Supporting Knowledge Acquisition by End Users: Tools and Representations

Andreas Girgensohn, Frank M. Shipman
University of Colorado at Boulder
Department of Computer Science and Institute of Cognitive Science

Abstract
End-user modifiability allows different users to tailor a system in order to pursue different tasks, to have different preferences, and to adapt it to changing needs over time. In knowledge-based systems, end-user modifiability can help eliminate the knowledge acquisition bottleneck: domain experts can extend the knowledge base in such systems without having to rely on knowledge engineers. Experiments with end users have led to the identification of several principles that must be observed in order to make systems end-user modifiable. These principles have been followed in the implementation of MODIFIER, an end-user modification component for design environments, and NETWORK, a design environment for local area networks.

Introduction
Knowledge-based systems, such as expert systems, are becoming widely used for both large and small tasks. The difficulty of the knowledge acquisition process for such systems has been documented in [26] and [27]. This process is costly in both the time and money required to develop the knowledge base. Our work focuses on minimizing the need for knowledge engineers in the knowledge acquisition process by providing tools and representations that allow end users to make modifications to the knowledge base of the system they are using. Besides making existing knowledge-based applications more affordable to create and maintain, we believe these techniques allow for a new range of applications of knowledge-based systems which were previously unachievable due to limitations in existing knowledge acquisition techniques.

End-user modifiable systems support their users in modifying the systems according to the users' needs. The intended users of end-user modifiable systems are knowledgeable in the application domain but unable or unwilling to modify a system on the programming language level. The usefulness of end-user modifiability is not restricted to knowledge-based systems. Many systems are end-user modifiable to some degree. A successful example is the Buttons system [24] that enables users to tailor their work station environment.

Two knowledge-based, end-user modifiable systems have been implemented. MODIFIER is an end-user modification component to JANUS, a design environment for kitchen design [13]. NETWORK is a design environment for computer networks. A design environment is composed of a construction kit, similar to a CAD system, and a knowledge-based critiquing component. Critics notify the user of suboptimal design situations and provide connections into a hypermedia document explaining the critique.

In the next section we describe related work in knowledge acquisition. Section 3 describes principles for allowing end users to add and edit knowledge in knowledge-based systems. Section 4 discusses our use of these principles in the implementation of two knowledge-based design environments. Evaluation of our implementations through user-testing is presented in Section 5.

Knowledge Acquisition
Knowledge acquisition tools that can be used directly by domain experts have been a goal for a long time. End-user modifiable systems address this problem in part. End-user modifiability enables the domain experts to update and extend a knowledge base during their daily work without having to rely on knowledge engineers. Knowledge engineers or different knowledge acquisition tools are still needed during the development of an initial seed for the knowledge base and later for periodic large-scale revisions of the evolving knowledge base.

Automated knowledge acquisition tools are also intended to be used by domain experts. Examples for such tools are Terasias [5], ETS [4], MOLE [7], KNACK [19], and OPAL [28]. Knowledge acquisition workbenches like KRITON [6] or AQUINAS [18] are intended to be used by knowledge engineers and differ in their goals from MODIFIER and NETWORK.

There are also important differences in the goals of automated knowledge acquisition tools and the goal of our systems to support end users in modifying a knowledge-based design environment [12]. Knowledge acquisition tools acquire knowledge for expert systems. Expert systems fall into two major categories according to their general function: analysis or synthesis. A design environment differs from a synthesis expert system in that it supports users in creating a solution instead of automatically creating a solution from a given specification. In a knowledge-based design environment the user does the construction and the knowledge-based support system analyzes the work of the user.
Figure 1: Layers of Abstraction to Reduce the Transformation Distance between Problem Domain and System Space

and critiques it. Critics [8] are advisory systems that act as demons watching out for conditions such as inconsistencies, omissions, and inefficiencies. Critics are similar to analysis expert systems in their general function. But since critics augment users in their work instead of doing the work for them, incomplete knowledge in critics does not reduce their value much whereas incomplete knowledge in an expert system can lead to serious errors.

Automated knowledge acquisition tools derive their power by presupposing the problem solving method, e.g., MOLE, or by exploiting an explicit domain model, e.g., OPAL. MODIFIER and NETWORK use both approaches. They contain knowledge about the purpose of different types of objects in the system like classes, critic rules, and production rules and can use this knowledge to help the domain expert in modifying the knowledge base. These objects are intentionally kept simple so that they are easily understandable by end users. The loss of expressional power does not seem to be critical in design environments since they are not responsible for building a complete solution. The domain model is part of the seed of the knowledge base initially created by the knowledge engineer. End users interact with objects that have a semantic meaning in the problem domain.

The HITS Knowledge Editor [32] provides an interface to CYC [22]. It is similar to MODIFIER in some of the methods it uses such as cooperative problem solving, advertisement, and critiquing. An important difference is that a design environment can make use of the knowledge of how the modified objects will be used whereas the HITS Knowledge Editor does not make any assumptions about the use of the knowledge being entered.

MODIFIER is similar to OPAL in that it uses an intermediate representation that is translated into another form for doing the actual work. In the case of MODIFIER the intermediate representation is translated into CLOS classes and methods [3], production rules, and LISP functions. NETWORK does not use an intermediate representation, keeping all interactions with the user in the application domain.

Principles for Achieving End-User Modifiability

Making a system end-user modifiable introduces additional costs during the system development. However, Henderson and Kyng [16] argue that end-user modifiability is still advantageous since the resources saved in the initial development by ignoring end-user modifiability will be spent several times over during the system’s lifetime. Several principles for making systems end-user modifiable have been identified during the development of MODIFIER and NETWORK:

Layered Architectures. Layered architectures have been used successfully in many areas, e.g., the design of operating systems [31]. They close the gap between the system space and the problem domain (see Figure 1). Modifications can be performed on different layers of abstraction. Modifications on higher layers are closer to the problem domain and therefore easier to do but less flexible. If a change extends beyond the functionality provided by one layer, users are not immediately thrown back to the system space but can descend one layer at a time. Henderson and Kyng [16] advocate that an important step in providing more user-accessible tailoring will be to develop new high level, or application-oriented, building blocks together with application-oriented ways of manipulating them.
Certain problems need to be considered in the design of a layered architecture. The lower layers should be invisible while looking at a higher layer. Another issue is the ease of descending the layers. The additional amount of knowledge necessary to understand the next lower layer should be small.

**Parameterization.** Parameterization relieves the user of the task of modifying program code and locating the program parts responsible for a certain behavior. Facets in the behavior of the system that need to be modifiable or extensible by the user have to be identified. These facets are parameterized so that end users only have to change those parameters in order to change the behavior of the system. These parameters have a meaning at higher layers of the layered architecture. The user can choose values for these parameters that the designer of the system did not foresee but it is not possible to modify unparameterized parts of the system without resorting to programming.

**Help System.** The system must provide help about the possible values and the purpose of a parameter a user wants to modify. For example, during the modification of a class in an object-oriented system, the user might want to change the superclasses of that class. The user can ask the system to list all classes that could be used as superclasses. Listing all possible values frees the user from having to remember these values. In addition, an explanation should be provided for class inheritance and what consequences would result from a change in the inheritance hierarchy. The need for the latter was found especially important during experiments with end users. Without an explanation about the purpose of a parameter, it is difficult to make an informed decision how to modify it.

**Task Agendas.** Experiments showed that users often got stuck during modification tasks and did not use the available information sources efficiently. The system needs to determine the relevant issues and to direct the user’s attention to them. The user’s comprehension can be aided by appropriate presentations on the screen. One way to make tools more comprehensible is to simulate their behavior. Baeccker, Small, and Mander [2] showed that such a simulation makes the tools in HYPERCARD easier to use. Decomposing the task into manageable elements is an important step in the solution process. According to Jeffries et al. [17], novices are incapable of performing and recursively refining such a task decomposition. Therefore, the system has to aid users in decomposing the task.

**Framer [21] provides a checklist for directing the user’s attention and supporting them in task decomposition. DETENTE [33] maintains a task agenda embedded into the application interface and advertises task recommendations. Advertisement is a method to draw the user’s attention to the materials that require more work. The HITS Knowledge Editor [32] combines the two approaches by advertising the objects that need work and maintaining a checklist for each of these objects.

**Critics.** Critics [8] are advisory systems that act as demons watching out for conditions such as inconsistencies, omissions, and inefficiencies. When such a condition exists, the corresponding critic notifies the user and optionally proposes remedial actions. There are several uses for critics during a modification process. For example, they can signal that constraints between different fields of an object are violated. Or they may make users aware that there is an object similar to the newly defined one and suggest merging the two objects.

**Automation.** Some parts of the modification process can be automated. For example, during the definition of a new class, the system can find an appropriate place in the inheritance hierarchy after the user has described the class.

**Varying Formality of Information.** When provided with representations that are easier to add to or modify there is a greater chance that users will take the time to add their knowledge. One approach to providing easier addition or modification of information is to allow more information to remain implicit. Formal knowledge representations require that information be stated explicitly. One informal representation that users have a lot of experience with is natural language text. By using text, users can leave much information implicit, but this makes using the knowledge difficult for the system.

Hypermedia, which often uses typed links and nodes, is a representation that fills the gap between the unstructured natural language text and the formal representations used in knowledge-based systems. Systems that are considered to be hypermedia can vary between being almost completely unstructured to being close to formal knowledge representation systems. KMS [1] and VNS [30] are examples of general purpose hypermedia systems and have little capability to represent formal knowledge, instead providing the user with an easy to use system in which to organize their information. CoAUTHOR [15] and SIBYL [20] are systems which are in between hypermedia systems and being formal knowledge representation systems.

Some evidence as to the effectiveness of providing a variety of less formal representations than are provided in most knowledge-based systems comes from a study at IBM that compared the use of a hypermedia system with the use of an expert system for the diagnosis of problems in a complex datalink network [29]. A usability study showed the performance of the hypermedia system to be comparable to that of the expert system. Besides being less expensive and easier to create, the hyperdocument allowed end users to modify the domain knowledge without any programming expertise.

**Familiar Interaction Styles.** Taking advantage of what users already know is a key to creating successful systems. By determining what tools are already used by the user group, it is more likely that tools can be created which will be acceptable to the users. Currently many computer users are familiar with using direct manipulation and dialog box-based interfaces. By keeping most interaction in these forms, interfaces are likely to be usable to average computer users. If the user group can be better specified then their known interaction styles may be more specifically defined.

**End-User Modifiability in JANUS and NETWORK.**

The principles described in the previous section have been developed and tested in the context of two knowledge-based design environments, JANUS and NETWORK. They have been designed according to the principles described in the previous
JANUS: a Design Environment for Kitchen Design

The implementation of these systems and the experiments with end users have been instrumental for refining the conceptual framework for end-user modifiability.

JANUS, a knowledge-based design environment for kitchen design [13] (see Figure 2), combines a direct-manipulation interface for drawing floor plans with critics that critique suboptimal design situations. The JANUS system and the end-user modification component MODIFIER are implemented in CLOS [3] on a SYMBOLICS LISP machine.

We have also been building a design environment, NETWORK, for the design of computer networks. This domain is more technical and detail oriented than kitchen design. Network design is rapidly changing due to the constant addition of new technologies and devices, which makes end-user modifiability critical in a network design environment.

MODIFIER

Objects in JANUS. Parameters are represented as slots of objects in JANUS. Modifications of these parameters cause changes in the behavior of the system. The items in the appliance palette in JANUS are represented as classes in an inheritance hierarchy. Design units in the work area are instances of these classes. Critic rules critique suboptimal design situations. They are applicable to instances of certain design unit classes. The applicability tester for critic rules uses the inheritance hierarchy of classes, e.g., a rule for appliances is also applicable for stoves and refrigerators. Design unit classes have fact descriptions like “(Cooks Self Food)” that are used for testing the applicability of critic rules as well. The different classes are embedded in a layered architecture. For example, the conditions of critic rules are represented by spatial relationships. In most cases, a new critic rule can be defined by using an existing relationship. Occasionally, a new spatial relationship is needed. The conditions of spatial relationships are described by a LISP expression. Usually, only a very small subset of LISP and a few library functions are needed for defining such expressions.

Help System. The objects in JANUS can be edited in a form-based interface. The fields of these forms are associated with help displayers. In many cases the help window displays all possible values, e.g., all fact descriptions used in JANUS as candidates for the “Descriptions” field (see Figure 3). All objects in help windows are mouse-sensitive. It is possible to choose any of these values with the mouse so that the help window acts as a menu. Additional information about each object can be requested with a mouse click.

Defining a New Class. The incomplete definition of the new design unit class is used as the specification for the retrieval of similar classes. The user can copy the descriptions from the similar classes in the “Matching Classes” window in Figure 3. Instead of deciding which additional descriptions are needed, the user only has to decide which of the copied descriptions do not apply, a much simpler task. The system also retrieves critic rules whose requirements match the descriptions of the new class. Copying a critic rule into the “Rules to Apply” window means that the rule will be made applicable for the new class.

MODIFIER employs an “advertising” strategy [33] by representing the task of defining a new class in several steps. Each step has a precondition that indicates whether it is executable and a postcondition that checks whether the step has been performed. These...
steps are ordered in a sequence so that the precondition of a step is fulfilled by the steps before it in the sequence. MODIFIER advertises the next step in the sequence by highlighting the area of the screen that is the starting point for executing the step, e.g., a field of a form that needs to be filled, and displays a description of the step (see windows “Matching Classes” and “Suggestions” in Figure 3).

After the definition of the new class has been completed, the system automatically generates suggestions for the place of the new class in the class hierarchy (see Figure 4). The goal is to keep the inheritance hierarchy free from redundancy, i.e., a description or critic rule should be associated with only one class and not be duplicated for different classes. The system employs three basic strategies for generating suggestions that are combined in order to account for subsets of the descriptions and rules of the new class:

1. The new class can be a subclass of an existing class if that class does not have descriptions or critic rules that are not part of the new class (“Alternative 3” in Figure 4).
2. The new class can be a superclass of an existing class. Descriptions and critic rules are removed from that class if they can be inherited from the new class (“Alternative 1”).
3. A class can be generated that becomes a superclass of the new class and an existing class. The generated class holds descriptions and rules that the new and the existing class have in common (“Alternative 2”).

Critics. MODIFIER supports the modification of critic rules used in JANUS but it also uses critic rules for critiquing modification tasks. These critics inform the user of inconsistencies. For example, a critic rule checks whether the number of parameters of a JANUS critic rule being modified corresponds to the number of parameters of the spatial relation used as the rule condition (see Figure 5).

NETWORK

While the work on NETWORK, shown in Figure 6, is not as mature as the work on JANUS, it does emphasize several aspects of supporting end-user knowledge acquisition that are not emphasized in JANUS. First, it provides the user with a representation that includes informal natural language text objects as well as formal representations. Second, in the development of NETWORK, the decision on how the user should interact with the knowledge representations is centered around the issue of what interactions are already familiar to the users.

NETWORK has been designed with ease of use as the primary goal. In NETWORK, all interaction with users occurs through direct manipulation of objects or through dialog windows. Direct manipulation interfaces help make the system transparent to the user so the user is communicating in the problem domain. Dialog windows aid the user in the knowledge acquisition process by leading the user through more complex tasks. Much of the interaction style is similar to drawing programs such as MacDraw.

Each object in NETWORK has a display method and attribute/value pairs. The display method is a set of graphics primitives, like draw a circle of a certain size, and can be created and edited by users in a graphical editor similar to MacDraw. The graphical editor is an example of a tool at a lower level in the layered architecture of NETWORK.

There are a variety of interaction styles for manipulating objects,
Figure 4: Suggestions for Embedding a New Class into the Inheritance Hierarchy

Figure 5: Critic Message during a Modification

depending upon which component the object is displayed in. Figure 6 shows the NETWORK with the construction and palette components displayed. Objects can move between components and can be displayed in multiple components at once. Interaction styles were chosen, in part, by their familiarity to the network designers. An example is that when talking to network designers we found several who used drawing programs to make maps of their networks.

Objects can be grouped into compound objects much in the way one groups pieces of a drawing in MacDraw. Compound objects allow for intermediate abstractions, such as a subnet or a combination of a workstation with its peripherals and drop cable. Abstractions like this are important for allowing designers to move between different levels of detail when working on a design. By building their own sets of intermediate abstractions designers can add new layers to the layered architecture.

All knowledge is kept in a single database of objects. Design units, like the 'Decstation 3100' and 'Laserwriter II' objects in Figure 7, are represented by an object with a number of specifications encoded as attributes and values. Textual notes, like the description of services positioned above the Decstation in the top-left corner of Figure 7, are represented as objects with a textual display component. Links between objects can connect formal and informal knowledge and so allow the designers to choose the degree of formality of knowledge they add to the system.

Attributes and values of objects provide a generic form of parameterization and can be created and edited in a property sheet, like the one shown in Figure 7. The user can specify between various types for attributes, which will then be enforced. Attributes of type link have other objects as values. Knowledge about relationships between objects is added by creating new links. Inheritance of attributes between objects is represented through special links between objects. Users can add and delete objects on the list to change the inheritance. Any object can inherit attributes from any other object. There is no class-instance distinction so that the users do not have to learn these knowledge engineering concepts. The users are concerned with their design, not in the intricacies of aesthetically pleasing knowledge representations. Multiple inheritance is resolved in a breadth first search through inheritance links. NETWORK also allows for loops to form in the inheritance network.

NETWORK, while not providing as much support as MODIFIER, has been built with end-user modifiability as a major goal. Of the principles discussed in the previous section, NETWORK currently includes a layered architecture, parameterization, unstructured information, and familiar interaction styles. Future work on NETWORK will include integrating more of the principles into this
new design environment.

Evaluation

Experiments with end users showed the appropriateness of the principles described in this paper. These experiments also uncovered some problems that helped refine the principles for achieving end-user modifiability. The concepts developed in JANUS and the domain of kitchen design proved to be transferable to other domains (network design and explanation of graphics programs).

Experiments with End Users. During the experiments, each subject was asked to complete six different tasks in MODIFIER such as “defining a critic rule that critiques design units which are positioned away from walls,” or “adding a microwave oven to the palette.” The experiments showed that the general approach for achieving end-user modifiability is correct. MODIFIER enabled the subjects to perform modifications they could not have done without the support of an end-user modification component. Some of the tasks could be completed without difficulties. At the same time, some issues were found to need further consideration. All of the subjects completed the tasks with some problems that could be resolved with the help of the experiment supervisor. The difficulties fall into three categories: (1) the interaction style on a SYMBOLICS LISP machine was unfamiliar to the subjects, (2) the purpose of some of the objects to be modified was not understood completely, and (3) in some situations the subjects did not know how to get started, how to continue, and how to find and use relevant information.

The first type of problems can be overcome with some training for the subjects and some improvements in the interface. Regular users of JANUS might not have had these difficulties since they would be familiar with the SYMBOLICS interaction style. The second class of problems has been addressed by improving the quality of the help provided by the system substantially. The third problem was caused by the inability of the subjects to decompose the modification task and to identify relevant information. The advertisement strategy described earlier solves this problem.

Transferability of the Results to Other Domains. An encouraging result is that the principles used in MODIFIER have been transferred to other domains. A version of JANUS without the knowledge of kitchen design was extended to an early prototype for NETWORK [9]. The adaptation was simplified because both domains use a spatial metaphor. The prototype design environment needed some work from a programmer since some domain concepts, such as the concept of connectivity, do not exist in the kitchen domain and cannot be introduced by MODIFIER. The conceptual framework and the implementation of MODIFIER was also transferred to EXPLAINER, an environment for explaining graphics programs [11].

Discussion

There are still issues left to be addressed in order to make end-user modifiability successful in the field. Once the technical difficulty of allowing knowledge acquisition by end users is solved the problem becomes one of motivating the users to take the time to add their knowledge to the system. Literature documenting the difficulties of CSCW systems [14, 25] have em-
phasized the need of systems to provide perceivable benefits to the users who are requested to do extra work. If the benefits are perceived to be primarily by others, individuals will not do the extra work.

Mackay [23] and MacLean et al. [24] show the importance of sharing modifications performed by individuals among a group of users. MODIFIER keeps all modifications in a common knowledge base so that all users share the same set of modifications. This is no sufficient answer to the need for sharing. Some users might not want to use all modifications introduced by other users or they might not understand them.

Another problem is that end-user modifiability may cause a lack of standardization leading to a lack of portability for new components of a system. Modifications that work in one environment might not work in another because of different settings. Henderson and Kyng [16] report that users have difficulties using the tailored workstation environments of their colleagues.

Conclusions

Through our experience applying the techniques described to the domains of kitchen design and computer network design, we believe that our approach will transfer to many other domains quite directly. In applying the principles to different domains we have also been able to emphasize different aspects of knowledge acquisition and to examine how these can be integrated.

This paper has described principles for making systems end-user modifiable. The principles have been refined by insights gained during the implementation of MODIFIER and NETWORK, and by the results of experiments with end users. These principles have been instrumental in developing support tools to avoid these problems. Further evaluations of the systems will improve the understanding of modification processes and the principles for achieving end-user modifiability.

Acknowledgements

We would like to thank our advisor Gerhard Fischer. The research was supported by grants No. IRI-8722792 and No. IRI-9015441 from the National Science Foundation.

References


List of Figures

Figure 1: Layers of Abstraction to Reduce the Transformation Distance between Problem Domain and System Space 2
Figure 2: JANUS: a Design Environment for Kitchen Design 4
Figure 3: Defining a Microwave Oven 5
Figure 4: Suggestions for Embedding a New Class into the Inheritance Hierarchy 6
Figure 5: Critic Message during a Modification 6
Figure 6: NETWORK: A Design Environment for Computer Network Design 7
Figure 7: An Object with Property Sheet in NETWORK 8