Available online at www.sciencedirect.com
SCIENCE DIRECT®
ELSEVIER
Int. J. Human-Computer Studies 58 (2003) 89–123
International Journal of Human-Computer Studies
www.elsevier.com/locate/ijhcs

The evolution of Protégé: an environment for knowledge-based systems development

John H. Gennari[a,*], Mark A. Musen[b], Ray W. Fergerson[b], William E. Grosso[d], Monica Crubézy[b], Henrik Eriksson[c], Natalya F. Noy[b], Samson W. Tu[b]

[a] *Biomedical and Health Informatics, University of Washington, USA*
[b] *Stanford Medical Informatics, Stanford University, USA*
[c] *Department of Computer and Information Science, Linköping University, Sweden*
[d] *Hipbone Inc., San Carlos, CA, USA*

Received 4 October 2002; accepted 10 October 2002
Paper accepted for publication by Editor, B.R. Gaines

Abstract

The Protégé project has come a long way since Mark Musen first built the Protégé meta-tool for knowledge-based systems in 1987. The original tool was a small application, aimed at building knowledge-acquisition tools for a few specialized programs in medical planning. From this initial tool, the Protégé system has evolved into a durable, extensible platform for knowledge-based systems development and research. The current version, Protégé-2000, can be run on a variety of platforms, supports customized user-interface extensions, incorporates the Open Knowledge-Base Connectivity (OKBC) knowledge model, interacts with standard storage formats such as relational databases, XML, and RDF, and has been used by hundreds of individuals and research groups. In this paper, we follow the evolution of the Protégé project through three distinct re-implementations. We describe our overall methodology, our design decisions, and the lessons we have learned over the duration of the project. We believe that our success is one of infrastructure: Protégé is a flexible, well-supported, and robust development environment. Using Protégé, developers and domain experts can easily build effective knowledge-based systems, and researchers can explore ideas in a variety of knowledge-based domains.

Keywords: Knowledge-bases; Problem-solving method; Protégé meta-tool

*Corresponding author. Department of Medical Education and Biomedical Informatics, Washington School of Medicine, 1959 NE Pacific St., Box 35 72 40, Seattle, WA 98195-7240, USA. Tel.: +1-206-616-6641.

E-mail address: gennari@u.washington.edu (J.H. Gennari).

1071-5819/03/$ - see front matter
doi:10.1016/S1071-5819(02)00127-1

1. Motivation and protégé timeline

The Protégé system is an environment for knowledge-based systems development that has been evolving for over a decade. Protégé began as a small application designed for a medical domain (protocol-based therapy planning), but has evolved into a much more general-purpose set of tools. More recently, Protégé has developed a world-wide community of users, who themselves are adding to Protégé's capabilities, and directing its further evolution.

The original goal of Protégé was to reduce the knowledge-acquisition bottleneck (Hayes-Roth et al., 1983) by minimizing the role of the knowledge engineer in constructing knowledge bases. In order to do this, Musen posited that knowledge-acquisition proceeds in well-defined stages and that knowledge acquired in one stage could be used to generate and customize knowledge-acquisition tools for subsequent stages (Musen, 1989a, b). Thus, the original version of the Protégé software (hereafter referred to as Protégé-I) was an application that took advantage of structured information to simplify the knowledge-acquisition process. Musen described Protégé-I as follows:

> Protégé is neither an expert system itself nor a program that builds expert systems directly; instead, Protégé is a tool that helps users build *other tools* that are custom-tailored to assist with knowledge-acquisition for expert systems in specific application areas. (Musen, 1989a, p. 2)

Protégé-I demonstrated the viability of this approach, and of the use of task-specific knowledge to generate and customize knowledge-acquisition tools. However, it was custom-tailored to support a particular type of application. In particular, Protégé-I grew out of the Oncocin project and subsequent attempts to build expert systems for protocol-based medical therapy planning (Shortliffe et al., 1981). Thus, the original system was limited in its application.

Over more than a decade, through four distinct releases, the Knowledge Modeling Group at Stanford Medical Informatics has worked to turn Protégé into a general-purpose environment for knowledge modeling. Fig. 1 shows a chronology of these four releases, along with a few of the salient features of each release. The current system, Protégé-2000, is far more general than the original version, yet it maintains the original focus on the use of meta-knowledge to create usable knowledge-acquisition tools.

We have guided the evolution of Protégé by a build-and-test methodology: We built systems and tested each generation of Protégé on real-world problems so as to solicit user feedback about strengths and weaknesses. Thus, our motivation has been more pragmatic than theoretic: Our primary goal has been to make Protégé an easily used environment, with the assumption that research issues will arise from use. Our research in knowledge-based systems has been grounded in this cycle of development: New versions of Protégé have not only extended its technical capability, but also uncovered new research questions to investigate.

In this paper, we describe the evolution of Protégé. We begin with a discussion of knowledge acquisition in the mid-1980s and a description of Opal, the immediate

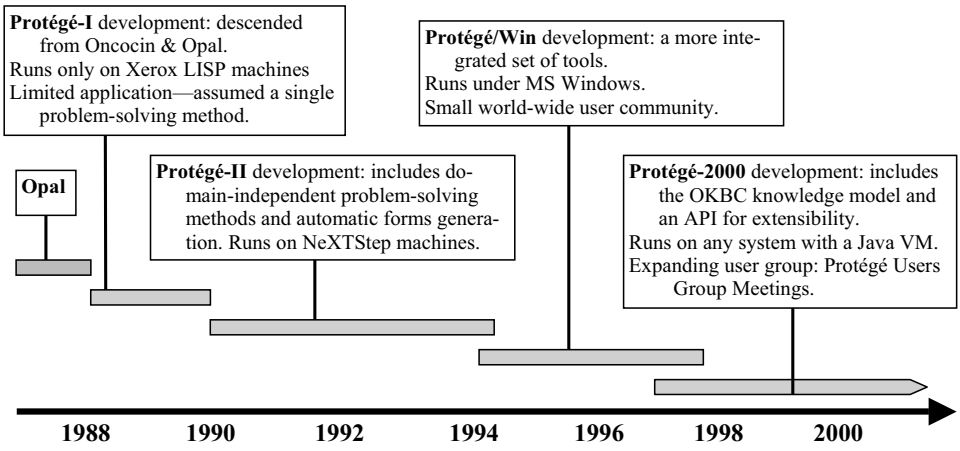


Fig. 1. A chronology of Protégé development.

ancestor of Protégé-I. We then follow Protégé through the three re-implementations shown in Fig. 1, discussing how they differ, and how they have led to a broader and more general-purpose environment. One conclusion we draw from this evolution is the significance of testing ideas on real-world problems. Our iterative methodology insured that the requirements for Protégé and the architecture for its implementation were constantly questioned and reconsidered. A related conclusion we draw is that long-term projects such as Protégé that span more than a decade can yield important results—the sorts of re-implementations we built could not have been carried out in the time span that is more typical for academic research projects (often 3 years, and at most 5 years).

2. Protégé roots: expert systems and knowledge acquisition in the 1980s

The early 1980s were a heady time for Artificial Intelligence (AI). Expert-systems research had produced some stunning successes (Bachant and McDermott, 1984; Buchanan and Shortliffe, 1984). To many people in the field, it seemed that AI was on the verge of a dramatic breakthrough. Perhaps, Hayes-Roth et al. put it best when they wrote:

> Over time, the knowledge engineering field will have an impact on all areas of human activity where knowledge provides the power for solving important problems. We can foresee two beneficial effects. The first and most obvious will be the development of knowledge systems that replicate and autonomously apply human expertise. For these systems, knowledge engineering will provide the technology for converting human knowledge into industrial power. The second benefit may be less obvious. As an inevitable side effect, knowledge engineering will catalyze a global effort to collect, codify, exchange and exploit applicable

Table 1
The classical model of expert system development (after Buchanan et al., 1983)

Stage	Description	Performed by
1. Identification	Characterize important aspects of problem. Identify participants, problem characteristics, resources, and goals.	Domain experts, knowledge engineer
2. Conceptualization	Make key concepts and relations from the Identification stage explicit.	Knowledge engineer
3. Formalization	Identified concepts are represented in a formal language.	Knowledge engineer
4. Implementation	Knowledge from formalization stage is represented in an expert-system shell.	Knowledge engineer
5. Testing	The completed system is tested on sample cases and weaknesses are identified	Domain experts, knowledge engineer
6. Revision	Redesign and reimplement the system, in light of the results from testing.	Knowledge engineer

> forms of human knowledge. In this way, knowledge engineering will accelerate the development, clarification, and expansion of human knowledge. (Hayes-Roth et al., 1983, p. xi)

However, even in this volume, which in some ways represents the high-water mark of expert-system optimism, the bulk of the papers discussed difficulties inherent in developing expert systems. In particular, the volume discussed the difficulties inherent in modeling expert knowledge, beginning with an examination of the role of the knowledge engineer, who is involved in all phases of system construction (Buchanan et al., 1983). The knowledge engineer must become familiar with the problem domain, characterize the reasoning tasks necessary to solve the problem, identify the major domain concepts, categorize the type of knowledge necessary to solve the problem, identify the reasoning strategies used by experts, define an inference structure for the resulting application, and formalize all this knowledge in a generic and reusable way. Table 1 summarizes this general-purpose iterative model of system development, where the knowledge engineers participate in all stages of development, while domain experts are seen simply as resources for knowledge engineers to draw upon.

2.1. Expert-systems shells

The classical model of expert-system development is based on the idea of an expert-system *shell*—an inference engine that knowledge engineers can reuse with different knowledge bases, to produce different expert systems. Fig. 2 diagrams this early view of expert-system development. In this view, *knowledge acquisition* was the work a knowledge engineer carried out when building a particular knowledge base. As became apparent, not only was it difficult and time-consuming to build knowledge bases, the introduction of a knowledge engineer between the domain

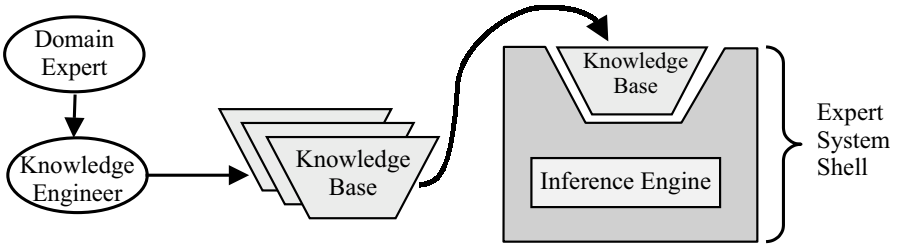


Fig. 2. Expert-system development from a reusable shell and knowledge bases.

expert and the knowledge base could lead to errors and misunderstandings. One of the main goals of Protégé was to overcome this problem, allowing domain experts to build knowledge bases more directly.

2.2. Protégé ancestry: Oncocin and Opal

The evolution of Protégé began with the development of Opal (Musen, 1987), a knowledge-acquisition tool for the Oncocin system (Shortliffe et al., 1981). Oncocin was an advice system for protocol-based cancer therapy, where information about the history of a specific patient could be entered by a domain user (a physician or nurse) and the system would give advice about treatment and tests. The original Oncocin was developed using the methodology of Table 1 and Fig. 2—knowledge engineers talked to medical specialists and then created the knowledge base used in the Oncocin system. The knowledge base was a set of if–then rules and other data structures that captured the semantics of cancer clinical trial protocols.

The goal of the Opal knowledge-acquisition tool was to allow the domain expert to enter *directly* some forms of domain-specific knowledge (i.e. to remove the knowledge engineer from stage 4 of Table 1). Opal used domain-specific concepts and ideas in order to present cancer specialists with carefully designed forms for structured data entry. Instead of typing individual production rules, the physicians themselves described complete cancer protocols by filling out special-purpose graphical forms.

Opal then translated the expert's input into Oncocin's internal representation. Fig. 3 shows the data flow that resulted from this approach. This design implicitly asserts a qualitative division among the types of information a knowledge-based system requires.[1] There are at least three different types of knowledge implicit in Fig. 3: (1) the knowledge engineer needs *structural domain concepts* in order to build the Opal knowledge-acquisition tool, (2) a domain expert (oncologist) must instantiate these concepts with *domain knowledge* (for Oncocin, these are the oncology protocols), and (3) the end-user provides *case data* to the resulting expert

[1] Sometime during the early 1990s, the term "expert system" became less fashionable. Since then, the more general term of "knowledge-based system" has been used, to emphasize knowledge, and to indicate that such systems are not intended as replacements for human expertise.

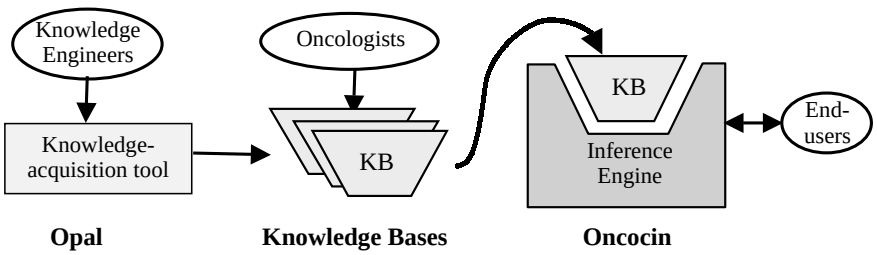


Fig. 3. The Oncocin expert system and Opal, a tool for building Oncocin knowledge bases. Knowledge engineers who understand the domain build Opal, and provide it with *structural knowledge*; oncologists use Opal to build *domain knowledge* into knowledge bases; and end-users provide Oncocin with *case data* at run-time.

system for decision support in particular cases of oncology treatment. The assertion that information can be partitioned in this way was crucial to Opal, Oncocin, and, as we will show, to Protégé.

By adopting this information-partitioning hypothesis, Oncocin and Opal modified the classical model of knowledge-based system development in two ways. First, the structural domain concepts identified in stage 2 (see Table 1) were built into actual tools developed for knowledge acquisition—Opal included concepts that were specific to the domain of protocol-based health care. Second, in stage 4, the domain expert is directly responsible for building the knowledge base. Opal aimed to improve the classical expert-system development approach by moving tasks from the knowledge engineers to the domain experts, thereby reducing the likelihood of errors and streamlining knowledge base construction (Sandahl, 1994).

3. Protégé-I

Protégé-I (Musen, 1989a, b) began as a generalization of the Oncocin/Opal architecture—rather than expecting knowledge engineers to build new knowledge-acquisition tools like Opal for every new domain, the Protégé meta-tool *generated* a knowledge-acquisition tool (KA-tool) from a set of structural concepts. Thus, we designed Protégé-I to further reduce the load on a knowledge engineer. Fig. 4 shows a flow diagram for the use of Protégé-I, with three classes of users: (1) knowledge engineers provide structural concepts and use Protégé-I to build the KA-tool, (2) domain experts use the KA tool to build and edit specific knowledge bases, and (3) end-users interact with the final expert system for decision support. This figure makes clear that Protégé-I adopts the information-partitioning hypothesis, dividing the knowledge and uses of a knowledge-based system into three types.

The general methodology is that knowledge is acquired in stages, and the information acquired in each stage is meta-knowledge for subsequent stages that

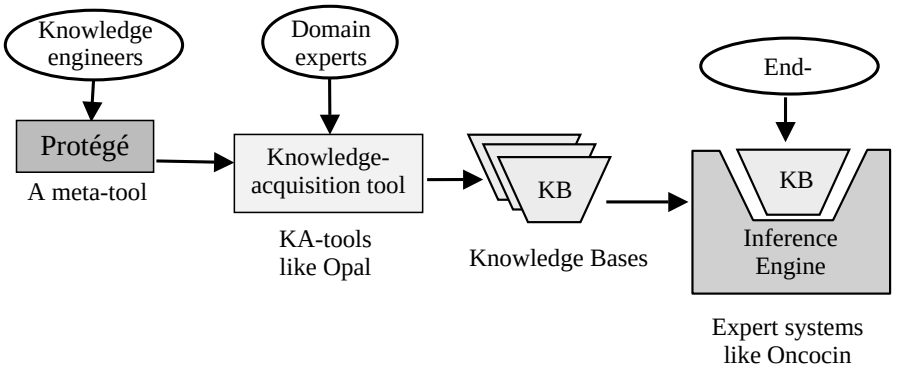


Fig. 4. A diagram showing the use of the Protégé-1. Opal has been replaced by a customizable, knowledge-acquisition tool automatically generated from previously acquired domain structural concepts (Musen, 1989a, b).

helps lower the barriers to knowledge acquisition. Acquiring the domain structural concepts is simplified by the presence of core structural concepts. In turn, the domain structural concepts are used to automatically generate user-interface forms, which make it easier to acquire domain knowledge from experts.

3.1. Assumptions of Protégé-I

In order to build a system in which this methodology could be quickly tested, Musen initially made some strong methodological assumptions (Musen, 1989a). These assumptions were natural given the early need to show that that the overall system could actually turn over and produce a usable knowledge base. As we evolved the Protégé environment to be more general purpose, we were able to relax each of the following assumptions in later versions of Protégé.

3.1.1. Knowledge bases are problem-specific artifacts

Protégé grew out of the Oncocin and Opal systems, and therefore assumed that the knowledge bases being constructed were for use with the Oncocin inference engine. As part of this inference engine, Oncocin used the *episodic skeletal-plan refinement* method, or ESPR (Tu et al., 1989), a planning method based on the instantiation and gradual refinement of skeletal plans for specific problems.

Because it assumed knowledge about the problem-solving process during knowledge acquisition, and because the knowledge-acquisition process was so highly structured, Protégé-I knowledge bases included only those concepts and distinctions that were important for the ESPR problem solving method. Therefore, the KA-tools generated by Protégé-I and the resulting knowledge bases were really only well-suited for applications that used some sort of skeletal planning algorithm.

3.1.2. The problem-solving method provides semantics

In Protégé-I, very little attention was paid to specifying a *formal* knowledge model, such as promoted by Hayes, (1979), or by Chaudhri et al. (1998). In some sense, a formal model for specifying the semantics of Protégé-I knowledge bases was unnecessary because the ESPR problem-solving method operationally defined these semantics. A more formal semantic model became necessary only when Protégé knowledge bases were used by multiple systems and multiple problem-solving methods.

3.1.3. Knowledge bases and problem-solving methods are atomic

Finally, when Protégé-I was developed, we assumed that knowledge bases and inference engines were atomic. That is, the acquired knowledge bases were entirely self-contained—they did not reference any other knowledge bases or knowledge sources at all. Likewise, we assumed that there was only a single, monolithic problem-solving method—the ESPR algorithm. Later, as Protégé applications and knowledge bases become larger and more complex, we adopted a more componential view of both knowledge bases and problem-solving methods (Musen and Tu, 1993).

3.2. Historical counterparts to Protégé-I

Protégé was not the only knowledge-acquisition tool built in the mid-1980s that attempted to use knowledge about the target inference algorithm to simplify knowledge acquisition. Systems such as MOLE (Eshelman et al., 1987) and SALT (Marcus and McDermott, 1989) made similar assumptions about their inference algorithms, also with the goal of simplifying knowledge acquisition. However, these systems, and their ideological descendants, such as Expect (Gil and Melz, 1996), used this knowledge in a very different way from Protégé. Rather than focusing on the construction of an easy-to-use KA-tool, systems like MOLE aimed to correct and complete the user's knowledge base:

> MOLE understands that experts are not very good at providing such information and so does not require that the expert provide a fully specified network of associations. Instead, MOLE relies on its expectations about the world and how experts enter data in order to mold an under-specified network of associations into a consistent and unambiguous knowledge-base. (Eshelman et al., 1987, p. 46)

This approach requires a stronger set of assumptions about the inference algorithm that will be used with the knowledge base. As we will see, we took the opposite tack, weakening such assumptions, so that Protégé knowledge bases would become more general purpose, and reusable by multiple problem-solving methods.

3.3. Summary of Protégé-I

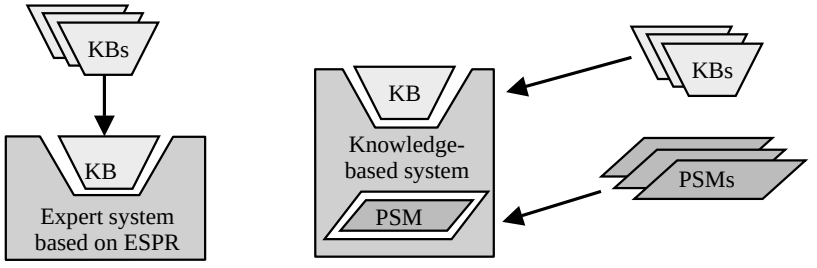
Protégé-I succeeded in substantially lowering the barriers to knowledge acquisition for medical advice systems. Protégé-I was important because it introduced the

idea of generating knowledge-acquisition tools from structured meta-knowledge. However, the knowledge bases that these systems built were neither reusable nor general purpose—Protégé-I knowledge bases were method specific (for use with skeletal planners only), and lacked any formal semantics that might make them interpretable in other settings. The assumptions made for Protégé-I had the effect of limiting the system's use to well-understood medical examples where episodic skeletal-plan refinement could be applied. To generalize Protégé, we had to go beyond ESPR, and this was a major goal of the Protégé-II development.

4. Protégé-II: problem-solving methods and the downhill flow assumption

The most significant difference between the original Protégé and the Protégé-II version was the idea of *reusable problem-solving methods*. Following Chandrasekaran (1983, 1986), Protégé-II allowed developers to build inference mechanisms in an entirely separate component, a problem-solving method, which could be developed independently from the knowledge base. These problem-solving methods (PSMs) were generic algorithms that could be used with different knowledge bases to solve different real-world tasks. Examples of problem-solving methods include the episodic planning algorithm used by Protégé-I, and methods such as "propose-and-revise" used by the SALT system to solve constraint-satisfaction problems. As Fig. 5a shows, Protégé was initially designed with a single problem-solving method (the ESPR method) which could be applied to different knowledge bases. Fig. 5b shows the shift to a more generic, component-based approach, where alternative problem-solving methods are applied to knowledge bases.

As we describe in greater detail below, to achieve the component-based reuse shown in Fig. 5b, we needed additional constructs and modifications to the Protégé methodology. In general, to support these new capabilities, we needed to use the notion of an *ontology*, a formal model of a shared domain of discourse (Guarino and Giaretti, 1995). In Protégé, our domain ontologies became the basis for the



(a) Protégé-I (b) The Protégé-II approach

Fig. 5. The evolving Protégé methodology. (a) Although different knowledge bases (KBs) could be created with Protégé-I, they were all expected to work with the ESPR problem-solving method, and this structure was built into the knowledge bases. (b) In Protégé-II, both knowledge bases and problem-solving methods are reusable components that become part of the resulting knowledge-based system.

generation of knowledge-acquisition tools. We also used ontologies to specify the knowledge representational and communication needs of problem-solving methods. Finally, our development of a methodology for building ontologies, problem-solving methods, and knowledge bases led to an extension of the "information partitioning hypothesis" that we presented earlier: Definitions of classes in an ontology lead "downhill" to knowledge-acquisition tools, which then lead downhill to specific knowledge bases.

4.1. Developing knowledge-based systems with Protégé-II

Protégé-II extended the original two-step process—generating a knowledge-acquisition tool and using it to instantiating a knowledge base—with additional steps that dealt with the problem-solving method. The Protégé-II methodology consisted of: (1) developing or reusing a problem-solving method, (2) defining an appropriate domain ontology, (3) generating a knowledge-acquisition tool, (4) building a knowledge base using the tool, and (5) integrating these components into a knowledge-based system by defining mappings between problem-solving methods and specific knowledge bases. Below, we describe each of these steps in detail.

4.1.1. Developing or reusing a problem-solving method

One benefit of separating problem-solving methods from knowledge bases, is that the former are relatively stable. Domain knowledge can quickly become out-of-date and require substantial modifications. In contrast, problem-solving methods, such as propose-and-revise seem well understood and less likely to change. Thus, once a problem-solving method is defined, it can be reused with a number of different knowledge bases.

To further enhance reuse benefits, the Protégé-II methodology included the idea of *decomposable* problem-solving methods (Puerta et al., 1992; Puerta et al., 1993; Eriksson et al., 1995). Thus, methods could be decomposed into *sub-methods* for carrying out smaller *sub-tasks* within the larger method. This approach might allow developers to configure a generic problem solver for more specific tasks by selecting the appropriate sub-methods for the problem at hand. Eriksson et al. described how to specialize chronological backtracking to other methods, such as the board-game method and the propose-and-revise methods (Eriksson et al., 1995).

4.1.2. Defining an ontology

Protégé-II formalized the ontologies that constrained its knowledge bases and knowledge-acquisition tools. Protégé-II was designed to leverage the CLIPS expert-system shell and its knowledge representation system (see http://www.ghg.net/clips/CLIPS.html). Because this was an object-oriented sub-system, we designed our ontologies to use a frame-based formalism. Thus, Protégé-II took an important first step by providing a precise language for its ontologies, based on standard frame language concepts of *classes*, *instances*, *slots*, and *facets*. Multiple inheritance among classes was supported. Being more precise about how to specify ontologies provided

several benefits:

- It clarified the conflation between structural knowledge about the domain vs. more specific knowledge supplied by the domain expert. Protégé-II used classes for the former, and instances for the latter sort of knowledge.
- It allowed graphical tools to manipulate ontologies in a user-friendly way. Protégé-II included a tool called Maître, which enabled developers to edit ontologies graphically (Gennari, 1993).
- It enabled early experiments in knowledge-base reuse across knowledge-modeling frameworks. For example, the Ontolingua system (Gruber, 1993) included a Protégé translator that allowed Ontolingua knowledge bases to be exported to and imported from Protégé-II. Although this capability did not result in significant knowledge sharing, this attempt at interoperation led to the development of more robust knowledge representation standards, which we embraced with Protégé-2000.

Ontologies can be used for different purposes, and the Protégé-II approach included three classes of ontologies—*domain*, *method*, and *application* ontologies (Gennari et al., 1994; Eriksson et al., 1995). *Domain* ontologies define the concepts related to an application domain (e.g. different symptoms, anomalies, and remedies). *Method* ontologies specify the data requirements of the problem-solving methods (i.e. the input and output structure of each method). *Application* ontologies define the concepts that are specific to a particular application or implementation. Thus, domain and method ontologies can be potentially reused across several applications, whereas the application ontologies are specific and not intended for reuse. Later in this section, we describe how these different ontologies can be integrated via mappings that connect particular elements across the ontologies.

4.1.3. Generating a knowledge-acquisition tool

Like Protégé-I, Protégé-II was a meta-level knowledge acquisition system that generated domain-specific knowledge-acquisition tools (KA-tools). Unlike Protégé-I, we built Protégé-II to be independent of any particular problem-solving method. Instead, the domain ontology is the basis for KA-tool generation. Fig. 6 shows a flow through the three subcomponents of Protégé-II: Maître, for building ontologies, Dash, for manipulating the default layout of the KA-tool, and Meditor, which domain experts used to build and edit knowledge bases. As shown in the figure, the process started with ontology construction. The class definitions in this ontology defined the forms in the KA-tool that Dash produced, and these forms would then be used within Meditor to acquire instances of the classes that made up the knowledge base.

Dash provided a significant benefit by allowing users to manipulate the format and arrangement of all knowledge-acquisition forms, thereby custom-tailoring the layout of the KA-tool (Eriksson et al., 1994). The most significant benefit of the new, ontology-based approach is that it allowed more rapid KA-tool development. Users could more easily experiment with different versions of a KA-tool, and developers could refine ontologies and their corresponding KA-tools incrementally and in parallel.

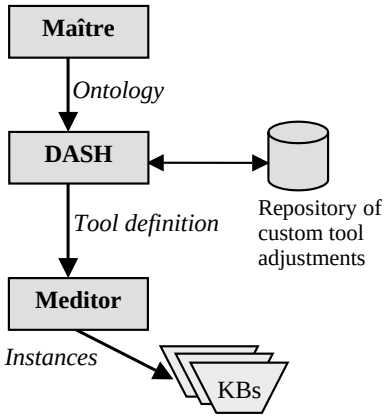


Fig. 6. Protégé-II components for building knowledge bases. The knowledge engineer uses Maître to define ontologies. Dash generates a tool definition based on the ontology. Finally, Meditor runs the tool definition and allows domain experts to build and edit knowledge bases.

4.1.4. Building a Knowledge Base using the tool

Once the knowledge-acquisition tool is completed the next step is to build the knowledge base. For Protégé-II, this meant that domain experts created instances of those classes that were pre-defined in the domain ontology. Thus, there was an assumption that instances can be created only after their classes have been fully defined. This assumption became part of a general "downhill flow" model of knowledge acquisition. Knowledge-base construction begins at the "top," with higher-level class definitions, and then proceeds downhill to the process of creating instances of those classes. As Fig. 6 shows, knowledge-base developers had to (1) define an ontology, (2) generate a KA-tool, and (3) build the knowledge base in that order. Diverting from the downhill flow often meant additional work. Making changes to the ontology after instances in the knowledge base had been created resulted in certain problems:

- Changes to a class definition affected all instances of that class in a knowledge base, possibly removing information or making information invalid in those instances.
- Changes to a class definition affected the forms in the knowledge-acquisition tool generated by Dash and therefore could change (e.g. invalidate or alter) domain knowledge.

Dash included functionality for partially alleviating the latter problem by storing custom adjustments from previous sessions in a repository (see Fig. 6). Nonetheless, we found that the Protégé-II approach made it difficult for developers to refine or change the ontology (e.g. creating new classes) after a cycle of KA-tool generation and knowledge-base construction.

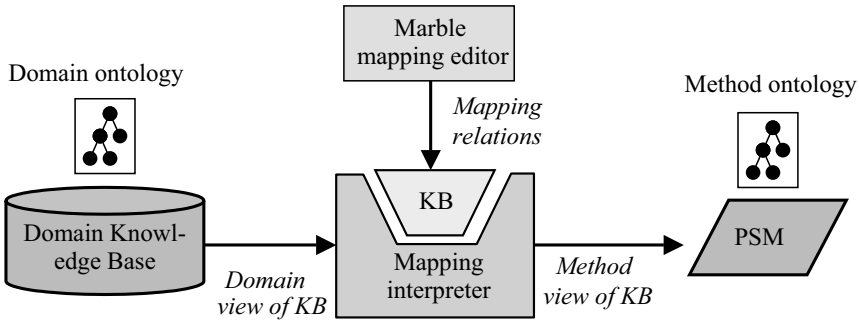


Fig. 7. Using mappings to connect reusable PSMs to knowledge bases. Marble was a special KA-tool for building knowledge bases of mapping relations. These relations describe how the input requirements of a particular problem-solving method are met by a particular knowledge base. The method and domain ontologies are used to build the mapping relations.

4.1.5. Integrating the components of a knowledge-based system—mappings

A knowledge-based system is not complete until its components have been integrated into a working unit. After developers have selected the problem-solving method, built a domain ontology, created a KA-tool, and populated the knowledge base, the resulting system must be assembled for actual use.[2] However, since the Protégé-II methodology separated the processes of problem-solving method development from the task of knowledge base development, it is likely that the requirements of the problem-solving method were not exactly matched by the classes and instances in the knowledge base. Thus, we introduced the idea of *mapping relations* to connect knowledge bases and problem-solving methods, while allowing each of these components to be independent and reusable (Gennari et al., 1994).

To connect a particular knowledge base to a particular problem-solving method, developers must examine the source and target class definitions and create an appropriate set of mapping relations. To help with this task, we developed a special KA-tool, known as Marble, that built knowledge bases of mappings. Mappings created with Marble were then applied by a generic mappings interpreter to a source knowledge base, thereby producing an appropriate view of this information for the target (the problem-solving method). Fig. 7 illustrates this process.

In an earlier publication, we discussed a basic example of this style of reuse in Protégé-II (Gennari et al., 1995). We implemented a simple constraint-satisfaction method, propose-and-revise, and then reused this method in an application that worked with ribosomal configurations. As one would expect, the knowledge base about base-pair sequences and configuration constraints had to be viewed from a particular perspective before we could apply the propose-and-revise problem-solving method. Rather than modifying the ribosomal knowledge base, we built mapping relations to provide this method-specific view of the knowledge base.

[2] It is sometimes useful to build knowledge bases **without** any problem-solving method or performance system, such as knowledge bases for documentation or information visualization. However, when we developed Protégé-II, we viewed such knowledge bases as unusual and exceptional.

The introduction of mappings and Marble demonstrated how component reuse could occur with Protégé knowledge-based systems. Although these mapping relations do not guarantee successful reuse, they help make it more practical. One compelling, longer-term goal is to provide partial automation to the process of connecting components. For example, given a library of problem-solving methods, and a task specification, it might be feasible to select an appropriate method and then to generate putative mappings to a particular domain ontology (Crubezy et al., 2001). Such automation has been the focus of recent research in our group.

4.2. Historical counterparts to Protégé-II

The idea of reusable problem-solving methods was not unique to Protégé. Chandrasekaran (1983, 1986) was one of the first researchers to recognize the need for reusable methods. His approach to *generic tasks* attempted to create a taxonomy of different types of problem solving. By the late-1980s, researchers had begun discussing and formalizing the idea of libraries of inference algorithms, which could be easily reused across a number of different applications (McDermott, 1988).

One approach to include a library of reusable inference patterns was the KADS methodology (Wielinga et al., 1992). Like Protégé, KADS is a methodology for developing knowledge-based systems. However, KADS takes a larger, more systems-level view of the process, including early stage requirements and knowledge management analysis. Protégé focuses exclusively on the latter stages of knowledge-base development, which KADS refers to as the *model of expertise*. With the KADS methodology, developers build up the model of expertise by providing four layers of knowledge: (1) the *domain layer*, for static knowledge about the domain, (2) the *inference layer*, for procedural knowledge about the application task, (3) the *task layer*, for sequencing a set of inferences to provide a complete problem-solving solution, and (4) the *strategy layer*, for selecting among alternative tasks. In Protégé-II, the domain layer was captured in the domain ontology, while the other three layers were folded into the choice and development of an appropriate problem-solving method.

A second significant difference between KADS and Protégé is that Protégé models are operational: The ontology is compiled into a knowledge-acquisition tool, which is used to build a knowledge base that we expect to be directly accessed and used by the selected problem-solving method. In contrast, a KADS model of expertise is most typically a written specification, which mainly provides guidance for engineers who must design and build the system as a separate step. In many settings, the organizational and knowledge management problems overwhelm any technical issues with implementation. In such situations, the KADS approach seems appropriate, because it focuses on solving these problems, and relegates implementation as an easier final step of construction from the specifications. Protégé focuses exclusively on this last step—thus, it could be used in conjunction with a KADS approach, and in settings where there is significant complexity in the knowledge modeling and application-building phases of development.

At around the same time, Steels implemented the KREST work bench (Steels, 1990), based on his view of *components of expertise* (Steels, 1992). KREST had similarities to Protégé-II, including the use of domain ontologies (models) and problem-solving methods as the basic building blocks. Except for requiring a method ontology, Protégé-II treated the actual development of a problem-solving method as an external task, which allowed for independence from programming language choices. In contrast, KREST required an internal development of a problem-solving method, and exposed the implementation details (in Common Lisp) to the system builder.

As with Protégé, KADS also has evolved over time. CommonKADS arose by using more precise, formal languages for models of expertise as well as by including some ideas from Steels' components of expertise (Schreiber et al., 1994, 1999). CommonKADS also introduced the idea of a library of reusable problem-solving methods, built up from the KADS inference layer and task layer models (Breuker and Velde, 1994).

In the United States, researchers began to experiment with the incorporation of discrete problem-solving methods within systems based on *description logic* (Gil and Melz, 1996). The problem-solving methods included within the EXPECT architecture tended to be more domain-specific and of smaller granularity than those used within Protégé-II, but the approach corroborated the utility of using problem-solving methods as an abstraction for control knowledge in large knowledge-based systems. EXPECT also demonstrated the value of description logic in facilitating the acquisition of knowledge-base instances.

4.3. Summary of Protégé-II

Protégé-II was a significant extension and generalization of the original Protégé system. Important changes included:

- Reuse of problem-solving methods as components (rather than the use of a single monolithic problem-solving method). The goal was to remove the implicit semantics of the ESPR method from Protégé-I, and to allow for alternative problem-solving methods.
- Ontologies and the adoption of a more formal frame-based representation language. Ontologies play an important role in Protégé-II. Both as input and output definitions for problem-solving methods and as the basis for generation of knowledge-acquisition tools.
- Generation of knowledge-acquisition tools from any ontology (rather than from the instantiation of ESPR). Protégé-II took the meta-tool idea one step further and streamlined generation and custom adjustments of knowledge-acquisition tools.
- The "downhill flow" assumption of classes over instances. The Protégé-II development process assumed that ontologies (classes) were more durable than knowledge bases (instances). We expected that knowledge engineers would use

one tool to define classes, and then domain experts would use a separate tool (the KA-tool) to create and edit instances.
- Declarative mappings between knowledge bases. Protégé-II introduced mappings as an approach to bridge the gap between different types of ontologies, and to enable reuse of both ontologies and problem-solving methods.

Protégé-II was designed as a suite of applications (Maitre, Dash, Meditor, Marble) to support the developer and to define the Protégé methodology. Although these tools clarified the distinct steps in the process of building a knowledge-based system, as we gained more experience and more users, we found that switching back and forth among the distinct tools eventually became burdensome. As a partial response, we added a control-panel application to Protégé-II, from which each of the separate applications could be launched. This control panel was a forerunner for the more tightly integrated Protégé/Win and Protégé-2000 systems.

An interesting aspect of this phase of Protégé development is that, in retrospect, many of the important changes involved moving from an informal model to a more formal one. The central change, the removal of method dependencies from the performance system, required a more formal knowledge model, which led to a deeper understanding of the knowledge-base structures required by problem-solving methods, and which spurred our development of mappings.

5. Protégé/Win: popularizing knowledge-based systems

Protégé-II introduced a number of significant conceptual changes to the basic Protégé idea. In contrast, the development of Protégé/Win was primarily motivated by a pragmatic concern: Protégé-II was built to run only on the NeXTStep operating system, and to expand our user base, we needed to re-implement our system to run under the Windows operating system.[3] However, given the requirement to re-implement, we choose to take this as an opportunity to improve on our system in several ways:

- We allowed for the use of modular ontologies, via an ontology inclusion mechanism.
- We designed for a more integrated, streamlined set of tools.
- We improved the task of custom-tailoring the knowledge-acquisition tool, storing layout and format information separately from either the ontology or the knowledge base.

However, the most important contribution of Protégé/Win was the development of a significant external users group. Protégé/Win was freely available as an easy-to-install application for any academic user. For the first time, we began to get off-site,

[3] This choice of operating system was also motivated by our funding sources. In any event, moving away from a NeXTStep-based system proved fortuitous, as the company went out of business circa 1996.

real-world feedback about problems and features of the system. As we describe, this feedback strongly influenced our development.

5.1. Includable ontologies

As users began making larger and larger knowledge bases, it became difficult to model an entire domain with a monolithic ontology. Because similar applications often share common concepts and terminology, we wanted to identify and build *reusable ontologies* that contained these common sets of abstractions. For example, many knowledge-based applications in the health-care domain share concepts such as *drugs* and *laboratory tests*. If we could do a good job building an ontology of these concepts, then we could reuse this work across several different knowledge bases for different medical applications.

To enable this type of ontology reuse, Protégé/Win implemented the idea of *ontology inclusion*. When building a particular knowledge base, users could choose to include all of the concepts of some pre-defined shared ontology. The concepts in the included ontology could not be altered, but they could be referenced by other classes and instances in the knowledge base, as well as subclassed to introduce specializations of the shared concepts. Thus, ontology inclusion furthered the idea of knowledge-base component reuse by enabling ontology reuse, complementing the problem-solving method reuse introduced by Protégé-II. Ontology inclusion allowed users to build large knowledge bases by "gluing" together a set of smaller, modular ontologies. As with software, modular ontologies scale to large problems better than monolithic ones.

The idea that a common set of abstractions (a shared ontology) could be used by multiple, related knowledge bases was not unique to the Protégé work: The Ontolingua library was designed for this sort of ontology reuse, and included low-level ontologies such as "units & measures" designed for inclusion into other, less generic ontologies (Farqahar et al., 1995). However, unlike users of the static Ontolingua library, Protégé/Win users could combine ontologies with problem-solving methods, and could construct knowledge-acquisition tools from composite ontologies.

5.2. Integrated tools

In aiming to make the Protégé environment easier to use, we wanted to streamline and integrate the set of Protégé-II tools for ontology building, KA-tool generation, and knowledge-base building. Most critically, we found that the generation of a KA-tool from an ontology was particularly cumbersome, in part because this process is highly iterative: During ontology construction, developers would want to see a prototype tool, which might inspire further changes to the ontology, and then they would want to see the resulting, next-generation KA-tool, and so on. In Protégé-II, the generation of a new KA-tool required a cumbersome compilation-style process which was time-consuming and potentially frustrating.

In response, we built a "synchronization" capability into Protégé/Win: without reading or writing to files, the KA-tool generator could be updated with the most recent version of an ontology. In this way, changes to the ontology could be seen immediately in a KA-tool, which made it easier to move between ontology editing and adjusting the layout of the corresponding KA-tool.

Fig. 8 shows screens from two Protégé/Win tools with an example knowledge base about wines. The ontology editor shows classes, and the layout interpreter (or KA-tool) shows instances. The layout editor (not shown) helps users custom tailor the layout and appearance of forms and widgets in the KA-tool. (As with Protégé-II, all three tools could be launched from a control-panel application.) As users edited a particular ontology, they could "synchronize" their work with the layout editor, and thereby preview the resulting KA-tool. Although our design did not completely

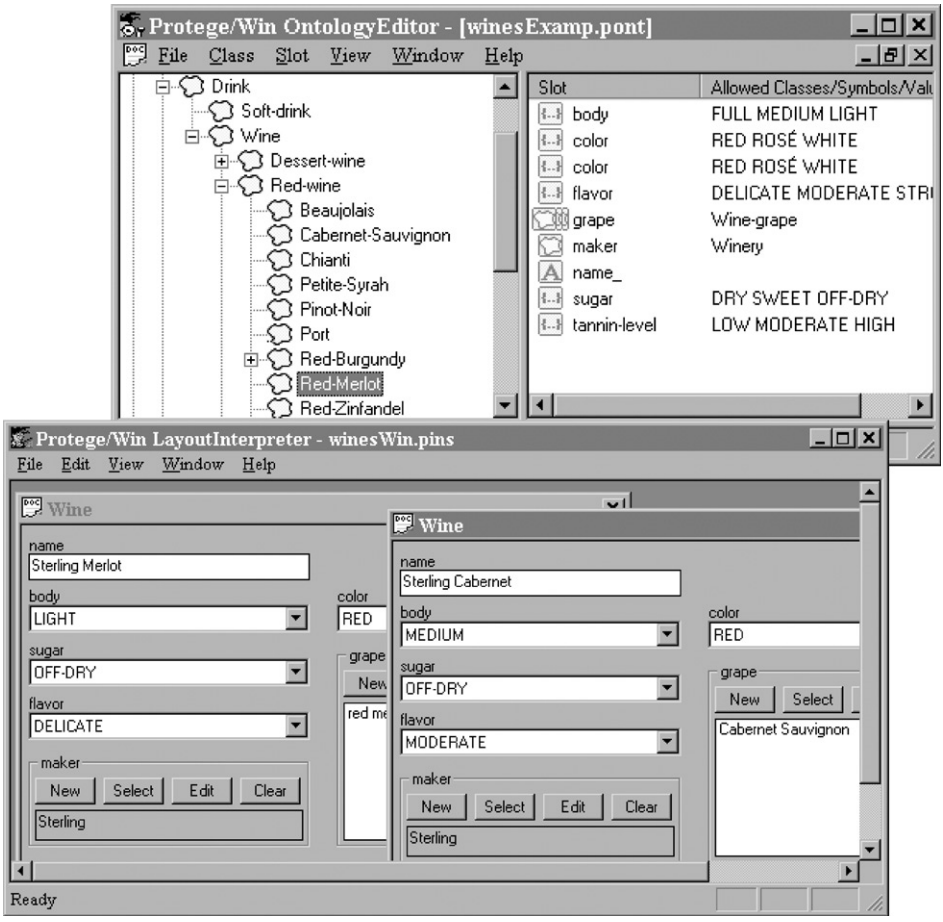


Fig. 8. The Protégé/Win interface. The Ontology Editor (above) shows classes on the left and slots on the right from an ontology about wines. The Layout Interpreter (below) shows two instances of the Wine class.

integrate the tools, the synchronization facility did make Protégé/Win easier to use than Protégé-II. Furthermore, this development led the way to a complete tool integration in Protégé-2000.

5.3. Reifying forms in the knowledge-acquisition tool

With Protégé-II, we removed the assumption that ESPR would be the underlying method for the resulting expert system, and thus, we also removed ancillary assumptions that were built in to the generation of Protégé-I KA-tools. However, Protégé-II knowledge bases still contained information that was about the formatting of the KA-tool itself, rather than any aspect of the domain knowledge being modeled. In Protégé/Win we viewed ontologies more as reusable components—models that could be used in different contexts (perhaps via the inclusion mechanism), in different applications. Therefore, we added a separate repository to store information that was specific to a particular KA-tool, and distinct from either the domain ontology or the knowledge base of instances. This repository reified the KA-tool information, treating information about format as first class objects, distinct from the domain knowledge.

For the first time, developers could generate multiple KA-tools, with different views and formats, from a single ontology. For example, as part of a project to model health-care guidelines, we built two tools from a single guideline ontology (Fridsma et al., 1996). One tool, aimed at the national or regional level, allowed domain experts to define a health-care guideline at a high level, without the implementation details that would be required to actually carry out the guideline. A second tool, aimed at local hospitals, allowed domain experts to instantiate the tool with details about the guideline, including resource and personnel allocation decisions that must be made at a local level. The second tool exposed and made available a number of classes that the national-level tool had kept hidden. This approach was possible only because Protégé/Win stored view-level information separately from the domain-level information.

In general, the KA-tool generator of Protégé/Win strengthened and codified parts of the "downhill flow" assumption. In particular, after users built an ontology of classes, they were then expected to use the layout editor to format and custom tailor the KA-tool that domain users would ultimately use to build a knowledge base of instances. The layout editor assumed that: (1) every class in an ontology had an associated KA-form and (2) every slot in a given class is associated with a knowledge-entry widget on the KA-form. These widgets were selected from a pre-enumerated list, associated with slot types, and they included widgets such as text fields and Boolean check boxes.

In developing Protégé/Win, we assumed that a KA-tool composed of these types of forms would be sufficiently expressive for knowledge-acquisition and knowledge-base visualization. For many domains, this assumption worked well; by 1998 we had a large user community who found the Protégé/Win tools to be worthwhile in their particular application areas. On the other hand, it became apparent that other

domains would require more specialized editing and visualization widgets. Implementing this idea became one of our goals for Protégé-2000.

5.4. Historical counterparts of Protégé/Win

The Ontolingua language was originally designed to support knowledge sharing and communication across multiple knowledge base systems (Gruber, 1993). It was built up from a formal set-theoretic language known as the Knowledge Interchange Format (Genesereth and Fikes, 1992). At about the same time as our development of Protégé/Win, researchers at the Knowledge Systems Laboratory built the Ontolingua ontology server and editor (Farqahar et al., 1995, 1997). Like Protégé, Ontolingua focused on the declarative specification of knowledge, rather than the procedural knowledge of inference mechanisms and problem-solving methods. For its time (early 1995), the Ontolingua editor was unique in that it was solely a web-based system. This design choice led to some significant user-interface differences when contrasted with Protégé/Win, a stand-alone, conventional application. However, at a deeper level, the knowledge representation of Ontolingua had cleaner, more consistent semantics than did Protégé/Win. Because it was designed from a logical substrate, and aimed to provide well-defined semantics across multiple-knowledge-based systems, it provided a more logical and complete implementation of the basic concepts of *class, instance,* and *slot*. As we will see, Ontolingua's more well-defined semantics directly affected our development of Protégé-2000.

Researchers at the University of Amsterdam compared the Ontolingua Editor and Protégé/Win as well as with three other ontology editing systems (Duineveld et al., 2000). By the time this comparison was carried out, Protégé/Win was a mature, well-tested tool. (In fact, we had just released the first version of Protégé-2000.) Perhaps because of its maturity, Protégé did quite well in categories such as stability, usability, and help systems. On the other hand, tools like the Ontolingua editor offered features and expressive power not available in Protégé/Win. (In some cases these deficiencies were remedied by Protégé-2000.) Disappointingly, the overall conclusion of the comparison was that none of the tools was really suitable for direct use by domain experts. Instead, all required some training and expertise with knowledge representation and modeling.

In contrast to either Ontolingua or Protégé/Win, the VITAL project focused on the development of appropriate problem-solving methods for knowledge-based systems (Motta et al., 1996). A novel aspect of this work was the idea of a "generalized directive model" to provide a grammar for composing knowledge-based system building blocks—in particular, the inference mechanisms and their links to knowledge bases (O'Hara et al., 1998). In addition, unlike much of the KADS-related work, VITAL included a formal language, OCML, that could make the resulting inference procedure operational. However, in many ways VITAL took an opposite approach from ours: Whereas VITAL focused on the inference procedures, and on a compositional grammar to drive the construction of

knowledge-based systems, Protégé/Win focused on the declarative ontology, using that to build an inference-independent KA-tool.

5.5. Summary of Protégé/Win

One the most important achievements of Protégé/Win was to expand our user community by providing an easy-to-use system that ran on a widely available operating system. Part of our strategy was to increase the number of users and the number of knowledge bases built with Protégé, with the idea that our users would provide input and guidance for the design of future improvements. Eventually, the Protégé/Win user base grew to about 200 researchers, working in applications that ranged from medical guidelines (Johnson et al., 2000) to meta-modeling about problem-solving methods (Fensel et al., 1999).

Although it included some novel extensions (such as ontology inclusion), the Protégé/Win implementation was largely a re-engineering of the ideas demonstrated in Protégé-II. However, this view should not diminish its overall contribution to the Protégé project. Engineering improvements such as tighter integration of the knowledge-base development tools (e.g. the ontology editor and the knowledge-acquisition tool) led directly to improved usability of the overall system, which in turn, led to the growth of a Protégé user community.

6. Protégé-2000: the current implementation

As the Protégé/Win user community grew, and as we received ideas (and feature requests) from this community, we realized it was time to re-engineer the Protégé environment one more time. In contrast to previous iterations, we were motivated neither by the need to drastically change our approach, nor by external forces, such as the need to change hardware or operating system platforms. Instead, we were responding to users' requests for improving the functionality and generality of the Protégé methodology. Particularly challenging were well-intentioned requests that ultimately required a domain-specific adaptation of Protégé for a particular class of applications. Although we remained attached to the goal of a domain-independent architecture, we began to realize that many users had needs that could not be adequately addressed in a domain-independent manner.

Another problem uncovered by our users was the insistence that our downhill flow assumption was too limiting. Because this was one of the fundamental underpinnings of previous versions of Protégé, allowing for other modes of interaction required some soul-searching on part. However, we eventually accepted the user need both to work with instances during class definitions, and to create and refer to classes during knowledge acquisition.

Given these sorts of modifications, we realized that a revolutionary re-engineering of the system would be more appropriate than would any evolutionary modification of Protégé/Win. With Protégé-2000, we provided at least three significant augmentations. First, Protégé-2000 included an overhaul of the underlying

knowledge model of Protégé. In order to improve the expressivity of our knowledge bases, we worked with other knowledge-base system developers to conform to a more consensus knowledge model for frame-based systems. Our aim, like that of Ontolingua, was to allow Protégé knowledge-based systems to *interoperate* with other knowledge-base formalisms. Second, to further improve usability and to better match our new knowledge model, we built Protégé-2000 as a single unified application, continuing the trend from Protégé/Win. Finally, in order to provide greater flexibility and to better distribute the development effort, we designed Protégé-2000 based on a *plug-in* architecture, as supported by the Java programming language.

6.1. The Protégé-2000 knowledge model

In previous Protégé editions, relatively little design went into the underlying knowledge representational model or structure: In Protégé-II and Protégé/Win, we used a simple frame-based model provided by CLIPS; in Protégé-I, we hand-coded Lisp objects to capture the necessary semantics. In contrast, for Protégé-2000, we made an effort to evaluate knowledge representation formalisms from a number of other systems, especially those that were frame based. In particular, we were strongly influenced by the line of work begun by Karp and others as the *Generic Frame Protocol* (Karp et al., 1995), which evolved into the open knowledge-base connectivity (OKBC) protocol (Chaudhri et al., 1998). These protocols attempt to specify a set of common semantics that could be used to enable better interoperation among different knowledge-based systems. This work was closely tied to the Ontolingua system; indeed, in some ways Ontolingua provided the canonical implementation of the OKBC protocol.

For Protégé-2000, we abandoned the semantics of the CLIPS object system, and based our new knowledge model on OKBC. (For a detailed description of the Protégé-2000 knowledge model and its relationship to OKBC, see Noy et al., 2000). In comparison to the knowledge models used by earlier versions of Protégé, the OKBC model is significantly more flexible. For example, in previous versions of Protégé, we made a strong distinction between classes and instances—every object was either one or the other. Although this approach worked well for many domains, we discovered that in some cases, our knowledge model was too restrictive. Domain specialists sometimes needed to create new classes as well as new instances. Consistent with OKBC, our solution to this dilemma is to blur the distinction between classes and instances. In particular, Protégé-2000 allows for *meta-classes*, classes whose instances are themselves classes—perhaps, special classes that domain experts might build and edit.

Protégé-2000 still supports the idea that the labor of knowledge-base construction should be divided into: (1) overall ontology construction by a knowledge engineer and then (2) knowledge-base filling-in by a domain expert. However, via meta-classes, we can extend the sorts of objects that domain experts may create and edit. Unlike earlier versions, where domain experts were limited to creating instances, Protégé-2000 is more agnostic about what sort of objects (classes or instances) get

created when. This capability had an important implication for our user interface. While Protégé/Win had separate tools and mechanisms for editing instances vs. classes, in Protégé-2000, we must integrate these more tightly, allowing domain experts to use the same editing tool to add specific knowledge, whether as instances or as classes.

6.2. Tool integration via tabs

Although the Protégé/Win tools were more tightly integrated than previous implementations, they were still distinct tools: Developers used one application to edit the class structure and then a different application to modify the knowledge base of instances. Both Protégé/Win and Protégé-II did include a single unifying application: a control panel that launched the individual tools for ontology creation, forms editing, and knowledge acquisition. In practice, these control panels were mostly just used for the purpose of explanation and system demonstration. As such, they were very effective, transforming the suite of tools into a much more easily understood system. In fact, the existence of these control panels led directly to the design and implementation of Protégé-2000 as a single application that carries out all knowledge-base building tasks.

Fig. 9 shows the standard Protégé-2000 user interface with the same knowledge base shown in Fig. 8. The interface is divided into a set of tabs—Fig. 9 has three tabs visible, with the "classes & instances" tab on top. This tab includes the functionality of both the ontology editor (modifying classes) and the "layout interpreter" (modifying instances) of Protégé/Win. The left pane shows the class hierarchy, the middle shows instances, and the right shows details of either a selected class or a selected instance. The "forms" tab carries out the function of the Protégé/Win "layout editor", and the "slots" tab is new, added because the Protégé-2000 knowledge model treats slots as first-class objects, as per the OKBC protocol.

An important design consideration for Protégé-2000 was to retain the user-interface simplicity of the earlier knowledge model. Thus, the meta-class capability is largely hidden from naïve users, and the class/instance distinction is retained by the user interface. This design allows Protégé to be flexible and powerful for developers and knowledge engineers, yet also support simple tools that are easy for the domain specialist to understand and use. As part of this approach, any of the tabs in Protégé-2000 can be configured as "hidden", so that end-users see only the functionality that they need. In fact, to further shield the end user from the more abstract aspects of the knowledge base, we can build a completely custom-tailored user interface by constructing a domain-specific *plug-in*.

6.3. The Protégé-2000 plug-in architecture

We had two motivations for re-building Protégé with a plug-in architecture. First, as with any large software development effort, we gain substantial maintenance and scalability benefits from a more modular architecture. With the use of plug-ins, we can distribute the development workload across multiple programmers, including

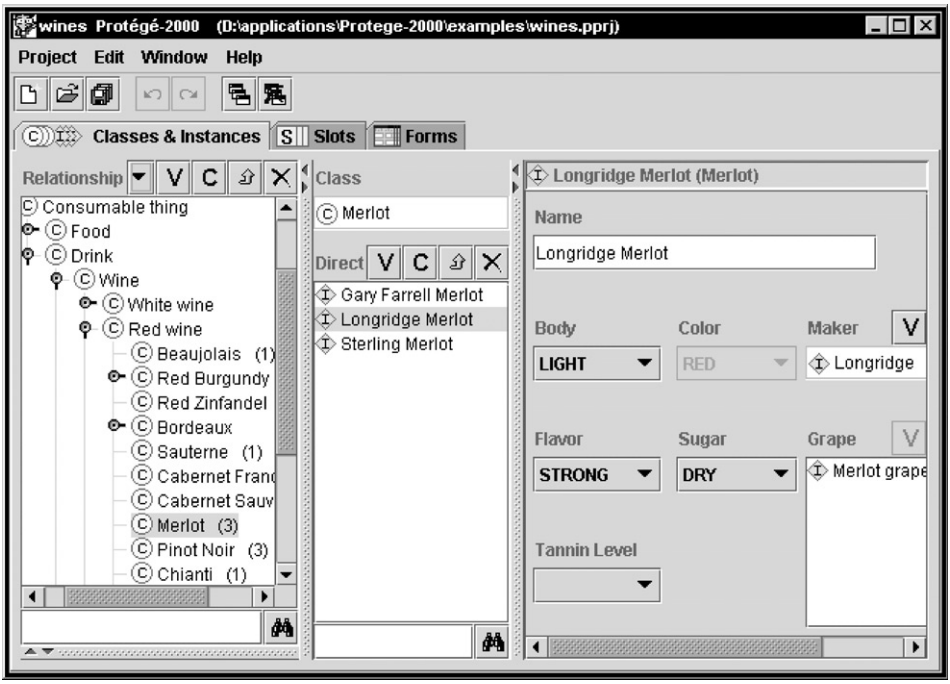


Fig. 9. The default user interface for Protégé-2000, showing a wine ontology and one instance of the class "Merlot".

external programmers from our user base. Second, as alluded to in Section 5, we wanted to provide a greater range of custom-tailored features to the Protégé user interface. As both the number of Protégé users and the variety of uses of Protégé grew, it became clear that the default user-interface widgets provided by Protégé/Win were insufficient. To provide knowledge-base tools that were domain specific and easy to use, we needed to allow programmers to build their own plug-ins to customize the interface and behavior of Protégé-2000.

Fig. 10 shows a schematic view of the Protégé-2000 plug-in architecture. At the core of Protégé-2000 is its knowledge model. Any programmatic interaction with the objects that reside in a knowledge base (instances, classes, etc) must be via the Protégé application programmers interface (API). The development and publication of this API is what allows independent developers to build plug-in components that add to or modify the functionality of Protégé. There are a variety of different ways to use this API.

6.3.1. External applications and Protégé-2000

At the coarsest level, developers may decide that the entire default Protégé user interface is inappropriate for a particular knowledge-acquisition situation or knowledge-based application. In this case, developers might use the default Protégé é KA-tool for designing and initializing the ontology, and perhaps for intermittent

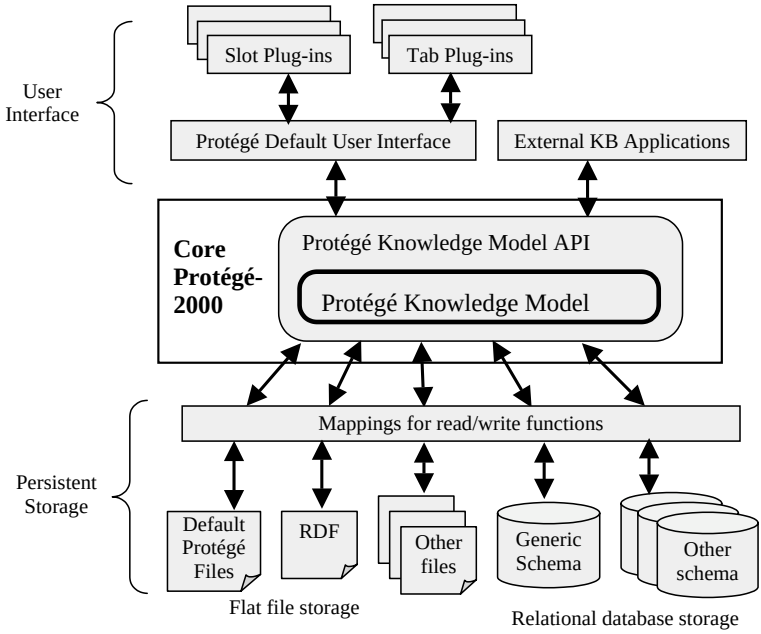


Fig. 10. The Protégé-2000 architecture. Developers may customize the default user interface via slot and tab plug-ins, or build completely custom user interfaces and interact only with the knowledge model API. Additionally, developers may build different persistent storage mechanisms, although mapping the Protégé knowledge model to the stored format is then the responsibility of the developer.

maintenance of the knowledge base, but then use an entirely separate, external application to control users' interaction with this knowledge base. This separate application would include calls to the API for retrieving or modifying Protégé knowledge-base elements.

One example of this type of external use of Protégé knowledge bases is our implementation of the ATHENA decision-support system for management of patients who have high blood pressure, currently at use at several VA hospitals (Goldstein et al., 2000). As is typical for medical domains, ATHENA has a set of particular user-interface requirements that make the default Protégé-2000 user interface insufficient for use in this complex medical work-flow setting. Thus, ATHENA is a custom-tailored, external application that provides appropriate interfaces at the point of care for decision support. Nonetheless, we view ATHENA as a Protégé knowledge-base system: It uses the Protégé-2000 API to query and retrieve information from its knowledge base of health-care guidelines.

6.3.2. Tab plug-ins

Providing an external, custom-tailored application that uses a Protégé knowledge base is the most extreme form of modifying the default Protégé user interface. As an alternative, developers may build tab-level plug-ins that provide a custom-tailored

interface or view into a Protégé knowledge base. With this approach, the developer is responsible for building Java code that fulfills the requirements of an abstract tab widget: something that provides the interface and layout information for a new Protégé tab. The resulting user interface uses the same basic menu items as the default Protégé user interface (see Fig. 9), but in addition to the default tabs, a new, custom-tailored tab will appear. Furthermore, since any tab can be hidden, the developer could design the interface so that end-users see only the custom-tailored tab. As with an external application, the tab plug-in may use any of the API calls to access information from the knowledge base.

There are several reasons for building a tab plug-in. First, a particular domain may have specific knowledge-acquisition needs that can best be satisfied by building a special-purpose knowledge-acquisition tool. Second, because tab plug-ins are (partially) independent software, it is possible to implement arbitrary functionality and inferencing capability into these plug-ins. Thus, one can build tab plug-ins that implement any generic problem-solving method, including the capability to connect that problem-solving method to a specific knowledge base. Finally, it is possible to build *domain-independent* tab plug-ins that provide new functionality for accessing, querying, or visualizing the entire knowledge base.

The OntoViz tab provides an example of a visualization tab plug-in, as well as an example of distributed software development. Built by Michael Sintek, a developer from Germany, this tab plug-in can be used to visualize any knowledge base as a diagram. Fig. 11 shows the tab with part of our example knowledge base of wines. This approach is in contrast to the default Protégé user interface, which displays classes and instances only as a collapsible hierarchy (see the left-hand column of Fig. 9). The Protégé knowledge model allows for multiple inheritance, and when this occurs, the OntoViz tab seems superior to a hierarchical view: When a class has multiple parents, this situation is awkward to visualize with a collapsible tree hierarchy. Of course, if the knowledge base has a very large number of objects, then the diagram can quickly become unmanageable. However, for small ontologies or for visualizing small parts of a larger ontology, the alternative diagramming visualization provided by OntoViz may be superior to Protégé's default user interface.

6.3.3. Slot-widget plugs-ins

At a much finer granularity, the Protégé-2000 plug-in architecture also allows developers to build plug-ins for individual elements of the user interface. Because elements of a form correspond to particular slots in a class, changing the user interface for these elements means making a new slot widget. Developers build slot widgets when they are satisfied with most of the default Protégé knowledge-acquisition tool, but need to modify the behavior for one particular type of data. For example, part of the data may be an image, perhaps stored as a GIF file, and the default Protégé KA-tool does not recognize this type of data.

To handle this sort of data, we have built a separate slot-widget plug-in that allows users to select a GIF or JPG file as the value of some slot (which is stored as a string in the knowledge base) and to display the corresponding image within the Protégé

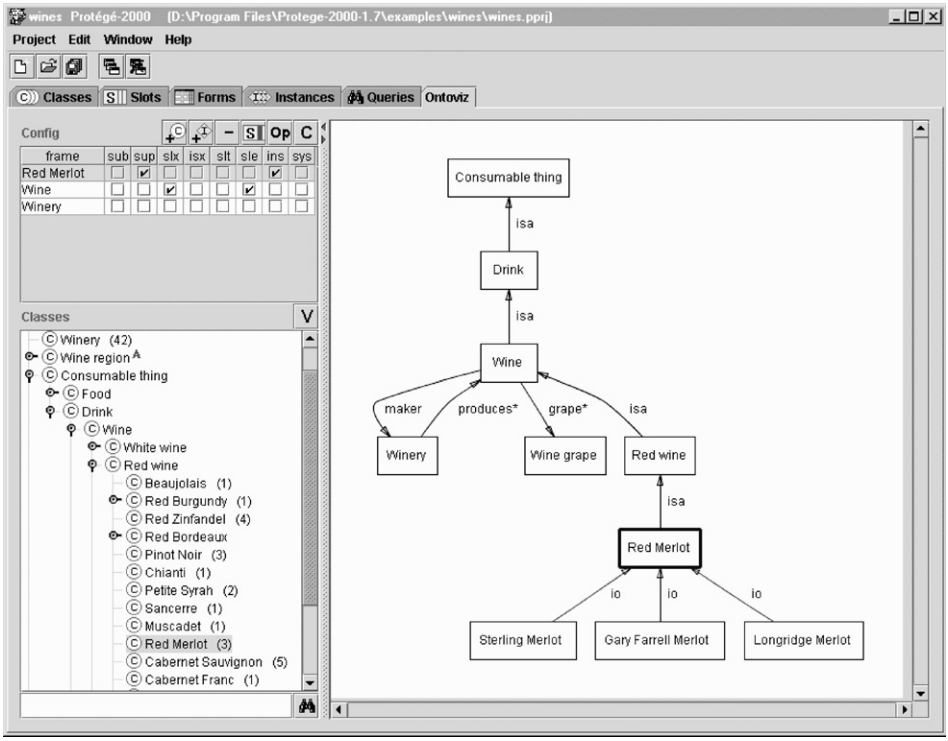


Fig. 11. The OntoViz tab plug-in used to give an alternative visualization for the Protégé wines knowledge base shown in Fig. 9.

KA-tool. This plug-in is a piece of external Java code that fulfills the requirements of an abstract slot widget: something that can display (and optionally allow for editing) objects of some particular type. This image-display slot widget is an example of a domain-independent plug-in: This slot widget might be appropriate for a variety of different domains that include images. However, because developers have access to the complete Protégé API, they can also design domain-specific slot-widget plug-ins.

Because any developer can build domain-independent slot and tab plug-ins, we have encouraged the user community to share these efforts, and we have built a library of these plug-ins, accessible via the Protégé home page. The existence and use of this library validates our approach: it is a modest example of the benefits of distributed software development and software reuse.

6.3.4. Backend plug-ins: file formats and databases

In addition to plug-ins for user-interface modifications, the Protégé-2000 architecture allows for "back-end" plug-ins that read and write to different storage formats (see Fig. 10). As a default, Protégé-2000 stores its knowledge bases in a special-purpose flat file format. However, it also has two other built-in storage back-end plug-ins. These allow for reading from and writing to resource description

framework (RDF) files and for using a Protégé-specific schema to read from and write to a relational database format.

The latter capability is essential for large knowledge bases: With any knowledge base that exceeds the physical size of primary memory, our tools must have an efficient way to retrieve and process information from secondary storage. An example is the Digital Anatomist Foundational Model, a large, rich ontology of human anatomy. The designers of this ontology must use the Protégé database backend to access the over 140 000 frames in this knowledge base (Noy et al., 2002).

Building a new "back-end" plug-in is more challenging than creating others sorts of plug-ins. To build a complete back-end, one must have a detailed understanding of the Protégé knowledge model so as to support reading and writing of all possible Protégé constructs. As an example, building the RDF backend exposed some subtle differences between our knowledge model and the semantics of RDF (Noy et al., 2001). We are currently in the process of developing a back-end plug-in that would use XML Schema. We are building a general-purpose XML schema that captures the Protégé knowledge model, so that individual knowledge bases can be written as XML files that conform to this schema.

6.4. Counterparts to Protégé-2000

By the time we released Protégé-2000, tools for ontology editing and knowledge base construction were becoming more common. For example, in addition to Protégé-2000 and the Ontolingua Editor, there were two other major ontology development tools built with the OKBC standard: the Ontosaurus system built on LOOM (MacGregor, 1991), and the GKB Editor (Karp et al., 1999).

Like the Ontolingua editor, the Ontosaurus system was built as a pure Web-based editing environment. As such, it suffered from some of the same drawbacks in user interface, and ease-of-use, as described in the University of Amsterdam comparison (Duineveld et al., 2000). However, the unique aspect of Ontosaurus was its underlying knowledge model: The LOOM system included a description language that unified frame-based knowledge with an inference procedure for classification of those frames. Effectively, this provided a consistent way of embedding rules and inference into frames. However, because LOOM is not a pure frame-based system for knowledge representation, this made it difficult and awkward to map Ontosaurus to the OKBC standard.

In contrast, the generic knowledge-base (GKB) Editor was developed hand-in-hand with the Generic Frame Protocol, and therefore it was easily mapped to OKBC. The GKB editor differed significantly from Protégé, Ontolingua, and Ontosaurus in that it provided a graphical nodes-and-arcs view of its knowledge bases (somewhat like OntoViz). In this approach, all knowledge construction occurs in the context of manipulating a diagram of nodes (objects or frames) and arcs (relations or slots). The WebOnto tool was another example of a knowledge-base building tool that used this type of visualization (Domingue, 1998).

OntoEdit borrows its user-interface design partially from Protégé, but differs significantly in its treatment of *axioms* about entities in a knowledge base (Staab and

Maedche, 2000). Protégé-2000 offers only very weak support for modeling such axioms, primarily because there are significant design and implementation difficulties in enforcing or validating a set of axioms against a particular knowledge base. Like Protégé-2000, OntoEdit is designed to read from and write to the RDF standard.

6.5. Summary of Protégé-2000

If one had to summarize the changes from Protégé/Win to Protégé-2000, the most important idea is that of *scalability*. In particular, Protégé-2000 is more extensible, more flexible, and better suited to the development of large knowledge bases than was Protégé/Win.

- Because of its extensible user interface, it can be used in situations that require specific interaction styles or user interfaces.
- Because it is a single tool with a more flexible knowledge model, it can be used in situations where domain users need to create instances as well as classes.
- Because of its capability to read and write to database systems, it can be used where knowledge bases exceed the size of physical machine memory.
- Because of its more standard knowledge model, it can be used in situations that require reading from a legacy knowledge base, or writing to other modeling formalisms (e.g. RDF).

To a certain degree, requests and feedback from the Protégé/Win user community led directly to each of these advances. To continue these cycles of iterative improvement, we have wanted to further expand our user community and to elicit additional feedback for Protégé-2000. For example, as an entrée into the W^3 RDF community, we decided to make Protégé-2000 "open source" under the Mozilla public license, making our system easily available to all. This choice had the benefit of making our system attractive to developers who might otherwise build similar systems from scratch.

The new plug-in architecture, together with the availability of the source code, has led to the growth of external Protégé *developers*: Java programmers who have developed plug-ins that extend the capabilities of Protégé. The existence of these developers and their contributions is evidence of the success of our architecture; they show that we have succeeded in building an extensible system, and that we have distributed the development effort beyond our own laboratory.

In fact, the biggest obstacles to Protégé's further growth and success are organizational rather than technical. As with any open-source development effort, it is a significant task to manage and work with external developers. For example, this group of users needs detailed, technical documentation for the Protégé API. Also, in order to keep the Protégé user and developer communities vibrant, we have maintained a lively "protege-discussion" mailing list, and spent considerable time and effort answering questions and providing technical support. We have also encouraged and helped organize a series of International Protégé Workshops both to solicit feedback from our users and to present the breadth of applications within our

community. We believe that our investment in documentation and user support is worthwhile, for we derive significant benefits from the feedback and real-world system testing that our user community provides.

7. Summary and discussion

The four generations of Protégé presented here represent over 16 years of research and system development. Protégé has evolved from a proof of concept and initial prototype to a comprehensive system with an active user community. As we described this evolution, we have highlighted the differences and augmentations from one version to the next. However, there have been some fundamental ideas of knowledge-based systems development that remained unchanged throughout this evolution:

- Knowledge-based systems should be designed for use by domain experts, rather than exclusively by knowledge engineers.
- Domain-specific knowledge-acquisition tools must be *generated* from an underlying domain model or ontology.
- During construction of these KA-tools, there is a division of labor between structural domain modeling and tool design (mostly carried out by the knowledge engineer), vs. filling in detailed domain knowledge (mostly carried out by the domain expert).
- Domain knowledge can be captured declaratively, without direct reference to an inference or problem-solving method. Conversely, inference methods can be isolated as problem-solving methods or (in Protégé-2000) as plug-in applications.

Finally, we have developed Protégé with a consistent meta-methodology that emphasizes pragmatics, real-world problems, and feedback from users. The longevity of these ideas across all of the Protégé implementations has resulted in a stream of derivative research efforts that leverage the flexibility of the Protégé system. These research efforts cover a wide variety of work in knowledge-based systems:

- *Knowledge-based software reuse*: Beginning in the mid-1990s, our group has carried out research that explores the ability to use the Protégé environment to support software reuse: allowing researchers to leverage older software development efforts for new (but related) tasks (Eriksson et al., 1995). We tested this capability with the task of constraint satisfaction in domains of computational biology, medical decision-making, and engineering (Gennari et al., 1995, 1998).
- *Protégé and the Semantic Web*: Recently, there has been a great deal of effort to annotate web information with richer semantics, thereby building a *Semantic Web* (Berners-Lee et al., 2001; Hendler, 2001). We describe how Protégé-2000 can be used to author RDF ontologies and content, and how Protégé could support other Semantic Web languages, via construction of other back-end plug-ins (Noy et al., 2001).

- *Ontology merging and alignment*: If multiple groups independently develop ontologies that cover the same domain content, then one often wishes to merge or at least partially align these ontologies. This problem is especially germane to web content, where ontologies proliferate rapidly, and many wish to leverage pre-existing content organizations. We describe a tool known as PROMPT for automatic ontology merging and alignment (Noy and Musen, 2000). This tool is implemented as a Protégé-2000 tab plug-in, and has been evaluated in several different domains.
- *Medical decision support systems*: The longest standing application of Protégé knowledge bases have been for medical decision support systems, beginning with protocol-based cancer therapy in Oncocin. This work was continued with the EON Project, which applied the ideas of component-based software and temporal databases to protocol therapy (Musen et al., 1996). Temporal information can be critical in medical applications, and thus, a number of Protégé KA tools have been designed for the management of this type of knowledge (Nguyen et al., 1999; Shahar et al., 1999). Most recently, by leveraging the Protégé-2000 tab plug-in capability, we have built systems that focused on supporting decisions with regard to patient eligibility into clinical trial protocols (Rubin et al., 1999; Gennari et al., 2001).
- *Knowledge representation for health-care guidelines*: The Protégé tools (beginning with Protégé/Win) have been applied throughout the development of the GLIF standard for representing healthcare guidelines (Ohno-Machado et al., 1998; Peleg et al., 2001). Protégé has been useful for this effort both because the models could be easily changed as GLIF evolved, and because Protégé supports a diagrammatic visualization of guideline elements.

Each of these efforts resulted in scientific contributions in their own domain. Thus, publications describing these endeavors minimize discussion of the underlying Protégé platform per se, and instead focus on their more specific research goals. In contrast, our development of Protégé itself was not driven by the need to build something completely novel, but rather by users' needs for a flexible, reliable, efficient, and easy-to-use environment. In fact, because Protégé is so extensible and general purpose, it is somewhat difficult to pinpoint the research contribution of the Protégé system itself. In truth, Protégé itself is no longer revolutionary research—however, it does provide a stable, robust, flexible infrastructure for carrying out more specific research in knowledge-based systems. Without the existence of a well-supported, stable infrastructure such as Protégé, all of the above research efforts would have been nearly impossible to carry out.

Protégé's longevity has been due to a number of factors. First, we recognized early that designing a system for the long term would have clear benefits. We now have a long history of experiences that help guide us when faced with different design choices. (In support of our aim to design for the long term, we have been fortunate to retain many of our scientific staff for relatively long periods of time.) Second, by happenstance, we were forced to re-implement our system more than once, thereby discovering that the resulting re-engineering led to a greatly improved system. The

opportunity to carry out complete revisions of the system has led to an architecture that is much more robust and flexible than any we could have designed in a single pass. Finally, we have strongly encouraged the growth of a user community. To do this, we have had to balance providing new functionality rapidly vs. providing stable software for our user community. We have also had to invest time and energy into documentation and user support.

The value of our user community cannot be overstated. Of course, the value provided by user feedback is well-recognized by the software engineering community and by commercial software enterprises. However, Protégé is unusual in that it is a hybrid—it is not a fully supported commercial software product, yet it is clearly much more robust and supported than typical research software produced by academic projects. Although the work required to transform research software into commercial software is mundane (documentation, thorough testing, user support), we argue that there are real research benefits to carrying out at least some of this effort. For any research effort that claims to be of general use, it is essential that some of these mundane services be provided, for otherwise, the generality claim cannot be proven. Unlike some academic research efforts, we have successfully demonstrated the generality of the Protégé architecture by means of our user community—a variety of different users have worked with Protégé to achieve a variety of different research goals. Our user community is large. Since the system is open source, it is hard to estimate the number of users, but as of this writing, we have over 900 members of the protege-discussion mailing list.

Unlike previous implementations, Protégé-2000 is extensible at the programming level: External developers can build plug-ins that change the behavior and appearance of the system. As a result, we now have a modest library of these plug-ins, and a community of Protégé developers outside of the Stanford group. We strongly encourage this community, since it represents a way to distribute the development effort in a controlled manner. Indeed, our hope is that the Protégé system will be able to follow the open-source model of distributed development, which has proven so successful with projects such as the Linux operating system or the Apache Web server.

The Protégé effort illustrates the importance of combining a usable implementation for practical applications with a research platform for scientific experimentation. Protégé has undergone several design iterations, which helped us evaluate Protégé progressively, understand and define the problems of using Protégé, and test new design solutions. The current Protégé-2000 architecture allows for distributed design, whereby developers can experiment with new functionality via plug-in construction, while taking advantage of the well-established knowledge representation infrastructure of the Protégé knowledge model. We believe that the Protégé system is an extensible and robust platform for a wide variety of users: from knowledge-based system developers who build practical systems, to domain experts who provide expert knowledge, to researchers who explore new scientific ideas. Although 16 years is already venerable in comparison to most academic projects, we believe that Protégé will remain a valuable tool for knowledge-base systems, and will continue to grow and evolve for many more years.

Acknowledgements

Parts of this work were funded by the High Performance Knowledge Base Project of the Defense Advanced Research Projects Agency (Contract N660001-97-C-8549) and the Space and Naval Warfare Systems Center (contract N6001-94-D-6052). As should be clear, we are greatly indebted to our users, for this project would not have flourished without their active and continued support. In addition to the authors, we acknowledge and thank the many people who have contributed to the development of Protégé. Protégé-2000 can be downloaded from http://protege.stanford.edu/.

References

Bachant, J., McDermott, J.D., 1984. R1 revisited: four years in the trenches. AI Magazine 5 (3), 21–32.

Berners-Lee, T., Hendler, J., Lassila, O., 2001. The semantic web. Scientific American 284 (5), 34–43.

Breuker, J.A., Velde, W.V.d., 1994. The CommonKADS Library for Expertise Modelling. IOS Press, Amsterdam, The Netherlands.

Buchanan, B.G., Shortliffe, E.H. (Eds.), 1984. Rule-Based Expert Systems: The Mycin Experiments of the Stanford Heuristic Programming Project, Addison-Wesley, Reading, MA.

Buchanan, B., Barstow, D., Bechtal, R., et al., 1983. Constructing an expert system. In: Hayes-Roth, F., Waterman, D., Lenat, D. (Eds.), Building Expert Systems. Addison-Wesley, Reading, MA.

Chandrasekaran, B., 1983. Towards a taxonomy of problem-solving types. AI Magazine 4 (1), 9–17.

Chandrasekaran, B., 1986. Generic tasks in knowledge-based reasoning: high-level building blocks for expert system design. IEEE Expert 1 (3), 23–30.

Chaudhri, V.K., Farquhar, A., Fikes, R., Karp, P.D., Rice, J.P., 1998. OKBC: a programmatic foundation for knowledge base interoperability. Proceedings of AAAI-'98, Madison, WI, pp. 600-607.

Crubezy, M., Lu, W., Motta, E., Musen, M.A., 2001. The internet reasoning service: delivering configurable problem-solving components to web users. Technical Report SMI-2001-0895, Stanford Medical Informatics.

Domingue, J., 1998. Tadzebao and WebOnto: Discussing, browsing, and editing ontologies on the web. Proceedings of the 11th Workshop on Knowledge Acquisition, Modeling and Management, Banff, CA.

Duineveld, A.J., Stoter, R., Weiden, M.R., Kenepa, B., Benjamins, V.R., 2000. Wondertools? A comparative study of ontological engineering tools. International Journal of Human–Computer Studies 52 (6), 1111–1133.

Eriksson, H., Puerta, A.R., Musen, M.A., 1994. Generation of knowledge-acquisition tools from domain ontologies. International Journal of Human–Computer Studies 41, 425–453.

Eriksson, H., Shahar, Y., Tu, S.W., Puerta, A.R., Musen, M.A., 1995. Task modeling with reusable problem-solving methods. Artificial Intelligence 79, 293–326.

Eshelman, L., Ehret, D., McDermott, J.D., Tan, M., 1987. MOLE: a tenacious knowledge-acquisition tool. International Journal of Man-Machine Studies 26, 41–54.

Farqahar, A., Fikes, R., Rice, J., 1997. The ontolingua server: a tool for collaborative ontology construction. International Journal of Human–Computer Studies 46 (6), 707–728.

Farqahar, A., Fikes, R., Pratt, W., Rice, J., 1995. Collaborative ontology construction for information integration. Technical Report KSL-95-63, Stanford Knowledge Systems Lab.

Fensel, D., Benjamins, V.R., Motta, E., Wielinga, B., 1999. UPML: A framework for knowledge system reuse. International Joint Conference on AI, Stockholm, Sweden.

Fridsma, D., Gennari, J.H., Musen, M.A., 1996. Making generic guidelines site-specific. Proceedings of the 20th AMIA Annual Fall Symposium, Washington, DC, pp. 597–601.

Genesereth, M.R., Fikes, R., 1992. Knowledge interchange format, Version 3.0 Reference Manual. Technical Report Logic-92-1, Stanford University Computer Science Department.

Gennari, J., 1993. A brief guide to MAITRE and MODEL: An ontology editor and a frame-based knowledge representation language. Stanford Medical Informatics technical report no. SMI-93-0486.

Gennari, J.H., Tu, S.W., Rothenfluh, T.E., Musen, M.A., 1994. Mapping domains to methods in support of reuse. International Journal of Human–Computer Studies 41, 399–424.

Gennari, J.H., Altman, R.B., Musen, M.A., 1995. Reuse with Protégé-II: from elevators to ribosomes. Proceedings of the ACM-SigSoft 1995 Symposium on Software Reusability, Seattle, WA, pp. 72–80.

Gennari, J.H., Cheng, H., Altman, R.B., Musen, M.A., 1998. Reuse, CORBA, and knowledge-based systems. International Journal of Human–Computer Studies 49, 523–546.

Gennari, J.H., Sklar, D., Silva, J.S., 2001. Protocol authoring to eligibility determination: cross-tool communication. Proceedings of the AMIA Annual Symposium, Washington, DC, pp. 199–203.

Gil, Y., Melz, E., 1996. Explicit representations of problem-solving strategies to support knowledge acquisition. Proceedings of the Thirteen National Conference on Artificial Intelligence (AAAI-96), Portland, OR, pp. 469–476.

Goldstein, M.K., Hoffman, B.B., Coleman, R.W., et al., 2000. Implementing clinical practice guidelines while taking account of evidence: ATHENA, an easily modifiable decision-support system for management of hypertension in primary care. Proceedings of the AMIA Annual Symposium, Los Angeles, CA, pp. 300–304.

Gruber, T.R., 1993. A translation approach to portable ontology specifications. Knowledge Acquisition 5 (2), 199–220.

Guarino, N., Giaretti, P., 1995. Ontologies and knowledge bases: towards a terminological clarification. In: Mars, N.J.I. (Ed.), Towards Very Large Knowledge Bases. IOS Press, Amsterdam, The Netherlands, pp. 25–32.

Hayes, P.J., 1979. The logic of frames. In: Brachman, R.J., Levesque, H.J.(Eds.), Readings in Knowledge Representation. Morgan Kaufmann, Los Altos, CA, pp. 287–295.

Hayes-Roth, F., Waterman, D., Lenat, D. (Eds.), 1983. Building Expert Systems. Addison-Wesley, Reading, MA.

Hendler, J., 2001. Agents and the semantic web. IEEE Intelligent Systems 16 (2), 30–37.

Johnson, P.D., Tu, S.W., Booth, N., Sugden, B., Purves, I.N., 2000. Using scenarios in chronic disease management guidelines for primary care. Proceedings of the AMIA Annual Symposium, Los Angeles, CA, pp. 389-393.

Karp, P., Myers, K., Gruber, T., 1995. The generic frame protocol. Proceedings of the 1995 International Joint Conference on Artificial Intelligence, pp. 768–774.

Karp, P.D., Chaudhri, V.K., Pauley, S.M., 1999. A collaborative environment for authoring large knowledge bases. Journal of Intelligent Information Systems 13, 155–194.

McDermott, J.D., 1988. Preliminary steps toward a taxonomy of problem-solving methods. In: Marcus, S. (Ed.), Automating Knowledge Acquisition for Expert Systems. Kluwer Academic Publishers, Dordrecht, pp. 225–256.

Marcus, S., McDermott, J.D., 1989. SALT: a knowledge acquisition language for propose-and-revise systems. Artificial Intelligence 39 (1), 1–37.

MacGregor, R.M., 1991. Using a description classifier to enhance deductive inference. Proceedings of the Seventh IEEE Conference on AI Applications, pp. 141–147.

Motta, E., O'Hara, K., Shadbolt, N., Stutt, A., Zdrahal, Z., 1996. Solving VT in VITAL: a study in model construction and knowledge reuse. International Journal of Human–Computer Studies 44, 333–371.

Musen, M.A., 1987. Use of a domain model to drive an interactive knowledge-editing tool. International Journal of Man-Machine Studies 26, 105–121.

Musen, M.A., 1989a. Automated Generation of Model-Based Knowledge-Acquisition Tools. Pitman Publishing, London.

Musen, M.A., 1989b. Automated support for building and extending expert models. Machine Learning 4, 347–376.

Musen, M.A., Tu, S.W., 1993. Problem-solving models for generation of task specific knowledge acquisition tools. In: Cuena, J. (Ed.), Knowledge-Oriented Software Design. Elsevier, Amsterdam.

Musen, M.A., Tu, S.W., Das, A., Shahar, Y., 1996. EON: a component-based approach to automation of protocol-directed therapy. Journal of the American Medical Informatics Association 3, 367–388.

Nguyen, J.H., Shahar, Y., Tu, S.W., Das, A., Musen, M.A., 1999. Integration of temporal reasoning and temporal-data maintenance into a reusable database mediator to answer abstract, time-oriented queries: the Tzolkin system. Journal of Intelligent Information Systems 13, 121–145.

Noy, N.F., Musen, M.A., 2000. PROMPT: algorithm and tool for automated ontology merging and alignment. 17th National Conference on Artificial Intelligence (AAAI-2000), Austin, TX, pp. 450–455.

Noy, N.F., Fergerson, R.W., Musen, M.A., 2000. The knowledge model of Protégé-2000: combining interoperability and flexibility. Second International Conference on Knowledge Engineering and Knowledge Management (EKAW'2000), Juan-les-Pins, France.

Noy, N.F., Sintek, M., Decker, S., et al., 2001. Creating semantic web contents with Protege-2000. IEEE Intelligent Systems 16 (2), 60–71.

Noy, N.F., Musen, M.A., Mejino, J.L.V., Rosse, C., 2002. Pushing the envelope: challenges in a frame-based representation of human anatomy. Technical Report SMI-2002-0925, Stanford Medical Informatics.

O'Hara, K., Shadbolt, N., van Heijst, G., 1998. Generalised directive models: integrating model development and knowledge acquisition. International Journal of Human–Computer Studies 49 (4), 497–522.

Ohno-Machado, L., Gennari, J.H., Murphy, S., et al., 1998. The guideline interchange format: a model for representing guidelines. Journal of the American Medical Informatics Association 5, 357–372.

Peleg, M., Boxwala, A.A., Bernstam, E., et al., 2001. Sharable representation of clinical guidelines in GLIF: relationship to the arden syntax. Journal of Biomedical Informatics 34 (3), 170–181.

Puerta, A.R., Egar, J.W., Tu, S.W., Musen, M.A., 1992. A multiple-method knowledge-acqusition shell for the automatic generation of knowledge-acquisition tools. Knowledge Acquisition 4, 171–196.

Puerta, A.R., Tu, S.W., Musen, M.A., 1993. Modeling tasks with mechanisms. International Journal of Intelligent Systems 8, 129–152.

Rubin, D., Gennari, J.H., Srinivas, S., et al.,1999. Tool support for authoring eligibility criteria for cancer trials. Proceedings of the AMIA Annual Symposium, Washington, DC, pp. 369–373.

Sandahl, K., 1994. Transferring knowledge from active expert to end-user environment. Knowledge Acquisition 6, 1–21.

Schreiber, A.T., Akkermans, J.M., Anjewierden, A., et al., 1999. Knowledge Engineering and Management: the CommonKADS methodology. MIT Press, Cambridge, MA.

Schreiber, A.T., Wielinga, B., Akkermans, J.M., van de Veld, W., deHoog, R., 1994. CommonKADS: a comprehensive methodology for KBS development. IEEE Expert 9, 28–37.

Shahar, Y., Chen, H., Stites, D.P., et al., 1999. Semi-automated entry of clinical temporal-abstraction knowledge. Journal of the American Medical Informatics Association 6 (6), 494–511.

Shortliffe, E.H., Scott, A.C., Bischoff, M.B., et al., 1981. ONCOCIN: an expert system for oncology protocol management. International Joint Conference on Artificial Intelligence (IJCAI '81), Vancouver, CA, pp. 876–881.

Staab, S., Maedche, A., 2000. Axioms are objects, too—Ontology engineering beyond the modeling of concepts and relations. Proceedings of the ECAI 2000 Workshop on Ontologies and Problem-Solving Methods, Berlin, Germany.

Steels, L., 1990. Components of expertise. AI Magazine 11, 30.

Steels, L., 1992. Reusability and configuration of applications by non-programmers. Technical Report VUB 92-4, Vrije Universiteit, Brussel.

Tu, S.W., Kahn, M.G., Musen, M.A., et al., 1989. Episodic skeletal-plan refinement based on temporal data. Communications of the ACM 32 (12), 1439–1455.

Wielinga, B., Schreiber, A.T., Breuker, J., 1992. KADS: a modeling approach to knowledge engineering. Knowledge Acquisition 4 (1), 5–53.