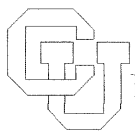
From Design to Redesign *

Gerhard Fischer
Andreas C. Lemke
Christian Rathke

CU-CS-368-87



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

* This research was supported by: Grant No. N00014-85-K-0842 from the Office of Naval Research, Grant No. DE-FG02-84ER1328 from the Department of Energy, the German mistry for Research and Technology, and Triumph Adler Corporation, Nuernberg.

From Design to Redesign

Gerhard Fischer
Andreas C. Lemke
Christian Rathke

CS-CU-368-87 June 1987

Department of Computer Science
Campus Box 430
University of Colorado,
Boulder, Colorado, 80309

This research was supported by: Grant No. N00014-85-K-0842 from the Office of Naval Research, Grant No. DE-FG02-84ER1328 from the Department of Energy, the German mistry for Research and Technology, and Triumph Adler Corporation, Nuernberg.

FROM DESIGN TO REDESIGN

Gerhard Fischer, Andreas C. Lemke, Christian Rathke

Department of Computer Science and Institute of Cognitive Science
University of Colorado, Campus Box 430
Boulder, CO 80309

ABSTRACT

Software Engineering environments have to support design methodologies whose main activity is not the generation of new independent programs, but the maintenance, integration, modification and explanation of existing ones. Especially for software systems in ill-structured problem domains where detailed specifications are not available (like Artificial Intelligence and Human-Computer Communication), incremental, evolutionary redesign has to be efficiently supported.

To achieve this goal we have designed and constructed an object-oriented, knowledge-based user interface construction kit and a large number of associated tools and intelligent support systems to be able to exploit this kit effectively. Answers to the "user interface design question" are given by providing appropriate building blocks that suggest the way user interfaces should be built. The object-oriented system architecture provides great flexibility, enhances the reusability of many building blocks, and supports redesign. Because existing objects can be used either directly or with minor modifications, the designer can base a new user interface on standard and well-tested components.

1. Introduction

Human-computer communication and knowledge-based systems are two research domains consisting mostly of ill-structured problems. In these domains it is seldom possible to provide a precise specification of intent; and without this specification, correctness is in general not a meaningful question. The main difficulty is not "correct" implementation of given specifications, but development of specifications that lead to effective solutions corresponding to real needs. In [Fischer, Schneider 84] we argued that life cycle models are inadequate for ill-structured problems and should be replaced by incremental design and a rapid prototyping methodology based on a communication model. Construction kits and support tools that allow the exploration of alternatives can considerably enhance this approach.

Redesign is a methodology which achieves that the prototyping process is *rapid* and it allows us to explore design alternatives for a problem. In addition, it supports that existing systems can be adapted to new requirements and can be tailored to special needs of individual users and user communities. Our redesign methodology is based on a construction kit (see Figure 2-1) which provides as building blocks the abstractions expected to be relevant for exploring the design space. In the domain of human-computer communication and user interfaces, primitives such as windows, menus, icons and editors are the basis for specific applications. Over the last six years we have designed, implemented and continuously enhanced a system for this domain called WLISP [Fabian 86, Boecker, Fabian, Lemke 85]*. WLISP's flexibility as a construction kit is provided by its object-oriented architecture. It contains a large amount of knowledge about the design of user interfaces. Additional knowledge is represented in support systems that guide the design process and critique intermediate results.

An especially interesting feature of WLISP is that it is one of the few systems implemented in a time sharing environment. It is currently running in FranzLisp and ObjTalk [Rathke 86] on VAX systems using powerful terminals and is not restricted to high performance personal computers. This environment provided both the necessity and the opportunity for implementing a distributed architecture between terminal and main frame.

From a designer's viewpoint, WLISP provides a visually based, interactive programming environment [Barstow, Shrobe, Sandewall 84]. WLISP has been an everyday operational environment at a number of institutions within universities and research organizations for several years. WLISP has served the designers of user interface software as well as the end-users.

Figure 1-1 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

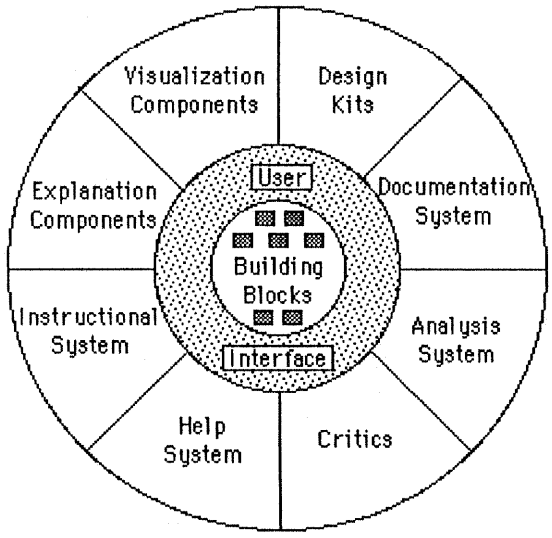


Figure 1-1: Architecture of a design environment

*The system was named WLISP because its development began 8 years ago with the goal of providing a window-based programming environment for LISP.

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 should be used to provide effective communication. We have built prototypical systems in many areas of the outer circle: documentation systems [Fischer, Schneider 84], help systems [Fischer, Lemke, Schwab 85], critics [Fischer 87], and visualization tools [Boecker, Fischer, Nieper 86].

In the following sections we illustrate the redesign process with a case study, describe the WLISP system and a prototypical design support tool (TRIKIT) and evaluate the strengths and weaknesses of our approach. In the last section we briefly describe ideas for future developments.

2. From Design to Redesign

Many large software systems are built as in Figure 2-1-a; a monolithic system is completely implemented in a general purpose programming language. Although these systems are usually structured in some way, this structure is oriented only towards the original specification of the problem. To overcome the difficulty of redesign that has been experienced with these systems, the design of one (or more) intermediate levels of abstractions must be an integral part of the software process (Figure 2-1-b). This strategy allows for both easy *redesign* by modifying the original design (Figure 2-1-c) and *reuse* by recombining the intermediate abstractions to form a different system (Figure 2-1-d).

Software Engineering environments of the future have to support design methodologies whose main activity is not the generation of new, independent programs, but the integration, modification, and explanation of existing ones [Winograd 79]. Just as one relies on already established theorems in a new mathematical proof, new systems should be built as much as possible using existing parts. In order to do so, the designer must understand the functioning of these parts. An important question concerns the level of understanding necessary for successful redesign: exactly how much does the user have to understand? Our methodologies (differential programming and programming by specialization [Kay 84] based on our object-oriented knowledge representation language ObjTalk; see Section 3.2) and support tools (e.g., the BROWSER; see Section 4.1) are steps in the direction of making it easier to modify an existing system than to create a new one. Inheritance is important for redesign because it enables objects that are almost like other objects to be created easily with a few incremental changes. Inheritance reduces the need to specify redundant information and simplifies updating and modification by allowing information to be entered and changed in one place.

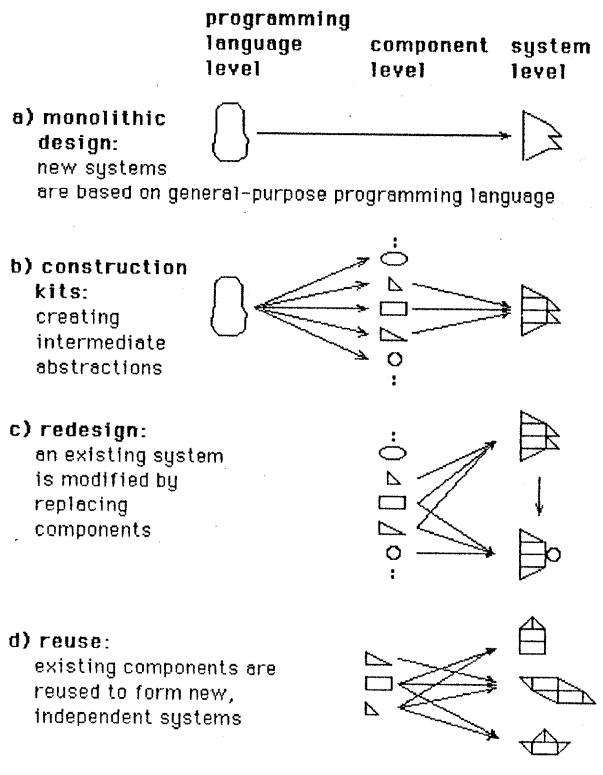


Figure 2-1: Reuse and redesign

A construction kit with a large number of generally useful building blocks provides a good basis for redesign. Simon [Simon 81] demonstrates that the evolution of a complex system proceeds much faster if stable intermediate parts exist. The abstractions, represented in WLISP by more than 200 ObjTalk classes, comprise stable intermediate parts for development of user interfaces. The large number of classes in WLISP is a mixed blessing. The advantage is that in all likelihood a building block or set of building blocks that either fits our needs or comes close to doing so already exists and has already been used and tested. The disadvantage is that they are useless unless the designer knows that they are available. Informal experiments [Fischer 87] indicate that the following problems prevent designers from successfully exploiting the potential of high functionality systems:

- designers do not know about the existence of needed objects (either building blocks or tools);
- designers do not know how to access objects;
- designers do not know when to use these objects;
- designers do not understand the results objects produce for them;
- designers cannot combine, adapt and modify an object for their specific needs.

Unless we are able to solve these problems, designers will constantly reinvent the wheel instead of taking advantage of already existing tools.

3. Description of WLISP

Our ideas, models and theories about the design of human-computer interfaces have become operational with the development of WLISP. The structure of the WLISP system defines a way of looking at and dealing with user interfaces. For users it provides a consistent world in which they can transfer interaction techniques among applications. For designers it provides a set of basic building blocks from which they can choose in designing a specific application.

3.1. A User's View

As a whole, WLISP can be described as a world with which the user interacts by screen manipulation. The system is *reactive* in the sense that each object on the screen exhibits a certain type of behavior. Actions that change the common world of user and system can be invoked either by the system (e.g., by updating some information about the state of an application) or by the user (e.g., by selecting some command from a menu). The designer using WLISP or the user of a specific application system deals with *screen objects* such as windows, icons, menus and buttons (see Figure 3-1), that are manipulated using the mouse as a pointing device. Many screen objects are independent of specific applications and serve as basic building blocks for different applications.

The user's view, which, depending on the task, is either that of a software engineer or an end-user is shown in a typical screen image of WLISP Figure 3-1:

- At the top of the screen some **status information** is shown. It is is updated continually by the underlying UNIX operating system. The status line displays time, load average, and number of users and tells about incoming and pending mail.
- A LISP **interpreter** is running in the **toplevel** window. Currently it displays the ObjTalk-Definition of a character object. On the right-hand side of the **toplevel** window a **static menu** is visible.
- Different systems can be activated by selecting the appropriate icon from the **Catalog**. The **Catalog** tells users which systems are available.

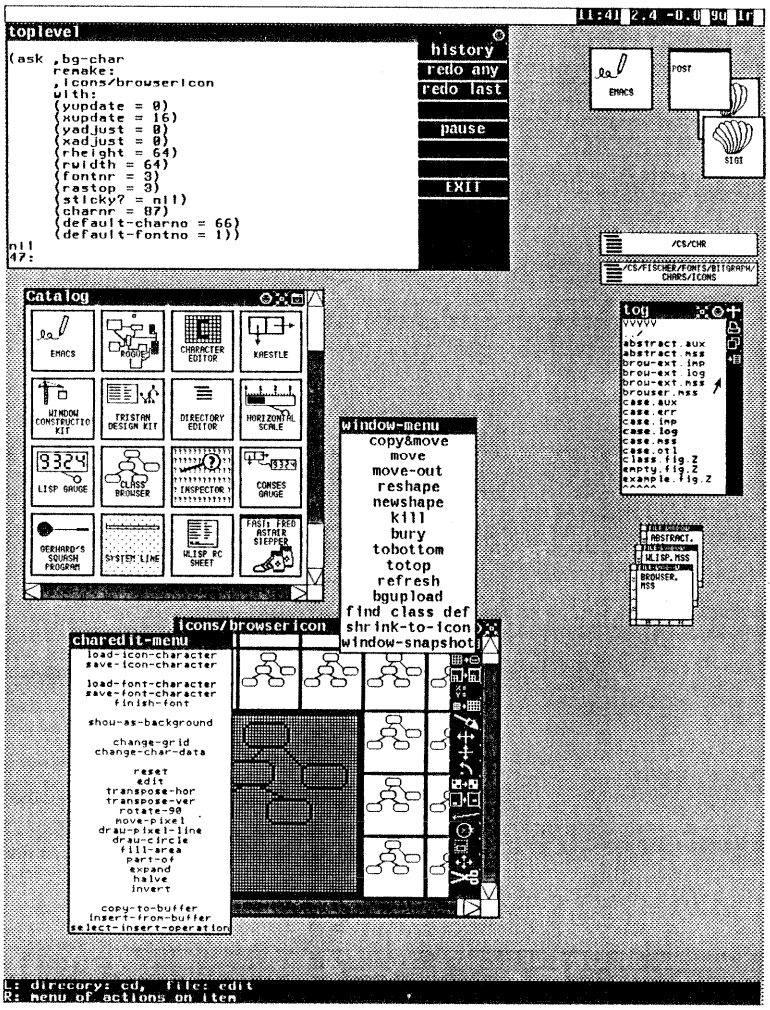


Figure 3-1: The user's view of WLISP

- At the bottom of the screen a **Character-Editor** window is shown that supports the definition of icons. All icons on the screen have been defined with the Character-Editor.
- A **window-menu** is associated with each window. It contains the basic operations on windows. Pressing a mouse button inside a window activates a pop-up menu with operations for the application itself. In the **Character-Editor** this menu displays operations to manipulate the pixel image of the character object.
- Some of these operations are also accessible through **buttons** on the right margin. Menus and buttons are alternative ways to perform the same operations.
- The **directory-editor** window gives access to the UNIX file system. It contains files of a certain directory.
- The **Catalog** and the **Character-Editor** windows have scroll bars on their right and lower sides. Windows in general show only a rectangular part of some larger region. Scrolling means to move this region below the window.
- Windows can be shrunk to icons when they are not needed. Some of them are shown on the right side. Icons are visual reminders of systems, functions and windows. They are organized in **clusters** that determines their spatial layout.
- In the **mouse-documentation-window** at the bottom of the screen information about mouse actions is displayed. As the user moves the mouse over windows, buttons or icons, this information is updated.

The association of applications with windows allows for an arbitrary number of applications to run at the same time. The user has direct access to each of them. WLISP provides a means for their integration.

3.2. A Designer's View

The components of WLISP are implemented in ObjTalk, an *object-oriented knowledge representation language* [Rathke 86]. Control is expressed in terms of a *message passing among objects* [Hewitt 77]. Objects *behave* based on the *interpretation of messages* and 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 hierarchies. They share the properties (i.e., methods and slots) of their parent classes. In this way properties are *inherited* by classes. All classes together 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.

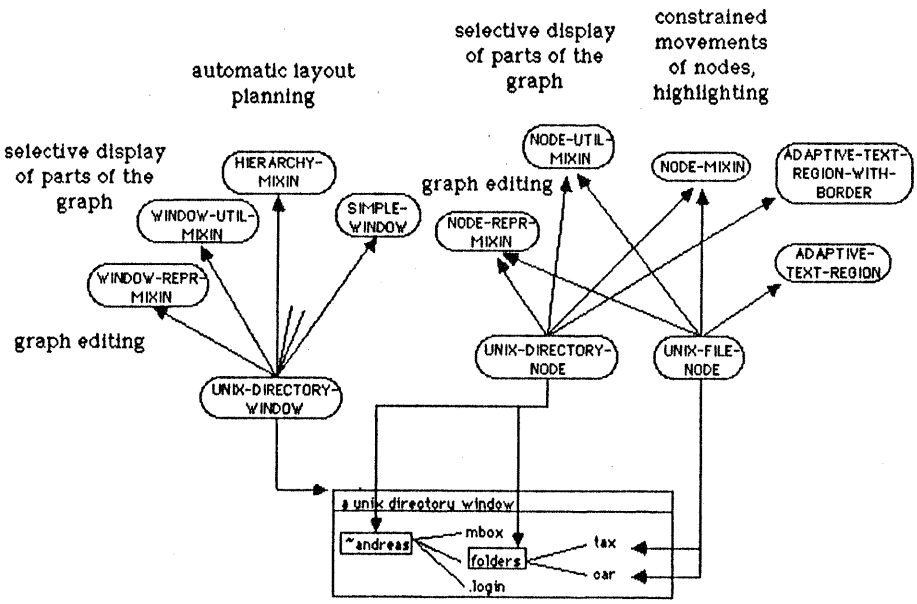


Figure 3-3: The TRISTAN classes

All components of WLISP are organized in inheritance hierarchies of classes. Classes describe screen objects such as windows or menus and components of screen objects such as borders, titles and buttons.

The design, development and use of WLISP have shown that the object-oriented architecture together with well established components and building blocks are extremely useful for the design of user interfaces. WLISP classes provide a set of abstractions on which the designer can rely. Existing systems like menus, icons, the directory editor, the character-editor, and the catalog serve as examples for the construction new user interfaces. The object-oriented architecture supports the reuse of components and building blocks.

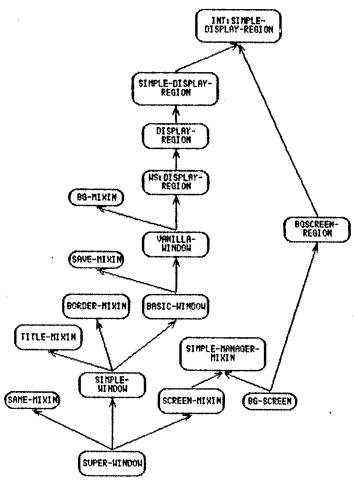


Figure 3-2: The basic window classes

Four categories of WLISP system classes can be identified:

1. **Basic system classes.** These classes describe rectangular areas on the screen such as regions, borders and titles. They are combined to construct output windows, text windows or dialogue windows. Superwindows act on subscreens, which can contain multiple subwindows. The inheritance hierarchy of the basic system classes is shown in Figure 3-2.

2. **Basic components of the user interface construction kit.** These classes form the application-independent parts of the user interface. Menus, icons, buttons, scroll bars, etc. inherit properties from the basic system classes. Parts of them can also be used as building blocks for application systems.

3. **Advanced components of the user interface construction kit.** TRISTAN [Nieper 85] is an example of a complex component providing high-level abstractions for displaying and editing graph structures through direct manipulation.

   Figure 3-3 shows the elements of TRISTAN as applied to a graphic UNIX directory editor. TRISTAN is independent of the particular node representation. It assumes only that the node representation is a subclass of the `simple-display-region` window class.

4. **Application systems.** Application systems use the components of the interface construction kit in different ways. Menus, icons, buttons or dialogue windows can be used directly within an application system. Other classes serve as a source from which properties are inherited. Applications usually define their own classes that incorporate the existing functionality by inheritance.

The reuse of WLISP system classes by means of inheritance and combination is supported by the object-oriented architecture:

- The creation of subclasses of existing classes allows the designer to create new objects that differ from existing objects in some desired aspects (e.g., different methods for some messages, additional slots) but that inherit almost all of the functionality of their ancestors.
- The use of predefined components is not an all-or-nothing decision. If the component has one undesired property, this component need not be completely abandoned. Even if the behavior of a superclass is to a very large extent undesired, it can still be used by shadowing all but the useful properties.

- Subclasses are not modified, independent copies of their superclasses. They benefit from any augmentations of their superclasses.
- Extensions can be done on different levels of the hierarchy, thereby affecting selected classes of objects.
- Like other object-oriented languages, ObjTalk supports dynamic modification of classes. Slots can be added without recompilation of other software. This not only makes rapid prototyping easier but also supports modifications by the end user.
- Each method can be a hook for modifying behavior. Methods can be augmented by adding procedures to be executed before, after, or instead of existing methods.

These architectural principles support new programming methodologies such as differential programming and programming by specialization and analogy, which is crucial to a redesign approach to system development.

4. A Case Study in Redesign

In this section we describe a prototypical task, the redesign of the interface to the ObjTalk Browser [Rathke 86] and how problems in redesigning this interface were solved by exploiting the architectural properties of WLISP.

4.1. The ObjTalk BROWSER

To accomplish our main goal, which is to facilitate the redesign of software and avoid building it from scratch, information retrieval tools for reusable software components like ObjTalk classes have become an absolute necessity. Browsers have become increasingly valuable for understanding and changing systems. They displace program listings because they present a program as a multidimensional structure being generated and filtered dynamically. In SMALLTALK [Goldberg 84], the browser is the main interface to the system and is used both for finding and analyzing existing pieces of software and for modifying and creating new software. It replaces the file system and the editor of conventional systems. The interaction style is one of moving and searching through an information space rather than directly accessing the space through names or descriptions.

The WLISP BROWSER displays the inheritance structure of a system. The interconnections between the system and the components it inherits from can be analyzed in more detail by selecting a component and looking at its slot descriptions, defaults, triggers, and methods (Figure 4-1).

The BROWSER exhibits some of the more advanced interface characteristics provided by WLISP. The main window is divided into several subwindows whose relationship is automatically maintained by WLISP and displayed in `paned windows`. Inside the ***Classes Window*** ObjTalk classes can be visualized by icons and are connected by arrows to show the superclass relationship. Selecting a class icon causes the other BROWSER windows to be filled with slots, methods, subclasses and instances of that class. This feature is implemented using the constraint mechanism of ObjTalk.

4.2. Description of the Redesign Process

The BROWSER had undergone several redesigns before it took its current shape. In an earlier development stage classes were represented as items of a scrollable menu (Figure 4-2). We describe the process of redesigning the menu-based BROWSER to show iconic representations for ObjTalk classes instead.

The Task. Instead of classes being visualized by strings in a scrollable menu, we would like to display classes as icons that are connected by arrows to show the superclass relationship. Everything else should stay unchanged, especially the dependencies between the selected class and the contents of the other windows.

The Redesign. The class of the window to be replaced (BROWSER-CLASSES-MENU) has two superclasses (Figure 4-3): BROWSER-SELECTION-MIXIN and CLASSES-MENU. BROWSER-SELECTION-MIXIN provides BROWSER-CLASSES-MENU with the application-dependent knowledge. It describes how the consequences of a class selection are propagated to the other BROWSER windows. CLASSES-MENU provides BROWSER-CLASSES-MENU with the knowledge for scrolling and menu selection. The redesign consists of

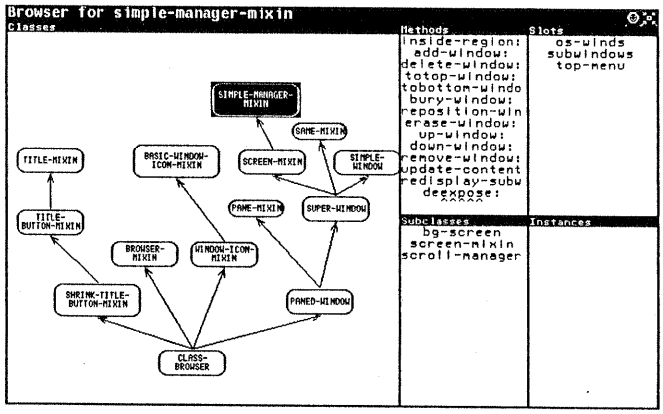


Figure 4-1: The BROWSER with windows to show the class hierarchy, methods, slots, instances and subclasses of a selected class

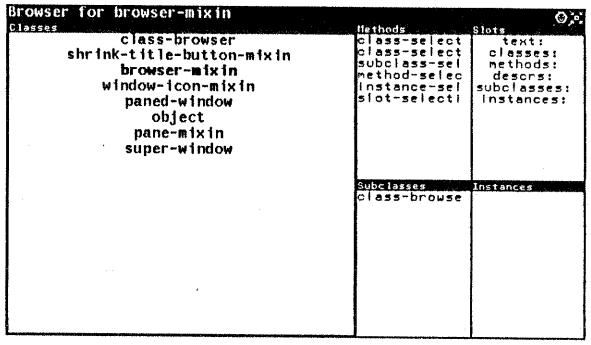


Figure 4-2: The original BROWSER

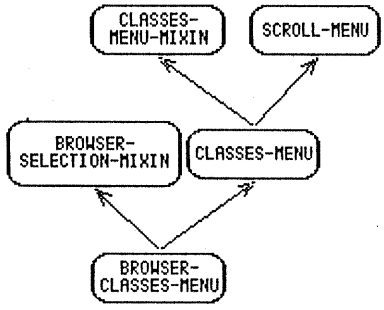


Figure 4-3: The inheritance hierarchy of BROWSER-CLASSES-MENU

---

replacing this class by a new one, thereby maintaining the important interactive properties of selecting, adding and deleting classes.

In order to find out about the desired properties of the new class we take a closer look at the definition of BROWSER-CLASSES-MENU (Figure 4-3):

SCROLL-MENU is combined with CLASSES-MENU-MIXIN to form the CLASSES-MENU class. Windows of this class are able to display classes as strings. BROWSER-CLASSES-MENU is constructed by mixing BROWSER-SELECTION-MIXIN in with CLASSES-MENU.

In BROWSER-SELECTION-MIXIN the dependencies between the selected class and the other BROWSER windows are represented: Whenever a class is selected, the contents of these windows will be updated to display methods, slots, subclasses and instances of the selected class. Because this functionality is not to be changed, we can use BROWSER-SELECTION-MIXIN without any modifications.

Instead of SCROLL-MENU we use SIMPLE-WINDOW as the basis for the new window. Simple windows provide the basic window capabilities of graphical windows in which icons can be placed and lines can be drawn. It sets up a coordinate system with its origin at the lower left corner. Scrolling the contents of a simple window is interpreted as moving the origin of the graphic coordinate system. CLASSES-MENU-MIXIN is replaced by a new mixin class (CLASS-NET-WINDOW-MIXIN) that specifies methods for selection, addition and deletion of classes. These operations are different from the ones in the preexisting implementation because they now have to display, add and delete icons and connect them by arrows (Figure 4-4).

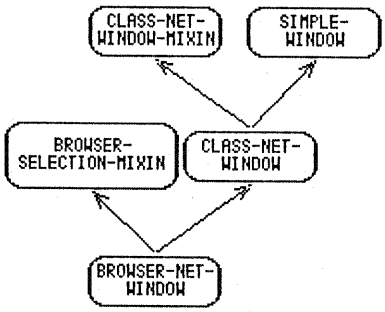


Figure 4-4: The inheritance hierarchy of BROWSER-NET-WINDOW

The implementation of the functionality for CLASS-NET-WINDOW-MIXIN would be a difficult task if there were no support for manipulating icons that contain text and that are connected by lines. Icons are an integrated part of the WLISP system. There are components that deal with text manipulation within icons (e.g., hyphenation and adjustment to the horizontal and vertical center), borders of various shapes and selection feedback. Also, the capability of connecting icons by arrows is supplied by a number of classes from WLISP's net package.

As the final step, the new class CLASS-NET-WINDOW-MIXIN is combined with SIMPLE-WINDOW to form CLASS-NET-WINDOW. Together with BROWSER-SELECTION-MIXIN the final BROWSER-NET-WINDOW is constructed (Figure 4-4).

5. Intelligent Support Tools for Reuse and Redesign

In addition to suitable languages and system architectures, support systems are necessary to enhance reuse and redesign processes. Rich computing environments contain at least hundreds of components that can be combined in many different ways. The existence of a component alone does not guarantee that it is readily available and that its usefulness is apparent. Therefore, support tools are needed that have knowledge about the structure of systems and about the use of existing facilities, that aid in making design decisions, carry out low level details, analyze or criticize intermediate versions, and visualize their structure. We will call these support tools *design kits*.

In this section we describe TRIKIT as an example of this type of systems (for a detailed description see [Fischer, Lemke 87]). TRIKIT supports design and redesign of network displays and editors. With its knowledge of this task domain it can aid the designer by automatically offering interesting design choices, selecting appropriate components, and combining them to make a functioning system. The designer needs much less knowledge and is able to produce higher quality results in shorter time.

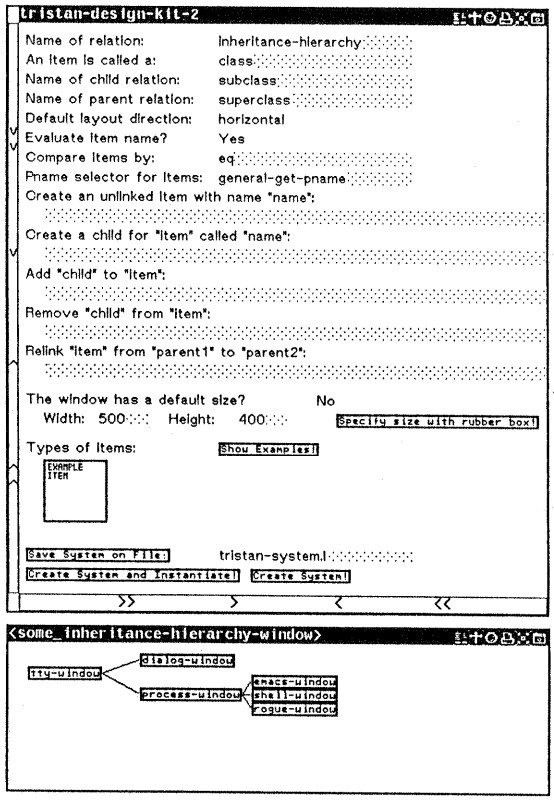


Figure 5-1: Initial state of the main form and an inheritance hierarchy window generated from it

The display and modification of hierarchical and network structures is a common problem in application systems including those that deal with structures of databases, directory trees, inheritance hierarchies, or dependency graphs are examples. TRISTAN, the user interface component described in Section 3.2, is a large set of object-oriented construction components that is applicable to many different types of graphical representations. We have used it to graphically display UNIX file system hierarchies, inheritance networks in object-oriented languages, and dependency relationships between rules in rule-based expert systems.

TRIKIT is a design kit for combining the TRISTAN components with a specific application. It presents itself to the user as a collection of interaction sheets as shown in Figures 5-1 (top window) and 5-2. On these interaction sheets the designer specifies the interface to the application. The designer specifies in terms of the application what it means to create and delete a node or to insert and remove a link. Here the designer chooses the desired graphical representation for the nodes of the graph and controls the creation of the user interface. This design process is carried out above the code level. TRIKIT hides from the designer program code that is generated when the network editor is created or modified.

Initially, the form is filled in with an example application - an ObjTalk inheritance hierarchy viewer; the window at the bottom of Figure 5-1 shows an instance. This allows users to familiarize themselves with TRIKIT and to modify parameters and explore their significance. The system supports redesign by providing designers with prototypical solutions and allowing them to take advantage of as much previous work and knowledge as possible.

Designers using TRIKIT need to know very little about the TRISTAN component. However, they do need to fill in code to access the application, for example, to retrieve the nodes of the graph, or to signal events that changed it. In addition to the specification of the application interface, many options of TRIKIT can be controlled. The representation of different types of nodes of the graph (different fonts, sizes, the use of pictorial representations) or

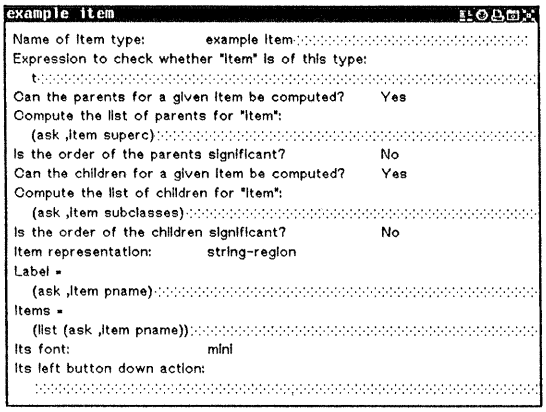


Figure 5-2: Initial state of the node form describing properties of individual nodes.

the layout direction (horizontal or vertical) can be adjusted to an individual application. Although the design space is limited by the available options in the forms, it is still possible to use this system to create a prototype to be further refined on a lower level (ObjTalk).

We have preferred a form-based specification over a specification language because form-based specification does not require the user to learn a language in order to be able to use the tool. An alternative that we will explore further is a direct manipulation interface [Hutchins, Hollan, Norman 86]. However, the abstract nature of many parameters (e.g., access functions for nodes, layout direction, distinguishing characteristics of node types) may make a direct manipulation interface infeasible.

6. The Evolutionary Development of Wlisp

The development of the WLISP system itself has been an incremental evolutionary process in which redesign has played a crucial role. It began many years ago with the implementation of a window system, and new interface components have been added incrementally. Ideas for further development and enhancement originated from the following sources:

- Systems constructed with an existing version of WLISP led to the development of new features that were at first built from scratch, quite often by different persons. Subsequently, the general usability of these features was recognized, and new, general components were built and integrated into WLISP.
- There was a strong interplay between the development of new applications and the recognition of the shortcoming of the available construction blocks and tools. For example, there was a long-felt need for superwindows and paned windows. Many applications resulted in special solutions until our understanding was good enough to develop them as general classes.
- Other developments of similar systems were carefully analyzed (e.g., the SMALLTalk system, the MacIntosh, the LISP machines, expert system shells like KEE, ART and LOOPS).

We believe that this evolutionary process has led to a set of abstractions, represented as classes in an inheritance network, that represent "tools of thought" for the designer and a "weak theory" of user interfaces. Using the current kit guarantees the development of reasonable interfaces with modest efforts and supports a redesign methodology.

Even today, the inheritance lattice of WLISP components is not static. Changes such as moving classes up in the lattice or isolating certain characteristics in separate mixins to increase the amount of shared information, reflect our growing understanding of the hierarchical structure and decomposition of a problem domain. The development of WLISP demonstrates that ill-structured problem domains, such as user interface and AI programming [Fischer, Schneider 84], require the coevolution of specification and implementation to achieve adequate and useful solutions [Swartout, Balzer 82].

7. Evaluation

Observing designers in dealing with complex software systems, one can see that they do not engage in redesign processes (even if they would like the system to behave differently) because redesign is not supported well enough. The effort to change a system or to explore design alternatives is too expensive in most software production environments. Our experience with our systems and tools makes us believe that if the cost of making changes is cheap enough, designers will start to experiment thereby gaining experience and insight leading to better designs. The existence of editors and formatting systems has shown that the willingness of writers to modify existing solutions increases.

Reuse and redesign processes in our environment are supported by WLISP and tools like the browser and TRIKIT. The reaction of designers using these tools has been largely positive, and a large number of complex systems with different user interfaces have been built (e.g., the NEWTON-Interface [Rathke 87]). The increased willingness to redesign has led to the construction of systems which fit an environment of needs. We feel very strongly that the specification of graphical, dynamic interfaces with a static, linear description language has severe limitations. Interfaces have a feel and an aesthetic quality that have to be experienced and that cannot be derived formally from specification languages. We were able to study empirically the consequences of specific design choices. Systems that make use of WLISP's abstractions are relatively easy to modify, maintain and adapt to changing needs. The object-oriented architecture provides a much greater flexibility and range of application than traditional subroutine libraries, which have failed as tools for redesign and reuse, because they are filled with specific implementations [Balzer, Cheatham, Green 83].

We are, however, aware of problems that remain to be solved. As described in Section 2, knowing about the existence of components is not trivial, especially as the number of available components is growing. In Section 4 it remains unclear how the components for building CLASS-NET-WINDOW-MIXIN have been determined. One can only search for something if one knows that something like it might exist.

If one has found a potentially useful component, one has to determine how it has to be used and combined with the other components. One has to understand its functionality and its properties. Design kits such as TRIKIT are an attempt at solving this problem. The problem of understanding, however, remains. The fields of TRIKIT's forms use a certain terminology, which not everybody is familiar with. The design space that is supported by TRIKIT must be extended to make it a more generally applicable tool.

We need more *active* tools or agents that have sufficient self-knowledge to offer their services. Our active help system [Fischer, Lemke, Schwab 85] and our critic [Fischer 87] are other steps in the direction of filling this need.

8. Conclusions

To cope with the ever-increasing complexity of designing, constructing, maintaining and enhancing software products, we have to replace discipline (by the software engineer) with better tools. A formalized, computer-assisted software paradigm must supersede the current informal, person-based software paradigm.

We believe that we have made an important step towards this goal. The classes of WLISP reflect our understanding of how to decompose the domain of modern user interfaces. WLISP contains many building blocks allowing existing software to a be reused and supporting design by redesign. Intelligent support tools guide the user in selecting among the numerous building blocks. Part of the knowledge about required or useful combinations about these components can be represented within WLISP itself. In further development stages of WLISP we will formalize the experts' knowledge about the use of components in ObjTalk *meta-classes*. Meta-classes allow the representation of structural and behavioral properties of classes. For instance, the ObjTalk classes that describe ***paned windows*** (Figure 4-1) are instances of a special meta-class WPANE that automatically supplies its instances with a required set of superclasses.

Design for Redesign is a promising methodology for keeping software "soft" and preventing it from becoming ossified and brittle with age. Modification within our methodology is not automated, but it is greatly facilitated. Redesign of software is similar to interior design of buildings,

where contexts and boundaries are given: in our case contexts and boundaries are provided by the components of the construction kit. Made with this approach, incremental improvements will be cheap enough that software engineers can experiment with alternative implementations, thereby gaining experience and insight leading to better designs.

Acknowledgments

The authors would like to thank their former colleagues on the project INFORM at the University of Stuttgart and their current colleagues and students in the research group "Knowledge-based Systems and Human-Computer Communication" at the University of Colorado, Boulder, who all contributed to making WLISP a usable, enjoyable and productive computational environment which allows us to explore new methodologies like redesigning and reusing software.

The research was supported by: grant No. N00014-85-K-0842 from the Office of Naval Research, grant No. DE-FG02-84ER1328 from the Department of Energy, the German Ministry for Research and Technology, and Triumph Adler Corporation, Nuernberg.

References

[Balzer, Cheatham, Green 83]
R. Balzer, T.E. Cheatham, C. Green, *Software Technology in the 1990's: Using a New Paradigm*, Computer, 1983.

[Barstow, Shrobe, Sandewall 84]
D.R. Barstow, H.E. Shrobe, E. Sandewall, *Interactive Programming Environments*, McGraw-Hill Book Company, New York, 1984.

[Boecker, Fabian, Lemke 85]
H.-D. Boecker, F. Fabian Jr., A.C. Lemke, *WLisp: A Window Based Programming Environment for FranzLisp*, Proceedings of the First Pan Pacific Computer Conference, The Australian Computer Society, Melbourne, Australia, September 1985, pp. 580-595.

[Boecker, Fischer, Nieper 86]
H.-D. Boecker, G. Fischer, H. Nieper, *The Enhancement of Understanding through Visual Representations*, Human Factors in Computing Systems, CHI'86 Conference Proceedings (Boston, MA), ACM, New York, April 1986, pp. 44-50.

[Fabian 86]
F. Fabian, *Fenster- und Menuesysteme in der MCK*, in G. Fischer, R. Gunzenhaeuser (eds.), *Methoden und Werkzeuge zur Gestaltung benutzergerechter Computersysteme*, Walter de Gruyter, Berlin - New York, Mensch-Computer-Kommunikation Vol. 1, 1986, pp. 101-119, ch. V.

[Fischer 87]
G. Fischer, *A Critic for LISP*, Technical Report, University of Colorado, Boulder, 1987.

[Fischer, Lemke 87]
G. Fischer, A.C. Lemke, *Design Kits: Steps Toward Human Problem-Domain Communication*, Paper submitted to 'Human-Computer Interaction - Special Issue on User Interfaces to Expert Systems', University of Colorado, Boulder, 1987.

[Fischer, Lemke, Schwab 85]
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.

[Fischer, Schneider 84]
G. Fischer, M. Schneider, *Knowledge-Based Communication Processes in Software Engineering*, Proceedings of the 7th International Conference on Software Engineering, Orlando, Florida, March 1984, pp. 358-368.

[Goldberg 84]
A. Goldberg, *Smalltalk-80, The Interactive Programming Environment*, Addison-Wesley Publishing Company, Reading, MA, 1984.

[Hewitt 77]
C. Hewitt, *Viewing Control Structures as Patterns of Passing Messages*, Artificial Intelligence Journal, Vol. 8, 1977, pp. 323-364.

[Hutchins, Hollan, Norman 86]
E.L. Hutchins, J.D. Hollan, D.A. Norman, *Direct Manipulation Interfaces*, in D.A. Norman, S.W. Draper (eds.), *User Centered System Design, New Perspectives on Human-Computer Interaction*, Lawrence Erlbaum Associates, Inc., Hillsdale, NJ, 1986, pp. 87-124, ch. 5.

[Kay 84]
A. Kay, *Computer Software*, Scientific American, Vol. 251, No. 3, September 1984, pp. 52-59.

[Nieper 85]
H. Nieper, *TRISTAN: A Generic Display and Editing System for Hierarchical Structures*, Technical Report, Department of Computer Science, University of Colorado, Boulder, 1985.

[Rathke 86]
C. Rathke, *ObjTalk: Repraesentation von Wissen in einer objektorientierten Sprache*, PhD Dissertation, Universitaet Stuttgart, Fakultaet fuer Mathematik und Informatik, 1986.

[Rathke 87]
C. Rathke, *Human-Computer Communication Meets Software Engineering*, Proceedings of the 9th International Conference on Software Engineering, IEEE, March 1987.

[Simon 81]
H.A. Simon, *The Sciences of the Artificial*, The MIT Press, Cambridge, MA, 1981.

[Swartout, Balzer 82]
W.R. Swartout, R. Balzer, *On the Inevitable Intertwining of Specification and Implementation*, Communications of the ACM, Vol. 25, No. 7, July 1982, pp. 438-439.

[Winograd 79]
T. Winograd, *Beyond Programming Languages*, Communications of the ACM, Vol. 22, No. 7, July 1979, pp. 391-401.