University of Colorado at Boulder

Department of Computer Science

ECOT 7-7 Engineering Center
Campus Box 430
Boulder, Colorado 80309-0430
(303) 492-7514, FAX: (303) 492-2844

End-User Modifiability in Design Environments

Gerhard Fischer and Andreas Girgensohn

Department of Computer Science and Institute of Cognitive Science
Campus Box 430
University of Colorado, Boulder, Colorado 80309
Arpanet: gerhard@boulder.colorado.edu; andreasg@boulder.colorado.edu

IN CHI'90 CONFERENCE PROCEEDINGS, SEATTLE, WASHINGTON, APRIL 1-5, 1990

Abstract: Convivial systems encourage users to be actively engaged in generating creative extensions to the artifacts given to them. Convivial systems have the potential to break down the counterproductive barrier between programming and using programs.

Knowledge-based design environments are prototypes for convivial systems. These environments support human problem-domain communication, letting users to work within their domains of expertise. One of the design rationales behind design environments is to ease the construction and modification of artifacts designed *within* the environment. But because design environments are intentionally not general purpose programming environments, situations will arise that require modifications to the design environment itself. The rationale and the techniques for these later modifications are discussed in this paper.

Our conceptual framework for end-user modifiability is illustrated in the context of JANUS, an environment for architectural design. Evaluating our system building efforts against our objectives shows the subtleties of integrating end-user modifiability in these kinds of systems.

Acknowledgements: The authors would like to thank Anders Morch and Raymond McCall, who were major contributors in the development of the JANUS system. Thomas Mastaglio, David Redmiles, Brent Reeves, and Curt Stevens provided valuable comments on earlier drafts of this paper.

The research was partially supported by grant No. IRI-8722792 from the National Science Foundation, grant No. MDA903-86-C0143 from the Army Research Institute, and grants from the Intelligent Systems Group at NYNEX and from Software Research Associates (SRA), Tokyo.

End-User Modifiability in Design Environments

Gerhard Fischer and Andreas Girgensohn

Department of Computer Science and Institute of Cognitive Science
Campus Box 430
University of Colorado, Boulder, Colorado 80309
e-mail: gerhard@boulder.colorado.edu; andreasg@boulder.colorado.edu

ABSTRACT

Convivial systems encourage users to be actively engaged in generating creative extensions to the artifacts given to them. Convivial systems have the potential to break down the counterproductive barrier between programming and using programs.

Knowledge-based design environments are prototypes for convivial systems. These environments support human problem-domain communication, letting users work within their domains of expertise. One of the design rationales behind design environments is to ease the construction and modification of artifacts designed *within* the environment. But because design environments are intentionally not general purpose programming environments, situations will arise that require modifications to the design environment itself. The rationale and the techniques for these later modifications are discussed in this paper.

Our conceptual framework for end-user modifiability is illustrated in the context of JANUS, an environment for architectural design. Evaluating our system building efforts against our objectives shows the subtleties of integrating end-user modifiability in these kinds of systems.

INTRODUCTION

Convivial tools and systems (as defined by [8]) allow users "to invest the world with *their* meaning, to enrich the environment with the fruits of *their* vision and to use them for the accomplishment of a purpose *they have chosen.*" Conviviality is a dimension which sets computers apart from other communication and information technologies (e.g., television, videodiscs, interactive videotex) that are passive and cannot conform to the users' own tastes and tasks. Passive technologies offer some selective power, but they cannot be extended in ways which the designer of those systems did not directly foresee.

Unfortunately, the potential for conviviality exists only in principle for most current computer systems. Many users perceive computer systems as unfriendly, un-cooperative and too time consuming. They find that they are dependent on human specialists for help, they notice that *software is not soft* (i.e., the behavior of a system can not be changed without reprogramming it substantially), and they spend more time fighting the computer than solving their problems.

Knowledge-based design environments contribute to the goal of convivial computing. They resolve the conflict between the generality, power and rich functionality of modern computer systems and the limited time and effort that domain specialists are willing to spend in solving their problems.

In this paper we first develop a conceptual framework for end-user modifiability. We illustrate this framework in the context of JANUS, a knowledge-based design environment for architectural design that allows end-user modifiability. These system building efforts are compared to the conceptual framework, providing ideas for extensions and future research in the conceptual as well as the system building area.

END-USER MODIFIABILITY: WHAT IS IT AND WHY IS IT IMPORTANT?

Most current computer systems belong into one of two categories:

- *General purpose programming languages:* Users can do anything with them, but they are hard to learn. They are too far removed from the conceptual structure of the problem, and it takes too long to get a task or a problem solved.
- *Turn-key systems:* They are easy to use, but they do not support the broad range of activities required by many problem domains.

Starting at either end, there are promising technologies for making systems more convivial. Coming from the general purpose programming languages end, object-oriented programming, user interface management systems, program-

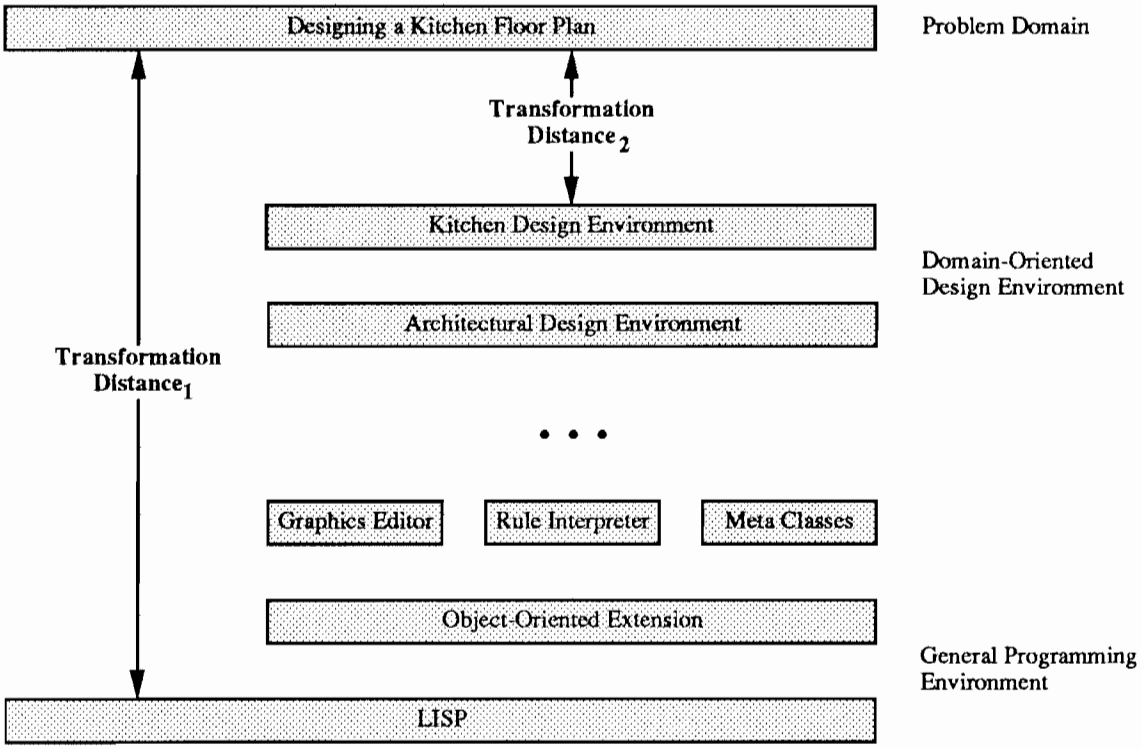

Figure 1: Layers of Abstraction to Reduce the Transformation Distance between Problem Domain and System Space

ming environments and command languages like the UNIX shell are efforts to make systems more accessible and usable. Coming from the other end, good turn-key systems contain features that make them modifiable by users without having to change internal structures. Editors allow users to define their own keys (e.g., "key-board macros"), and modern user interfaces allow users to create and manipulate windows, menus, icons etc. at an easy to learn level.

In our work we have tried to replace *human computer communication* with *human problem domain communication* [3]. The latter approach makes the computer an invisible instrument allowing knowledgeable, task-oriented scientists and designers to work with the abstractions and concepts of their domains. To achieve this goal, we have constructed knowledge-based design environments with an underlying layered architecture [10, 6]. Figure 1 shows the layered architecture underlying the JANUS system discussed in this paper.

End-user modifiability in the context of a layered architecture means that users can change the behavior of the system in the layers near the top where changes remain in the context of the problem space. 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.

A Taxonomy of End-User Modifiability. End-user modifiability is of crucial importance in knowledge-based design environments, because these systems do not try to serve as general purpose programming environments but provide support for specific tasks. In cases where designers of these environments have not anticipated specific activities, users must be able to modify the design environment itself. The changes supported by a modifiable system include the following (for slightly different taxonomies see [16] and [4]):

- setting parameters (e.g., with the help of property sheets),
- adding functionality to existing objects,
- creating new objects by modifying existing objects, and
- defining new objects from scratch.

End-user modifiability makes systems *adaptable*, in contrast to *adaptive* systems [5] which change themselves based on the user's behavior.

Why is End-User Modifiability Important? Pre-designed systems are too encapsulated for problems whose nature and specifications change and evolve. A useful system must accommodate changing needs. Domain experts must have some control over the system because they understand the semantics of the problem domain best. End-user

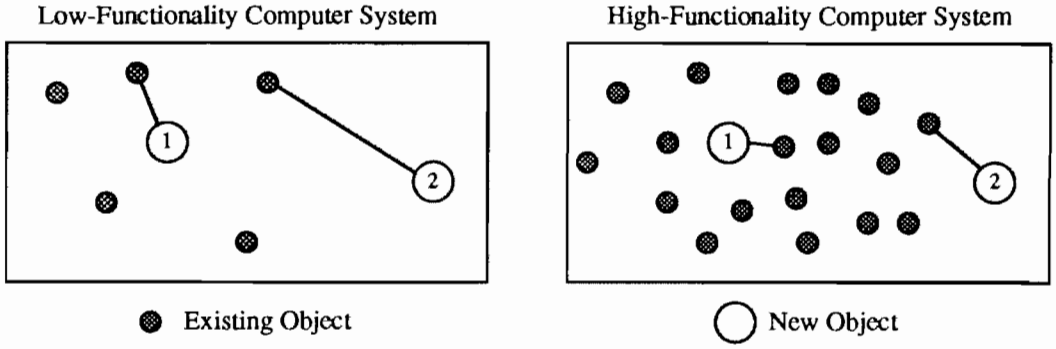

Figure 2: End-User Modifiability in a Low-Functionality System versus a High-Functionality System

The trade-off between low-functionality and high-functionality systems with respect to end-user modifiability is: it is easier to locate existing objects in low-functionality systems, but the potential for finding an existing object, which is close to what one needs, is higher in high-functionality systems. The figure also illustrates that the extend of the required modification depends on the desired new object: object "1" is closer than object "2" to the existing functionality.

semantics of the problem domain best. End-user modifiability is not a luxury, but a necessity in cases where the systems do not fit a particular task, a particular style of working or a personal sense of aesthetics.

End-user modifiability is equally important for knowledge-based systems which by their nature are never completely specified, and undergo continuous change and growth. The evaluation of the MYCIN system [2] showed that one of the major reasons that MYCIN was never used in a real setting was that the system's knowledge base was outdated by the time the system was finished. This lack of modifiability prohibited an evolution of the underlying knowledge base. Before machine learning can make major contributions to the knowledge acquisition problem, end-user modifiability is a promising approach to increase the amount of shared knowledge between a system and a user.

There are high costs associated with a failure to support end-user modifiability. Users are not in control of the interaction designed to achieve their goals. They have to put up with an unsatisfactory state of affairs, or they may not use a system at all, if it does not fit their needs.

ISSUES FOR END-USER MODIFIABILITY

High-Functionality Computer Systems. High-functionality computer systems [10] contain a large number of abstractions in an integrated software environment. Such systems can increase our productivity and efficiency by providing many built-in facilities that users would otherwise have to construct. They have the potential to support a "copy&edit" strategy [11, 12] for making modifications from a rich, initial foundation. Instead of starting from scratch, new functionality can be achieved by modifying an existing part of the system. Figure 2 illustrates the difference of supporting end-user modifiability in a low-functionality system versus a high-functionality system.

Locating Existing System Functionality. The limited success of modification as a major programming methodology is in our opinion directly related to the lack of support tools for exploiting the power of high-functionality systems. Having a large set of existing building blocks without good retrieval tools is a mixed blessing. The advantage of reuse and redesign is that existing buildings blocks - which have been used and tested before - already fits the users' needs or comes close to doing so. The problem is that it may take a long time to discover these suitable building blocks or to find out that none exists.

Comprehending Existing System Functionality. Locating promising parts of the system is just the first step in the modification process. In the next, step users have to comprehend an existing object in order to carry out the modifications. External modifications that do not require an understanding of the internal workings of an existing object are preferable to internal modifications. In addition, a system constructed using a layered architecture (see Figure 1) is very helpful. In such an architecture, users can remain in the higher layers during the comprehension process.

Supporting Changes. The last step in the modification process is to carry out the modifications. To do so, users should have a clear understanding of the implications of the change with respect to the problem domain. The system should support the mechanics of the change (e.g., with property sheets, animated examples, context-sensitive help at every stage). A uniform interface for the comprehension process and the modification process makes changes more natural, a principle violated by many systems (e.g., the Symbolics Document Examiner [17] offers a different

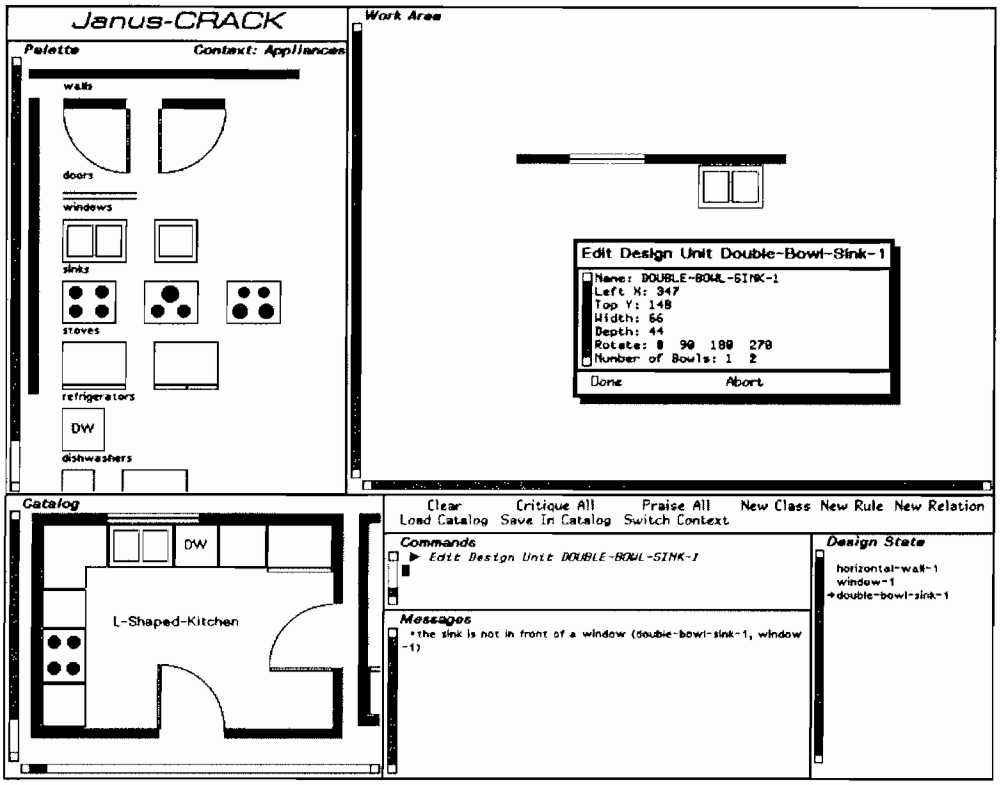

Figure 3: JANUS-CRACK: the Constructive Design Component of JANUS

This screen image shows the different components of JANUS-CRACK: the *Palette* contains the building blocks; the *Work Area* is used for the design and modification of a kitchen; the *Catalog* offers existing kitchen designs which can be used for redesign; the menu bar contains often used commands; the *Commands* pane is used issuing less frequently used commands; the *Messages* pane shows the critique generated by the critics and the *Design State* pane enumerates the objects in the current design.

writer's and reader's interface).

END-USER MODIFIABILITY IN KNOWLEDGE-BASED ENVIRONMENTS

The JANUS System

JANUS [6, 7] is a design environment that allows designers to construct architectural floor plan layouts of kitchens and at the same time to learn about the general principles underlying such constructions. JANUS does not try to automate the design process by replacing the human designer with an expert system, but rather it cooperates with the designer to enrich traditional design practices, amplifying the power of human designers rather than "deskilling" them.

JANUS contains two integrated subsystems: JANUS-CRACK and JANUS-VIEWPOINTS. JANUS-CRACK (see Figure 3) is a knowledge-based system supporting the construction of designs. JANUS-VIEWPOINTS is an issue-based hypertext system containing useful information about general principles of design. This integration allows argumentation to resolve problematic situations encountered during construction.

JANUS-CRACK supports *human problem-domain communication* as a construction kit. The building blocks (contained in the Palette) represent a design vocabulary and define a design space. Empirical evidence [14, 6] demonstrates that construction kits are necessary but not sufficient conditions for useful design environments. Design environments need embedded knowledge for distinguishing "good" designs from "bad" designs and explanations for justifying these distinctions. Kitchen design is more than selecting appliances from a palette; it also involves knowing how to combine these simple building blocks into functional kitchens. Knowledge about kitchen design includes design principles based on building codes, safety standards, and functional preferences. This knowledge, combined with critics which can use it, extend construction kits to design environments.

Critics in JANUS-CRACK are implemented as rules. They detect and critique partial solutions developed by the designer. The critics display critiques (such as: *"sink not in*

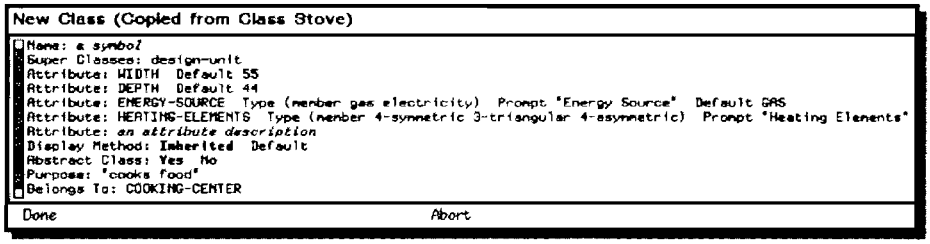

Figure 4: Definition of a New Class by Copying an Existing One

This figure illustrates how a copy of the class *stove* is used for the definition of a *microwave*. A class has the attributes *name, super classes, attributes, display method, abstract class, purpose*, and *belongs-to.*

---

front of a window") in a critic window (the message window in Figure 3). If designers want to get more information, request a suggestion, an explanation or want to understand the argumentation behind the critic's view, they can switch to the JANUS-VIEWPOINTS subsystem.

JANUS-VIEWPOINTS is a hypertext system for argumentative kitchen design based on the PHI design methodology [13], an extension of IBIS [9]. The elements of VIEWPOINTS are issues, answers, and arguments. JANUS-VIEWPOINTS is implemented within the Symbolics Document Examiner [17].

End-User Modifiability in JANUS

The possibilities for modification in earlier versions of JANUS were restricted to making modifications easy for artifacts constructed *within* the design environment. Experimental use of JANUS by professional and amateur kitchen designers indicated that situations arise that require *the modification of the design environment itself.*

We have extended the JANUS systems with knowledge-based components to support the following types of modifications:

1. introducing new classes of objects into the palette (e.g., a "microwave"),
2. adding new critic rules to the system (e.g., "the microwave should be next to the refrigerator"),
3. allowing the definition of new relationships (e.g., "between"), and
4. supporting the creation of composite objects (e.g., a "cleanup center").

These different types of modifications will be described below in more detail.

The knowledge-based components supporting these modifications provide a uniform interface for all modifications in the form of property sheets. These components give context-sensitive help at each step in the modification process, allow users to modify an existing example, take advantage of the layered architecture of the system by minimizing the number of layers that users have to cross, and exploit the properties of object-oriented representation supporting differential descriptions.

Definition of New Classes. Situations will arise in which users want to design a kitchen with appliances that are not provided by the design environment. For example, the palette in JANUS (see Figure 3) does not contain a *microwave*. The command "New Class" activates a system component supporting the addition of new elements to the palette. Property sheets help users define new design unit classes or modify existing ones by eliminating the need to remember names of attributes. The modification process is supported with context-sensitive help (e.g., showing users constraints for the value of a field). If values are required, users cannot leave the sheet without providing these values.

Although a new class can be defined from scratch, it is much easier to find an existing class that comes closest to the new one and to copy and edit it (see Figure 4). This changes the modification task from *"telling the system about* x" to *"finding an already known* x' *that's similar to* x". Even if a new class is defined from scratch, it is not necessary to specify it completely by integrating it into the inheritance hierarchy below an existing class. The system supports the finding of an appropriate class by displaying the class hierarchy. Under the assumption that each class in the hierarchy has a name meaningful in the problem space, the classification of the new class is reduced to a *selection* problem among meaningful names. In addition, users can look at the definition of each existing class and list their applicable critic rules (see Figure 5).

Depending on the type of the modification, it is sometimes better to introduce a new common superclass (of the existing and the newly defined class) rather than making the new class a subclass of an existing one. For example, a *microwave* and a *stove* could have the common superclass *cooking unit.* The rules and attributes of *stove* that also apply to *microwave* can then be moved to *cooking unit* [15]. The support of the system for this modification can be further improved by describing the purpose of every class with respect to the problem space. A *classifier* (such as the one in KL-ONE [1]) can then try to find the correct

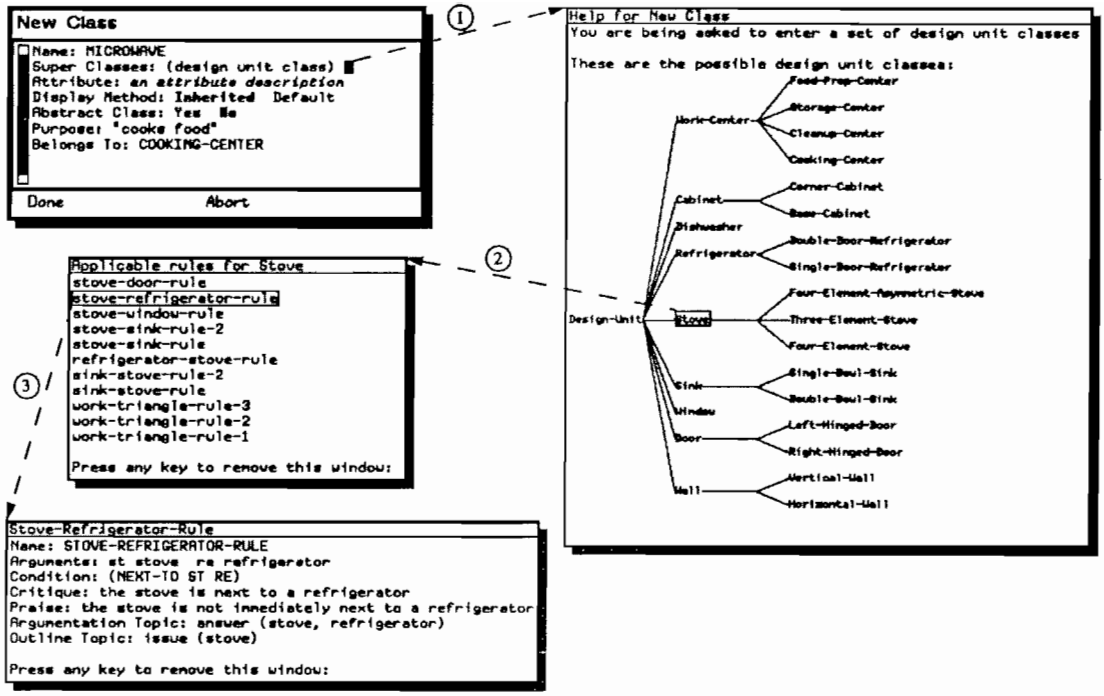

Figure 5: Rules Applicable for a Class

After the user presses the HELP key in the field "Super Classes," a window with the class hierarchy is displayed (1). Every class in the hierarchy is mouse-sensitive. A window with the list of rules that are applicable to a class is displayed by a mouse click on that class (2). The rule names are also mouse-sensitive, and a mouse click opens a window with the definition of a rule (3).

---

place for the new class in the hierarchy from its description. It may not be possible to do this automatically but the system could at least assist the user. For example, if an existing class and a new class both have the description *"cooks food"* (see Figure 4), the system could offer three options: make the new class a subclass of the old one, do it vice versa, or generate a common superclass that contains the common feature.

After a new class is located within the hierarchy, the differences between it and its superclass need to be defined. This is done by defining new attributes and redefining inherited ones. The system supports users in this task by providing a list of all inherited attributes and by giving help for the definition of new ones (see Figure 6).

Definition of New Critic Rules. A new rule can be defined with the "New Rule" command. A property sheet similar to the one used for the definition of design unit classes helps the user define the rule (see Figure 7).

The system provides additional support for the specification of a rule condition. JANUS contains different kinds of rules. On the one hand, a statement such as, "a sink should be under a window," means that for every sink there should be a window behind it. But the converse is not true, that there should be a sink in front of every window. On the other hand, users probably do not want to have different rules for the statements "a refrigerator should be near a sink" and "a sink should be near to a refrigerator." If there is more than one sink or more than one refrigerator, a mechanism is necessary for specifying whether these statements should apply for all possible combinations or only for one sink-refrigerator pair.

Definition of New Relations. Relations usually describe spatial relationships between design units (*near*, *next-to*, *away-from*, etc.). They are used in rule conditions. In JANUS, most relations are defined in terms of *distance* between design units. As shown in Figure 8, the *between* relation can be defined in terms of other relations.

A deeper knowledge about the relations in the system helps to detect conflicts between rules. For example, it is not possible for design units to accommodate the relations *near* and *away-from* at the same time. It is desirable for a user to be able to describe spatial relationships by showing an example to the system. Unfortunately, one can usually extract more than one relationship from an example. A solution could be a system that presents all relationships extracted from an example to users allowing them to select the relationship that is closest to their purpose and to refine

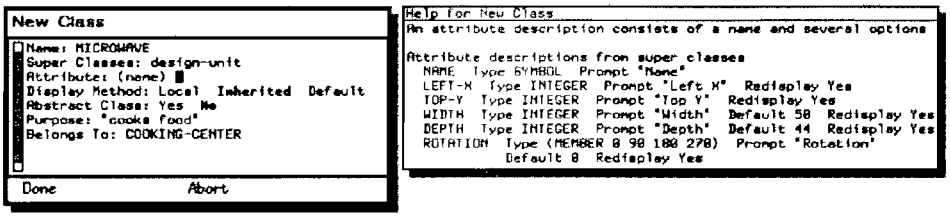

Figure 6: Definition of Attributes for a Class

An attribute is described by a name and "keyword-value" pairs. When specifying a new attribute, pressing the HELP key shows all inherited attribute descriptions. Users can redefine inherited attributes, e.g., by specifying a new prompt or a new default value. The attributes inherited from the class *design-unit* deal with size and position of a design unit. Other classes define problem-domain specific attributes (e.g., *energy-source* in the class *stove*).

---

it.

The relations have a layered architecture. In the current version of the system, any LISP expression can be used to define the condition part of a relation making it hard to assist users in the definition of relations. At least three layers are required:

1. existing relations and logical operators (and, or, not),
2. distance functions between design units, accessing functions for the position and size of design units, arithmetic and comparison functions (+, *, ..., <, >, ...), and
3. more useful LISP functions (trigonometric functions, conditionals, ...).

New relations should be defined within the top layer whenever possible.

Composite Objects. Our cooperation with domain experts [6] showed that kitchen designers use *intermediate abstractions*, i.e., a combination of several design units, during design. In JANUS, these intermediate abstractions are called "work centers." For example, a *sink* and a *dishwasher* could be combined into a *cleanup center.* Designers start designing a kitchen with work centers and replace them later with their components (see Figure 9).

The components supporting end-user modifiability in JANUS allow the creation of composite objects to be performed on different levels (similar to macro definitions in editors). On a lower level, JANUS supports user-definable grouping of graphical presentations of design units so that they can be moved and rotated together. On a higher level, JANUS provides a palette of work centers; the command "Switch Context" switches between the palette of work centers and the palette of appliances. Each work center has its own critic rules. Composite objects introduce an additional layer of abstraction for a design environment.

EVALUATION

The described end-user modifiability features have only been informally evaluated in order to help us understand to what extend our current system building efforts have instantiated our conceptual framework. The informal evaluations indicated that the additional power and flexibility does not come for free. Learning is required at different levels: users have to operate on different descriptive levels and they must familiarize themselves with the interaction mechanisms that are necessary for the modifications.

Explorations with the current systems taught us the following lessons:

- Modifiable systems may have side-effects on other system components (e.g., static structures such as the layout of the palette suddenly need to be changed).
- Making systems modifiable has to be a design goal for the original system. Adding system components to allow for end-user modifiability in a system constructed without this goal in mind is a nearly impossible task.
- A deeper understanding of the cognitive issues surrounding end-user modifiability is required [12], e.g., it is necessary to understand when classes or instances should be changed and how these different changes affect each other.
- The taxonomy needs to be extended and refined. It should include an illustration of the consequences of different approaches. Modifications need to be classified whether they are local to one user's environment or global to a whole user community, and whether they are temporary or permanent.
- End-user modifiable systems have the potential drawback that they result in multiple versions of the same systems violating standardization and portability goals.

EXTENSIONS

In the current implementation, design units are presented as icons by using methods that call graphics functions. Modifying icons or creating new ones requires changing method definitions. That forces users to descend into the lower layers of our architecture (see Figure 1). Another solution must be found. In the meantime, users have the

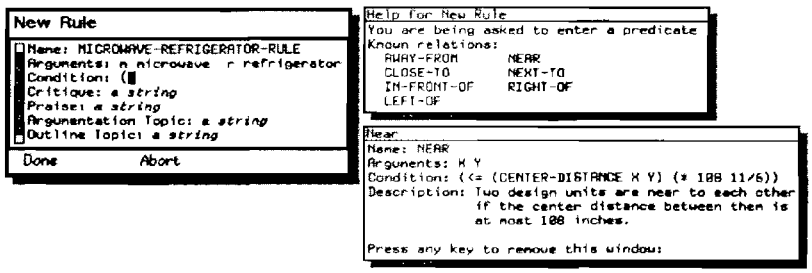

Figure 7: Definition of a Rule

After the user presses the HELP key in the field "Condition," a window with the names of all currently available relations between design units is shown. These names are mouse-sensitive. The definition of a relation is displayed by a mouse click.

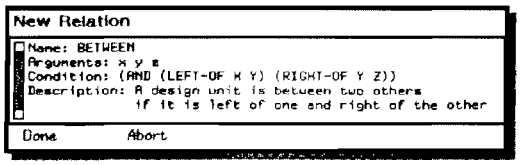

Figure 8: Property Sheet for the Definition of a Relation

choice between an inherited, predefined display method and a default display method drawing a rectangle with the class name in it.

The current system also makes it impossible to define an *on-top-of* relation at a higher layer in the architecture, because the system represents only a two dimensional space. A on-top-of relation is needed if users want to have a rule that enforces that one design unit is always on top of another (e.g., a microwave on top of a cabinet). An alternative solution for this problem might be the introduction of a new class that models the two units together. The system should have knowledge structures that assist users in exploring these alternatives.

Modifiable systems generate new problems as consequences of these modifications. In earlier versions of JANUS, the layout of the palette could be determined at the time the design environment was created. The current layout of the palette puts classes that have the same superclass in the same row. Designers might want to have different layouts, e.g., positioning the design units most important to their work at the top of the palette. By allowing users to dynamically add new objects to the palette, the layout of the palette will change as well. This requires either that users determine the new structure (and are able to save it) or that the system has enough knowledge to compute it.

At the current stage of development, our efforts towards making JANUS more end-user modifiable have concentrated on JANUS-CRACK, the constructive design component. In the future, we will investigate modifiability issues in relationship to the Catalog (see Figure 3) and JANUS-VIEWPOINTS. Hypertext systems [16] have to be treated differently, because information structures can be interpreted by other humans and need not be understood by the system.

CONCLUSIONS

The goal of making systems modifiable by users does not transfer responsibility for good system design to users. Average users will never build systems of the quality a professional designer would; but this is not the goal of convivial systems. Only if a system does not satisfy the needs and the taste of its users (which they know best themselves) should they be able to carry out a constrained design process to modify it. The strongest test of a system with respect to user modifiability and user control is not how well its features conform to anticipated needs, but how easy it is to modify it in order to perform a task the designer did not foresee.

ACKNOWLEDGMENTS

The authors would like to thank Raymond McCall and Anders Morch, who were major contributors in the development of the JANUS system. Thomas Mastaglio, David Redmiles, Brent Reeves, and Curt Stevens provided valuable comments on earlier drafts of this paper.

The research was partially supported by grant No. IRI-8722792 from the National Science Foundation, grant No. MDA903-86-

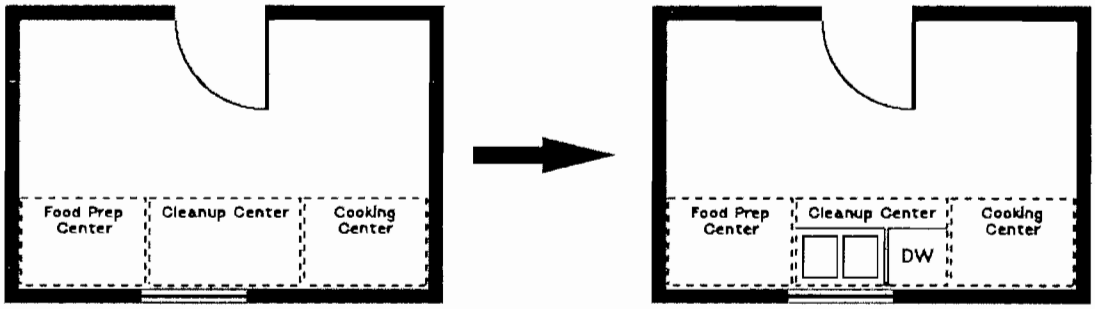

Figure 9: Work Centers

Composite objects allow designers to design at different levels of abstractions. After having completed the design of the kitchen at the work center level, users can proceed to the detailed design of the centers. The figure shows how the *cleanup center* is expanded to its components *sink* and *dishwasher*.

C0143 from the Army Research Institute, and grants from the Intelligent Systems Group at NYNEX and from Software Research Associates (SRA), Tokyo.

REFERENCES

1. R.J. Brachman. On the Epistemological Status of Semantic Networks. In N.V. Findler (Ed.), *Associative Networks - Representation and Use of Knowledge by Computers*, Academic Press, New York, 1979.

2. B.G. Buchanan, E.H. Shortliffe. *Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project*. Addison-Wesley Publishing Company, Reading, MA, 1984.

3. G. Fischer, A.C. Lemke. Construction Kits and Design Environments: Steps Toward Human Problem-Domain Communication. *Human-Computer Interaction 3*, 3 (1988), 179-222.

4. G. Fischer, A.C. Lemke. Constrained Design Processes: Steps Towards Convivial Computing. In R. Guindon (Ed.), *Cognitive Science and its Application for Human-Computer Interaction*, Lawrence Erlbaum Associates, Hillsdale, NJ, 1988, Chap. 1, pp. 1-58.

5. G. Fischer, A.C. Lemke, T. Schwab. Knowledge-Based Help Systems. *Human Factors in Computing Systems, CHI'85 Conference Proceedings (San Francisco, CA)*, ACM, New York, April, 1985, pp. 161-167.

6. G. Fischer, R. McCall, A. Morch. Design Environments for Constructive and Argumentative Design. *Human Factors in Computing Systems, CHI'89 Conference Proceedings (Austin, TX)*, ACM, New York, May, 1989, pp. 269-275.

7. G. Fischer, R. McCall, A. Morch. JANUS: Integrating Hypertext with a Knowledge-Based Design Environment. *Proceedings of Hypertext'89*, ACM, New York, November, 1989, pp. 105-117.

8. I. Illich. *Tools for Conviviality*. Harper and Row, New York, 1973.

9. W. Kunz, H.W.J. Rittel. *Issues as Elements of Information Systems*. Working Paper 131, Center for Planning and Development Research, University of California, Berkely, CA, 1970.

10. A.C. Lemke. *Design Environments for High-Functionality Computer Systems*. Ph.D. Thesis, Department of Computer Science, University of Colorado, Boulder, CO, July 1989.

11. D. Lenat, M. Prakash, M. Shepherd. CYC: Using Common Sense Knowledge to Overcome Brittleness and Knowledge Acquisition Bottlenecks. *AI Magazine 6*, 4 (Winter 1986), 65-85.

12. C.H. Lewis, G.M. Olson. Can the Principles of Cognition Lower the Barriers of Programming? In G.M. Olson, E. Soloway, S. Sheppard (Eds.), *Empirical Studies of Programmers (Vol. 2)*, Ablex Publishing Corporation, Lawrence Erlbaum Associates, Norwood, NJ - Hillsdale, NJ, 1987.

13. R. McCall. PHIBIS: Procedurally Hierarchical Issue-Based Information Systems. *Proceedings of the Conference on Architecture at the International Congress on Planning and Design Theory*, American Society of Mechanical Engineers, New York, 1987.

14. D.A. Norman. Cognitive Engineering. In D.A. Norman, S.W. Draper (Eds.), *User Centered System Design, New Perspectives on Human-Computer Interaction*, Lawrence Erlbaum Associates, Hillsdale, NJ, 1986, Chap. 3, pp. 31-62.

15. M.J. Stefik, D.G. Bobrow. Object-Oriented Programming: Themes and Variations. *AI Magazine 6*, 4 (Winter 1986).

16. R.H. Trigg, T.P. Moran, F.G. Halasz. Adaptability and Tailorability in NoteCards. H.-J. Bullinger, B. Shackel (Eds.), *Proceedings of INTERACT'87, 2nd IFIP Conference on Human-Computer Interaction (Stuttgart, FRG)*, North-Holland, Amsterdam, September, 1987, pp. 723-728.

17. J.H. Walker. Document Examiner: Delivery Interface for Hypertext Documents. *Hypertext'87 Papers*, University of North Carolina, Chapel Hill, NC, November, 1987, pp. 307-323.