University of Colorado at Boulder

Department of Computer Science

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

An Object-oriented Construction and Tool Kit for Human-Computer Communication

GERHARD FISCHER
Department of Computer Science and Institute of Cognitive Science
University of Colorado, Boulder

In Computer Graphics, April 1987

Abstract

Over the last several years, we have developed an object-oriented, knowledge-based construction kit for human-computer communication (HCC) and a large number of associated tools and intelligent support systems needed to exploit this kit effectively. Answers to the "HCC design question" are given by providing appropriate building blocks which suggest the way HCC's should be built. The object-oriented system architecture is highly flexible and enhances the reusability of many building blocks. In designing new HCC capabilities the designer can use either existing objects or objects with minor modifications, and can thereby rely on standard and well-tested components.

Our support tools analyze HCC systems, provide assistance and guidance in building them, and create informative graphical displays of their structure. They help the designer regain control over systems that have become too complex to master without support tools. Our research efforts have given us an understanding of how the concepts and abstractions of our HCC toolkit have evolved and of what it means today to design new systems that make use of various types of graphical representations, icons, windows, and menus.

Acknowledgements

The author would like to thank his colleagues and students at the University of Stuttgart and at the University of Colorado, Boulder who have developed the tools and systems on which the contents of this paper is based. Without their contributions, the described research effort would not have been possible.

Table of Contents

List of Figures

An Object-oriented Construction and Tool Kit for Human-Computer Communication

Gerhard Fischer
Department of Computer Science and Institute of Cognitive Science
University of Colorado, Boulder, Colorado 80309

Abstract: Over the last several years, we have developed an object-oriented, knowledge-based construction kit for human-computer communication (HCC) and a large number of associated tools and intelligent support systems needed to exploit this kit effectively. Answers to the "HCC design question" are given by providing appropriate building blocks which suggest the way HCC's should be built. The object-oriented system architecture is highly flexible and enhances the reusability of many building blocks. In designing new HCC capabilities the designer can use either existing objects or objects with minor modifications, and can thereby rely on standard and well-tested components.

Our support tools analyze HCC systems, provide assistance and guidance in building them, and create informative graphical displays of their structure. They help the designer regain control over systems that have become too complex to master without support tools. Our research efforts have given us an understanding of how the concepts and abstractions of our HCC toolkit have evolved and of what it means today to design new systems that make use of various types of graphical representations, icons, windows, and menus.

1. Introduction

The user interface is the external representation of the capabilities of a system. The user *feels* the computer through the user interface, which is the software that mediates between the user and the programs that shape the computer into a tool for a specific goal, whether the goal is to compose music, design a computer game, do scientific computations, or write an article. The user interface was traditionally the last part of the development process; now it should be the first. It is more than just an additional component; it is an integral and important part of the whole system. The traditional design process, proceeding from the "inside" to the "outside," should be reversed whenever possible. The design, development, and evaluation of new information systems -- indeed of any new technology -- should start with an understanding of the overall social and technical environment in which they are embedded.

The user interface is crucial for novices and professionals alike because what it presents to their senses *is* the computer. "The *user illusion* is the simplified myth (the mental model) everyone builds to explain (and makes guesses about) the system's actions and what should be done next" [Kay 84].

2. Requirements for the Design of User Interfaces

The challenge for the designers of user interface software is in meeting a large number of requirements. Designers must

- take advantage of modern hardware and basic software capabilities (e.g., use screens as a two-dimensional world that can be edited; use windows, menus and mouse interaction). *All* software should be reconceptualized to take advantage of these interface possibilities;
- mirror the abstractions of the application domain to reduce the transformation distance between the domain expert's description of the task and its representation as a computer program;
- support active exploration by allowing reversal of operations and recovery from errors;
- help break the complexity barrier by supporting dynamic unfolding and user- and task-specific filters;
- help break the utility barrier, defined as the ratio of value to effort expended, either by increasing the value of systems or by decreasing the effort necessary to learn and use them;
- support the development of a mental model by building in consistency and predictability;
- not only provide extensive error recovery mechanisms but also organize systems in such a way that the user cannot make errors in the first place (e.g., list only those menu items that are applicable);
- give control to users when they want or need it and do things automatically when users do not want to be bothered with doing things themselves.

The design space of modern user interfaces is enormous, and designing a complex user interface often takes years (designing the Xerox Star, for example, took more than six years). To make this task manageable, we have over the last six years designed, implemented, and enhanced a construction kit called WLISP [Boecker, Fabian, Lemke 85; Fabian 86]. WLISP is based on an object-oriented architecture and contains a large amount of knowledge about the design of user interfaces. The WLISP system 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.

Most other systems which support windows, menus, icons and mouse interaction are based on high performance personal computers. WLISP is implemented in a time sharing environment using powerful terminals (currently running on VAX systems, using FranzLisp and our object-oriented knowledge representation language OBJTALK [Rathke 86]). This environment provided a necessity and a chance to implement a distributed architecture between terminals and main frame. WLISP has recently also been made available on Symbolics LISP machines.

3. A Knowledge-Based Architecture for Human-Computer Communication

Communication between humans and computers is more than what can be seen on the screen. Therefore we argue that the term *user interface* should be replaced by *human-computer communication* to convey our real objective. Knowledge-based systems are the most promising approach to improving human-computer communication. When communication is based on

shared knowledge structures, it is no longer necessary to explicitly exchange all information between the user and the system.

In our work, all knowledge-based facilities have elaborate, interactive graphic interfaces, which provide specialized viewers and specialized information management systems to support their use. A good user interface is vital to a knowledge-based system; but interfaces whose sophisticated graphic facilities lack rich, supporting information structures are of little use. In our approach, human-computer communication is analogous to human-to-human communication: a person who knows a lot but cannot transmit this knowledge to others is as useless as a teacher, advisor, or knowledge worker who can communicate well but has nothing to say.

User Interface Management Systems [Olsen et al. 84] provide graphic primitives and tools to specify dialogue structures. A user interface management system architecture requires limited information exchange and strong separation between user interface and application system -- a reasonable approach for some problems. Based on our architecture and on the kinds of problems we try to solve -- for example, building intelligent support systems such as help, documentation and explanation systems -- we claim that a strong separation between interface and application is limiting because the user interface has to have extensive access to the state and actions of the application system. We regard and model a computational system as a collection of communicating objects or agents, each of them having an internal state and an external view; and we provide mechanisms, such as the constraints in OBJTALK, to maintain their consistency.

4. The Object-Oriented Architecture of WLISP

The design, development, and use of WLISP has shown that object-oriented architectures are extremely useful for user interfaces. The user communicates with the system through a world, represented on the display screen, that is composed of *active objects.* Each screen object has its visual representation, a state that defines its appearance on the screen and its relations to other screen objects; and it behaves according to its functional role. The advantages of object-oriented architectures are recognized in a variety of systems. Many successful user interface systems, including SMALLTALK, the STAR system, and the LISP machines, have object-oriented architectures. In addition to the representational aspects of user interfaces they provide substantial support in user interface design:

- 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 a component has one undesired property, the designer is not forced to abandon it completely. Even if most of the behavior of a superclass is undesired, the designer can still use it by overwriting all but the useful properties.
- Subclasses are not independent copies of their superclasses. They benefit from any augmentations of their superclasses.
- Extensions can be made on different levels of the hierarchy, thereby affecting selected classes of objects.
- 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* like differential programming, programming by specialization, and analogy, which are crucial to a reusability and redesign approach to system development. The OBJTALK-BROWSER, implemented in WLISP (see Figure 4-1), is an important tool for supporting these programming methodologies.

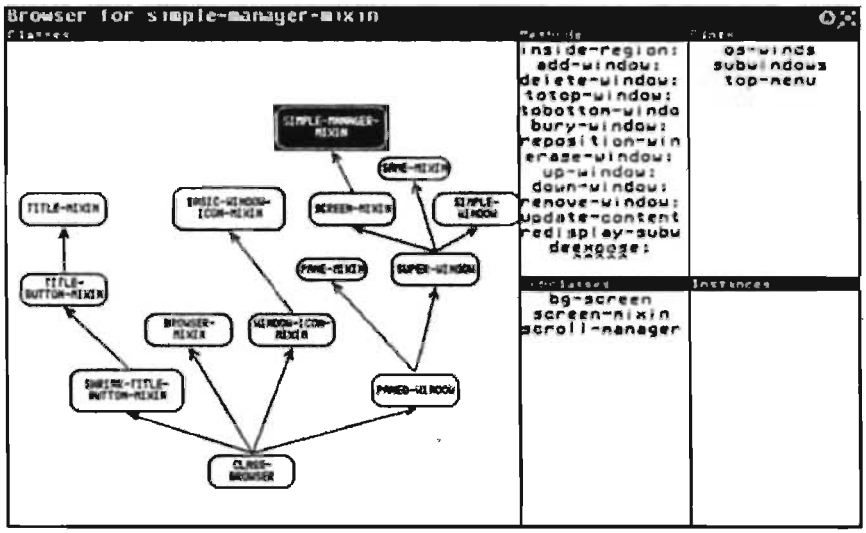

Figure 4-1: The OBJTALK-BROWSER with windows to show the class hierarchy, methods, slots, instances, and subclasses of a selected class

5. Design by Reuse and Redesign

WLISP is a good example of the class of systems that evolve to fit an environment of needs, rather than carrying out a single, well-specified task. In these systems, the main activity of programming is no longer the origination of new programs but the redesign of existing ones and the reuse of building blocks. To make this transition, the designer must thoroughly understand how these parts function. The level of understanding necessary for successful redesign and reuse is an important question: Exactly how much does the user have to understand about existing objects? Our new programming methodologies and support tools (e.g., the BROWSER; see Figure 4-1) are steps toward making it easier to modify an existing system than to create a new one. Inheritance is important for redesign and reuse, because it enables the easy creation of objects that are almost like other objects with a few incremental changes. Inheritance reduces the need to specify redundant information and simplifies updating and modification because information can be entered and changed in one place.

A construction kit with a large number of generally useful building blocks provides a good basis

for redesign. WLISP consists of more than 200 OBJTALK classes. This number is both an advantage and a disadvantage for a system of this kind. The advantage is that in all probability, an existing building block or set of building blocks -- which have been used and tested before -- either fit the users' needs directly or come close to doing so. The disadvantage is that the user may never discover their existence.

Informal experiments [Fischer 86] indicate that the following problems prevent users from successfully exploiting the potential of high-functionality systems:

- Users do not know about the existence of objects such as a building block or tool;
- Users do not know how to access objects;
- Users do not know when to use these objects;
- Users do not understand the results that objects produce for them;
- Users cannot combine, adapt, and modify an object to their specific needs.

Unless we solve these problems, users will constantly reinvent the wheel instead of taking advantage of already existing tools.

6. Intelligent Support Tools for Wlisp

Support tools are needed to augment a construction kit. They should provide guidance in building user interfaces, and they should analyze or criticize them and visualize their structure. These tools are systems that incorporate knowledge about user interfaces that goes beyond what went into the design of individual components.

Figure 6-1 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. 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 86], and visualization tools [Boecker, Fischer, Nieper 86].

Currently, the use of the WLISP user-interface components requires considerable expertise, which has to be acquired through an extended learning and experimentation period. We have constructed a number of *design kits* [Fischer, Lemke 86] to support the modification and construction of new systems from sets of predefined components. In contrast to simple software construction kits, which present the designer only with the available parts and the operations to put them together and to run the resulting system, our design kits give additional support. They incorporate knowledge about which components fit together and how they do so; and they contain a critic that recognizes errors and inefficient or useless structures. They are able to deal with multiple representations of the design.

Design kits constrain the design space and leave beginners with fewer choices by providing defaults and grouping the available functions. They reduce the amount of knowledge designers have to acquire before they can do useful work. This is especially important when the design environment contains many special-purpose components, each of which is used only rarely even by a full-time designer.

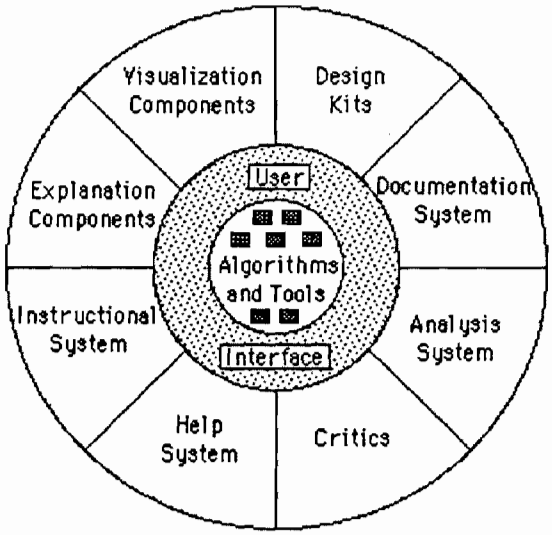

Figure 6-1: An Architecture for Intelligent Support Systems

7. Evaluation

No formal assessment has been made of the WLISP system, but user reaction has been largely positive, and a large number of complex systems using different user interfaces have been built with WLISP. Our system-building effort reinforced our basic assumption that in user interface design *"there are no optimal solutions, only trade-offs."* How do we balance the effort and the large training costs necessary to learn a complex interface with the power of having extensive functionality? Should we design the user interface completely or should we develop meta-systems that enable end-users to alter interfaces according to their needs? Should systems be *adaptive* that is, should the system itself change its behavior based on a model of the user and the task or should systems *be adaptable by the user*? Will systems that give control to the user be *less complicated* because they do not have to anticipate all possible futures? Who has control over the system? Does the system or the user make decisions about such things as the size of a window? In our experience, these trade-offs cannot be made in the abstract, but have to be carefully balanced for specific situations.

The **strengths** of WLISP are substantial. It is a powerful environment for rapid prototyping of a large class of interfaces. It provides a large class of building blocks that guarantee the construction of high-quality interfaces with relatively low construction costs. It contains many associated support tools that make it easier to learn and work with a complex system. The object-oriented architecture achieves uniformity, extensibility, and incremental development.

The **weaknesses** of WLISP are that it is a complex system and requires considerable learning time. The rich set of classes allows a user to select and specialize, but there is insufficient knowledge to support these processes. The question is: *How do we find what we do not know?*

The system needs more *active* tools or agents that have sufficient self-knowledge to offer their services to the user. And WLISP provides little support for transforming a *problem model*, which is the user's vague and imprecise conceptualization of the task, into the corresponding *system model*.

8. Extensions

Construction and tool kits are promising steps toward our goal of replacing human-computer communication with *human problem-domain communication*. Human problem-domain communication is an important step forward because users operate within the semantics of their domain of expertise and the formal descriptions closely match the structures of the problem domain. Whenever the user of a system can directly manipulate the concepts of an application, programs become more understandable and the distinction between programmers and non-programmers vanishes.

We will investigate the trade-off between *simple tools* and *intelligent tools*. Both kinds of tools have problems. Simple tools require too much skill and too much time and effort from the user; for example, it is still non-trivial to construct an interesting user interface with WLISP. Most intelligent tools fail to give any indication of how they operate and what they are doing; the user feels like an observer watching while unexplained operations take place. This mode of operation results in a lack of control over events, one of the most frustrating experiences of dealing with computer systems.

Traditionally, in most programs, interaction between the human and the program has been limited to an exchange of small pieces of information. We are concerned with a new class of computer systems that support *cooperative problem-solving processes* by providing advice, criticism, and explanation. In these systems, the boundaries between the user interface portion and the application system become much less clear than in traditional systems. WLISP is a good first approach to dealing with new systems for cooperative problem solving.

References

[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), 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 Kommunication Vol. 1, 1986, pp. 101-120, ch. V.

[Fischer 86]
G. Fischer, *Enhancing Incremental Learning Processes with Knowledge-Based Systems*, in H. Mandl, A. Lesgold (eds.), *Learning Issues for Intelligent Tutoring Systems*, Springer-Verlag, Berlin - Heidelberg - New York, 1986.

[Fischer, Lemke 86]
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, Inc., Hillsdale, NJ, 1986.

[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), 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.

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

[Olsen et al. 84]
D.R. Olsen Jr., W. Buxton, R. Ehrich, D.J. Kasik, J.R. Rhyne, J. Sibert, *A Context for User Interface Management*, IEEE Computer Graphics and Applications, December 1984, pp. 33-42.

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