181–215.
- [31] H. Eriksson, et al., Task Modeling with Reusable Problem-Solving Methods. *Artificial Intelligence* **79** (1995) 293–326.
- [32] L. Eschelman, et al., MOLE: A Tenacious Knowledge Acquisition Tool. *International Journal of Man-Machine Studies* **26**(1) (1987) 41–54.
- [33] S. Marcus and J. McDermott, SALT: A Knowledge Acquisition Language for Propose-and-Revise Systems. *Artificial Intelligence* **39**(1) (1989) 1–37.
- [34] S.W. Tu, et al., Episodic Skeletal-Plan Refinement Based on Temporal Data. *Communications of the ACM* **32** (1989) 1439–55.
- [35] G. Schreiber, et al., Knowledge Engineering and Management: The CommonKADS Methodology. MIT Press, Cambridge, Massachusetts, 2000.
- [36] J.A. Brueker and W. Van de Velde (eds), The CommonKADS Library for Expertise Modeling. IOS Press, Amsterdam, 1994.

- [37] M.A. Musen, et al., Component-Based Support for Building Knowledge Acquisition Systems. Proceedings of the Conference on Intelligent Information Processing (IIP 2000) of the IFIP Sixteenth World Computer Congress (WCC 2000), Beijing, China, August, 2000, pp. 18–22.
- [38] W.E. Grosso, et al., Knowledge Modeling at the Millennium: The Design and Evolution of Protégé-2000. Proceedings of the Twelfth Knowledge Acquisition for Knowledge-Based Systems Workshop. Banff, Alberta, Canada, October, 1999.
- [39] Y. Gil and E. Melz, Explicit Representations of Problem-Solving Strategies to Support Knowledge Acquisition. In: National Conference on Artificial Intelligence. MIT Press, Portland, OR, 1996.
- [40] M.A. Musen, and S.W. Tu, Problem-Solving Models for Generation of Task-Specific Knowledge-Acquisition Tools. In: J. Cuenca (ed), Knowledge-Oriented Software Design, Elsevier, Amsterdam, 1993, pp. 23–50.
- [41] M. Molina and J. Cuenca, Using Knowledge Modelling Tools for Agent-Based Systems: The Experience of KSM. In: J. Cuenca, et al., (eds.) Knowledge Engineering and Agent Technology. IOS Press, Amsterdam, 2000, this volume.
- [42] Y. Demazeau and M. Ocelllo, Systems Development as Societies of Agents. In: J. Cuenca, et al., (eds.) Knowledge Engineering and Agent Technology. IOS Press, Amsterdam, 2000, this volume.
- [43] H. Eriksson, A.R. Puerta, and M.A. Musen, Generation of Knowledge-Acquisition Tools from Domain Ontologies. *International Journal of Human-Computer Studies* **41** (1994) 425–53.
- [44] Y. Shahar, et al., Semiautomated Entry of Clinical Temporal-Abstraction Knowledge. *Journal of the American Medical Informatics Association*, **6** (1999) 494–511.
- [45] T.E. Rothenfluh, et al., Reusable Ontologies, Knowledge-Acquisition Tools, and Performance Systems: PROTÉGÉ-II Solutions to Sisyphus-2. *International Journal of Human-Computer Studies* **44** (1996) 303–32.
- [46] J.H. Gennari, R.B. Altman, and M.A. Musen, Reuse with PROTÉGÉ-II: From Elevators to Ribosomes. In: SSR'95: ACM-SIGSOFT Symposium on Software Reusability, Seattle, WA, 1995.
- [47] J.Y. Park and M.A. Musen, VM-in-Protégé: A Study of Software Reuse. In: A.T. McCray, B. Cesnik, and J.R. Scherrer (eds), MEDINFO '98, Seoul, South Korea, IOS Press, Amsterdam, 1998, pp. 644–648.
- [48] D.S. Smith, J.Y. Park, and M.A. Musen, Therapy Planning as Constraint Satisfaction: A Computer-Based Antiretroviral Therapy Advisor for the Management of HIV. In: C.G. Chute (ed), 1998 AMIA Annual Fall Symposium. Orlando, FL, 1998, pp.627–631.
- [49] J.H. Gennari, et al., Mapping Domains to Methods in Support of Reuse. *International Journal of Human-Computer Studies* **41** (1994) 399–424.
- [50] J.Y. Park and M.A. Musen, Mappings for Reuse in Knowledge-Based Systems. In: R. Roy (ed), Industrial Knowledge Management. A Micro Level Approach, Springer-Verlag, London, in press.
- [51] D. Fensel and E. Motta, Structured Development of Protein Solving Methods. In: 11th Banff Knowledge Acquisition for Knowledge-Based Systems Workshop, Banff, Alberta, Canada, 1998.
- [52] S.W. Tu and M.A. Musen, Episodic Refinement of Episodic Skeletal Plan Refinement. *International Journal of Human-Computer Studies* **48** (1998) 475–497.
- [53] World Wide Web Consortium, Resource Description Framework (RDF), online: <http://www.w3.org/RDF/>