IEEE Software

Cognitive View of Reuse and Redesign

Reusable components are not enough. Program designers need tools that help them understand the components and how to use them. Fortunately, some support tools do exist.

Gerhard Fischer, University of Colorado at Boulder

The microelectronics revolution of the 1970s made computer systems cheaper and more compact — and greatly increased the range of their capabilities. Much of the available computing power is wasted, however, if users don't understand and use the systems' full potential.

In the past, too much attention has been given to systems technology and not enough to the effects of that technology — which has produced inadequate solutions to real-world problems, imposed unnecessary constraints on users, and failed to respond to changing needs.

Most users feel that computer systems are unfriendly, uncooperative, and require too much time and effort to get something done. They feel dependent on specialists. They notice that software is *not* soft: A system's behavior cannot be changed without major reprogramming. To let users take advantage of previous work, software environments must support reuse and redesign.

Software environments must support design methods whose main activity is not only generating new programs but also maintaining, integrating, modifying, and explaining existing ones.[1] For software systems in ill-structured problem domains where detailed specifications are not available (like artificial intelligence and human-computer interaction), incremental, evolutionary reuse and redesign must be efficiently supported.

Successful reuse and redesign face several challenging problems that are primarily cognitive. It is a big mistake to assume that reuse and redesign pose no new design challenges. New architectures and new support tools are needed to reduce some of the cognitive demands. To understand and overcome these problems, my colleagues at the universities of Stuttgart and Colorado and I have defined a theoretical framework and constructed several systems for reuse and redesign that include many tools and intelligent support systems.

0740-7459/87/0700/0060/$01.00 ©1987 IEEE

Our object-oriented system architecture provides great flexibility, enhances the reusability of many building blocks, and supports redesign. In this framework, existing objects can be used as is or with minor modifications, and the designer can base a new system on standard, well-tested components.

To make these reuse methods viable, systems need many functions. However, such complex systems are a mixed blessing. The advantage is that a set of building blocks probably fits our needs (or comes close to doing so) and has already been used and tested. The disadvantage is that building blocks are useless unless the designer knows that they are available and how the right one can be found.

Reuse and redesign

Just as you rely on established theorems in a new mathematical proof, you should build new systems as much as possible with existing parts. To do so, the designer must understand how these existing parts function.

Abstraction levels. Figure 1 shows how many large software systems are built: A monolithic system is completely implemented in a general-purpose programming language. To make design methods like reuse and redesign possible, the design of intermediate abstraction levels must be an integral part of the software process. This strategy allows both easy redesign by modifying the original design and easy reuse by recombining the intermediate abstractions to form a different system.

The intermediate abstraction levels are modeled as construction kits that contain a set of building blocks for specific problem domains. The building blocks define a design space (the set of all possible designs that can be created by combining these blocks). The building blocks in our system are organized as inheritance networks in an object-oriented architecture, which provides components on multiple levels and makes it easier to extend the set of available blocks. Simon has argued convincingly that the evolution of a complex system proceeds much faster if stable intermediate parts exist.[2]

Cognitive issues. Several cognitive problems prevent users from successfully exploiting their function-rich systems. Users do not know

- that tools (building blocks and support systems) exist,
- how to access tools,
- when to use these tools,
- what the tools do, or
- how to combine, adapt, and modify tools to their specific needs.

Software design tasks can be characterized with two models: the situation model and the system model.

The situation model is the user's mental representation of the situation, including the problems motivating the design task, general ideas about how a solution will be obtained, and a characterization of the desired goal state. The first stage of problem-solving is transforming this verbal, imprecise situation model into a formal system model.

The system model is a set of operations that will result in the desired solution. For each operation, the system must have commands to execute it. Thus, the system model is a sequence of operations that can be instantiated through a command.

In the situation model, goals refer to actions and states in the real world. Goals may be precise or imprecise, but they do *not* belong to the computer world. In contrast, the system model is part of the formal world of computers. Depending on the problem, goals can be expressed in terms of real-world actions and events or in terms of formal, computational methods.

Design tasks require translation of the problems expressed in the situation model to the system model and require construction of the right sequence of operations to yield the solution. To do this, search

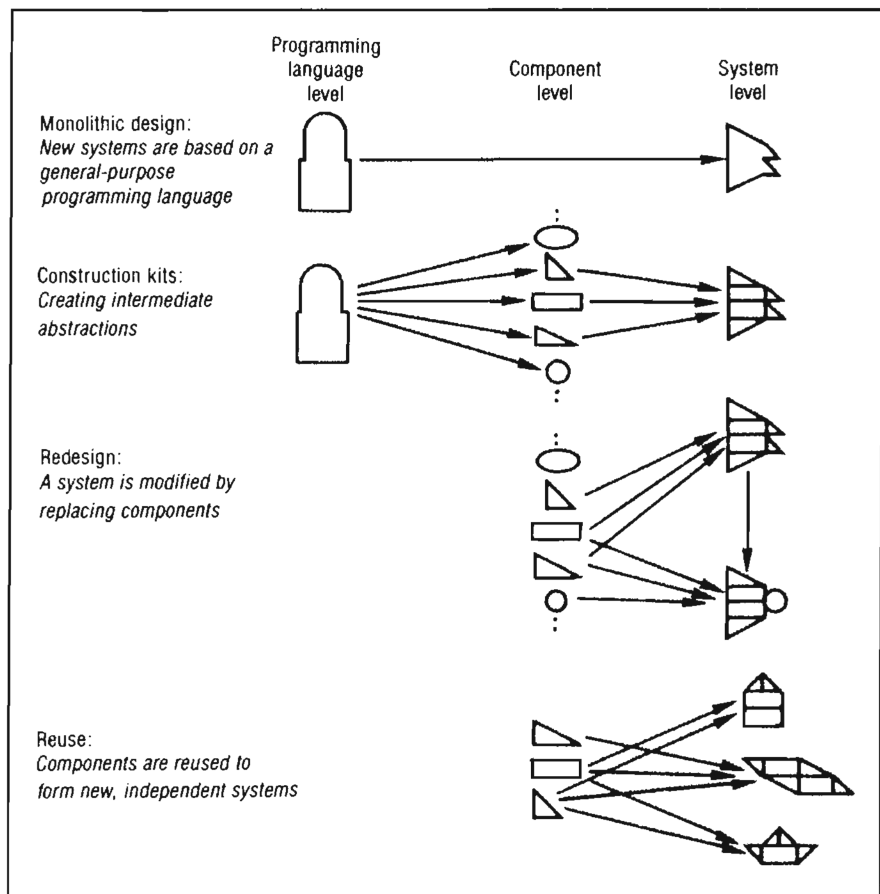

Figure 1. Reuse and redesign.

Number of Computational Objects in Systems

EMACS:
- 170 function keys and 462 commands

Unix:
- More than 700 commands and many embedded systems

Lisp systems:
- Franz Lisp: 685 functions
- WLisp: 2590 Lisp functions and more than 200 ObjTalk classes
- Symbolics Lisp Machines: 19,000 functions and 2300 flavors

Amount of Written Documentation

Symbolics Lisp Machines:
- 10 books with 300 pages
- Does not include any application programs

Sun workstations:
- 15 books with 4600 pages
- Additional beginner's guides: eight books totaling 800 pages

Figure 2. Example quantitative analyses.

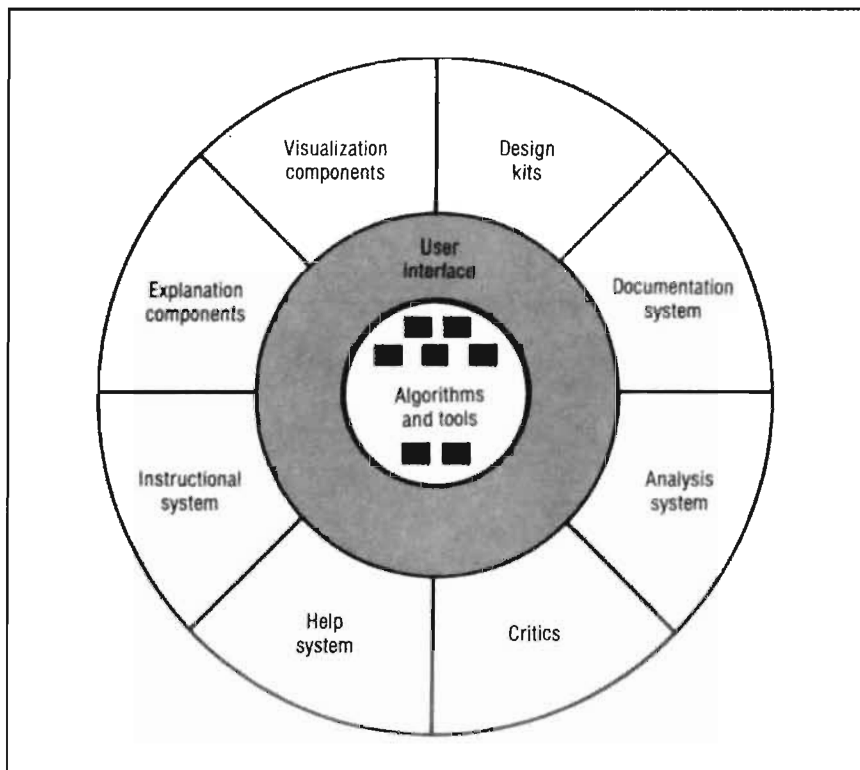

Figure 3. Intelligent design environment architecture.

strategies must be used. Psychological research has shown that there are great differences between the efficient and successful strategies used by experts and the inefficient and ineffective search strategies of novices.[3]

Sometimes a design task is approached with a fully elaborated, precise situation model: Designers know exactly what they want. The problem is figuring out how to do it — how to find a sequence of methods that define a workable system model. More typically, the situation model is inadequate, imprecise, incomplete, and inconsistent. In this case, the situation model must be modified.

Help systems. This may be why help systems are inferior to human advice. Help systems focus on problems at the level of the system model — the actual operations and commands needed — but users may have a confused situation model that prevents them from using the help facilities successfully. A human expert, however, will try to figure out what the user ***really*** wants or needs, before jumping into the technical details.

Systems that try to model many different problem domains (like Unix and Lisp) must be large and complex to provide all the necessary abstractions. Figure 2 shows a quantitative analysis of some of these systems.

With all this information available, it seems that these systems should be ideal candidates for reuse and redesign. But informal investigations[4,5] indicate that these systems are underused and that users quickly settle at a plateau of suboptimal behavior. This finding is not surprising, given the inability of the systems to promote incremental learning and support learning on demand — capabilities prerequisite for reuse and redesign. What is needed is not more information but new, better ways to structure and present information.

To ask a question, you must know enough to know what is not known! How can you find out about all the relevant abstractions and building blocks if you don't even know what to ask? Having a vague situation model means that you cannot ask the right questions. We need active systems that volunteer help in appropriate situations rather than respond to explicit requests.

Level of understanding. An important question in redesign is the level of understanding necessary for successful redesign: Exactly how much must users understand? Differential programming, programming by specialization (based on our object-oriented knowledge representation language Objtalk[6]), and tools will help make it easier to modify an existing system than to create a new one.

In ill-structured problem domains, it is seldom possible to provide a precise specification of intent. Design instabilities and frequent redesigns are necessarily common. The main difficulty is not a correct implementation but the development of specifications that lead to effective solutions for real needs.

Life-cycle models are inadequate for ill-structured problems and should be

replaced by incremental design and a rapid prototyping method based on a communication model.[7] Reuse and redesign (supported by adequate tools) that allow the exploration of alternatives can considerably enhance this approach.

Human problem-domain communication provides a new level of quality in human-computer interaction because it lets us build an application's important abstract operations and objects directly into the exploratory environment. The domain-oriented abstractions bridge the gap between the situation model and the system model. Reuse and redesign in such a system let the domain experts operate with their own abstractions.

In most cases, we do not want to eliminate the semantics of a problem domain by reducing the information to formulas in first-order logic or to general graphs. Whenever users can directly manipulate the concepts of an application, programs become more understandable, and the distinction between programmers and nonprogrammers vanishes.

Design environment architecture. Figure 3 describes our vision of the architecture of an intelligent design environment. This architecture is based on the belief that the intelligence of a complex tool must contribute to its ease of use.

Truly intelligent and knowledgeable humans, such as good teachers, use a substantial part of their knowledge to explain their expertise to others. In the same way, the intelligence of a computer system (by incorporating knowledge about the user, the tasks being carried out, and the communication process) should be used to provide effective communication.

We have built prototype systems in many areas of Figure 3's outer ring: documentation systems, help systems, critics, and visualization tools.

By being part of our design environment, intelligent support systems can enhance the learning and understanding of complex software environments. Different modes of learning can complement each other. The modes used depend on whether the users' goal is to complete an action or to acquire new knowledge and whether users are inexperienced with a system or familiar with it.

Our intelligent design environment supports several major learning modes:

- Unguided, active exploration. This mode lets users fully control what they would like to do and how they would like to do it. It is important that an environment for this kind of learning support safe experimentation — undo mechanisms are crucial, for example — and that the environment be intuitive.
- Tutoring. This mode is adequate to begin learning a new system. It lets you predesign a sequence of microworlds[4] that make up learning space of a complex system, which a user can master only incrementally. But tutoring does not support learning on demand well when intermediate users are exploring their own areas. Tutoring is not task-driven because the total set of tasks cannot be anticipated. Instead, the system controls the dialogue, and users have little control over what to do next.

Intelligent support systems can enhance the learning and understanding of computer software environments.

- Asking for help. In passive help systems, users must actively seek help. But knowing what you can ask for and finding information in complex systems is hardly easy. Finding information is difficult because there is usually a huge gap between the initial mental form of the query (articulated in the situation model) and the corresponding expression required by the system.
- Answers first, then questions. To ask a question, you must know how to ask it, and you cannot ask questions about knowledge that you don't know exists. We plan to investigate programs, like active help systems and critics, that volunteer information and support the acquisition of information by chance.
- Learning on demand. Active help systems and critics also support learning on demand. Users often do not want to learn more about a system or a tool than necessary for the immediate solution of their problem. To successfully cope with new problems as they arise, users require a consultant that generates advice tailored to their needs. A support system for learning on demand provides information only when it becomes relevant. This approach eliminates the burden of learning many things when users do not know whether the information will ever be used and when learners have difficulty imagining an application for the knowledge.
- Human assistance. If available on a personal level, human assistance is still the most useful source of advice. The mode of learning can best be characterized as cooperative problem-solving. Learners can ask a question in an infinite variety of ways. They get assistance in formulating the question. And they can articulate their problem in terms of the situation model rather than being required to do so in terms of a system model.

All these learning modes must be supported so users can understand and exploit function-rich systems for reuse and redesign.

Methods, tools, and systems

Object-oriented system architectures support new methods like differential programming and programming by specialization (which let users design systems like "*X* is like *Y*, except . . . "). Inheritance promotes reuse and redesign because it lets objects that are almost like other objects be created with only a few incremental changes. Inheritance reduces the need to specify redundant information and simplifies updating and modification by letting information be entered and changed in one place.

Support tools like the Objtalk Browser and the Objtalk Navigator address the retrievability problem by letting users locate resources without knowing and remembering names. Design kits go beyond construction kits in that they use general knowledge about design (which meaningful artifacts can be constructed, how and which building blocks can be combined with each other) that is useful for the designer.

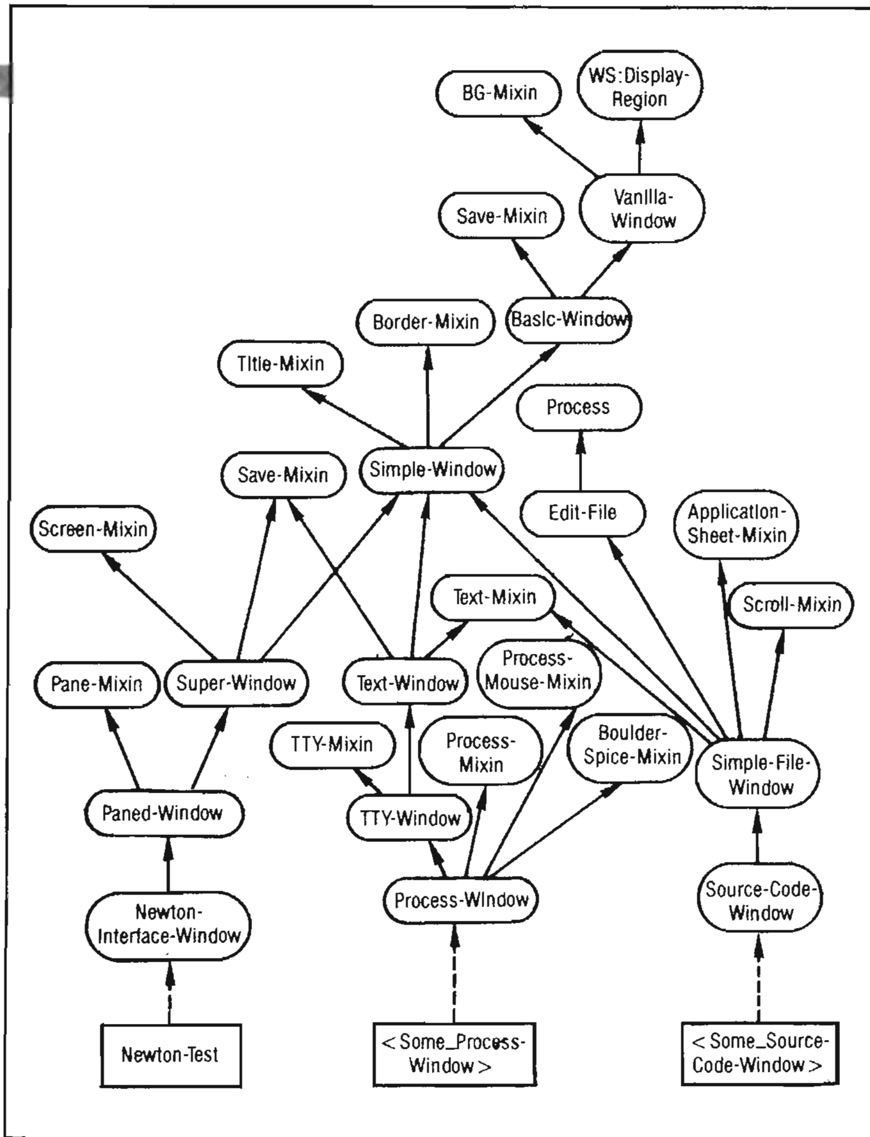

Figure 4. Inheritance of the Newton interface using paned windows (detailed in Figure 15). Objects with round corners are classes. Solid arrows represent the superclass relationship. Dashed arrows show the relationship of objects to their class.

Objtalk architecture. Objtalk is an object-oriented knowledge representation language that supports multiple inheritance. Control is expressed in terms of message-passing among objects. Object behavior is based on interpreting messages and on their internal state. Objects are organized in classes. Objects of the same class — the instances of that class — have the same methods and slots and show the same type of behavior.

Classes are arranged in a hierarchy. They share the properties (methods and slots) of their parent classes. Properties are thus inherited by classes. All classes form an inheritance hierarchy with a special class at the root called Object. The fundamental properties of all objects are defined in Object. They can be modified or extended in any of its subclasses. Figure 4 shows one interface's inheritance hierarchy.

Object-oriented formalisms support constrained design processes through instantiation of existing classes and through creation of subclasses that can inherit large amounts of information from their superclasses. You can accomplish many tasks before you must use the full generality of the formalism by defining new classes.

WLisp construction kit. WLisp[8] is a user interface toolkit implemented in Objtalk and Franz Lisp (see Figure 5). Users interact with WLisp primarily by direct manipulation. Actions that change the common world of the user and the system can be invoked by the system (by updating some information about the state of an application) or by the user (by selecting some command from a menu).

All WLisp components are organized in an inheritance hierarchy of classes (Figure 4 shows a part of the hierarchy). Classes describe screen objects (such as windows and menus) and components of screen objects (such as borders, titles, and buttons).

The uniform representation on all levels is one of WLisp's major design properties. New interfaces can be constructed quickly by providing appropriate building blocks that suggest good designs. The object-oriented system architecture is highly flexible and enhances the reusability of many building blocks. When creating new interfaces, the designer may use existing objects as is or with minor modifications and can thus rely on standard, well-tested components.

All systems embedded in WLisp use the inheritance hierarchies by defining subclasses of one of the window types. An application system can use menus, icons, buttons, and dialogue windows directly. Other classes serve as sources from which properties are inherited. Applications usually define their own classes, which use existing functionality by inheritance. All WLisp classes can be used as building blocks for more sophisticated user interface components. They provide the basic functionality of screen objects. We found them useful and often necessary components of a new interface design.

A construction kit with many generally useful building blocks (WLisp has more than 200) provides a good basis for redesign. The abstractions make up stable intermediate parts user interfaces. WLisp, which has evolved with experience gained in its applications, is now based on several abstractions that characterize the domain of user interface design. These abstractions make up a preliminary "theory" for a class of user interfaces. The evolutionary development of such a framework, driven by testing the validity of abstractions in different applications, is a prerequisite for the development of a system that supports reuse and redesign.

Objtalk Browser. The Objtalk Browser

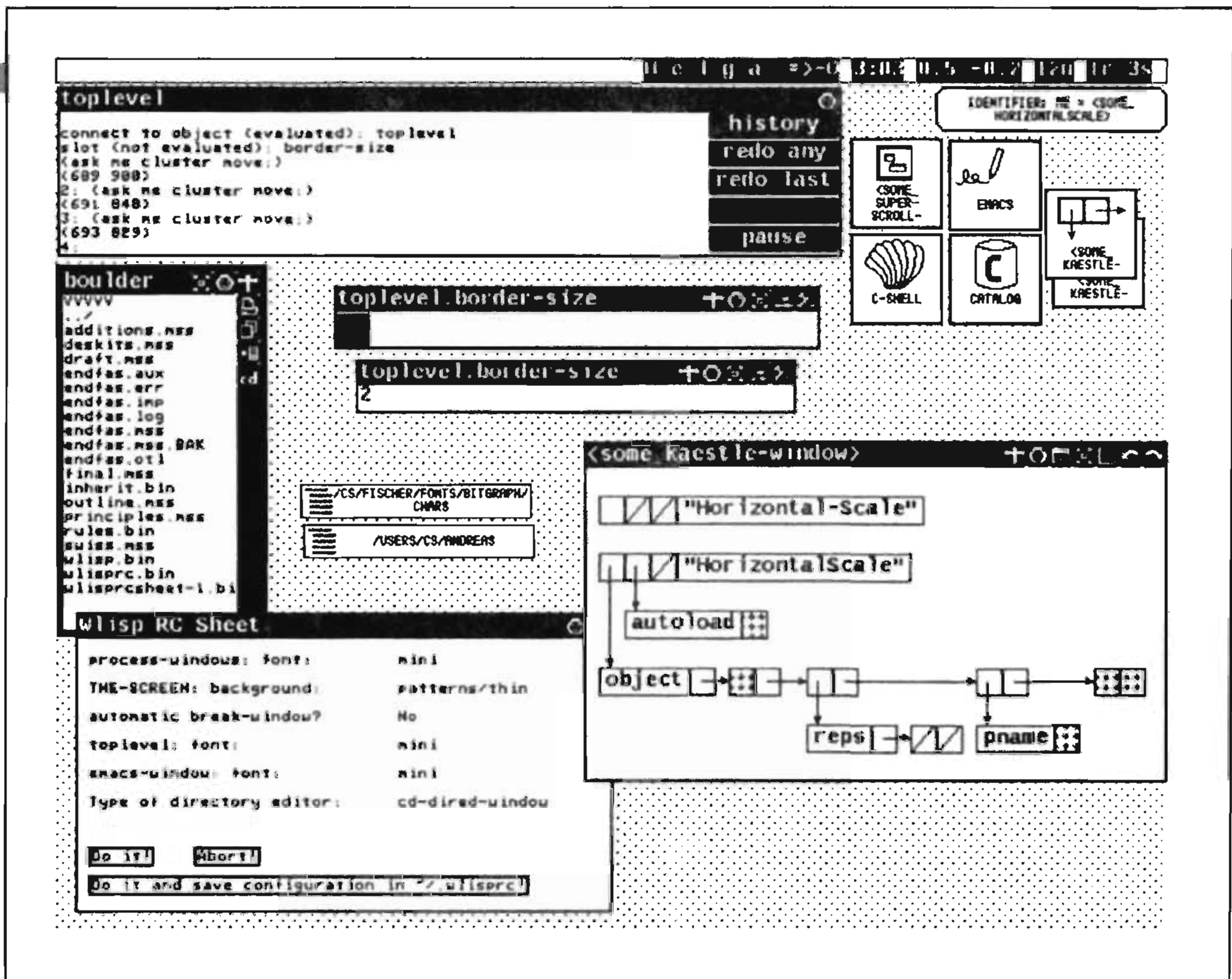

Figure 5. WLisp.

tool (see Figure 6) explores a large space of building blocks embedded in an inheritance network. The browser displays the inheritance structure of a system and lets users look around a system with an unfamiliar structure and search for building blocks. The connections between the system and the components it inherits can be analyzed in more detail by selecting a component and looking at its slot descriptions, defaults, triggers, and methods. Smalltalk has similar tools. Both the Objtalk Browser and the Objtalk Navigator (described next) are steps towards solving the retrievability problem.

Objtalk Navigator. A tool like the browser does not support location of inherited slots or methods. Working with Objtalk and WLisp, we have observed that there is a design trade-off in the design of inheritance systems: Trying to reuse building blocks as much as possible leads to a wide distribution of information throughout the inheritance network, making it difficult to understand a specific behavior. The Objtalk Navigator tool helps locate functions regardless of where they might be defined in the hierarchy.

In addition to the browser, the navigator (see Figure 7) shows all inherited slots and methods, lets users selectively view any subset of the inherited slots and methods, and lets users go directly to the superclass containing the inherited slot or method. All inherited slots and methods are presented in a scroll menu in the Inherited Slots and Inherited Methods windows. The Inherited From window is empty until a slot or method is selected with the mouse; the superclasses that contain this slot or method are then displayed. Users can also specify keywords to search all methods and slot names.

The navigator was built as an extension of the browser, providing further empirical evidence for the reuse and redesign possibilities offered by Objtalk and WLisp.

Constrained design processes. If we base our system design efforts on reuse and redesign, the question is how we combine and modify the existing pieces with mechanisms other than conditionals, loops, and recursion: What is the semantics of the arrows in Figure 1 combining components into systems?

Unix lets us treat an existing program as independent building blocks. If one of these building blocks does not meet our needs, we need not — and should not — change the component itself. Instead of rewriting the building block, we can wrap some program code around it that provides the needed properties. To protect users from the overwhelming complexity of function-rich systems, you need to support

- enhanced incremental learning of

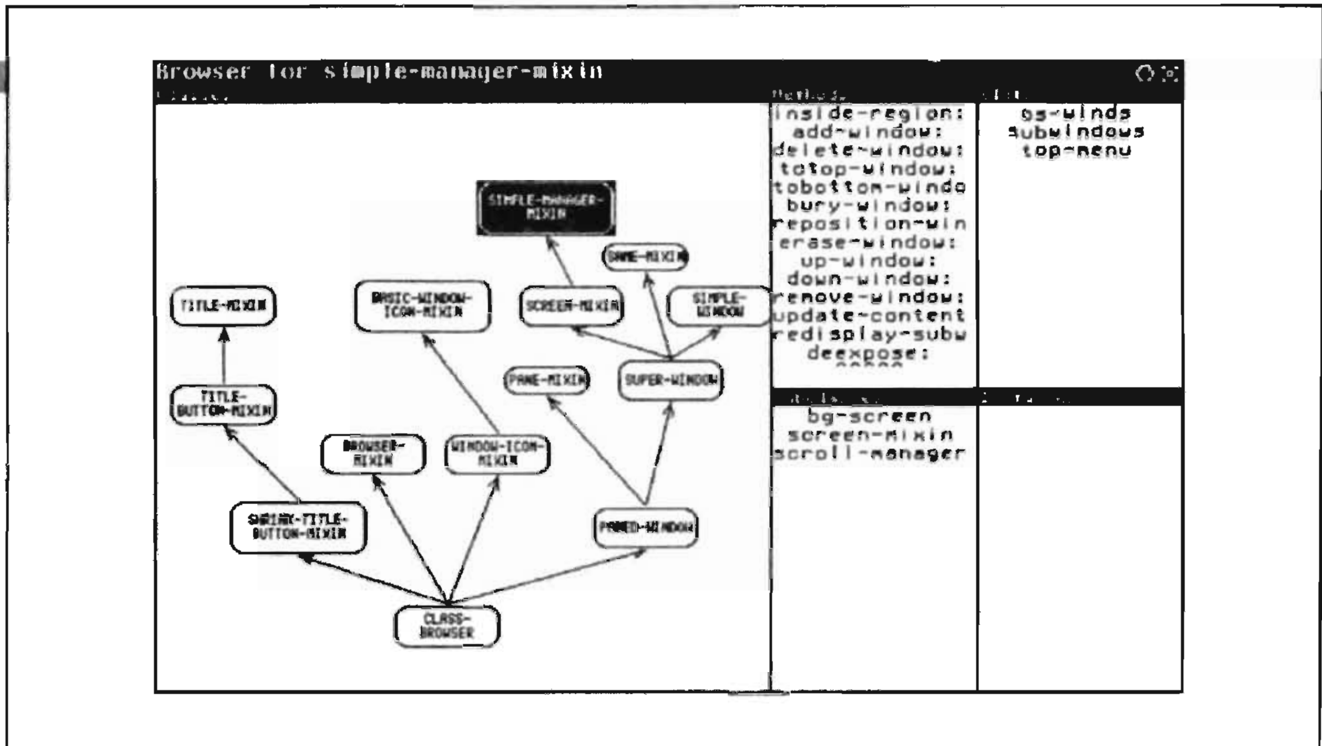

Figure 6. Objtalk Browser with windows to show class hierarchy.

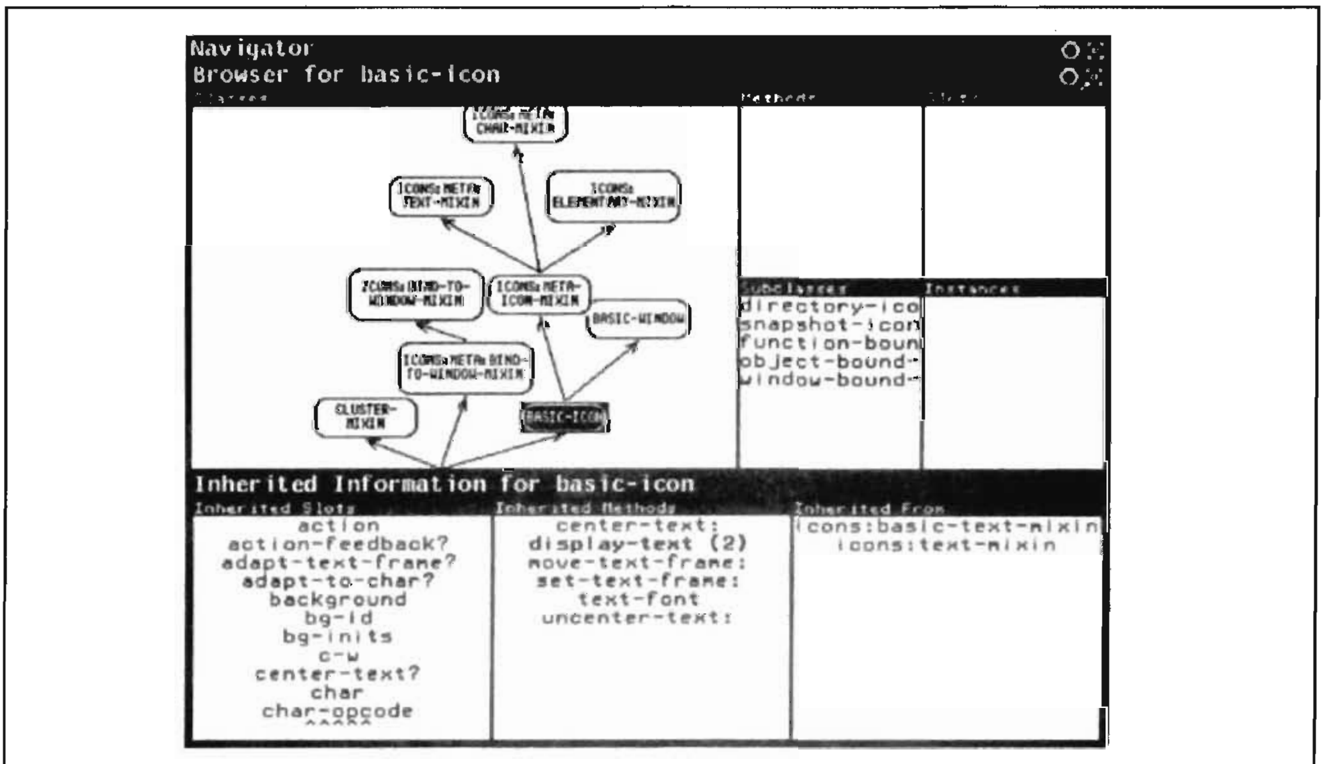

Figure 7. Objtalk Navigator.

complex systems and to delimit useful microworlds (inexperienced users should be able to get started and do useful work when they know only a small part of the system),

• increased subjective computability (by eliminating prerequisite knowledge and skills and by raising the abstraction level),

• more efficient experts (they can rely on tested building blocks, need not worry about details, and can engage in rapid prototyping),

• contextual guidance (so users can choose the next steps), and

• directed movement from chaos to order (for example, a language's primitives and a technical construction kit's basic elements give users little guidance on how to construct a complex artifact).

The goal of making tools modifiable by the user does not mean transferring the

responsibility of good tool design to the user. Typical users will never build tools with the same quality that a professional designer would. Only if the tool does not satisfy the users' needs and the tastes (and users know best what they want) should a user carry out constrained design to adapt the tool.

The strongest test of user modifiability and control is not how well its features conform to anticipated needs but how well it performs when you want to do something that the designer did not specifically foresee but that the system can do. Most systems are too encapsulated for problems whose nature and specifications change and evolve. A useful system must accommodate these changing needs by incorporating reuse and redesign as explicit objectives into the original design.

Constrained design is not only useful for producing transient objects by nonexperts: Experts can also take advantage of these objects if they support the required functionality. The advantages of restricting yourself to the limits of constrained design are that the human effort is smaller (there is less code to write), the process is less error-prone (because you can rely on tested building blocks), and users must know less to succeed (there are no worries about low-level details). These advantages are especially important for a rapid prototyping method where several experimental systems must be constructed quickly.

Selecting tools from a set of tools is the least demanding method to carry out a constrained design. But in realistic situations, it is far from trivial. For example, a Swiss army knife has at most 15 different tools while a function-rich computer system has hundreds or thousands of tools.

Catalog, a tool to access the many tools and application systems in WLisp, simplifies tool selection. The iconic representations help users see what is there and give clues about systems users might be interested in (see Figure 8).

If you accept recursive function theory, you know you can compute anything with a set of very simple functions and powerful combination methods like function definition and recursion. But many combinations are too complicated and require too many intermediate levels to get to the abstraction level that users can operate at.

Simple combination processes let users define keyboard macros in extensible editors. The combination method in this case is a simple sequencing operation. Another example is the concept of a Unix pipe, which uses the output of one tool as the input to another tool. Direct manipulation styles of human-computer interaction are also based on a simple combination process: Any output that appears on the screen can be used as input.

Combination processes become more difficult if there are many building blocks to work with, if the number of links necessary to build a connection increases, and if compatibility between parts is not obvious.

Instantiation is another method to carry out constrained design. A restricted form of programming in an object-oriented formalism like Objtalk can be done by creating instances of existing classes. Classes provide a set of abstract descriptions and, if enough classes exist, you can generate a broad range of behavior.

Systems to support reuse and redesign should not be restricted to providing only design building blocks; they should also support composition of systems within the application domain. The building blocks should have self-knowledge and should be more like active agents than like passive objects.

Design kits are an instance of intelligent support systems and should be integral parts of future computer systems. System that allow user modifiability should have associated design kits. Design kits can contribute to resolving the basic design conflict between generality and power versus ease of use.

Design kits provide prototype solutions and examples that can be modified and extended to achieve a new goal instead of starting from scratch — they support a copy-and-edit method of constructing systems through reuse and redesign of existing components.

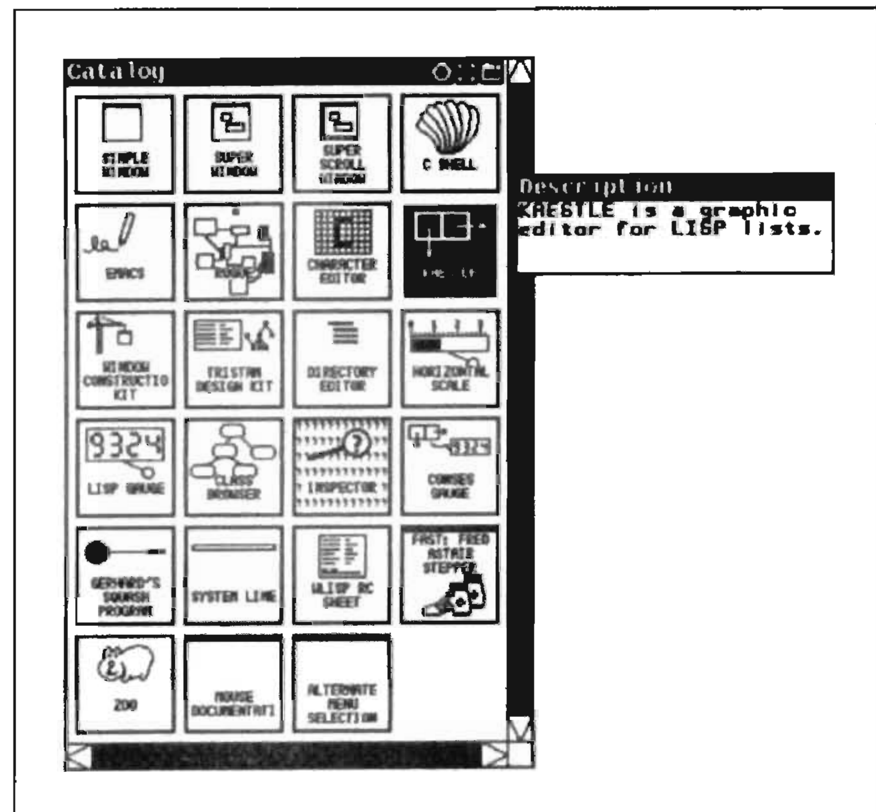

Figure 8. Catalog tool to simplify selection.

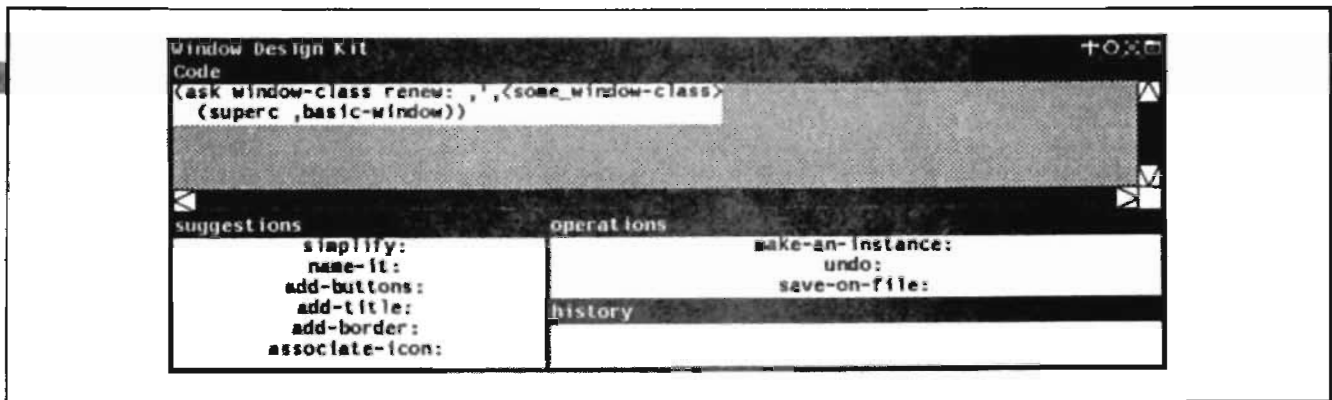

Figure 9. Wides initial state.

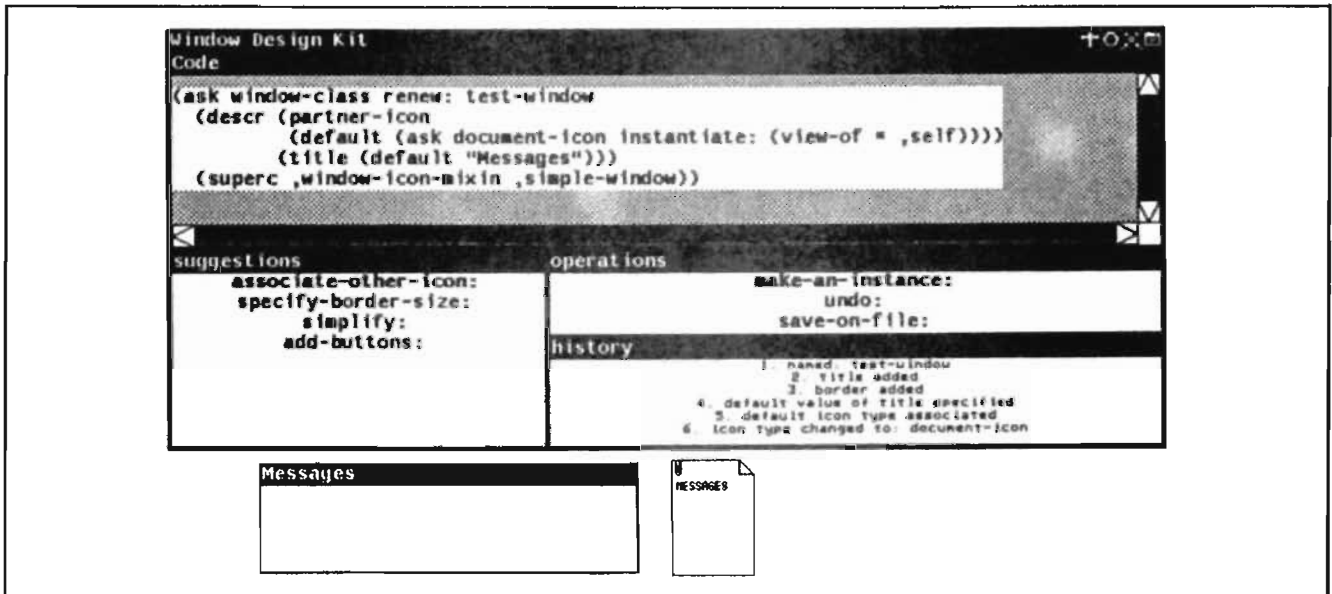

Figure 10. Wides window and its associated icon.

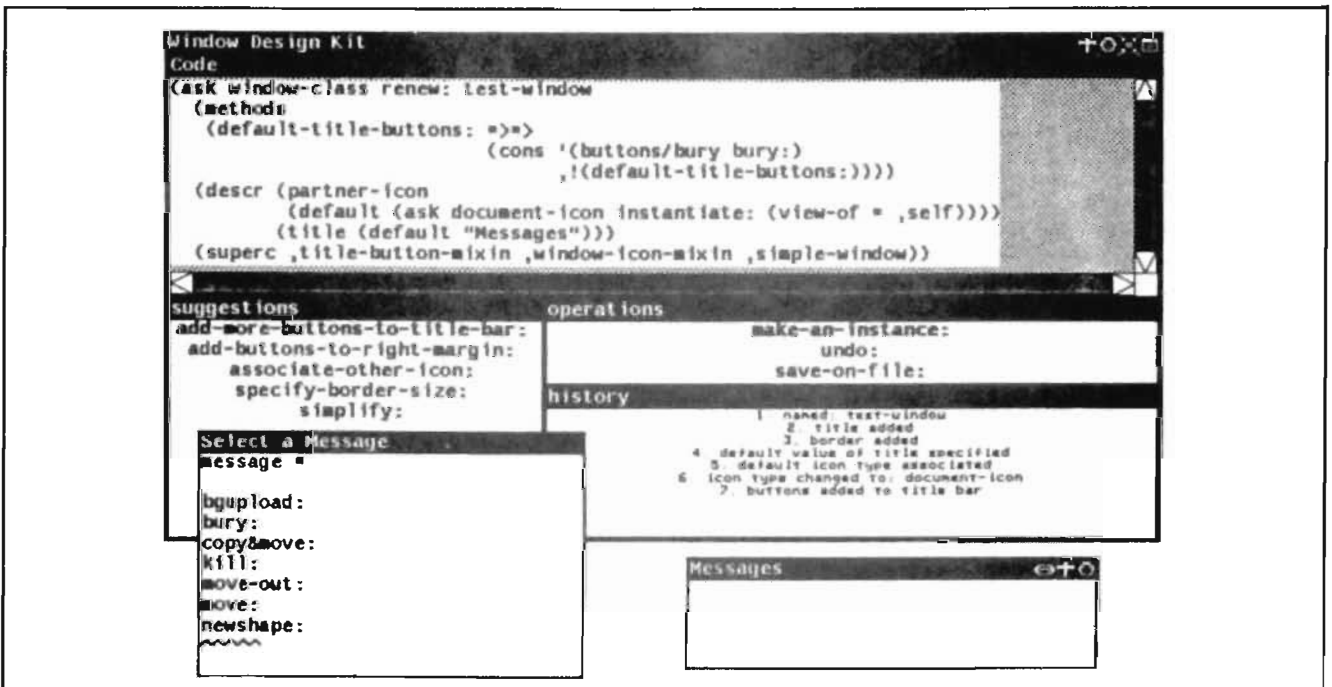

Figure 11. Adding a button to the Wides title bar.

Wides design kit. Window systems and user interface tool kits have rich functionality. There are text windows and graphic windows that have controls like menus and pushbuttons. There are various text, graphic, and network editors that may be adapted. Using these components now requires considerable expertise that may only be acquired through extended learning and experimentation.

Wides, a window design kit for WLisp, lets designers build window-based systems at a high abstraction level. It generates the programs in the background. Wides' goals are to reduce the knowledge required to use the components, to support learning, and to shape a toolkit's structure so useful work can be done when users know only a small part of the structure.

Wides provides a safe learning environment where no fatal errors are possible and where users always have enough information to proceed. The design kit lets users create simple window types for their applications. Figure 9 shows the system's initial state, a window with four panes:

- a code pane that displays the current definition of the window type,
- a menu of suggestions for enhancements to the window type,
- a history list, and
- a menu of general operations.

Figure 10 shows a window and an icon of the selected type.

Figure 11 shows an even more complex modification: Windows can be associated with pushbuttons such as those in the upper right corner of the Wides window. Clicking the button with the mouse sends a message to the window. As an example of complex modification, extend the default pushbuttons (the two rightmost ones) in the title bar by adding a pushbutton that buries the window out of view.

After selecting Add-More-Buttons-to-Title-Bar from the suggestions menu, you must choose a button icon and a message from two menus. In Figure 11, the reduce button appears as the leftmost button in the instance of Test-Window. The Save-on-File operation may be used to save the final definition for later use.

Although the system generates little code (because it uses many high-level building blocks), Wides represents a significant advantage for the user. To construct a new window type, users no longer must know what building blocks (superclasses like Title-Mixin) exist, what their names are, and how they are applied. Users no longer must know that new superclasses must be added, for example, to the Superc description of a class. Also, Wides determines the correct superclass order. The system knows what types of icons are available and how an icon is associated with a window.

To construct a new window type, users no longer must know what building blocks exist, what their names are, and how they are applied.

User interface techniques like prompts and menus make it easy to experiment with window construction. The system makes sure that errors are impossible — but this does not mean that these techniques make sure that users always *understand* what they are doing.

These methods can be applied easily to well-structured domains. There are two problems, however, that must be addressed: (1) Seeing an option in a menu does not imply that its significance is obvious. For example, what does Associate-Icon mean? What is the function of a window's icon? (2) There may be too many options.

We did not look into the problem of too many options because it does not occur in our relatively small system, but future systems may offer hundreds of choices. For these design kits, a system of reasonable defaults may provide some help if combined with a set of predefined samples that are themselves specific starting points.

Trikit design kit. A common user interface problem is displaying and modifying hierarchical and network structures. Examples include application systems that deal with structures of databases, directory trees, and dependency graphs.

Tristan is a generic tool to display these structures graphically on the screen. To combine the generic tool with specific application systems, we built Trikit, a design kit for graph display/edit tools. Figure 12 illustrates its use.

The application programmer who is an expert for the application system — but not for building user interfaces — adjusts parameters of a generic tool and specifies the links between it and the application. The result of the design is a new, application-specific tool to display and edit the underlying data structure.

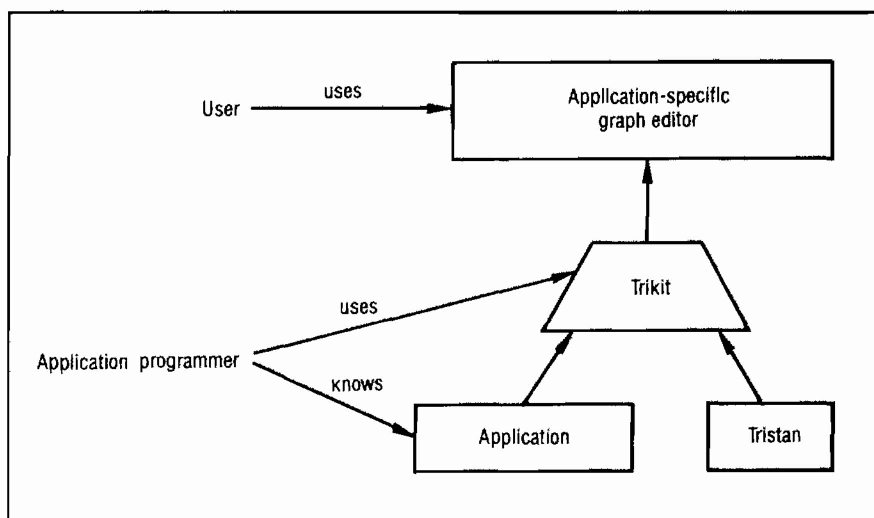

Figure 12. Using Trikit.

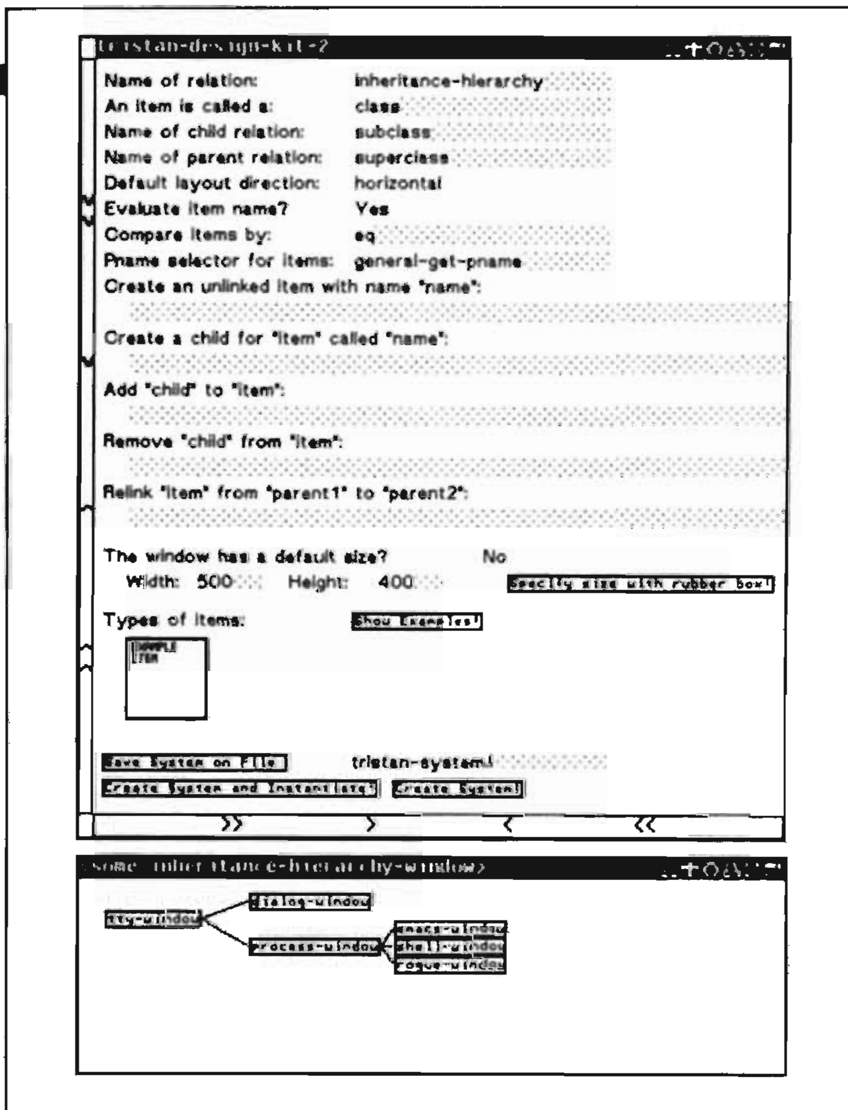

Figure 13. Initial state of Trikit's main form and an inheritance-hierarchy window generated from it.

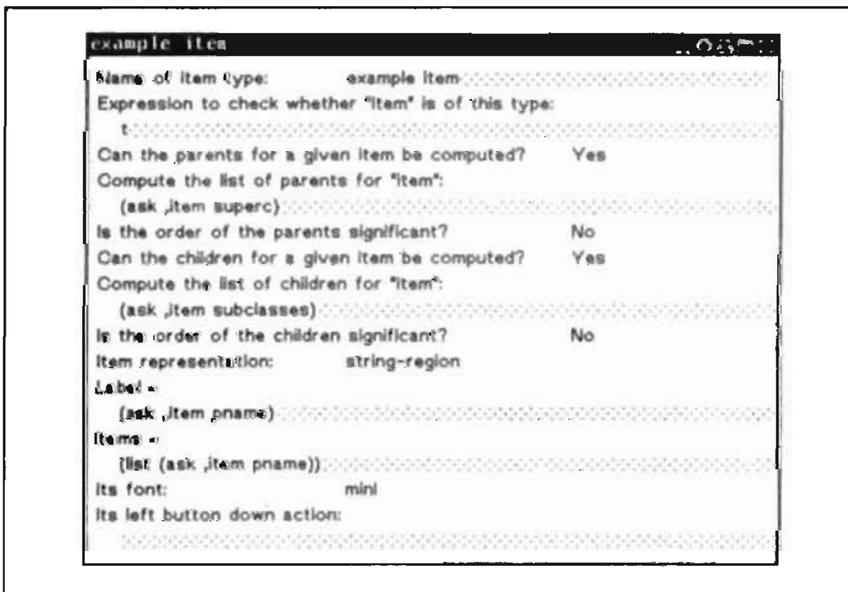

Figure 14. Initial state of the node form.

Trikit presents itself to users as an interaction sheet, as Figure 13's top window shows. It is where users specify the interface to the application, choose a graphical representation for the nodes, and control the creation of the user interface.

The initial form is filled in with an example application (see Figure 13's bottom window). This lets users familiarize themselves with the system, modify parameters, and find out about their significance. Clicking the square representing the Example Item subform produces the form in Figure 14. While the main form is associated with the graph in general, the subforms describe the properties of its nodes.

Trikit can construct useful systems without knowing the details of the selected building blocks. Although the design space is limited by the available options, you can still use this system to create a prototype that may be refined on a lower level. We have used the system to build an Objtalk inheritance-hierarchy editor, Unix directory editor, subwindow-hierarchy display, a project-team hierarchy, and an Emycin rule-dependency display.

Enhancement with WLisp. Newton[9] and WLisp were designed and implemented independently by two different research groups. Newton is a tool that lets you inspect and debug Fortran programs at execution time. We wanted to integrate the two systems to enhance the usability and understandability of an existing software tool (in this case, Newton) and to test the reusability and redesign possibilities of the user interface toolkit (WLisp).

Newton originally ran on a teletype-oriented terminal with no support for multiple viewports or a mouse. With a modest effort,[6] Newton was embedded as a fully functional system into the environment provided by WLisp (see Figure 15). It provides all the functionality of the original system plus many features gained from the integration into WLisp. It uses paned windows, menus, and mouse-sensitive text extensively.

This system-building effort showed how the structural properties of WLisp can be used in reuse and redesign to construct a user interface for an existing system. We met many objectives by instantiating exist-

ing classes — only a few required that we exploit the inheritance mechanism and the extension possibilities in Objtalk.

Findings and questions

Kernighan claims that "many Unix systems have the source code for commands, libraries, etc. on-line and accessible to all users. Accordingly, it is straightforward to find a program, read its code, and use it as a starting point for a new program."[10] There is absolutely no evidence for this "straightforwardness" in the empirical work that we carried out on Unix.

Whoever thinks that reuse and redesign have no costs is wrong. If reuse and redesign are such great ideas and so easy to master, why have they had only a limited success in software construction?

When observing designers dealing with complex software systems, you can reason that they do not engage in reuse and redesign (even though they would like to have a new system or make the system behave differently) because these methods are not adequately supported. The effort to change a system or to explore design alternatives is too expensive in most production environments.

Our experience indicates that if the cost of making changes is cheap enough, users will start to experiment — gaining experience and insight leading to better designs. The existence of editing and formatting systems has shown that writers are increasingly willing to modify manuscripts, articles, and books because the overhead in doing so is now acceptable.

Our environment supports reuse and redesign with basic formalisms, tools, and support systems. The reaction of designers (students, software engineers, and user-interface designers) using these tools has been largely positive, and these designers have built many complex systems with different user interfaces (examples are shown in the figures).

The increased willingness to reuse and to redesign has led to the construction of systems that fit an environment of needs. Systems that use WLisp's abstractions are relatively easy to modify, maintain, and adapt. The object-oriented architecture provides a much greater flexibility and range of application than traditional subroutine libraries, which have failed as tools for reuse and redesign because they are filled with specific implementations.

Several problems remain unsolved. Knowing about the existence of components is not trivial, especially as the number of components grows. And if you do find a potentially useful component, you must determine how it must be used and combined with the other components. You must understand its functionality and its properties (design kits such as Wides and Trikit try to solve this problem).

The problem of understanding, however, remains: For example, the fields in Trikit's forms use a certain terminology that not everyone is familiar with. We need more active tools and agents that have sufficient self-knowledge to offer their services. This self-knowledge must be represented in metaclasses that allow the representation of structural and behavioral properties of classes.

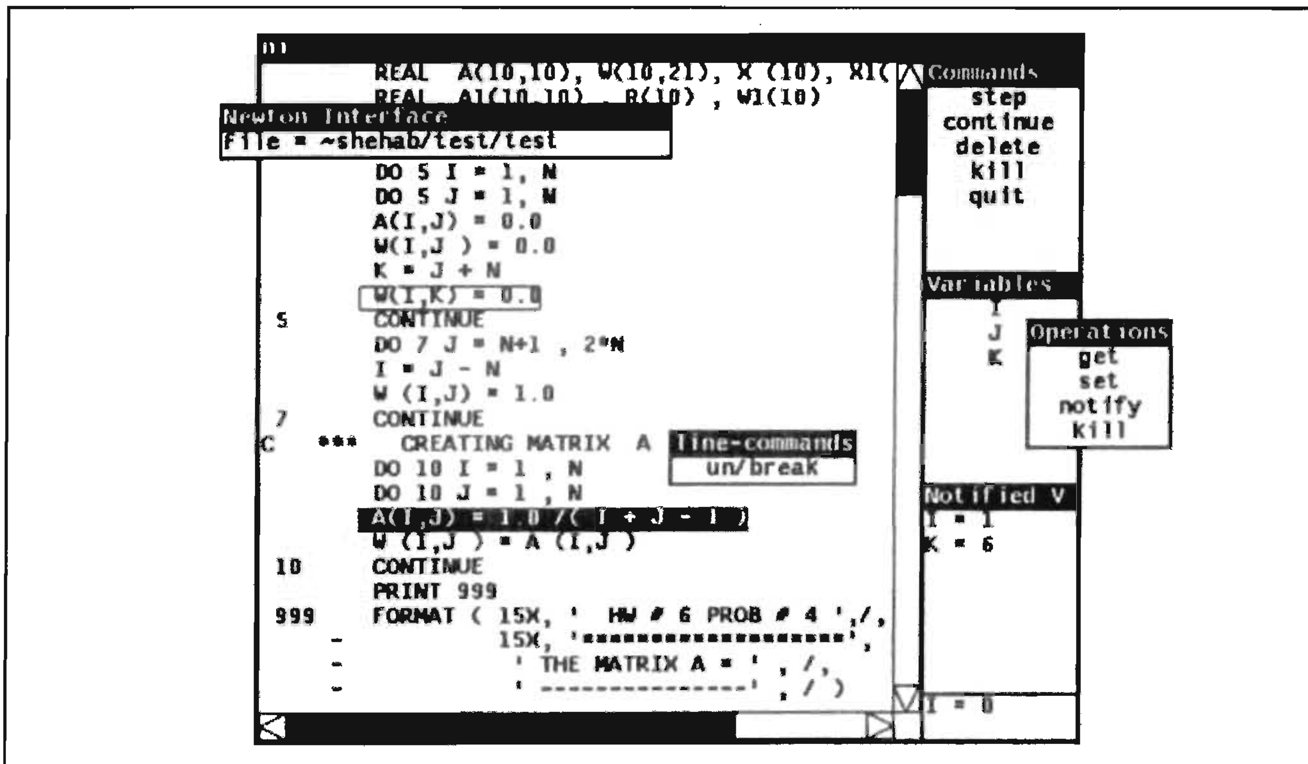

Figure 15. Newton integrated in the WLisp environment.

To cope with the ever-increasing complexity of designing, constructing, maintaining, and enhancing software, we need more than intellectual capabilities and discipline from software engineers. We need new methods and better tools to amplify and augment human intelligence. Reuse and redesign will play a crucial role in the future because they let knowledge about problem domains accumulate and be shared — just like in other scientific disciplines, which mature as abstractions and concepts are created.

But the existence of information alone is not good enough. Intelligent support systems are needed to overcome some of the cognitive problems associated with highly functional computer systems.

Design for reuse and redesign is a promising method for keeping software soft and preventing it from becoming obsolete with age. The creation of new systems (through reuse) and the modification of existing ones (through redesign) is not automated in this framework — but it is greatly aided.

Software reuse and redesign is similar to interior design of buildings, where contexts and boundaries are given: In our case, contexts and boundaries are provided by the abstractions modeled as components of the construction kits. In systems developed with this method, incremental improvements will be cheap enough that software engineers can experiment with alternative implementations, gaining experience and insight leading to better designs. □

Acknowledgments

I thank my former colleagues and students from the Inform project at the University of Stuttgart, West Germany, and my current colleagues and students at the University of Colorado at Boulder — especially Franz Fabian, who created the foundations of WLisp; Christian Rathke, who developed Objtalk, the Objtalk Browser, and the Newton interface; Andreas Lemke, who contributed many ideas and conceptualizations and who developed Wides and Trikit; Helga Nieper, who developed Tristan; Jim Sullivan, who developed the Objtalk Navigator; and Hal Eden, who makes sure that all our systems do what they are supposed to do. Without their contributions, this research effort would not have been possible. The reviewers provided many insightful comments. Richard Byrd and Paul Harter helped me to overcome the shortcomings of not being a native speaker.

The research was supported by National Science Foundation grant DCR-8420944 and Army Research Institute grant MDA-903-86-C0143.

References

1. T. Winograd, "Beyond Programming Languages," *Comm. ACM*, July 1979, pp. 391-401.
2. H.A. Simon, *The Sciences of the Artificial*, MIT Press, Cambridge, Mass., 1981.
3. W. Kintsch and J.G. Greeno, "Understanding and Solving Word Arithmetic Problems," *Psychological Rev.*, Vol. 92, 1985, pp. 109-129.
4. G. Fischer, "A Critic for Lisp," to appear in *Proc. 10th Int'l Conf. Artificial Intelligence*, AAAI, New York, 1987.
5. G. Fischer, A.C. Lemke, and T. Schwab, "Knowledge-Based Help Systems," *Proc. Human Factors in Computing Conf.*, ACM, New York, 1985, pp. 161-167.
6. C. Rathke, "Human-Computer Communication Meets Software Engineering," *Proc. Ninth Int'l Conf. Software Eng.*, CS Press, Los Alamitos, Calif., 1987, pp. 216-224.
7. G. Fischer and M. Schneider, "Knowledge-Based Communication Processes in Software Engineering," *Proc. Seventh Int'l Conf. Software Eng.*, CS Press, Los Alamitos, Calif., 1984, pp. 358-368.
8. H.-D. Boecker, F. Fabian, Jr., and A.C. Lemke, "WLisp: A Window-Based Programming Environment for Franz Lisp," *Proc. First Pan-Pacific Computer Conf.*, Australian Computer Society, Melbourne, Australia, 1985, pp. 580-595.
9. J.D. Feiber, R.N. Taylor, and L. Osterweil, "Newton: A Dynamic Analysis Tool Capabilities Specification," Tech. Report CU-CS-200-81, Computer Science Dept., Univ. of Colorado, Boulder, Colo., 1981.
10. B.W. Kernighan, "The Unix System and Software Reusability," *IEEE Trans. Software Eng.*, Sept. 1984, pp. 513-518.

Gerhard Fischer is an associate professor of computer science and a member of the Institute of Cognitive Science at the University of Colorado at Boulder. Fischer directs the university's Knowledge-Based Systems and Human-Computer Communication research group. Before joining the university, he directed a similar group at the University of Stuttgart in West Germany.

He has been a visiting research associate at Massachusetts Institute of Technology, Carnegie Mellon University, and Xerox Palo Alto Research Center. His research interests include artificial intelligence, human-computer communication, and software engineering.

Fischer received a PhD in computer science from the University of Hamburg.

He can be contacted at Computer Science Dept., University of Colorado, Campus Box 430, Boulder, CO 80309; CSnet gerhard@sigi.colorado.edu.

Copyright © 1987 The Institute of Electrical and Electronics Engineers, Inc.
Reprinted with permission from IEEE SOFTWARE,
10662 Los Vaqueros Circle, Los Alamitos, CA 90720