Comparing the Programming Demands of Single-User and Multi-User Applications

John F. Patterson

Bellcore
445 South St.
Morristown, NJ 07960-1910

ABSTRACT

Synchronous multi-user applications are designed to support two or more simultaneous users. The RENDEZVOUS[TM1] system is an infrastructure for building such multi-user applications. Several multi-user applications, such as a tic-tac-toe game, a multi-user CardTable application, and a multi-user whiteboard have been or are being constructed with the RENDEZVOUS system.

We argue that there are at least three dimensions of programming complexity that are differentially affected by the programming of multi-user applications as compared to the programming of single-user applications. The first, concurrency, addresses the need to cope with parallel activities. The second dimension, abstraction, addresses the need to separate the user-interface from an underlying application abstraction. The third dimension, roles, addresses the need to differentially characterize users and customize the user-interface appropriately. Certainly, single-user applications often deal with these complexities; we argue that multi-user applications cannot avoid them.

[Keywords: User Interface Management System, Computer-Supported Cooperative Work, groupware, programming, dialogue separation, concurrency, roles]

1. INTRODUCTION

We envision a future in which computer workstations and terminals support simultaneous interaction among users who are distributed across a communication network. This is frequently referred to as synchronous computer conferencing and includes numerous earlier systems (Forsdick [1985], Lantz [1986], Stefik et al. [1987], Gust [1988], Ensor et al. [1988], Ellis, Gibbs, and Rein [1991], and Patterson [1990]). Many of these systems provide a content-independent form of sharing; they are not concerned with how the users plan to interact or about what they hope to communicate. A different type of synchronous computer conferencing structures the user interactions in a very specific way, e.g., a multi-user card game or a multi-user training simulation. We refer to these customized computer conferences as multi-user applications. Our goal is to make it easy to program multi-user applications.

Towards this end we have developed an architecture for creating multi-user applications, called the RENDEZVOUS system (Patterson et al. [1990]). This system provides both an architecture for starting multi-user applications and a programming infrastructure (Hill [1991]) that has been designed to support multiple users. The RENDEZVOUS system has been, or is being, used to implement several multi-user applications, including tic-tac-toe, a CardTable application, and a multi-user whiteboard.

Drawing on our experience with the RENDEZVOUS system and the CardTable, we offer some speculations about how multi-user programming is different from single-user programming. We start by outlining our implementation of the CardTable. This background establishes a foundation for our speculations and portrays our biases. Next, we analyze the ways in which single-user and multi-user programming are differentially affected by three dimensions of programming complexity: concurrency, abstraction, and roles. While not exhaustive, these dimensions are important and sufficiently general to warrant analysis. For each dimension, we consider two hypothetical applications, a single-user CardTable for playing solitaire and a multi-user CardTable for playing gin. We assume that these *Solitaire* and *Gin* applications offer the same interface as that implemented with RENDEZVOUS. This permits us to hold interface details constant and ask how the two applications are fundamentally different. Our goal is to discover general advice about the new challenges that multi-user applications will pose to user-interface technologists[2].

2. THE CARDTABLE

The CardTable is a multi-user application permitting several geographically dispersed users to simultaneously interact with a virtual deck of cards. It does not support any particular card game; the rules of the game must be provided by the users. Instead, it offers a simulation of the physical environment provided when several players

1. RENDEZVOUS is a trademark of Bellcore.

2. For a general review of the topic of multi-user applications and their implications for system support, the reader is referred to Ellis, Gibbs, and Rein [1991].

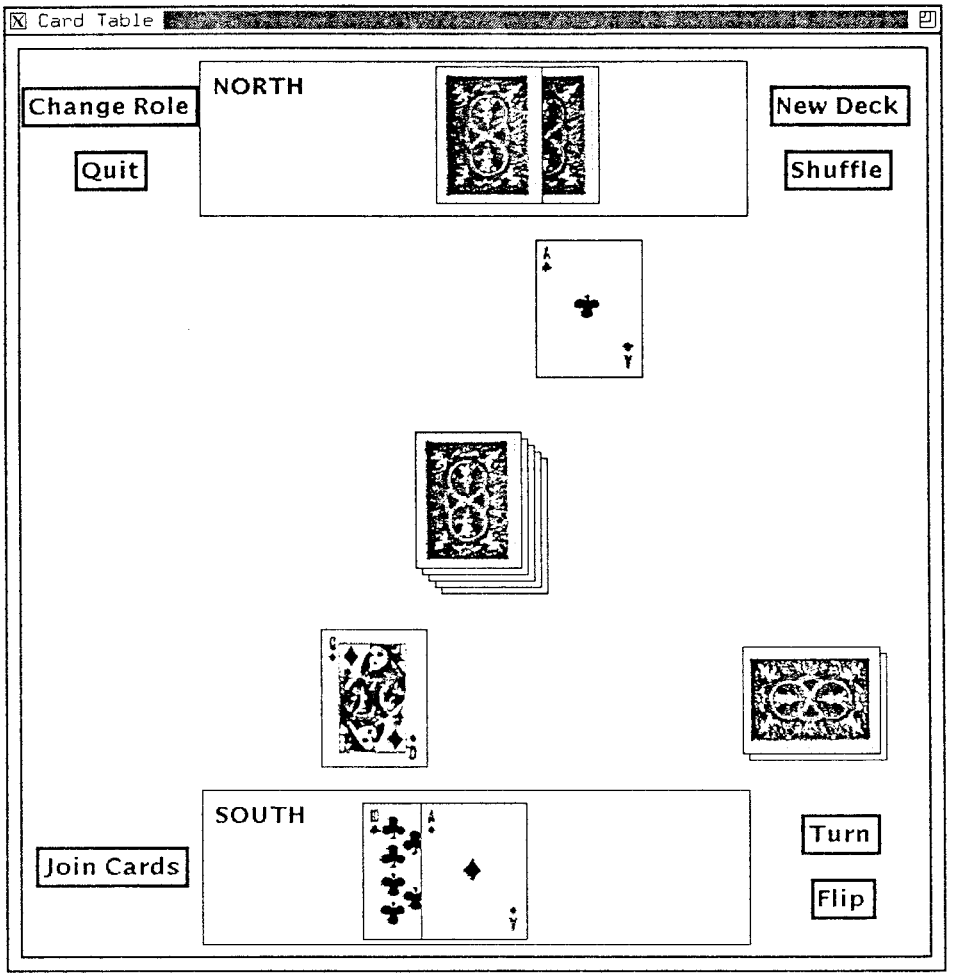

Figure 1. One User's View of the CardTable

sit around a table with a deck of cards.

2.1 CardTable Functionality

Users of the CardTable may assume one of four player roles or any of a variety of kibitzer roles. Figure 1 depicts the interface that one user of the CardTable might see. There are three major types of objects presented in this display: buttons, regions, and piles. The buttons are arranged at the corners of the layout and provide basic control over the application. For kibitzers, many of the buttons are suppressed to preclude the ability to manipulate the piles.

There are up to five regions on any user's display. The center region is the table. It is always present. The other regions are hands arranged around the four sides of the table region. If there is a player for a particular hand, then the hand is displayed. Otherwise, the hand is suppressed. The players have rotated views of the regions assuring that their own hand is always at the bottom of the screen. In figure 1 there are only two hands displayed, labeled as they might be in the game of Bridge, i.e., as compass points.

Piles are ordered sets of cards that may be either face up or face down. An individual card is a pile with only one element. Drop edges indicate how many cards are in the pile. Piles on the table are seen in the same way by all users, albeit rotated to the particular view. This is not true for piles in the hands. The face of these cards can only be seen by the player.

In the CardTable, almost all activities are actions on the piles, which may only occur when the pile is either on the table or in one's hand. Moving the cursor to a pile and pressing one mouse button permits the top card of the pile to be separated into a distinct pile. Pressing another button permits a pile to be dragged until the button is released. Piles are moved into or out of one's hand by simply dragging them from one region to another. It is not permitted, however, to move a pile in or out of someone else's hand.

Finally, the CardTable provides two buttons that are available to players and kibitzers alike. These are buttons to either quit or change one's role.

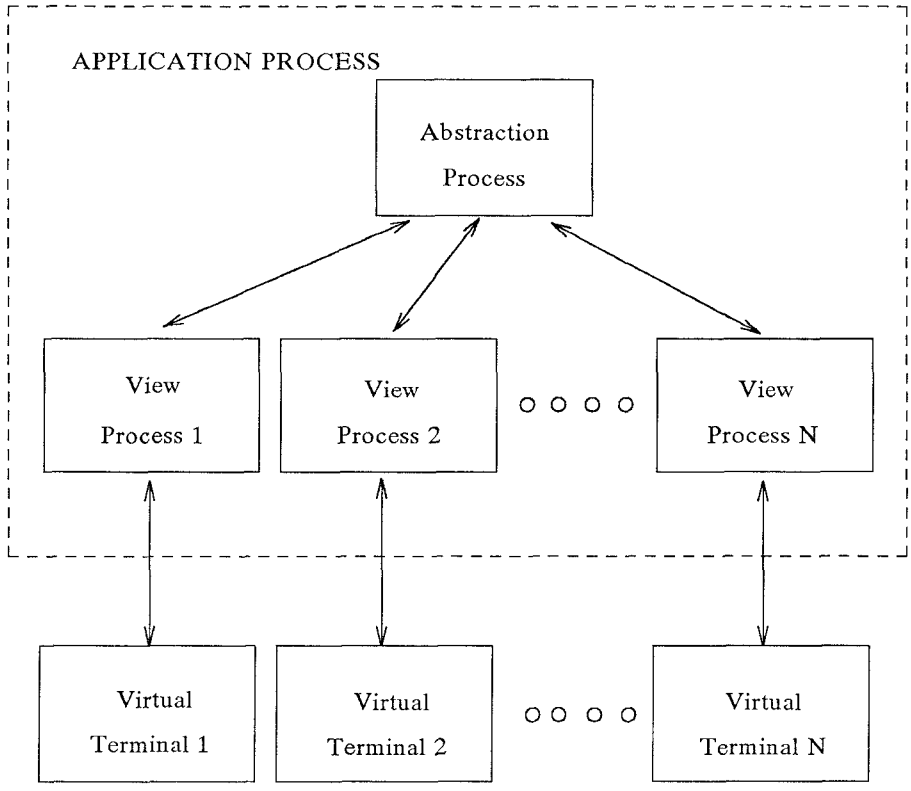

Figure 2. Generic Run-Time Architecture

2.2 CardTable Implementation

The basic design of any multi-user application in the RENDEZVOUS system is depicted in figure 2. Closest to the user is a virtual terminal[3]. This controls all direct input and output to the user and presents an abstract model of the terminal capabilities. We have elected to use the X Window®[4] System (Scheifler, Gettys, and Newman [1988]) as our virtual terminal.

The virtual terminals communicate with one centralized process that controls the application. This application process is decomposed into several light-weight processes. For each user there is a view process containing the view objects for that user. In addition, there is a single abstraction process containing the abstraction objects for the application. The abstraction objects model the application and generally facilitate the coordination among users. The view objects manage the display and interaction for their respective users without any direct concern for the presence of other users. (This distinction between views and abstractions is discussed further in section 4.)

Figure 3 elaborates on this theme by identifying some of the objects used in the CardTable. To simplify the presentation, figure 3 presents the situation for two users (North and South), three piles, and no buttons. The three boxes represent the three light-weight processes.

Ignoring, at first, the collection of arrows that tie together the three light-weight processes, the first thing to notice is that the processes contain identical hierarchies of objects. Although every object in one process has a counterpart in another process, this does not mean that the objects are identical. The abstraction's representation of an object contains no information about how to display an object; it contains only that state information that must somehow be shared between the multiple views. The view's representation of an object may lack certain fundamental state information, but must have any information that is needed to determine the display.

Consider the example of piles. The abstraction's version of a pile has an ordered list of cards, but has no information about the image associated with the top card. The view's version of the same pile contains information about the top card's image, but has no information about any cards other than the top card. The two objects, abstraction and view, are acting in the

3. We use the term virtual terminal to avoid implying any particular configuration concerning hardware. It is of little concern to us whether the user is running a PC-based X Window System, an X terminal, or an X server running on a workstation

4. X Window is a registered trademark of the Massachusetts Institute of Technology

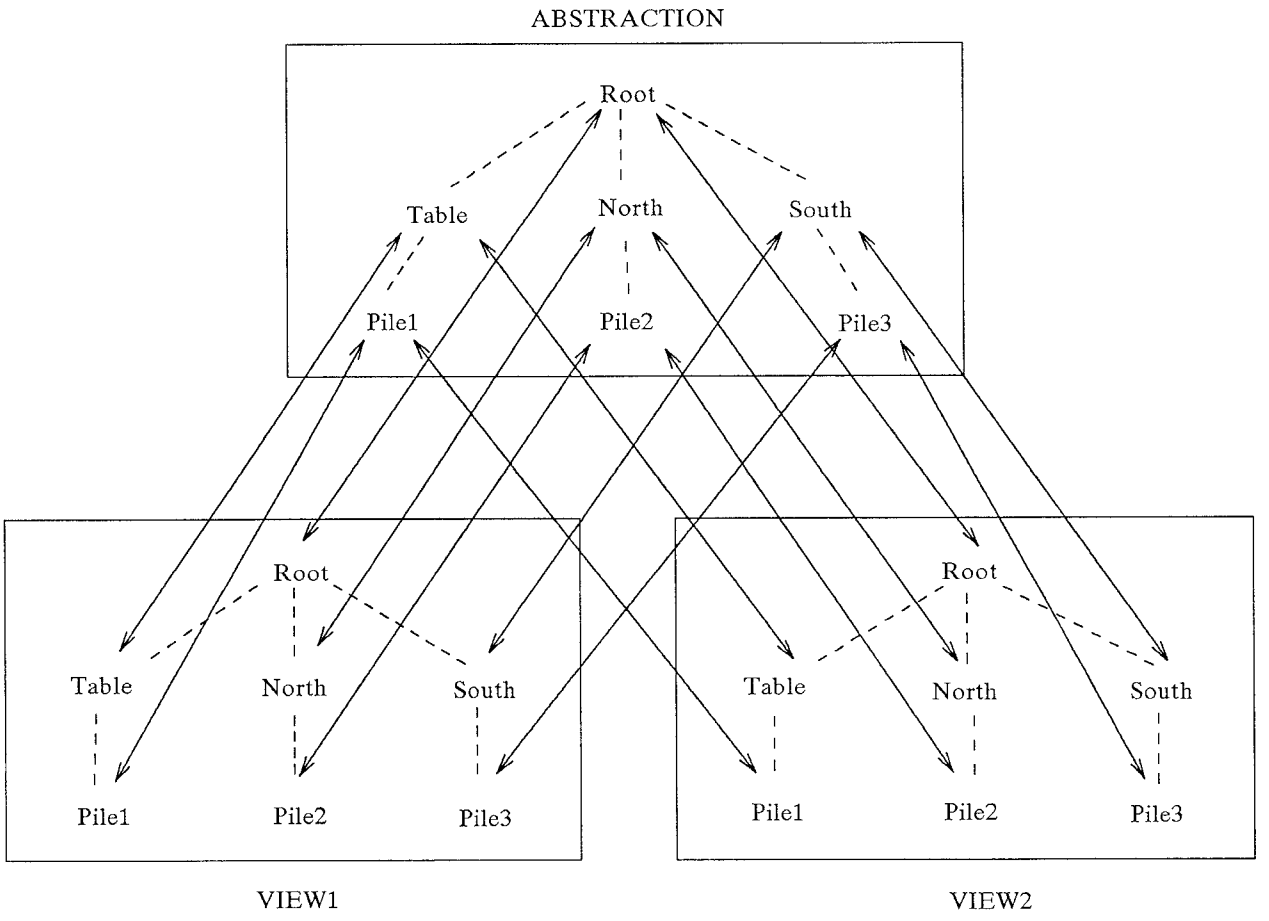

Figure 3. CardTable Implementation

service of one concept, but the task of representing the concept has been divided between them.

In the CardTable, the two view objects possess the same slots, but vary in their state settings[5]. For example, they will contain information indicating whether they are rotated, viewable, or permitting input. This is largely determined by the role of the user. (Roles are discussed more in section 5.)

Having considered the representation of state in the CardTable, we now turn to the issue of control. The RENDEZVOUS system provides two mechanisms by which activities may be expressed: rules and constraints. Rules are used to express procedural information; they associate a triggering event with an action in a fashion similar to a production system (Hayes-Roth, Watterman, and Lenat [1978]). In the CardTable, rules are only used for interaction with the outside. When a button is pushed or the mouse is moved, an event is delivered to the appropriate object and the event triggers one of the object's rules. Usually the consequence of activating the rule is to modify one of the slots of the object.

Constraints are used to express declarative information about relationships that must be maintained. One of the primary uses of constraints in the CardTable is to maintain the relationship between view objects and abstraction objects. This brings us back to the collection of arrows in figure 3. These arrows correspond to a special type of object called a Link. This object contains constraints expressing the relationship between abstraction and view objects, so that changes in one will yield the appropriate changes in the other. In a sense, it is the viewing transformation between the abstraction and the view.

Roughly speaking, rules provide a mechanism for activities outside the CardTable to cause changes in the view objects. Within a light-weight process, these rules are non-preemptive and must run to completion before another rule can fire. Between light-weight processes, the rules are preemptive, which assures some degree of fairness in the performance of the multiple views. (The importance of preemptive and nonpreemptive scheduling will be discussed further in section 3.) Constraints express relationships that must be maintained despite any

5 While the RENDEZVOUS system permits views of vastly different types, i.e., views with different slots, this was not necessary for the CardTable.

exogenous changes. In a sense, the constraints describe an equilibrium to which an application constantly returns.

3. CONCURRENCY

An essential aspect of user-interface management is the scheduling or control regime. This is the procedure by which input to the application is managed. We address the way that nonpreemptive and preemptive scheduling (Deitel [1984]) differentially affect single-user and multi-user programming[6].

Under nonpreemptive scheduling many different sources of input are simultaneously monitored, but whichever arrives first activates an action that procedes without preemption until control is released. There is much to recommend this technique for the Solitaire application. Time-sensitive activities, like dragging a card, are programmed as a sequence of very brief actions in response to mouse movements. This assures maximum responsivity to changes in the input. Lengthy computations, like cutting a pile — which requires the creation of a new object — may force the user to wait, but it is a delay of the user's making and therefore understandable to the user.

For the Gin application, nonpreemptive scheduling is less appealing. Delays in the responsiveness of one user's interface cause delays in the other user's interface as well. While one user is dragging a pile, the other might cut a pile. The user who is dragging a pile will suddenly experience a stutter or outright halt of the pile's movement. With two or more users it is no longer possible to be certain that these activities are not interleaved in an unanticipated manner.

Under preemptive scheduling many different sources of input are simultaneously monitored. When input arrives an action is activated and its execution can be interleaved with other actions. Functionally, activities must be treated as if they are simultaneously executing. In the Solitaire application, this would permit card dragging to occur while a new pile is being created. It is not clear why this capability is needed for the Solitaire application, but other single-user applications could benefit by the ability to interrupt lengthy activities or to launch simultaneous lengthy activities. For the Gin application, this preemption is necessary to ensure that the users obtain relatively independent responsiveness from the system. Each may slow the other slightly, but neither may grab the processing exclusively.

The acceptability of nonpreemptive scheduling in single-user applications is based on two design assumptions: a predictability assumption and a delay-tolerance assumption. The *predictability assumption* is that:

> Users schedule their activities in somewhat predictable patterns.

They do not usually try to push buttons and drag objects at the same time. They do not usually push two buttons at the same time. These patterns may be used by the interface designer to ensure that a nonpreemptive scheduling regime does not seem overly restrictive.

The *delay-tolerance assumption* is that:

> Users tolerate brief delays resulting from their own requests of the system.

Certainly, when the delays do not exceed about two seconds, users are fairly tolerant. (See Shneiderman [1984].)

The problem in multi-user applications is that while these assumptions remain largely valid for the input from any one user, they are largely invalid for the input from two or more users. The interleaved input of two users will not follow the predictable patterns of a single user. Also, users cannot be expected to tolerate even brief delays introduced by someone else. Indeed, delays introduced by others will be perceived as increases in the variance of the overall system response time, which is not desirable. (See Shneiderman [1984].) These observations establish a need for some form of preemptive scheduling for multi-user applications.

Full preemptive scheduling may be more capability than is absolutely required. The RENDEZVOUS system has adopted a compromise approach. The view objects for each user and the abstraction object for sharing between users are in separate light-weight processes. They can process events preemptively with respect to each other. Within a light-weight process, however, the various events are handled by a nonpreemptive regime. This means that the design assumptions of nonpreemptive interfaces are still employed for each user, but not applicable between users. So far, this has proven effective.

4. ABSTRACTION

In the RENDEZVOUS system, the term *abstraction* is used to mean an interface-independent representation of the underlying information associated with an application. Although our use of the terms abstraction and view may be somewhat idiosyncratic, the separation of underlying application information from interface information is hardly novel. Hartson and Hix [1989] call this *dialogue independence* and state that, "Almost all modern approaches to human-computer interface management are based to some extent on dialogue independence." (pg 13). How, then, are single-user and multi-user applications differentially affected by abstraction?

We offer two arguments concerning the impact of abstraction. First, although in single-user applications abstraction is often a nicety, in multi-user applications it

6. Hartson and Hix [1989] identify three types of control regimes: sequential dialogue, asynchronous dialogue, and concurrent dialogue. This corresponds to blocking I/O, nonpreemptive scheduling, and preemptive scheduling respectively. Sequential dialogue, or blocking I/O, is not discussed because it is clearly unacceptable to totally block one user while waiting for input from another.

is a necessity. Second, given that one concedes a need for an abstraction in both cases, multi-user applications may make abstractions easier to discover. Let's examine these arguments in the context of the hypothetical Solitaire and Gin applications.

4.1 Is an abstraction necessary in a multi-user application?

For an implementor of the Solitaire application, there is little in the application specification that compels the abstraction/view distinction. Indeed, it would not be surprising if the abstraction state were simply buried in the view objects. Since there is only one user, only one view need be supported; the separation of view and abstraction seems unnecessary and redundant. Admittedly, it is considered good programming practice to separate views and abstractions even in single-user applications, but this advice is often ignored given the application specification.

Three things foster an abstraction/view separation in single-user applications. The first is acceptance of the admonition that it is good practice. This may be enough for some, but it is thin support when the pressures for rapid production or improved performance run afoul of good design. The second is tools that support the separation. The Andrew Toolkit (Palay et al. [1988]), which builds in the distinction between view and data objects, is a good example of this. Finally, there is the need for multiple views within a single-user application. For example, one user might want multiple perspectives on a graphical scene; or, a programmer might want to ensure that users can customize their interfaces.

For an implementor of the Gin application, the situation is quite different. Though the interface to any one user is much the same as for Solitaire, there is a need for a mechanism to coordinate and communicate between the multiple views[7]. We believe that, as a first approximation, the abstraction or underlying application model contains exactly that state information that one might want to share between the multiple views of a multi-user application. By separating this information from any particular view and providing the mechanism for views to be kept consistent with this abstraction, the coordination of views is assured.

While one might concede that the abstraction is a sufficient representation for coordinating multiple views, is it necessary? We find it useful to consider how one might create a Gin application from two of the abstractionless Solitaire implementations. Given a mechanism for communicating between the two Solitaire view processes — either via shared memory or messaging — it is necessary to identify the critical state that must be kept consistent between the two views and construct a set of messages or memory assignments to establish and maintain the necessary consistency. If this were done with a shared memory system, then the collection of shared memory would constitute an abstraction. If we accept the view of Carriero and Gelernter [1989] that all shared memory systems can be mapped into message passing systems and vice versa, then even a message passing approach must, in some sense, identify the abstraction. It strikes us as considerably less painful to identify the critical information at the outset.

4.2 Are abstractions easier to identify in multi-user applications?

Let's return to the Solitaire and Gin applications and ask a somewhat different question. Suppose that we have decided, in either case, that the abstraction/view split is a good thing. Which abstraction will be easier to identify?

One of the frustrating aspects of the advice to separate the abstraction from the view in single-user applications is that there is so little in the application specification that explains how this split should be made. When programming Solitaire, for example, should the suits and ranks of cards be in the abstraction? Aren't they simply used to identify the card image to display? Why should this information be separated out from the view? In the absence of a multiple purpose to which a piece of information will be placed, it is hard to determine that an abstraction is needed.

Of course, the reason that abstraction is desirable is that there will eventually be multiple purposes. The first specification of an interface is never quite right and it is desirable to ensure that an abstraction will support the inevitable modifications that will be demanded. Unfortunately, an understanding of these sorts of multiple purposes requires prescience concerning the inadequacies of the first interface. The usual substitutes for such prescience are experience in the construction of user interfaces and an adequate understanding of the application domain.

It may seem paradoxical, but the Gin application may yield an abstraction more easily than the Solitaire application. Are ranks and suits in the abstraction? If the information must be available to both views, then the answer is yes. Is the position of a pile on the table part of the abstraction? If both users must see the pile in the same relative position, albeit rotated, then positional information must be part of the abstraction. The multi-user application provides a built-in multiple purpose for some state information and, at least as a first approximation, this is the information that must be in the abstraction. Experience and knowledge of the application domain will still be invaluable, but the need for multiple views acts as its own goad to developing an abstraction.

7. Some multi-user applications require views that are literally identical, e.g., as in window sharing (Lantz [1986], Ensor et al. [1988], and Patterson [1990]). These applications may not require an abstraction as much as a mechanism for copying output. In general, however, multi-user applications will require the flexibility demonstrated in the CardTable to present different views to different users and to treat some display information, e.g., highlighting, as local to a view and therefore unshared

In summary, the separation of an abstraction from a view has a complicated differential impact on single-user as opposed to multi-user programming. On the one hand, single-user applications may often ignore the need for a separation thereby decreasing their complexity and speeding their development. Multi-user applications do not enjoy this luxury. For them, abstraction is a necessity. On the other hand, it may be easier to discover the abstraction with multi-user applications because the application specification provides a first approximation to the type of information that must be abstracted.

5. ROLES

Think of a user as an object in the object-oriented programming sense. A user-interface is simply an elaborate mechanism for moving messages back and forth between the user object and other objects within the application — especially abstraction objects. A *role* is an abstract user type. It is an object class of which a user is an instance. Admittedly, a user may assume many roles, but we will treat this as a form of multiple inheritance. Given the roles of the user, the messages understood by the user are known.

In addition to characterizing the messages understood by the user, roles also characterize a cluster of capabilities given by the application to the user. We think of these capabilities as similar to those described by Mullender and Tanenbaum [1986] in the following:

> "Associated with each object are one or more 'capabilities' which are used to control access to the object, both in terms of *who* may use the object and what operations *he* may perform on it." (pg. 289-290, emphasis added)

Although this quote implies that capabilities are assigned to people, they are actually assigned to objects that act on behalf of a user. The capability is represented by a data structure which supports encryption and acts as the key to unlock an object's operations. We are not especially concerned with the security issues, but do wish to control access, albeit in a form that is meaningful to the user. Rather than require a user to remember and understand a large number of individual capabilities, roles cluster the capabilities into meaningful sets for the particular application. It becomes the job of the user interface to manage the set of access privileges implied by the user's role.

Let's turn now to considering roles in the context of our hypothetical Solitaire and Gin applications. For Solitaire it is difficult to understand why roles are needed. For the most part the user is simply *the user;* nothing else needs to be known about him/her. Everything that is possible with the interface is made available at all times; there is no need for access control.

The closest that most single-user interfaces come to needing roles is when a user description is used to modify an interface. For example, user preferences might be thought of as assigning the user to different class types thereby characterizing the messages that can be sent to the user. Typically, however, this merely tailors the output and lacks any concept of access control. A more compelling example arises when a single-user interface supports different skill levels. By characterizing the user as a novice or an expert, the interface may provide very different interfaces to limit novice access to certain capabilities.

In the Gin application roles are essential. Two users must be identified as players and not the same player. With this information the interface can provide open access to objects on the table and disallow access to objects in the other players' hands. A kibitzer role can be added to permit users the ability to watch without interfering. Also, since roles are meaningful to users in the context of the application, the roles assigned to the participating users can be displayed as status information and made available to be changed.

In the RENDEZVOUS system, there is not yet any explicit mechanism for handling roles. The CardTable treats a user's role as an attribute of the view process. The role is used to enable and disable an object's visibility and to act as a filter on the input events. We will need much more experience with multi-user applications before we could advocate a less application-specific approach.

6. CONCLUDING COMMENTS

We have identified three dimensions of programming complexity: concurrency, abstraction, and roles, and argued that while single-user applications may often ignore these dimensions, multi-user applications cannot. We conclude by asking what support will be needed for developing multi-user applications?

Multi-user applications need some support for concurrency. With several users, interface designers can no longer rely on the predictability of user input sequences and the tolerance of user-caused delays as design assumptions permitting nonpreemptive scheduling. Perhaps, as in the RENDEZVOUS system, it will be possible to support preemption between views and maintain nonpreemptive scheduling within views. Some level of concurrency will be needed, however, and it remains to be seen how well these half-measures will serve.

Multi-user applications also need support for the construction of abstractions. This is hardly a novel request in the field of User Interface Management Systems and many systems are designed to provide this facility. (See Hartson and Hix [1989] for review.) The only ones of these that we should reject are the ones that Hartson and Hix [1989] call dialogue-dominant. In these systems the abstraction is invoked from the view. This is a problematic approach when one view may alter the abstraction thereby requiring that the abstraction notify all other views. In the RENDEZVOUS system, the abstraction and view are peers.

Finally, most multi-user applications require an ability to describe users in terms of roles. We are not yet certain how roles should be supported. Should access be controlled in the view or the abstraction? Is there such a thing as an application-independent role and, if so, how might it best be encapsulated? Roles seem to have a global impact throughout an application's view, how should this be accomplished? At present, we know that there must be mechanisms to enable and disable the input and output for a user, but we are uncertain how to best map roles into these controls.

If multi-user applications are to flourish in the future, then programmers will require support for building these applications. Much of what is needed is not new. Capabilities like concurrent control, separation of abstraction and view, and customization based on user classification are available but may be avoided or ignored when programming a single-user application. As we move into an era of multi-user applications we must abandon our timidity and embrace these more advanced programming techniques despite the increased complexity they entail.

7. ACKNOWLEDGEMENTS

Progress on the CardTable would have been impossible without the efforts of Ralph Hill and Steve Rohall to implement the RENDEZVOUS system's run-time infrastructure. In addition, this paper has benefitted greatly from the careful reviews of Ralph Hill, Louis Gomez, Laurence Brothers, and Nathaniel Borenstein, Finally, I would like to acknowledge the intellectual support of the other members of the RENDEZVOUS team: Debbie Bloom, Tom Brinck, and Wayne Wilner.

8. REFERENCES

Nicholas Carriero and David Gelernter, How to Write Parallel Programs: A Guide to the Perplexed, *ACM Computing Surveys 21*,3 (September 1989), 323-357.

Harvey M. Deitel, in *An Introduction to Operating Systems*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1984.

C. A. Ellis, S. J. Gibbs, and G. L. Rein, Groupware: Some Issues and Experiences, *Communications of the ACM 34*,1 (January, 1991), 38-58.

J. R. Ensor, S. R. Ahuja, D. N. Horn, and S. E. Lucco, The Rapport Multimedia Conferencing System - A Software Overview, *Proceedings of the IEEE Conference on Computer Workstations*, Santa Clara, California, March 7-10, 1988, 52-58.

H. Forsdick, Exploration into Real-time Multimedia Conferencing, *Proceedings of the Second International Symposium on Computer Message Systems*, September 1985, 299-315.

P. Gust, Shared X: X in a Distributed Group Work Environment, Unpublished paper presented at the Second Annual X Technical Conference, January, 1988.

H. Rex Hartson and Deborah Hix, Human-Computer Interface Development: concepts and Systems for Its Management, *ACM Computing Surveys 21*,1 (March 1989), 5-92.

F. Hayes-Roth, D. A. Watterman, and D. B. Lenat, Principles of Pattern-Directed Inference Systems, in *Pattern-Directed Inference Systems*, Academic Press, New York, 1978, 577-601.

R. D. Hill, A 2-D Graphics System for Multi-User Interactive Graphics Based on Objects and Constraints, in *Advances in Object Oriented Graphics*, E. Blake and P. Wisskirchen (editors), Springer-Verlag, Berlin, 1991.

K. A. Lantz, An Experiment in Integrated Multimedia Conferencing, *Proceedings of CSCW '86 Conference on Computer-supported Cooperative Work*, Austin, TX, 1986, 267-275.

S. J. Mullender and A. S. Tanenbaum, The Design of a Capability-Based Distributed Operating System, *The Computer Journal 29*,4 (August 1986), 289-299.

A. J. Palay, W. J. Hanson, M. Sherman, M. G. Wadlow, T. P. Neuendorffer, Z. Stern, M. Bader, and T. Peters, The Andrew Toolkit - An Overview, *Proceedings of the USENIX Winter Conference*, Dallas, TX, February 9-12, 1988.

J. F. Patterson, R. D. Hill, S. L. Rohall, and W. S. Meeks, Rendezvous: An Architecture for Synchronous Multi-User Applications, *CSCW '90 Proceedings (Los Angeles, October 8-10)*, New York, 1990.

J. F. Patterson, The Good, the Bad, and the Ugly of Window Sharing in X, *Proceedings of the Fourth Annual X Technical Conference (Boston, January 15-17)*, January, 1990.

R. W. Scheifler, J. Gettys, and R. Newman, *X Window System: C Library and Protocol Reference*, Digital Press, Bedford, Massachusetts , 1988.

B. Shneiderman, Response Time and Display Rate in Human Performance with Computers, *ACM Computing Surveys 16*,3 (September 1984), 265-285.

M. Stefik, D. G. Bobrow, G. Foster, S. Lanning, and D. Tatar, WYSIWIS Revised: Early Experiences with Multiuser Interfaces, *ACM Transactions on Office Information Systems* 5,2 (April 1987), 147-167.