THE UNIVERSITY OF CALGARY

Design of a Real-Time Groupware Toolkit

by

Mark Roseman

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

FEBRUARY, 1993

ISBN 0-315-83238-X

THE UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled "Design of a Real-Time Groupware Toolkit" submitted by Mark Roseman in partial fulfillment of the requirements for the degree of Master of Science.

[illegible]

Supervisor, Saul Greenberg

Department of Computer Science

[illegible]

Brian Gaines,

Department of Computer Science

[illegible]

David Hill,

Department of Computer Science

[illegible]

Stephen Hayne,

Department of Management Information Systems

March 5/1993

(Date)

Abstract

Real-time groupware systems, where several users work simultaneously with the same information, are notoriously difficult to construct. This thesis describes the design and implementation of a toolkit for building real-time groupware. Following a user-centered methodology, a number of requirements for groupware toolkits are presented. The requirements are both human-centered, such as support for work surface activities, flexible group processes, and integration with conventional work, and also programmer-centered, such as communications and a shared graphics model. Based on these requirements, a prototype toolkit called GROUPKIT is described. It contains three sets of components used to meet the requirements: a communications infrastructure; overlays for work surface activities; and open protocols for flexible group processes. Other concerns in the toolkit design are minimizing the developer's work; encouraging use; extensibility; and flexibility. A number of sample applications built with GROUPKIT are described.

Acknowledgements

It would be most ironic if a thesis about collaborative work were a truly individual effort. Many people have contributed to this work, and it has been enriched because of them.

Most importantly, Saul Greenberg introduced me to the new and exciting area of Computer Supported Cooperative Work, initially hiring me as a research assistant over the summer of 1991. He also managed to keep my interest in the area throughout my MSc., despite the numerous threats to defect to the Philosophy department. He has provided excellent ideas and perspectives on the work as it evolved; and provided both the guidance and the freedom that made this a most rewarding experience.

The open and friendly atmosphere in the labs may not have always improved productivity, but did serve as a source of new ideas, victims for proofreading, and many much-needed diversions. Thanks to Doug Schaffer, my partner in crime, INTERVIEWS frustrations, and 6:30 a.m. squash, for the never-ending "over the shoulder" chats. Thanks also to Shelli Dubs, Eric Schenk, Natascha Schuler, Sonja Branskat, Rob Kremer, and Ted O'Grady for the enlightening discussions and inspiration.

Thanks to Ron Baecker for sponsoring my visit to DGP, which provided a work environment that somehow allowed large portions of this text to be generated. Gifford Louie, Alex Mitchell, Beverly Harrison, Michel Beaudouin-Lafon and George Fitzmaurice made my stay in Toronto a useful and enjoyable one. Feedback from the Computer Science Department at the U of T as well as from the world-wide CSCW community has helped to improve this work.

This research was supported by the National Sciences and Engineering Research Council.

Table of Contents

List of Tables

List of Figures

Chapter 1

Motivation and Goals

Groupware is a generic term for software systems designed to assist groups of people working together (Johansen 1988). This is in contrast to most software systems that only support interaction between a single user and the computer. Although the roots of groupware can be traced to the late 1960's (Engelbart and English 1968), the systems have only recently begun to proliferate. The research discipline that studies groupware, as well as the wider context in which groupware is used, is referred to as Computer Supported Cooperative Work, or CSCW (Grief 1988).

The development of groupware systems has shown that there are many difficulties associated with building these systems (Greenberg, Roseman, Webster and Bohnet 1992). Developers must consider technical issues such as synchronization, concurrency, communications, registration and more. Several "human-centered" issues — issues arising out of the social context in which the groupware is used — must be dealt with, or the systems may be rejected by their users (Grudin 1989). Finally, all the difficulties associated with single-user systems remain present when developing groupware systems. Time spent dealing with these difficulties is an obstacle to progress in CSCW research, where it is essential to evaluate real systems — systems that have to overcome these difficulties (Grudin 1989).

Several *groupware toolkits* are now emerging from universities and research labs to address these difficulties, assisting the software developer by providing software libraries that eliminate many of these difficulties. However, most of these toolkits have been developed in a restricted way, leaving many of the problems, particularly those surrounding human-centered issues, unresolved.

This thesis describes the design principles behind a groupware toolkit, and how they are instantiated in GROUPKIT (Roseman and Greenberg 1992b). GROUPKIT addresses several of the problems left unsolved by current toolkits. It is unique in that its design focus is on the needs of groupware developers, largely embodied in both the technical (or programmer-centered) and human-centered features of the applications they are constructing.

This chapter begins by providing background to the particular application domain the thesis considers — "real-time conferencing systems." The next section argues that toolkits are a reasonable approach to building groupware, and introduces the principle of "user-centered toolkit design." The chapter concludes by outlining the rest of the thesis.

Real-Time Conferencing Systems

Groupware systems are often divided into four categories, as shown in Figure 1.1 (Johansen 1988). This taxonomy is based on two dimensions, time and location. Groupware systems can support users working at the same time (synchronous) or at different times (asynchronous), as well as users working in the same physical location (co-located) or at a distance (distributed). While most systems support work in only one quadrant, a few are designed to span the dimensions. This thesis is particularly concerned with groupware systems used for real-time work, both in the same location or at a distance, called *real-time conferencing systems*.

There are many real-time conferencing systems available today. This category includes shared text editors, e.g. SHREDIT (Killey 1991), text-based chat and talk programs, e.g. CANTATA (Chang 1986), freehand sketching systems, e.g. GROUPSKETCH (Greenberg

	Same Time	Different Times
Same Place	face-to-face interaction	asynchronous interaction
Different Places	synchronous distributed interaction	asynchronous distributed interaction

Figure 1.1. Taxonomy of groupware systems.

and Bohnet 1991), structured drawing programs, e.g. GROUPDRAW (Greenberg, Roseman et al 1992), group decision support systems, e.g. GROUPSYSTEMS (Nunamaker, Dennis, Valacich, Vogel and George 1991), and video-based desktop conferencing systems, e.g. TEAMWORKSTATION (Ishii 1990).

Much of the recent interest in groupware has been in real-time conferencing systems. While most groupware should be developed to support both synchronous and asynchronous work (Dourish and Belloti 1992), the problems associated with building the two types of systems are sufficiently diverse that providing the necessary underlying support would be a large undertaking. Further, with the exception of studies of electronic mail usage, e.g. Eveland and Bikson (1988), much of the research into group behavior has occurred in synchronous domains, e.g. Tang(1991), Tatar, Foster and Bobrow (1991) and Mantei (1989). Additionally, there is a greater amount of research discussing implementation issues for synchronous systems, e.g. Greenberg, Roseman et al. (1992), Lauwers, Joseph, Lantz and Romanow (1990) and Ellis, Gibbs and Rein (1991). This wider body of research provides a substantially better basis for deriving design principles and generalizations for toolkits — particularly human-centered design principles — than is available in asynchronous domains.

User-Centered Toolkit Design

Software developers use toolkits to produce better programs faster. This not only saves programming time, but can increase the quality of the resulting program by using well-constructed and principled modules. The support is usually provided by a number of library subroutines, or increasingly, object-oriented class libraries (Booch 1990). These library routines or class libraries provide components with very general functionality, leaving only application-specific functionality to be written by the developer.

Toolkits exist for many different problem domains, providing components for building applications specific to those domains. An example domain is *Human Computer Interaction* (HCI) and specifically user interface design, where toolkits can simplify the development of interfaces (Myers and Rosson 1992). Though other types of development tools exist, such as *interface builders* or *User Interface Management Systems*, toolkits offer the greatest flexibility to the developer. Toolkit users can use pre-defined components, often can alter or extend them, or can eschew standard components for custom-made solutions. This flexibility can be important for non-standard applications, where desired features may be difficult to implement in more restrictive development tools. This flexibility is especially important to groupware, for it is far too early to conclude what a "standard" groupware application would be.

In order for a toolkit to be successful, it must provide the necessary components and functionality required by the groupware developer. Yet how can the developer's needs be determined?

Consider traditional interface design practice. When designing interfaces for end users, developers are taught to observe the users of their system, to focus on the type of work the users need to do, and to take into account their needs and preferences. Such *user-centered* *design* (Norman 1986) has proven to be reasonably successful at determining the requirements for new systems.

This thesis argues that a user-centered approach can also be applied to designing software intended for developing other software — toolkits. The users here are the application developers who build interface software. These developers have particular needs in building their applications that should be carefully considered by toolkit builders. These needs can be identified by examining their work practices. What problems do developers encounter when building software? What sorts of applications are they building? What features are necessary, which are important, and which will never be needed? Addressing these sorts of questions provides the foundations necessary to support developers.

An outline of steps that can be used when designing a toolkit using a user-centered approach is presented below. This particular set of steps is completely analogous to steps found in the traditional practice of user-centered design of applications for end users: observe users, design and prototype, elicit feedback, then repeat (Norman 1986; Gomoll and Nicol 1990; Tang 1991). Each of the steps is illustrated by considering the particular needs of groupware developers.

Specify Toolkit Domain

The first step is to identify the domain in which the toolkit will be used. Special needs application domains such as groupware can benefit from toolkit support unnecessary for most applications. This is important to determine what features will and will not be included in the toolkit. As with end-user applications, the more that is known about the domain the easier it is to build a good system that meets the specific needs of the users.

The toolkit domain considered in this thesis is that of real-time conferencing applications, as described previously. Asynchronous groupware applications such as electronic mail are explicitly excluded. While strong claims can be made that groupware systems should span the synchronous vs. asynchronous dimension, e.g. Cockburn and Greenberg (1993) and Dourish and Belloti (1992), the needs of the two types of groupware are different enough that supporting both is non-trivial. While an ideal groupware toolkit should provide support for both, this project is restricted to synchronous groupware.

Identify Developers

Having identified the target domain, it is then important to identify the developers who will use the toolkit. Knowing the types of developers will have important consequences for how the toolkit will be designed, the level of functionality provided, and how that functionality will be delivered to the developer. Several important considerations can arise. Toolkits should attempt to use programming languages already known by the developer. The selection of computing paradigms, such as an object-oriented paradigm will depend on the developers' skills and experiences. If developers have experience with other toolkits, familiar concepts from those toolkits should be exploited in the design. The acceptable learning curve for the toolkit depends on its use by developers, whether for short-term or long-term projects.

Since most groupware is being developed in research labs, it is sensible to assume sophisticated developers. This has several implications for building the toolkit, for example selection of an object-oriented language, or a sophisticated underlying user interface toolkit as a basis for the groupware toolkit.

Identify Use of Toolkit

Toolkits can either provide building blocks for making domain-specific objects, or provide the domain-specific objects themselves. In the first case, developers must act as widget builders, creating high-level components that are then used to build the end applications. In the second case, the high-level components in the toolkit are used directly in the end application, such as complete dialog boxes provided by many interface toolkits.

For the real-time conferencing domain, the first approach seems beneficial. While several high-level components can be identified and should be provided in the toolkit, in general the domain is too new and diverse to predict the set of high-level components that will be necessary. A toolkit supporting this domain should then be created in an open-ended and extensible way, encouraging developers to create new groupware components.

Consider Target Applications

The toolkit features required by developers depend heavily on the applications they are constructing. Studying these applications can suggest common needs that can be translated into functionality and features best incorporated into the toolkit. Existing applications in the selected domain are easiest to consider, but considering possible future applications can be valuable as well.

Many real-time conferencing applications exist today, e.g. sketchpads, structured drawing programs, and editors, as previously discussed. Much of the research in CSCW is concerned with evaluating the features and paradigms used by these applications (Greenberg, Roseman et al 1992; Grudin 1989; Stefik, Bobrow, Foster, Lanning and Tatar 1987). This research provides an excellent basis for deriving toolkit features. Chapter 2 provides examples of how this research can be transformed into toolkit design requirements.

Design for Proper Use

The previous steps should generate a set of necessary components and features of the toolkit, should raise issues about the flexibility of components, and should suggest information about the implementation language and other important concepts. Yet it is important not merely to piece together all these features and information into the toolkit, but to combine and structure the information in a way useful to the developer.

A toolkit provides more than an alphabetic list of routines in a library to the developer. A toolkit should contain a philosophy of how applications should be developed using the toolkit, and should encourage developers to build good programs properly. This can be done in several ways. The structure of the toolkit can suggest appropriate ways that applications can be created. Documentation can go beyond describing how to use particular components, and concentrate also on *when* to use them. Example programs included with the toolkit can highlight instances of good design, and are perhaps more likely to be used than documentation. One goal in GROUPKIT is to make it easier for developers to do the right things in building their systems, and examples of this are presented throughout the thesis.

Apply Design Affordances

A toolkit is more than simply a set of components or library routines. Developers must be encouraged to actually *understand* and *use* the appropriate components when the situation demands it. With careful design, toolkits can actually encourage the creation of better programs. This section develops the idea of a *design affordance* to address this issue.

Affordance theory has been applied to the design of interfaces by several people (Gaver 1991b; Norman 1988), where an *affordance* is defined as the properties of objects that suggest particular uses to users. For example, the raised appearance of buttons in modern interfaces suggests "pushing" to the user. Gaver (1991b) believes the theory can guide developers in designing artifacts that emphasize desired affordances and de-emphasize undesired ones.

The use of affordances can be extended beyond conventional applications in interface objects to the level of software objects. A *design affordance* is defined as a property of a toolkit object that suggests how it can be used. The design objects are at the level of underlying software tools, not end-user applications. Design affordances therefore involve interactions between such tools and program developers, not end users. In this sense a toolkit can have affordances, where the affordances suggest appropriate uses of the toolkit to the software developer.

It is important to distinguish between design affordances and "features." Features can be all too easily hidden within a toolkit, can be difficult to use, and it can be hard to comprehend when or why they would be used. In contrast, design affordances are situated in the overall toolkit to present themselves to developers when needed. Design affordances clearly suggest to the developer *how* a particular feature can be used and *why* it should be used. Affordances may also combine a set of individual features into a high-level concept or construct that can be easily applied to simplify development.

Design affordances can be used by toolkit builders to aid developers in building better programs. Most user interface toolkits, for example, provide "control panels" that collect buttons, valuators, menus, and so on. As an affordance, this suggests a particular interaction style to the developer, and indeed control panels are prevalent in many of today's applications. Other common toolkit constructs-as-affordances are graphical canvases, text editors, and terminal windows.

Similarly, a toolkit supplying consistent looking components will encourage building consistent looking interfaces. Supporting keyboard accelerators by default at the toolkit level will encourage their adoption. Programs built using a toolkit with an embeddable control language, e.g. TCL (Ousterhout 1990), will themselves be structured to support this language, providing user-level scripting and a highly tailorable system. Nothing forces developers to incorporate these features in their applications, but because of the toolkit's design affordances it is easy and natural to do so.

Iterate Design

Finally, it is essential to iterate the design, using preliminary versions of the toolkit to design standard applications, identify problems, redesign, and try again. Toolkit design is inherently a "wicked problem" (Rittel and Webber 1973), where there can be no definitive formulation of the problem. Developers have diverse and often conflicting needs, and these needs change rapidly in response to changes in technology and supporting research. As with applications, the chances of getting the design right the first time are extremely small. Iterating the design helps refine both the set of features and the ways of structuring those features.

The GROUPKIT system described in this thesis is the result of several iterations. Because of time restrictions, these are currently based on the experiences of the toolkit developer. Significant changes have resulted as a direct consequence of the iteration. An effort has been made to highlight these changes throughout the thesis, to highlight the ongoing design rationale rather than present the illusion of a static artifact resulting from a deterministic design process (Kuhn 1962). A preliminary version of the system has been released to the

CSCW research community, so that ongoing work on GROUPKIT can be based on feedback from developers.

This user-centered approach to designing software toolkits seems a natural and promising extension to the user-centered design of interfaces. The "observe, design, prototype, repeat" cycle should allow toolkit builders to design systems that more accurately reflect the needs of the application developers.

Outline of Thesis

The remainder of this thesis describes and evaluates the design and implementation of GROUPKIT, which is carried out using the user-centered toolkit design methodology presented in the previous section.

Chapter 2, "Design Requirements," surveys many of the existing groupware systems and how they are used to determine some of the features that are necessary for building real-time conferencing systems. These are generalized as human-centered and programmer-centered design requirements for groupware toolkits. The chapter then uses these principles to compare some of the existing toolkits for real-time groupware.

Chapter 3, "Design and Implementation" details the design and implementation process that resulted from these design requirements. The emphasis in this chapter will be on describing the set of features developed in GROUPKIT. These components include a communications infrastructure, overlays for providing work surface actions, and open protocols for flexible software policies.

Chapter 4, "Design Rationale" examines why particular design decisions were made in GROUPKIT, in the hope that this knowledge can guide other developers of toolkits. The components in Chapter 3 are considered in terms of minimizing the developer's work, encouraging use of components, extensibility, and flexibility.

Chapter 5, "Evaluation" reflects on the resulting design and implementation, evaluating the project in terms of the GROUPKIT implementation, as well as the underlying principles and requirements which directed the implementation.

Chapter 6, "Concluding Remarks" presents several areas for future work, and summarizes the contributions of this research.

Chapter 2

Design Requirements

This chapter surveys many of the existing groupware systems as well as studies of their use. This provides a basis for determining some of the features of groupware that are important for its success. These features can then be interpreted as design requirements for a groupware toolkit.

This chapter divides toolkit requirements into two categories: human-centered and programmer-centered. Human-centered requirements are motivated by the needs of groupware users. They suggest common features that people require to work together effectively, and that may be critical for the successful adoption of the system. These are the features that people see as part of groupware. Strictly speaking, a programmer can build groupware and ignore all the human-centered requirements, for they are not as fundamental in the engineering sense as, for example, network communications. Given that human-centered requirements may make difficult implementation demands, application developers may neglect to follow them. The result may well be an impoverished or unusable system. Embedding human-centered requirements in a toolkit encourages application developers to include them in software, arguably producing better programs.

In contrast, programmer-centered requirements are typically visible only to the application developer, and not to the end user. These are engineering requirements that are necessary for building nearly any groupware system providing a basic level of functionality. For example the ability to easily set up communication between different computer processes forms a basic programmer-centered requirement. When programmer-centered requirements are embedded in a toolkit, application developers gain easy access to well-designed features they would otherwise have to build themselves.

Table 2.1 summarizes the human-centered and programmer-centered design requirements discussed in this chapter.

The chapter closes by considering the support provided by some existing groupware toolkits, compared with the design requirements derived in this chapter. While this comparison is not a fair one — different toolkits were constructed in response to different needs — it does suggest that current toolkits have emphasized programmer-centered requirements (which developers have to worry about anyway) while largely ignoring human-centered requirements (which developers may overlook if support is not provided).

Requirements	Rationale	Examples
Human-centered requirements		
Supporting multi-user actions over a visual work surface	Human factors work suggests people often gesture over and annotate diverse artifacts	• provide support for gesturing • provide support for graphical annotation
Structuring group processes during a meeting	Many conferences need to be structured, but different groups require different sorts of structure to accept software	• provide various floor control policies • support different registration methods • support latecomers
Integration with conventional ways of doing work	People are comfortable with using other media, e.g. telephone and other programs	• integrate other forms of communication • allow use of single-user applications
Programmer-centered requirements		
Technical support of multiple and distributed processes	Programmers must, for any conference, manage multiple, distributed processes, and connections between them, including starting up connections, keeping them active, and tearing them down.	• provide processes for basic conference management • provide a robust communications infrastructure • provide support for persistent sessions
Technical support of a graphics model	Many applications can be seen as shared visual work surfaces, requiring textual and graphical primitives that can cope with issues such as concurrency and WYSIWIS view sharing	• provide primitives to a shared graphics library • provide object concurrency control • separate the view of an object from its underlying representation

Table 2.1. Summary of toolkit design requirements.

Human-Centered Requirements

This section looks at some of the human-centered requirements for real-time conferences that have been derived from CSCW research. Human-centered requirements are all too often ignored in software design (Mulligan, Altom and Simkin 1991), because they are not only difficult to build, but their benefits are often hard to quantify and evaluate. With CSCW, as with more traditional HCI, the requirements do not seem to affect the software's functionality, they do not add to lists of features in advertising brochures, and they are often termed non-essential by management in software development organizations.

However, Grudin (1989) claims that software ignoring human considerations, particularly groupware, will not be accepted or adopted by the group. Groups have particular work practices, and are reluctant to change these work practices. Computer systems that require changes to these work practices are likely to be left unused by the group. Unfortunately, it is a long and difficult process to design the necessary features required by the group (Pendergast 1992). Further, the way those features are packaged and presented to users — human considerations in the software — is critical to the program's success (Francik, Rudman, Cooper and Levine 1991). Development effort directed towards state-of-the-art technical features is wasted if the product is not used. Thus the inclusion of human-centered requirements is critical to the overall success of a software product.

By observation and study of work situations, requirements arise that cannot usually be determined *a priori,* through introspection or "desktop design." Such introspective methods are difficult enough to apply for single-user applications, but break down for groupware (Grudin 1989). These methods would require the ability to predict the reactions of all members of the work group, taking into account the interactions between the different group members in a variety of work situations. Therefore it is essential to use methods from the social sciences, such as task analysis and ethnography (Hughes, Randall and Shapiro 1992), observing how people actually work. As will be seen with the examples that follow, many of these principles and requirements seem "obvious." However, it is only through the studies that they became evident.

This section presents three critical human-centered requirements that have emerged from the literature. These are: support of generic multi-user actions over a work space (e.g. gesturing and annotation), flexibility in structuring group processes (e.g. floor control and conference registration), and integration with conventional ways of doing work (e.g. telephone, video and single-user applications). These requirements are now considered in turn, with comparisons being drawn to everyday work situations.

Supporting multi-user actions over a visual work surface

A common occurrence in many organizations is the small group design meeting. These meetings occur when a group of people get together informally to discuss a problem. A common feature of most of these meetings is the inclusion — and often central focus — on a shared surface, typically a whiteboard or a sheet of paper on a table. These *visual work surfaces* perform an important function during these meetings, serving as both a repository for ideas as well as a focus for group interaction.

Simulating this shared surface for meeting participants who are not co-located was an early application of groupware systems (Group Technologies Inc. 1990; Stefik, Bobrow et al 1987). Each user in the meeting was equipped with their own computer terminal running special software to simulate an electronic whiteboard. These systems permitted actions such as drawing and typing on one user's screen to be transmitted and displayed on all other users' screens. This transmission occurred at discrete intervals, often when an entire stroke or object was completely drawn. The commonly held belief was that since the work surface was a place merely to store information, that this would be enough to support design meetings at a distance.

However, when people actually used these systems, problems appeared. The following quotes were typical of observations of these systems, in this case COGNOTER (Tatar, Foster et al 1991).

"Why can't I see that?"

"I don't see what use it is to have a big screen if we can't all contribute to it."

"Click DONE so I can see it."

P1: "P2, do you have anything you want to say?"
P2: "I won't be able to see it up there, right?"

There were many problems with *reference* — people speaking with reference to items on their own screens, but other users could not identify the referenced items. What happened was that the *results* of drawing or typing were displayed on other users' screens, but not as the drawings or text were being *created.* The user creating the drawing would see it as it was being created, and would speak with reference to the drawing, but other users would not see the drawing until it was complete.

This motivated researchers at Xerox PARC (Bly 1988; Tang 1991) to study how people in design meetings use conventional work surfaces such as whiteboards. In several small group design sessions, actions around a surface were enumerated and classified. A framework was developed, containing three *actions* (list, draw, and gesture) and three *functions* of the actions (store information, express ideas, and mediate interaction). Table 2.2 shows the resulting classification from one of several studies run (Tang 1991). Results in other studies were similarly distributed.

	List	Draw	Gesture	Total
Store Information	18%	8%	1%	27%
Express ideas	1%	28%	15%	44%
Mediate interaction	0%	9%	20%	29%
Total	19%	45%	36%	100%

Table 2.2. Results of Xerox PARC classification of workspace activity. ($n = 225$)

Surprisingly, the conventional view of work surfaces (listing and drawing to store information) accounts for only 37% (59 actions) of all work space activity. Averaging over all the studies this figure is about 25%. The remaining functions, expressing ideas and mediating interaction, together account for a significant portion of work surface usage. Also gestures are used extensively to both express ideas and mediate interaction.

These considerations had not been included in the design of earlier systems. This was largely because incorrect theories of human communication were applied to the system designs. For example, a "parcel-post" model of communication (Tatar, Foster et al 1991) was often assumed, whereby a group of work surface actions were broadcast at discrete intervals to remote users. This is insufficient because the marks on the work surface are used as an expression of ideas, and are closely coupled with verbal communication. When there is a delay in broadcasting changes to the work surface, this communication became unsynchronized, resulting in confusion amongst users. Six design implications arose from this work.

1. Convey process. The process of creating and using drawings is often more important than the resulting artifact or drawing. Further, the process is usually closely synchronized with speech. It is important that systems transmit changes to the work surface immediately to other users, providing immediate fine-grained feedback rather than a "parcel-post" model of interaction. Systems based on Tang's recommendations such as GROUPSKETCH (Greenberg and Bohnet 1991), COMMUNE (Bly and Minneman 1990), and GROUPDRAW (Greenberg, Roseman et al 1992) provide immediate feedback to solve the referencing problems.

2. Simultaneity. The studies also highlighted frequent instances of several people working on the drawing surface at the same time. If it is desired to emulate such a surface, users must be able to interact at the same time, as opposed to following a strict turn-taking or floor control policy. Observations of computer systems supporting simultaneous actions by users, e.g. Greenberg, Roseman et al (1992), note that simultaneous work occurs often on the computer systems, where users do not physically impede each other.

3. Modeless interface. The studies identified many cases of frequent mode switches — users switching rapidly and frequently between listing, drawing and gesturing. Supporting this fluid switching in software requires that modes be eliminated or that the overhead of switching modes in the software is minimized.

4. Common view. Providing a common view to the shared work surface permits users to share a common orientation, so that it is easier to reference items in the work space. This has typically implied a "What You See Is What I See" (WYSIWIS) work surface (Stefik, Bobrow et al 1987) where all participants share an identical view.

5. Gestures. With conventional work surfaces, gestures are used by people to reference existing items on the work surfaces, and also to mediate access to the work surface. Early systems did not support any use of gestures, which increased the difficulty of references between participants.

Modern systems based on the Xerox studies support the use of gestures, using *telepointers* or *multiple cursors* — a cursor for each user that is visible on other users' screens — allowing all users to point to objects on the work surface with a mouse or stylus. Usability tests on these systems, e.g. Minneman and Bly (1991), Greenberg and Bohnet (1991) and Scrivener, Garner, Palmen, Smyth, Clark et al (1992), show that using computerized work surfaces is close to using conventional work surfaces when gesturing is provided.

Gesturing seems a generally useful activity, applicable to a wide variety of computer systems. For example, the ability to point to cells in a shared spreadsheet provides the same benefits as pointing to graphics or text in shared drawing surfaces. This is because many different applications can be envisioned as a shared work surface.

6. Annotation. A similar activity that can be valuable for shared surfaces is annotation, where a graphical or textual marking is attached to an artifact in the work surface. The Xerox studies noted many instances of using drawing or writing to express ideas. Annotation is commonly seen when, for example, a document is being proofread — graphical and textual marks are made, again referring to items or artifacts in a work surface. Several systems, such as FREESTYLE (Francik, Rudman et al 1991) and the PROOF-MARKS language in VMACS (Lakin 1990) support graphical annotation of objects in a work surface.

Flexible structuring of group processes during a meeting

A contentious issue in the CSCW literature is the role of *socially* vs. *technically-supported* protocols and processes for controlling a group's behavior during meetings. Social protocols imply the *absence* of explicit control in software for the group's behavior, allowing the group to negotiate protocols through normal channels. Technically-supported protocols, on the other hand, use the software to impose a particular model of social interaction on the group.

The choice between social and technically-supported protocols must be considered carefully. Advocates of social protocols, e.g. Dykstra and Carasik (1991), often argue that we don't understand the way that humans work together well enough to formalize a protocol describing that work. Further, different groups require different types of protocols, and entrenching a single protocol in the software will alienate many groups (Grudin 1989). Natural protocols evolve and change throughout the course of even a single meeting, and while humans can quickly and fluidly adapt to these changes, it may be more difficult to provide smooth transitions if protocols are embedded in software.

However, advocates of technically-supported protocols, e.g. Nunamaker, Dennis et al (1991), note the many problems or "process losses" associated with unstructured meetings, such as poor usage of time (Jay 1976). Management research has created numerous models of organizational behavior that can assist a group in improving their productivity. By embedding much of this theory in software, it is argued that computer meetings can be made more productive as well.

More moderate approaches are of course also possible, e.g. Greenberg (1991), Johnson-Lenz and Johnson-Lenz (1991). It is individual developers of groupware applications who must determine the role of social and technically-supported protocols based on the needs of their user groups, whether the group will accept and benefit from including structured processes in the software or not. However, toolkit developers would be advised to provide the *facility* whereby developers can build structured protocols into their software if needed. This facility should have several characteristics. First, developers should not be required to use structured protocols, if it is not desirable in a particular application. Second, the facility should allow a wide range of protocols to be implemented, to better accommodate the needs of different groups. Finally, it should be possible to build several different protocols for the same operation, allowing designers the ability to provide a library of protocols to end users or group moderators. This could allow end-users to select and switch between different protocols at run-time, adjusting their tools as their needs change.

This flexibility is of paramount importance if toolkits are to assist developers in meeting the needs of their end users. Three examples showing the need for flexibility in protocols are now discussed.

1. Floor control. Floor control can mediate access to shared objects in a real-time conference. For example, floor control used with a shared text editor can control which users can type at any given time. Many different floor control policies are possible, ranging from free floor control (anyone can type anytime), to preemptive floor control (any user can grab control from the active floor holder) to explicit release (users can grab control only when the floor holder explicitly releases the floor). Lauwers (1990) and Greenberg (1991) recommend that systems should "support a broad range of (floor control) policies" providing different policies to suit the users' needs. Mantei observed these needs are different between groups and may even change within a single meeting of one group, where the participants select different styles of working (Mantei 1989).

2. Registration. Every conference must provide a mechanism allowing users to join. Yet even the method of joining can vary between conferences. Some conferences are more open or informal, allowing anyone to join the conference at any time. For other conferences, perhaps those resembling formal business meetings, there should be tighter control of who can join a conference. New users might be required to appear on an "access list" or be approved by an existing conference user. As another example, sometimes more spontaneous creation of conferences is desired to simulate casual interactions (Kraut, Egido and Galegher 1990; Root 1988) while other situations may require a central facilitator to handle registration (Nunamaker, Dennis et al 1991). Toolkits should provide the flexibility to support any reasonable registration process.

3. Conference Latecomers. As a consequence of spontaneous conferences and due to the nature of conventional meetings, all users will rarely be present at the start of the meeting. There should exist ways for newcomers to join at any time, as well as existing members to leave. Strategies should also be supported to assist the newcomers in "getting up to speed." This may involve simply sending the current conference state to the new user (Greenberg, Roseman et al 1992) or may involve providing summary information, in the form of transcripts or snapshots (Dubs and Hayne 1992), on the progress of the conference from its beginning. Though many are interested in this problem, e.g. Patterson, Hill, Rohall and Meeks (1990) and Crowley, Baker, Forsdick, Milazzo and Tomlinson (1990), much research remains on exactly how latecomers can be smoothly integrated into a conference.

Integration with conventional ways of doing work

Groupware systems are only one set of tools used to do work. Groups can work together using other means, such as traditional face to face meetings not augmented by computers, e.g. Tang (1991) and Jay (1976), through telephone conversations, and through conventional paper documents. To further complicate matters, "individual" work, done with pen and paper or single-user computer applications must often be incorporated into "group" work, allowing individuals to share their work with the whole group. This section suggests human-centered requirements based on the challenge of integrating individual work into group scenarios.

1. Integrate with Non-Computer Artifacts. An excellent example of integrating daily work into computer conferences is the TEAMWORKSTATION system (Ishii 1990). That system, and others like it, e.g. VIDEODRAW (Tang and Minneman 1990), use video fusion techniques so that users working together can use both the computer and items on the normal desktop. Video cameras display one user's desktop on other users' monitors, with the "real" and "virtual" desktop fused together. In this way, computers can even be used to discuss paper documents between remote users without having them scanned in to the computer. Ishii refers to this important concept as "seamlessness" (Ishii, Kobayashi and Grudin 1992), where the barrier or seam separating computer-based and individual work is reduced. This approach provides excellent results, but includes extra costs such as the hardware involved. The following two methods for integrating conventional work at a low cost are also desirable.

2. Integrate Other Forms of Communication. Voice communication is an important factor in most conferences (Chapanis 1975). In the Xerox PARC observations for example, it was noted that voice was used frequently, and also directly related to artifacts on the drawing surfaces. Given the ubiquity of telephones, it can be assumed that a voice channel is available. Some systems have been built that closely integrates other forms of communication with computer conferencing, allowing a user to begin a normal telephone call or video-conference from their computer workstation, e.g. the RAPPORT conferencing system (Ensor, Ahuja, Horn and Lucco 1988). Given the close ties between computer conferencing and other forms of communication, it is important that they are handled in a consistent way, preferably through the conferencing system.

3. Integrate Single-User Applications. Groupware applications today are far from ubiquitous. There are many tasks that can currently be done only with single-user software. Further, many users are comfortable with particular single-user programs. If single-user applications can be made to function in the multi-user conference setting, both this extra functionality and extra familiarity can be used to the advantage of the group. Shared text screen systems like SHARE (Greenberg 1990), and shared window systems like SHAREDX (Garfinkel, Gust, Lemon and Lowder 1989) provide the ability to incorporate single-user systems into multi-user settings. Input from several participants is merged into a single input channel, perhaps moderated using floor control strategies, and then sent to the single-user application as if from a single-user. The application's output, normally sent to a single screen, is distributed to all the conference users' screens. Though this does not result in true multi-user aware applications, it can be an effective medium for presenting individual work to the group.

Programmer-Centered Requirements

The previous section dealt with the human-centered features that should be included in groupware applications to encourage user acceptance, and to provide tools that assist people in doing work. This section now deals with the *construction* of groupware, involving issues hidden from the end user but essential to the developer. The groupware developer is faced with a wide range of new challenges not encountered when developing single-user software. Toolkits can help by providing reusable components to be included in the software, limiting the amount of design and code that must be created from scratch.

This section looks at several fundamental components that can be provided, based on requirements derived by reviewing groupware systems. Studies of these systems highlight the commonalities between them and suggest toolkit components. By managing these programmer-centered requirements at the toolkit level, the application developer can save effort in designing, implementing and testing common groupware components. In this section two sets of requirements are identified, dealing with distributed processes and graphics respectively, which suggest the design of toolkit components.

Technical support of multiple and distributed processes

At a fundamental level, groupware systems for real-time distributed work require support for maintaining several different processes, often distributed across a network. Not only must these processes be maintained, but communications channels must be available so they can communicate and interact with one another. These processes and communications channels provide the mechanisms whereby higher-level components, such as conferencing tools, can be built and interact with each other. Several fundamental aspects of multiple and distributed process support are now discussed.

1. Conference Management. Conference management includes participant registration, conference initiation, conference maintenance, and conference teardown. Participant registration allows users to join conferences. Users must first locate existing conferences, or alternately create new ones, and then join the conference. Conference initiation actually creates a new conference process, whether locally invoked (e.g. a user asks to create a new conference) or remotely (e.g. a meeting facilitator opens a voting tool on all users' screens). Conference maintenance involves the communications between conference users, discussed in the next paragraph. Finally, conference teardown is the process whereby an existing conference is ended. Issues here mainly concern the disposition of conference artifacts.

Groupware toolkits should strive to support all aspects of basic conference management. This allows application developers to concentrate on the functionality specific to their application, without concern for establishing communications. In light of the human-centered requirements, conference management must be implemented flexibly, so that developers have choices where appropriate, such as different strategies for participant registration.

2. Communications Infrastructure. It must of course be possible for the distributed processes to communicate. At the minimum, any process must be capable of sending a message to any other process, without concern for low-level communications issues. Because there is often a need in groupware systems to send the same message to some or all conference participants, it is preferable if a multi-cast facility is available.

Communications architecture may be designed in one of two ways, centralized or replicated. .In centralized architectures, all messages are sent through a central machine, which forwards messages to other conference users. In replicated architectures, no central mediary is used and processes communicate directly with each other. The trade-offs between the two architectures have been well-documented (Ahuja, Ensor and Lucco 1990; Lauwers, Joseph et al 1990), with centralized architectures simplifying concurrency control and replicated architectures being more efficient and robust to machine failure. It is also possible to build hybrid architectures, where some services are provided in a replicated way (e.g. most communications between programs) while other services are provided in a centralized way (e.g. lock servers for concurrency).

3. Persistent Sessions. Often computer conferences will span more than a single session, as occurs frequently in decision support meetings (Nunamaker, Dennis et al 1991). It is desirable to maintain all session state information over the full duration of the conference. There should exist general mechanisms whereby conference objects can be made persistent. Facilities for persistence are discussed mainly in the distributed systems literature, for example the OBJECT REPOSITORY (Liskov, Gruber, Johnson and Shrira 1991) which provides a highly reliable generic facility for storing persistent objects. In groupware systems, the use of persistence can be seen in many hypertext-based systems, such as SEPIA (Haake and Wilson 1992).

Technical support of a graphics model

As previously seen, many groupware systems can be thought of as visual work surfaces, shared between conference participants. In light of this, groupware will require mechanisms to allow users access to such a shared work surface. Though in theory the basic communications discussed in the previous section can serve as a basis for a shared work surface, a higher level of support is desired. This support includes a framework for implementing shared visual objects, such as structured graphics or text, concurrency control for mediating access to these shared objects, and a method of separating the underlying structure of an object from its view, allowing users with different needs to view the objects differently.

1. Shared Visual Objects. Many groupware applications require shared visual objects for displaying objects such as shared lines, rectangles, and text. Much work is involved in creating such objects, and it is undesirable to repeat the work for each new type of object created. Greenberg, Roseman et al.'s discussion of GROUPDRAW (1992) describes technical issues of a shared object-oriented drawing package, and provides a design for an abstract drawing object that can be sub-classed into concrete objects such as shared lines. Some guidelines for the behavior of fine-grained editors for simple visual objects are provided by Bier and Freeman in their MMM system (Bier and Freeman 1991), while the RENDEZVOUS project has spawned a large collection of graphical editors (Brinck and Gomez 1992). UNIDRAW (Vlissides and Linton 1989) provides a single-user framework for creating graphical editors, allowing developers to create underlying components, graphical views on the components, and commands and tools for manipulating the components. Ideally, a groupware toolkit should provide the same support for creating *shared* graphical editors.

2. Object Concurrency Control. Many groupware systems support access to some sort of shared object, be it structured graphics or a text buffer. Concurrency control is often needed to mediate access to the object, for example in the case where two people try to manipulate the same point on a line (Greenberg, Roseman et al 1992). As will be seen shortly, concurrency schemes have already been a focus for several current generation groupware toolkits, such as RENDEZVOUS (Patterson, Hill et al 1990) and LIZA (Gibbs 1989). Ellis et al. (1991) discuss several ways that concurrency can be implemented, ranging from simple locking to more complicated schemes such as transactions, with the suitability of different schemes depending on the particular application. The ARJUNA system (Parrington and Shrivastava 1988) is one example of using object-oriented techniques to provide for flexible locking strategies, while Chin and Chanson (1991) provide a more general discussion of issues in concurrency and distributed objects. Flexible concurrency control for groupware systems has been identified as important by Dewan (1991), motivating the use of *coupling status* — objects that are private, public, or sharable between users—in systems like GROUPDRAW (Greenberg, Roseman et al 1992) and CAVEDRAW (Lu and Mantei 1991). Finally, a concurrency control algorithm well suited to real-time groupware systems is presented by Beaudouin-Lafon and Karsenty (1992) which uses a selective undo / redo of actions based on application semantics to guarantee messages received in different orders at different sites are processed in an equivalent manner.

3. Separate View from Representation. Many single-user graphical systems separate the underlying structure and properties of an object from its view on the screen. Patterson argues that this separation is critical in groupware (Patterson 1991; Patterson, Hill et al 1990), and that abstractions should be used to create an interface-independent representation of data. As a consequence, users can have multiple perspectives on the same data. The example given is of a card table, where different users see different cards differently (i.e. their own face-up), and the orientation is different for each player. Separating view from representation accommodates different users' needs for the data. For example, view separation in a multi-user CASE tool would allow different ways of viewing data for managers and programmers. Different users of a shared drawing tool could select from a different set of tools for working on the same drawing (Brinck and Gomez 1992).

Existing Toolkits

This section briefly reviews existing groupware toolkits. The toolkits will be reviewed using the requirements detailed in the previous sections. Again, this is not a fair comparison, as the different toolkits were constructed to meet different purposes. However, the intent is to highlight the fact that current toolkits have done well in satisfying programmer-centered requirements, but have limited their attention to the human-centered requirements.

Rendezvous

RENDEZVOUS (Patterson 1991; Patterson, Hill et al 1990) is a toolkit for developing synchronous multi-user applications that has been developed at BellCore. The system is derived from the MEL User Interface Management System, a language extension to Common Lisp providing support for graphics, object-oriented programming, and constraint management. RENDEZVOUS consists of two parts, a *start-up architecture* and a *run-time architecture*. The start-up architecture is used to separate or decouple the chores of registration from the application itself. Users join a RENDEZVOUS conference through the *Rendezvous Access Point*, or RAP. Communications are also handled via the start-up architecture.

However, it is RENDEZVOUS's run-time architecture that is impressive. Its constraints are used extensively to provide different dimensions of object sharing. They are used to share underlying information about objects between different users, maintaining all shared objects in a consistent state. Constraints have also been used to build a What You See Is What I See (WYSIWIS) abstraction for shared objects. RENDEZVOUS makes a clear separation between the underlying object and views on that object, allowing different members of the group to have different views on the same object. Constraints maintain consistency between the underlying object and its views. This is exemplified by a multi-user card game, where each user has a different view of the card table (Patterson 1991). Finally, constraints have been used — although the details are not clear from (Patterson, Hill et al 1990) — to build turn-taking or floor control protocols. Several applications, notably an interface design program based on the PICTIVE (Muller 1991) technique and a suite of drawing tools (Brinck and Gomez 1992) have been developed at BellCore using RENDEZVOUS.

As a toolkit, RENDEZVOUS supplies a strong, well-designed foundation on which to build. Unfortunately, its applications run very slowly, likely because of their reliance on Lisp and an elaborate constraint maintenance system. The communications architecture is centralized, leading to robustness problems when machines fail. In general, the system has not addressed many of the human-centered issues identified earlier in this chapter.

Several future directions for RENDEZVOUS are planned. Out of band communication channels, such as telephone and video links using the CRUISER system (Root 1988), are being more closely integrated into the RENDEZVOUS conferencing system. The graceful incorporation of conference latecomers into a session (potentially via summary information) is also being addressed, as are persistent sessions. Finally, the developers are looking at alternate ways of maintaining structural consistency between underlying objects and views, suggesting that the constraint-based mechanisms currently used are not as powerful as required.

MMConf

MMCONF is a tele-conferencing toolkit whose primary focus is the smooth integration of single-user applications into group conferencing situations (Crowley, Baker et al 1990). However, facilities are provided for the development of pure multi-user applications. The toolkit, implemented under Unix, contains several components for multi-user conferences. Each user is assigned a *conference manager* to oversee all aspects of basic conference management and communications. The conference startup process is limited in that the user creating the conference must at startup time specify all other users (by user and host name) that will be part of the conference. There are plans to provide a more general startup architecture that would permit latecomers.

Single-user applications are easily incorporated into a conference. MMCONF takes special care with these applications, changing their behavior when used by a group. For example, while continuous scrolling works well in single-user applications, it does not transfer well to group situations. MMCONF disables continuous scrolling for such applications when used by groups. Similarly, customizations made by individual users are ignored, and the system makes an effort to provide each user with local copies of data being used by the conference.

MMCONF has provisions for floor control, with several different policies implemented. These policies are based on flexible control of a single token, which can be passed to various users to establish a floor. The token may also be ignored, allowing a free floor policy to be adopted. While this implementation scheme does provide a number of useful protocols, it is not completely general, by not permitting several users (but not all) to have control of the floor.

Several applications have been built using MMCONF. The BBN/SLATE Document Editor is provided for editing multi-media documents that may contain voice or video information. A shared terminal emulator is provided called VIEWSHELL, along with a tool for creating slide shows (PRESENTER), and the VIDEO INFORMATION SERVER for sharing video resources among workstations. A simple graphics editor called SKETCH is also provided, which can be used as a normal drawing tool or to annotate snap-shots of conference windows.

Conference Toolkit

CONFERENCE TOOLKIT (Bonfiglio, Malatesa and Tisato 1989) is part of the ESPRIT Multimedia Integrated Workstation project. The toolkit is primarily concerned with basic connectivity requirements, permitting use of traditional single-user or specially designed conference aware applications. The toolkit, developed under Unix and the NeWS windowing environment, consists of several communicating processes. A *Multiplexer* process manages the communication between other processes, mediating the sharing of data. Multiplexers can be designed to optimize sharing of application-specific data, or can handle general transmission (e.g. voice). A *Bridge Manager* controls the behavior of a single conferencing applications, in particular negotiating which users and applications can communicate with the controlled application. This is done using floor control strategies — both for users and applications — which can be chosen from several different strategies provided by the toolkit. Finally, a *Conference Manager* collects the various pieces of the conference, including the set of connected users, the set of shared applications (i.e. bridges), and a number of conference attributes and data channels used to control the conference's behavior.

A prototype application, the CONFERENCE DESK, has been built using the toolkit. The CONFERENCE DESK supports a shared terminal emulator, a simple graphics viewing surface, integration of a voice channel, as well as an interface to the Conference Manager. Arbitrary single-user NeWS applications could also be inserted into the conference. As well, two conference aware applications have been built, the VOICE BLACKBOARD and the SLIDE MULTIPLE PROJECTOR.

Though much detail is provided on the technical aspects of managing the communication channels, there seems to be little focus on higher level components for groupware. Further, some details are left undiscussed. In particular, it is not clear if new features can be easily integrated into the toolkit, although object-oriented subclassing can be used for adding new components similar to existing components. Though the toolkit does not impose a particular process of interaction for the group, it is unclear how software-based group processes *could* be added. Finally, the status of the project is unclear, and the future work noted seems more involved with supporting technical problems (e.g. adding support for X as well as NeWS) rather than looking at the facilities for conferencing itself.

Liza

LIZA is an extensible groupware toolkit developed at MCC (Gibbs 1989). The conferencing applications built using LIZA consist of *active objects*, based on the notion of Actors (Agha 1986). These applications, called tools, run as clients to a single LIZA server running on each machine. Communication between the various tools is done via Unix sockets. Several tools have been developed using the LIZA system including: a ROOM TOOL (showing where the session participants are located), a GRAPH TOOL (allowing editing of a shared graph), a SLIDE TOOL (for viewing a slide show), and a VOTE TOOL (allowing voting on issues).

Unfortunately, Gibbs (1989) provides few details on the facilities and architecture provided by LIZA, and little can be concluded. It does appear that basic support is provided for conference management and communications mechanisms, but there is nothing said about higher level groupware constructs.

Other Systems

Some other systems have been described in the literature which, though not complete toolkits, provide valuable discussion of issues and techniques that could be incorporated into future toolkits. These include: SUITE which considers multi-user primitives, MMM which investigates fine-grained editing, GROUPDESIGN which provides object concurrency, GROUPDRAW which looks at issues for shared drawing, and SHARE, which discusses flexible floor control.

SUITE (Dewan 1990; Dewan and Choudhary 1991) provides primitives for programming multi-user interfaces, based on single-user interface primitives. Facilities have been provided for authenticating users, setting properties and executing code in a user's environment. As well, "coupling" or the degree of access to shared objects, has been explored, allowing different coupling strategies to be adopted in different situations.

MMM (Bier and Freeman 1991) is a user interface architecture allowing several users to share a single screen by attaching several input devices to a single machine. A prototype system has been developed allowing fine-grained editing of simple rectangles or a text buffer. The system looks at strategies for sharing both the devices and the shared objects, as well as handling user preferences within a single system. The ideas presented have potential to be developed into a useful architecture for fine-grained editing of objects.

GROUPDESIGN (Beaudouin-Lafon and Karsenty 1992) provides a general object concurrency algorithm the authors argue is suitable for a wide variety of groupware applications. The system, based on a single-user structured graphics editor, also addresses several human factors concerns.

GROUPDRAW (Greenberg, Roseman et al 1992) is a prototype system for shared object-based drawing.. A detailed description of the architecture for shared objects is discussed, using object-oriented techniques to provide behavior for an abstract object that can be inherited by specific programmer-defined objects. This behavior could be used to build general concurrency control and persistence for example.

SHARE (Greenberg 1990; Greenberg 1991) is a system for sharing terminal emulators. It provides a flexible and extensible notion of floor control, and the strategy described can be generalized to a number of uses beyond just floor control. The papers provide details of both the architecture and the implementation of several protocols.

Comparative Evaluation

Table 2.3 summarizes the support provided by RENDEZVOUS, MMCONF, CONFERENCE TOOLKIT, and LIZA with respect to the five design requirements presented earlier. Note that this summary is based on the features described in the papers cited previously, allowing the possibility of additional features that have been developed since publication. Further, note that all the toolkits were not developed for exactly the same purpose, and the implementation emphasis in each is different. Thus, a detailed comparison, based on a single set of standards, is at least to some degree hiding many issues and ideas addressed by the toolkits' developers.

In general, RENDEZVOUS has done an excellent job, particularly in their support of a shared graphics model, emphasizing flexibility and strong view separation using constraints to manage the dependencies in the model. MMCONF provides many interesting tools, yet significant effort has not yet been spent on the toolkit as a whole. CONFERENCE TOOLKIT, though providing a reasonable communications architecture, has not yet addressed many of the issues important for groupware development beyond simple connectivity. Finally, LIZA seems to again provide connectivity with little higher level support, though again, (Gibbs 1989) is not specific.

Overall, the current generation of synchronous groupware toolkits have done a great deal to provide basic connectivity. Higher level human-centered support, however, is scarce. Though MMCONF does provide several tools, they seem to be prototypes, rather than smoothly integrated into a style of developing groupware applications. Most of the other toolkits seem to have this problem as well. RENDEZVOUS has by far the most consistent and thoughtful design philosophy, and as this is applied to other areas of groupware support, shows excellent promise for the future.

Requirement	RENDEZ-VOUS	MMCONF	CONF. TOOLKIT	LIZA
Human-centered Requirements				
Actions over visual work surfaces				
convey process	+	o	o	o
simultaneity	+	+	o	o
modeless interface	.	?	o	o
common view	+	o	o	o
gestures	.	+	.	.
annotation	o	+	.	.
Flexible group processes				
floor control	+	++	+	?
registration	.	.	.	.
latecomers	.	.	.	.
Integration with conventional work				
non-computer artifacts	.	.	.	.
other communication	+	+	.	.
single-user apps	?	++	+	.
Programmer-centered requirements				
Multiple distributed processes				
conference management	+	+	+	+
communications infrastructure	+	+	++	+
persistent sessions	.	?	?	?
Shared graphics model				
shared visual objects	++	.	.	.
object concurrency	++	.	.	.
separate view	++	.	.	.

++	system provides good support
+	system provides support
o	system can handle but does not specifically support
.	system does not support
?	not clear if support is provided

Table 2.3. Summary of features provided by toolkits.

Summary

Groupware toolkits should be specifically designed to provide support for the types of applications people are building. The CSCW literature, as reviewed in this chapter, has suggested several design features and requirements that are important for synchronous groupware conferencing systems. These include a number of human-centered design requirements, such as supporting actions over a visual work surface, providing capabilities to structure group processes, as well as integrating group work with conventional work. Technical or programmer-centered requirements for groupware conferencing include support for multiple, distributed processes, as well as supporting a shared graphics model.

Contemporary toolkits have generally not provided the desired level of support, concentrating primarily on basic components for connectivity. While this is important, it leaves the application developer to deal with many human-centered concerns for groupware development. Future generations of toolkits, to prove useful, must begin to address these human-centered concerns to provide the level of support that groupware developers require.

Chapter 3

Design and Implementation

The previous chapter motivated the design of a groupware toolkit from both human and programmer-centered perspectives. This chapter describes how a number of the resulting design requirements are instantiated in GROUPKIT. Due to time limitations, the work on GROUPKIT does not cover all the design principles outlined in Chapter 2. Future work, described in Chapter 6, will result in more of the design requirements being satisfied, and further design requirements being derived.

In particular, this chapter discusses how the design requirements are satisfied, concentrating on descriptions of the GROUPKIT components, and how they would be used by application developers building groupware systems. Chapter 4 will provide some insights into *why* these components were designed in the way they were, and list the critical design decisions made.

GROUPKIT is written as a C++ class library, based on the INTERVIEWS toolkit (Linton, Calder and Vlissides 1988). Applications built using GROUPKIT run on any machine supporting INTERVIEWS, that is, Unix workstations running X-Windows. GROUPKIT relies on the INTERVIEWS glyph mechanism for its user interface constructs. Glyphs are lightweight objects (similar to widgets in other toolkits) that are composed to make interfaces. The INTERVIEWS Dispatch library — a front end to Unix sockets — is used as the backbone for communications.

The chapter opens by describing what an end user working with a GROUPKIT program would see. Later sections discuss the three sets of components implemented in

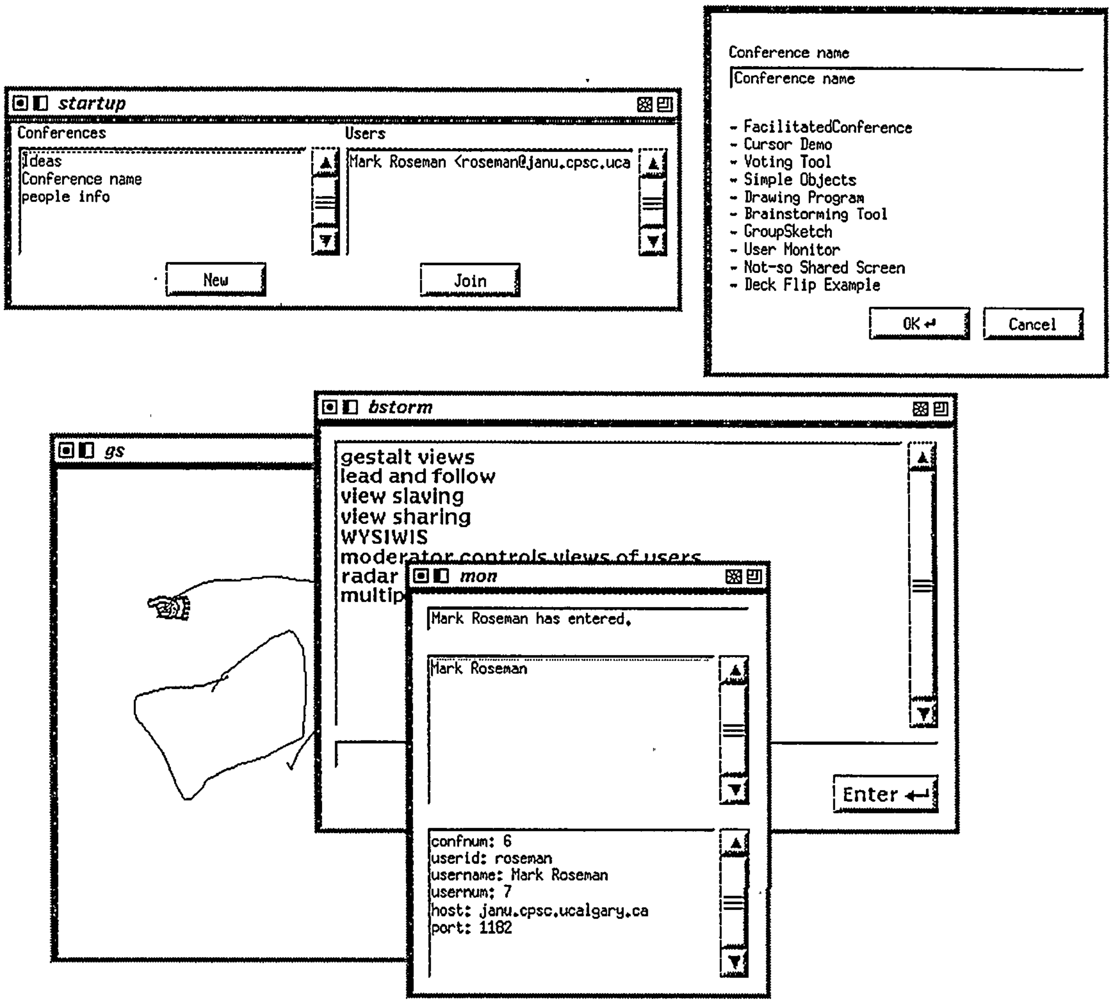
GROUPKIT. The first is the communications infrastructure and messaging scheme, which provides the facilities necessary for basic conference management, including registration, communication, and conference termination. Second, overlays provide an abstraction to easily add support for generic work surface activities, such as gesturing or annotation, to groupware applications. Finally, open protocols provide a scheme whereby group processes, such as for floor control or conference registration, can be encoded into groupware applications in a flexible way, accommodating different groups who require different working styles.

End User Perspective

To provide a context for the detailed discussion of underlying GROUPKIT components in the rest of the chapter, this section provides a brief description of what a typical end user of a GROUPKIT application would see.

Figure 3.1 shows a screen dump of a typical GROUPKIT usage scenario. The window at the top left of the screen labelled "startup" is a *Registrar Client.* A GROUPKIT user begins a session by running a Registrar Client program, of which there are several different kinds. All of them serve the same basic purpose, which is to let the user find out about groupware programs (called *Conferences* in GROUPKIT terminology) that other users are running, or to run new groupware programs. The Registrar Clients also allow users to join and leave groupware Conferences.

Each Registrar Client can be responsible for a number of different Conferences running at the same time. In the figure, three Conferences are running, as shown in the "Conferences" list of the Registrar Client. The dialog box on the top right of the screen dump (the untitled window) would allow the user to create new groupware Conferences.



```
Ideas
Conference name
people info

Mark Roseman <roseman@janu.cpsc.uca
```

```
Conference name
Conference name

- FacilitatedConference
- Cursor Demo
- Voting Tool
- Simple Objects
- Drawing Program
- Brainstorming Tool
- GroupSketch
- User Monitor
- Not-so Shared Screen
- Deck Flip Example
```

```
gestalt views
lead and follow
view slaving
view sharing
WYSIWIS
moderator controls views of users
radar
multip
```

```
Mark Roseman has entered.

Mark Roseman

confnum: 6
userid: roseman
username: Mark Roseman
usernum: 7
host: janu.cpsc.ucalgary.ca
port: 1182
```

Figure 3.1. End-user view of GROUPKIT applications.

To do so, the user gives the Conference a name and selects the type of Conference from the list of available types.

Finally, the three windows at the bottom of the screen dump are the three groupware Conferences that the Registrar Client is responsible for running. The three are a groupware sketchpad program (in the "gs" window), an anonymous group brainstorming tool (in the "bstorm" window) and a window to display information about Conference users (in the "mon" window). Though these programs were started by the Registrar Client, they run independently of it and each other.

The remainder of this chapter discusses the three sets of software components used by the application developer in constructing groupware applications with GROUPKIT.

Communications Infrastructure and Messages

A fundamental programmer-centered requirement is the provision of facilities to support multiple and distributed processes. GROUPKIT provides a communications infrastructure that supports this requirement. It offers basic conference management, including participant registration, conference maintenance, and conference teardown, as well as mechanisms for simple interprocess communication.

Overview

GROUPKIT applications consist of several processes, arranged in a distributed or replicated architecture on a number of machines, as illustrated in Figure 3.2.

The components are now briefly described in turn, providing a "process view" of the communications architecture. The example scenario following these descriptions elaborates on their behavior.

Registrar. The central Registrar provides the master record used to locate Conferences. It maintains a list of all the conferences active on the system, including their users, and responds to requests from other processes. The Registrar itself implements no policy on how conferences are created or deleted, or how users join or leave them. Commands are provided so that other processes can manipulate the lists, registering or deleting conferences and users. The Registrar is invisible to the end user.

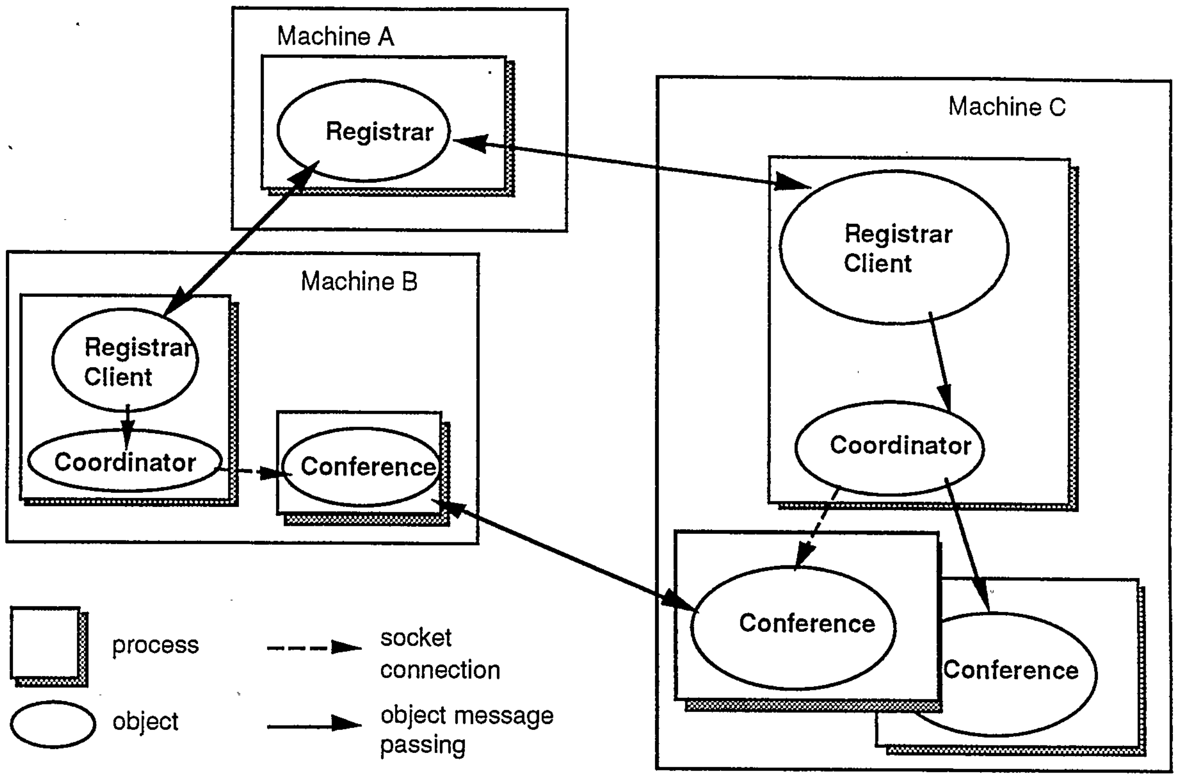


Figure 3.2. Communications infrastructure of GROUPKIT. Here, objects owned by one user (rightmost Registrar Client, Coordinator, and Conference) interact with objects owned by another use, as well as the central Registrar.

Registrar Client. The Registrar Client (one per user) allows users to create, delete, join or leave conferences. It interacts with other Registrar Clients through the central lists provided by the Registrar. The Registrar Client provides both a user interface as well as a policy dictating how conferences are created or deleted and how users enter or leave conferences. Different Registrar Clients can be created to suit different registration needs, and can even interact with each other to produce complex registration schemes.

Coordinator. The Coordinator (one per user) acts as an interface between application conferences and the Registrar Client that created them. The Coordinator responds to the registration system by creating and then maintaining connections to any number of conferences, allowing them all to share the same registration mechanism. The Coordinator is invisible to the end user.

Conference. The Conference is the heart of the groupware application itself, separate from the registration system. Spawned by the Coordinator object, the Conference application maintains communication facilities necessary for exchanging messages with Conferences run by other users. The user interface portions of groupware applications use these facilities extensively.

Example Scenario

To illustrate the interactions of the infrastructure components, this section provides an example usage scenario. The scenario consists of four parts: registration, conference initiation, conference maintenance, and conference termination.

Registration. Suppose a user wants to discuss a design problem with a remotely located colleague through a shared drawing surface . To initiate the GROUPKIT-based program, the user first creates a conference. This is done through the Registrar Client, which provides an interface to the central Registrar.

The Registrar allows the user to create, join or leave one or more conferences. The Registrar Client is responsible for implementing particular registration policies, e.g. deciding who enters the conference, how they do so, and what the interface looks like. Figure 3.3 shows the interface for one Registrar Client (other clients are possible). The communications interface between the Registrar and the Registrar Client is described in detail in the "Open Protocols" section.

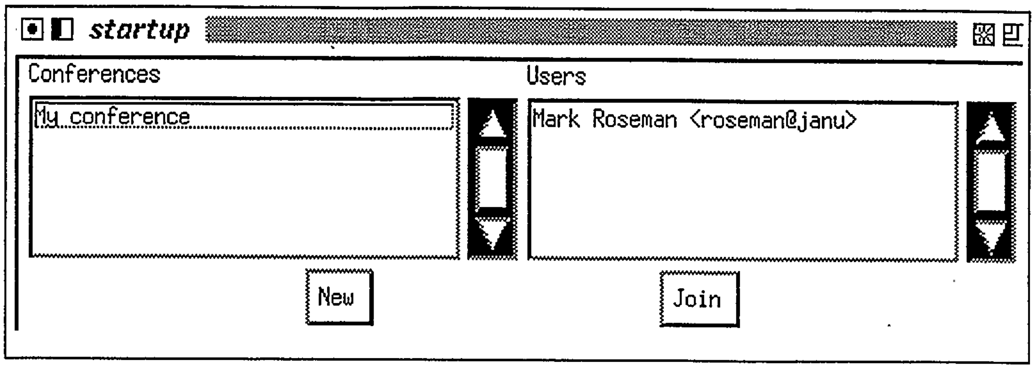


Figure 3.3 Interface for a typical Registrar Client.

The Registrar itself is an independent process (invisible to the user) that maintains a list of all conferences and their users. One central Registrar would exist at each installation. The Registrar itself is *policy-free*, and leaves it to the Registrar Clients to implement a particular registration policy and to present a reasonable interface to the user. This allows different policies to be implemented to accommodate group differences.

While GROUPKIT allows new Registrar Clients to be programmed, it also provides a library of predefined Registrar Clients implementing particular registration policies and user interfaces. For example, users may choose a Registrar Client allowing any user to enter any conference. On other occassions, users may choose a Registrar Client where new users must be "approved" before joining an existing conference. One novel aspect of this scheme is that it allows group members to use different Registrar Clients to enter a single conference.

Conference Initiation. In the scenario, the user has just requested a new conference through the Registrar Client, which in turn passes the request on to the Registrar. Next, the Registrar Client asks the Coordinator to create a new Conference object. The Coordinator acts as an intermediary between the Registrar Client and application

Conferences, permitting multiple conferences (e.g. sketching and editing applications) to share a common registration mechanism. Its main duties are to create Conferences at the request of the Registrar Client, and to forward information (such as new users joining) from the Client to the appropriate Conference.

The Coordinator spawns a new process containing a Conference object. This could be the shared drawing surface. The new Conference object connects back to the Coordinator, so that messages from the registration system (such as announcements of new users) can be received by the Conference object. It is the Conference object that actually runs the specific groupware application. GROUPKIT provides a generic Conference object, handling users joining and leaving, and providing support for interprocess communication. In the scenario, the generic Conference object is combined with a programmer-defined graphical interface supporting a shared drawing surface. The interface will rely heavily on the communications facilities provided by the Conference object.

Figure 3.4 provides a timing diagram illustrating the conference initiation phase of the scenario. Two Registrar Clients begin connected to the Registrar. One of the Registrar Clients creates a new Conference. It does so by sending a "create conference" message to the Registrar which is echoed to all Registrar Clients. It then asks its Coordinator to spawn the new Conference process, which connects back to the Coordinator to receive further information. At the same time, the Registrar Client sends a "new user" message to the Registrar in order to join the newly created Conference. Again, this is echoed so that all Registrar Clients are kept synchronized.

Conference Maintenance. Other users can also create Conference objects. As each Conference locates existing participants via the Registrar, communications channels are opened between all Conference objects. Facilities in the generic Conference are provided

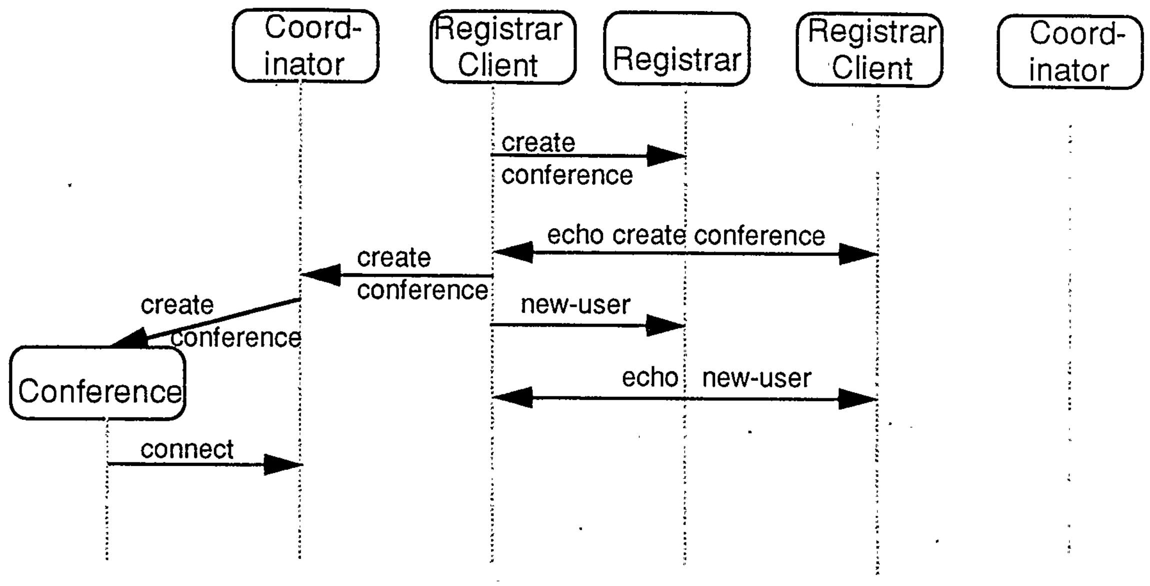


Figure 3.4. Object interactions during conference registration and initiation.

for exchanging messages with other user processes. Users within each conference are identified by a unique integer "user number" assigned by the Registrar.

Communications between distributed processes are maintained by messaging objects. The two types of messaging objects ("Writer" and "Reader") provide a convenient method of communicating with processes owned by remote conference participants. These objects, derived from the INTERVIEWS Dispatch library, provide a primitive Remote Procedure Call (RPC) facility. Writer objects specify messages they can send, while Reader objects provide functions to be called when particular messages are received.

Figure 3.5 illustrates the maintenance phase of the scenario. The other user joins the previously created conference, as before by sending a "new user" message to the Registrar which is echoed to the other Registrar Client. While the new user's Registrar Client spawns its Conference process, the other Registrar Client initiates a connection to the new Conference. The connection is initially made to the user's Registrar Client, which passes

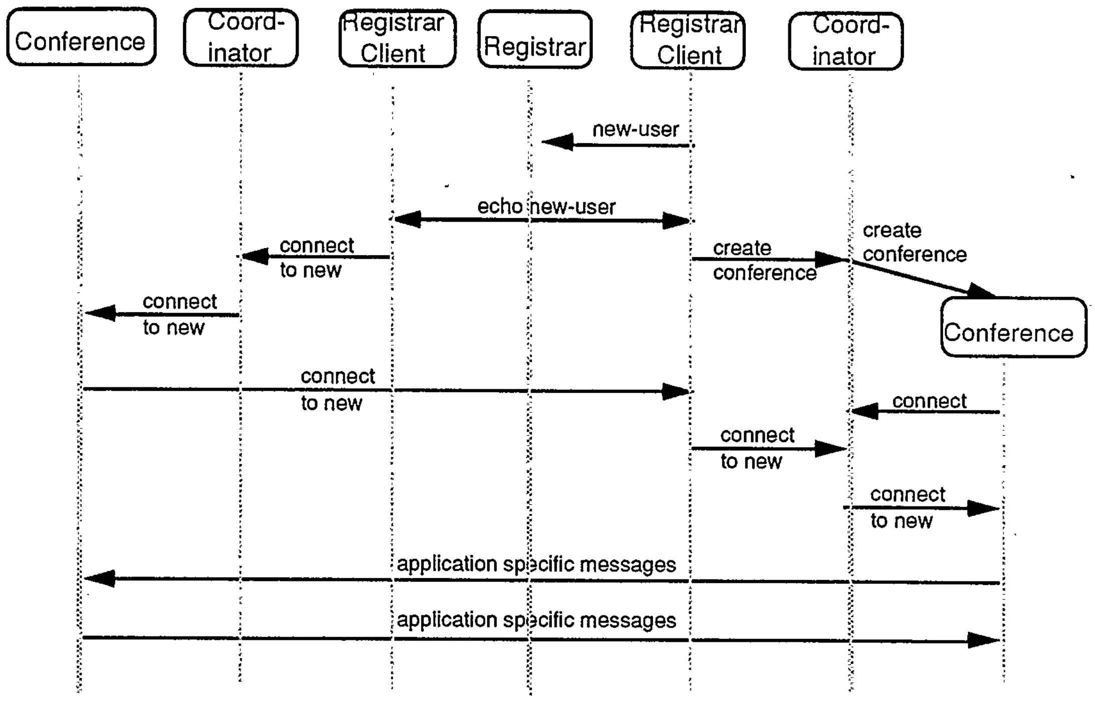


Figure 3.5. Interactions between objects during conference maintenance. A new user joins and messages are interchanged between the two conferences.

the connection via the Coordinator to the Conference once it has been created. From that point, communications between the two Conference objects are established and messages can be transmitted between them.

Conference Termination. As with initiation, conference leaving and termination is handled through the Registrar Client. If a user wishes to leave the conference without terminating it, their Registrar Client sends the "delete-user" message to the Registrar. Some Registrar Clients may permit explicit conference termination, allowing any user to end the conference, while more typically the Registrar Client for the last user to leave will end the conference. This is done by sending a "delete-conference" message to the Registrar, which will be broadcast to any remaining users' Registrar Clients.

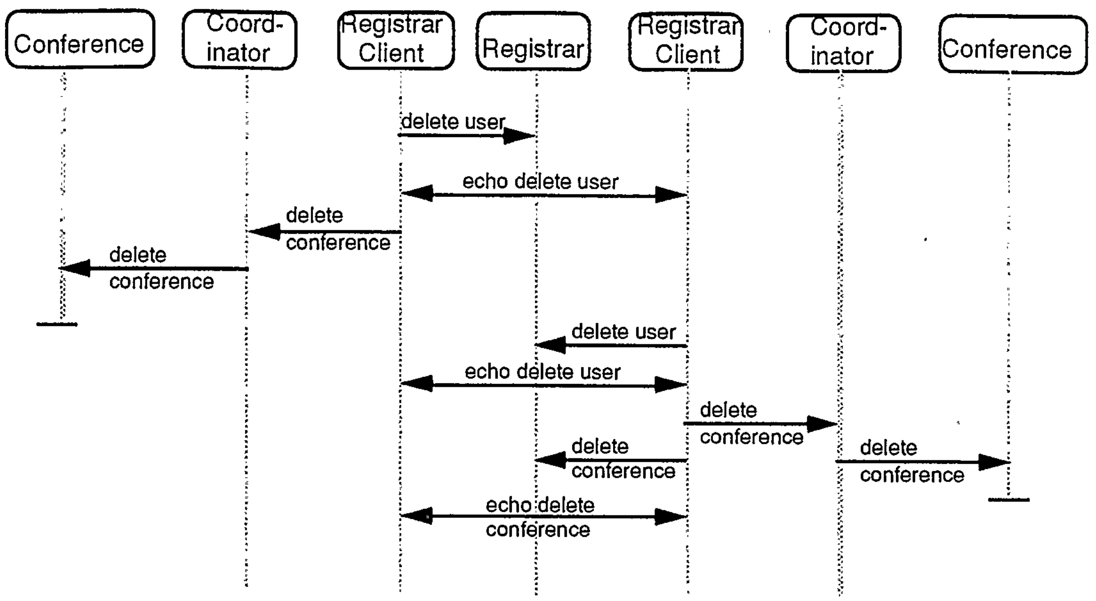


Figure 3.6. Interactions between objects during conference termination.

Figure 3.6 illustrates the conference termination phase of the scenario. Both users quit out of the Conference, by sending a "delete user" message to the Registrar. When this is echoed back, the Coordinators are instructed to delete the Conference, which is in turn delegated to the Conference, which deletes itself. The last user's Registrar Client, seeing it is the last user in the Conference, deletes the entry for the Conference by sending a "delete conference" message to the Registrar.

Messages

Messages are used to communicate between different applications. A message consists of a "message-type" which identifies the message, and an "option-string" which provides any information needed by the message-type. Message-types are just integers, and while a number are pre-defined, others can be trivially added. Option-strings are normal character strings, in any format, provided of course that the sender and receiver agree on the same format.

```
char option_string[80];

sprintf ( option_string, "%d:%d:%d:%d", x0, y0, x1, y1 );

conference->connections()->sendTo ( remote_user_number,

        new StrMsgSender ( DRAW_LINE, option_string ) );

conference->connections()->toAll (

        new StrMsgSender ( ERASE_LINE, option_string ) );
```

Figure 3.7. Code fragment for sending messages.

To send messages, a "ConnectionList" is provided as part of the Conference object, which maintains a list of Connections to all other conference users. The ConnectionList provides methods for sending messages to a single user (specified by their unique user number in the conference) or to all users of the conference. Messages are sent using special "Message-Sender" objects, which can send the same message over several Connections. Figure 3.7 illustrates sending messages to a single user or every other user in the conference.

To receive messages, a "callback" routine must be set up, so that whenever a particular message (of a given message-type) is received, the routine will be called, with the message's option-string passed as a parameter to the routine. The ConnectionList maintains a table of callback routines for different message-types, so that adding a callback is done using the code fragment in Figure 3.8.

Attribute Lists

While option-strings can be constructed and interpreted manually using standard string processing commands, GROUPKIT provides a higher-level construct, *attribute lists*.

```
conference->connections()->callbacks()->insert(
  DRAW_LINE,
  new StrActionCallback(Sketchpad)
      ( my_sketchpad, &Sketchpad::DrawLine ) );
```

Figure 3.8. Code fragment to add a callback for receiving messages.

Attribute lists maintain a list of <attribute,value> pairs, where all the attributes in a given list are guaranteed to be unique. Attributes can be added or removed from lists. This provides a general data structure that can describe a wide variety of objects or actions.

More relevant here, the attribute lists can be converted back and forth to strings in order to be transmitted. The conversion process takes into account special characters within the <attribute,value> pairs. Because messages often consist of commands accompanied by a number of parameters for the commands, attributes are a natural encoding for sending and receiving messages. Using attribute lists simplifies the programmer's task, removing the need for lengthy conversion code, as well as providing a generally useful abstract data type. Figure 3.9 shows examples of creating, encoding, and decoding attribute lists.

Monitoring the User List

By default, all changes to the conference's user list, such as when users first enter or when they leave, are automatically handled by the Conference object, so that the application developer need not explicitly be concerned with the changes. However, under many circumstances the application developer would like to be informed of users coming and going. A special "ConferenceMonitor" class can be used to receive notification about users entering or leaving the conference.

```
char option_string[1000], id[80];
AttributeList* al = new AttributeList();  // create attribute list

al->attribute("id", "123");              // add the pair <id,123>
al->attribute("name", "Mark" );          // add <name,Mark>
al->attribute("temp", "blah" );          // add <temp,blah>
al->attribute("id", "45" );              // replace <id,123> by <id,45>
al->remove_attribute("temp");            // remove <temp,blah>

al->write(option-string);                // encode list to string

AttributeList* al2 = AttributeList::read(option-string);
                              // decode the string into a new list

if (al2->find_attribute("id", id))         // if attribute in list
  printf("id is %s\n", id);                //   print out value
else printf("id not in list\n");
```

Figure 3.9. Code fragments showing use of attribute lists.

By registering a ConferenceMonitor with the Conference object, the programmer can be notified any time users enter or leave the conference. The ConferenceMonitor must define a “newUser” method that will be notified and receive information (as an attribute list) about any new users, while a “userLeaving” method is notified when users exit the conference. This is illustrated in Figure 3.10.

Implementation Notes

The communications facilities are based on the INTERVIEWS Dispatch library, a front-end to standard Unix TCP sockets. Writer objects are a simple extension to the INTERVIEWS “RpcWriter” class, adding a method to send messages having a particular message-type and

```
class Sketchpad : public ConferenceMonitor {
  // class definition...
};

Sketchpad::Sketchpad() {
   // tell the conference we want to monitor;
   // this sets up a callback so the routines below will be called
  conference->monitors()->append(this);
}

void Sketchpad::newUser (AttributeList* new_user_attrs )  { ... }
void Sketchpad::userLeaving( int leaving_user_id ) { ... }
```

Figure 3.10. Code fragment for monitoring the conference user list.

option-string. Reader objects extend the "RpcReader" class to use an arbitrary table for callbacks rather than an array, supporting discontinuous message-types. Callbacks for communications are specified in the same way that INTERVIEWS handles callbacks from interface components, using "Action" subclasses.

Overlays

GROUPKIT provides components that may be included in any conferencing application through an *overlay* strategy. Overlays are transparent "windows" placed on top of the main application graphics. Graphics in the overlay sit atop the application, without disturbing what is underneath. They are intended to satisfy the human-centered requirement of providing multi-user actions over a visual work surface. Currently, overlay components have been implemented for gestural communication (via multiple cursors over a surface) and annotative communication (via freehand drawing over a surface). The motivation is that such components could be useful for a variety of groupware applications, as discussed in Chapter 2.

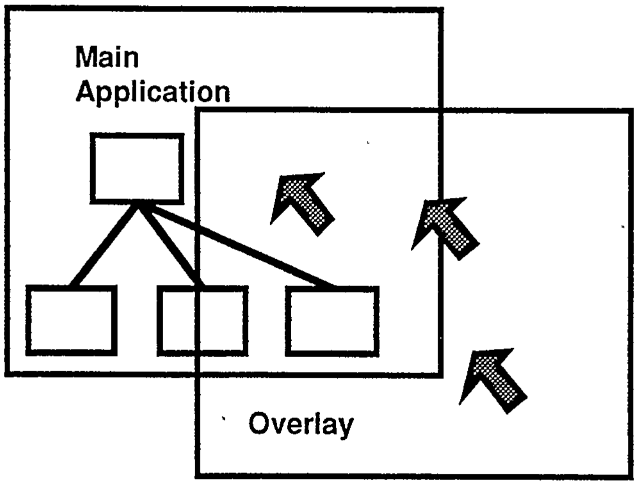


Figure 3.11. Adding a multiple cursor overlay to an application.

Figure 3.11 provides a conceptual picture of adding an overlay for displaying multiple cursors to an existing application graphic, here an organizational chart. The transparent cursor overlay is written as an INTERVIEWS glyph that overlays any other glyph. Neither the cursor glyph nor the main application glyph need any knowledge of the other. Local input events are received by the cursor glyph, which updates cursors as necessary. The local event (e.g. mouse move) is then passed to the application glyph, to use as needed. The application glyph need not even know the event went through the cursor glyph. As with single-cursor window systems, event-driven drawing operations are performed normally by the application glyph, based on these events, with the cursor glyph sketching the cursors on top of the normal graphics. Figure 3.12 illustrates this flow of control for both event handling and drawing.

To incorporate the multiple cursor overlay into an application, the programmer instantiates a "CursorOverlay" which surrounds the application's graphic. Code in the overlay's constructor allows the CursorOverlay — using the communications features discussed earlier in the chapter — to communicate with remote conference objects and ask about new

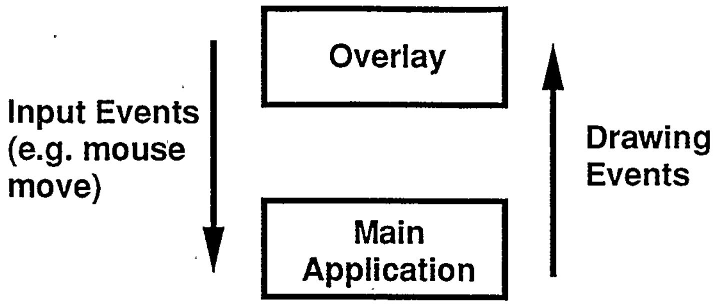


Figure 3.12. Flow of input events and drawing operations with overlays.

conference users. However, the application programmer need not be concerned with these details which are hidden within the CursorOverlay. Figure 3.13 shows a code fragment illustrating how a CursorOverlay is added to a GROUPKIT program. The underlying application need not even be aware that the overlay is present, but can continue to draw and process events normally.

An overlay has also been implemented to support primitive annotation, via bitmap sketching over work surface artifacts. Users can create simple sketches — a series of line segments — using the mouse, and such sketches are immediately transmitted to all other users. Further, overlays can be combined, so that multiple cursors can overlay bitmap sketches that in turn overlay a main application. Figure 3.14 illustrates a simple program combining the two overlays over an empty work surface, resulting in an application (see Figure 3.15) resembling simple bitmap sketchpads such as GROUPSKETCH (Greenberg and Bohnet 1991) or COMMUNE (Minneman and Bly 1991).

```
new ApplicationWindow(                          // create a window
  new CursorOverlay(                            // create an overlay
    new MyConfGlyph( style, conference )        // graphics under overlay
    , style, conference ) )
```

Figure 3.13. Adding a multiple cursor overlay to a GROUPKIT program.

```
int main(int argc, char** argv) {
  GroupSession* session = new GroupSession("GroupSketch", argc, argv);
  Conference* confer = session->conference();    // Conference object
  session->run_window(         // start program running
    new ApplicationWindow(     // in this window
      new CursorOverlay(       // outside layer is gesturing overlay
        new Sketchpad(         // inside that is annotation overlay
          new EmptyGlyph(session->style(), confer, 300, 300)
                               // and very inside is empty graphic
          ,session->style(), confer)
        ,session->style(), confer) ) );
}
```

Figure 3.14. Code to produce a simple sketchpad using overlays.

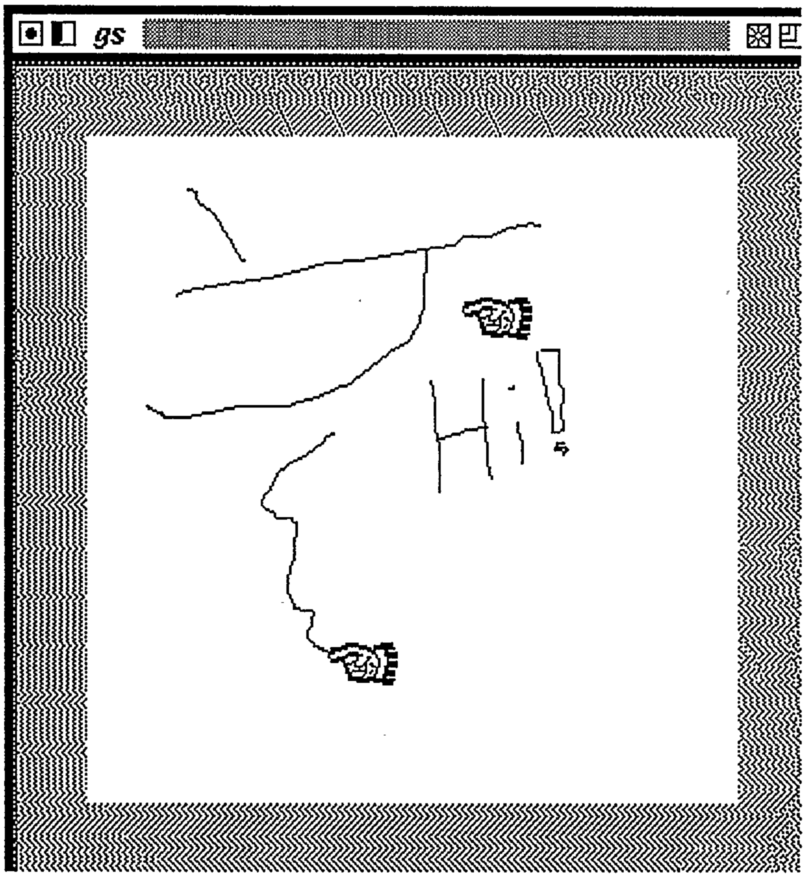


Figure 3.15. Simple sketchpad created using overlays.

This overlay technique seems promising. Through the composition mechanism, adding overlay components to applications is straightforward. As well, the overlays are kept separate — conceptually and in the code — from the underlying applications. The technique's strength is that the overlay does not interfere with the underlying graphics of the application, even if those graphics are changing. Because of this, it is a trivial matter to add, for example, annotation capabilities on top of a "live" shared terminal application. Unlike other systems that only allow annotation of static screen snapshots, e.g. MMCONF (Crowley, Baker et al 1990), with GROUPKIT's overlays the underlying application can be fully active. It is expected that further research will suggest other components that could be transparently added to a variety of conferences.

Implementation Notes

A virtual "Overlay" class is used to create a glyph that passes events to child glyph's after handling the events itself. This is done through extensive changes to the "pick" method of the glyph, which handles hit detection. Subclasses of Overlay handle the drawing and layout themselves, using other INTERVIEWS components. For example, the "CursorOverlay" uses an INTERVIEWS "Page" glyph, allowing arbitrary placement of the cursor bitmaps over a background glyph, the underlying application. The "Sketchpad" on the other hand uses the "LayoutKit::overlay" to control its layout and drawing.

Open Protocols

One human-centered design requirement for GROUPKIT is to provide flexible policies where appropriate, allowing group processes to be structured during a meeting, and to accommodate group differences. *Open protocols* provide a means of implementing this flexibility in software.

Open protocols have three components: a *controlled object* that maintains a state, a *controller object*, and a *protocol* describing how the two communicate. The controlled object's behavior does not incorporate any policy determining how its state can be manipulated. Instead, a protocol is defined for manipulating the state, and the controlled object obeys any external requests made to it to change its state. The controller is an external object that implements a particular policy by the requests it sends to the controlled object. This is in contrast to most client / server architectures, where access policy resides in the server, not the client. Using open protocols, the responsibility is for the client (controller), not the server (controlled) to implement access policies; the server merely obeys a set of general commands according to its protocol.

Open protocols are a generalization of work on floor control in the SHARE system (Greenberg 1991). Under that system, floor control was used to mediate access to a shared terminal — "floor holders" could type to the shared screen, while those not holding the floor could only observe. Rather than implementing a small, fixed set of floor control policies into the system, a protocol was defined whereby independent modules could be attached to the system to manage floor control. The shared terminal itself maintained a flag for each user in the conference. If the flag was set, the user could type to the shared terminal, while if the flag was reset, the user could not type. A protocol was defined whereby floor control modules could attach to the shared terminal and set or reset the flag for any user. This scheme allowed a wide variety of floor control policies to be built, including round-robin, free floor, preemptive, and explicit-release. The floor control modules were developed independently from the main shared terminal, and others could be added dynamically. Under this scheme, the shared terminal acted as the controlled object, maintaining state information in the form of the flags for each user. The floor control modules acted as the controller objects, specifying the pattern of state changes of the flags

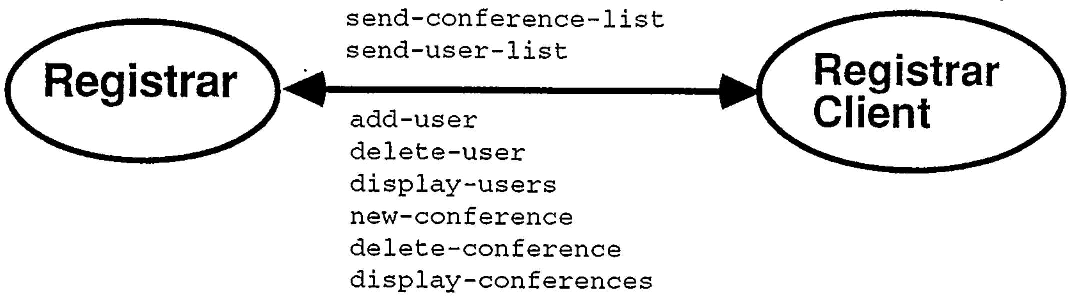


Figure 3.16. Open protocol between Registrar and Registrar Client.

in the shared terminal. Finally, the protocol between the shared terminal and the floor control modules was open in the sense that it supported a wide variety of different policies, defined by the floor control modules and not the shared terminal.

As an example of the use of open protocols, GROUPKIT defines open protocols to handle registration for conferences. As described earlier in the chapter, GROUPKIT decouples registration from the main groupware application. A central Registrar maintains lists of conferences and their users, while different Registrar Clients can connect to the Registrar to alter these lists. Here, the Registrar is the controlled object, and the Registrar Client is the controlling object. The two objects, and the protocol by which they communicate, are shown in Figure 3.16.

The Registrar responds to any request from its clients, broadcasting the result to all attached Registrar Clients. This allows any client to ask the Registrar to create a new conference, or conceivably even to delete any user from any conference. While this does make it possible to create a "super-user" version of the Registrar Client, it also provides the flexibility to create any number of other Registrar Clients interfacing to the Registrar, without making any changes to the Registrar itself.

As an example, consider the implementation of a *free* registration policy. Under such a policy, any user may create a conference, and users can join any existing conference, as in Figure 3.3. The implementation here is straightforward. To join an existing conference, the Registrar Client sends an "add-user" message to the Registrar, which is broadcast to the other Registrar Clients in the selected conference. The Registrar Client also requests the Coordinator to create a new application Conference, because under the policy, the user is guaranteed entry to the conference. The Conference makes connections with the other users, and interaction proceeds normally.

In contrast, a "sponsored" registration policy does not permit new users to join an existing conference unless sponsored by an existing conference participant. Here, the Registrar Client again sends an "add-user" message, which is broadcast to the other users. At this point, the local Registrar Client does *not* ask the Coordinator to create a new Conference. The remote users are asked by their Registrar Clients if the new user should be accepted. A remote user can accept the new user, and sends them a message, prompting the new user to create the conference as before. If rejected (either explicitly or by timeout), the "delete-user" message is sent to the Registrar.

Similarly, a *facilitated* Registrar Client can be created, emulating the registration policies often found in group support systems meetings (Nunamaker, Dennis et al 1991). A central facilitator running one client can create several conferences, such as brainstorming sessions or voting. Users in the facilitated meeting use a different client, which merely obeys the requests from the facilitator's client. The facilitator could cause a user to join a brainstorming session, by sending an "add-user" message for the user and conference to the Registrar. The user's client, receiving the message via the Registrar, would obey the request, creating the application Conference and connecting to other users. The facilitator could then remove the brainstorming session by sending a "delete-user" message for the user and conference. The user's client would again receive this message and obey it, deleting the brainstorming conference. Again, the flexibility of the open protocol permits a wide variety of registration schemes to be implemented.

As with overlays, the open protocol strategy appears useful in general. By providing a simple protocol to change states, building new policies becomes a matter of expressing the policy's semantics in the language of the protocol. While open protocols assume well-behaved clients, their flexibility allows application developer's to more easily build a wide variety of protocols in their system to accommodate different groups' preferences.

Implementation Notes

Open protocols are built on top of the communications facilities described earlier. Attribute lists are used extensively to represent system state — a user or conference is represented by an attribute list, and the protocol provides the means to change this attribute list. A virtual "RegistrarClient" class is provided that handles sending and receiving messages to the Registrar, but does not implement particular policies. Subclasses such as "OpenRegClient" must interpret the messages to implement particular policies. Clients have been implemented for the free registration policy, for a facilitator in a strictly facilitated conference, and for a user in a strictly facilitated conference.

Summary

This chapter has described the design and implementation of several GROUPKIT components that satisfy many of the design requirements described in Chapter 2. Future work on GROUPKIT will extend this subset of requirements.

Three sets of components were discussed. A communications infrastructure provides the developer with components for participant registration, conference maintenance, message passing between processes, and conference teardown. Overlays provide a method to incorporate generic work surface activities such as gesturing and annotation easily into groupware applications. Finally, open protocols provide a means to build very flexible group processes into the groupware application itself, allowing groups to select appropriate modules that reflect their work patterns.

Chapter 4

Design Rationale

This chapter documents many of the design decisions behind the features of GROUPKIT described in Chapter 3. It explains why particular features are designed the way they are, uncovering the underlying design rationale. This material is therefore particularly useful for others developing toolkit support for groupware or other domains.

According to the user-centered toolkit design approach, design should take place with the interests of the developer in mind. The purpose in constructing a toolkit is to provide useful tools for the developer. Therefore it is important not only that features are provided, but that they are provided in a way that is accessible to the developer. Design affordances, introduced in Chapter 1, provide a framework for delivering appropriate and accessible toolkit components to the developer.

Design rationale is considered in terms of four areas: minimizing the developer's work, encouraging the developer to use appropriate features, extensibility of the toolkit, and flexibility of toolkit components. Examples for each of these areas are drawn from the implementation described in Chapter 3. The issues raised suggest strategies for designing other toolkit components that meet the needs of application developers.

Minimize Developer's Work

A goal of any toolkit should be to minimize the work developers need to create their applications, while still producing high quality products. Appropriate toolkit components can encapsulate potentially large amounts of effort into easily used modules. Examples of this in GROUPKIT can be found in many components. Here, three of these are presented, including low-level socket setup and message passing, attribute lists, and the registration system.

1. Sockets and message passing. Communications facilities are non-trivial and time-consuming to develop. One obvious benefit of providing the communications infrastructure is that developers are able to concentrate solely on the specific application, without concern for generic communications tasks such as setting up sockets or sending messages. The infrastructure establishes connections to and from other machines, establishes callbacks for messages, maintains lists of all connected users, and manages broken socket connections.

It is important to note that the provided components work at a high level of abstraction. For example, received messages are automatically dispatched to appropriate message handlers by the system, so that developers need not worry about initial parsing of incoming messages. To send a message, whether to a single user or to every user in the conference, is an atomic operation. Common operations are kept simple.

2. Attribute lists. Attribute lists again raise the level of abstraction so that the developer is not concerned with encoding and decoding sets of parameters transmitted with commands — the option-strings in Chapter 2. As with message passing, this mechanism allows the developer to focus strictly on the task at hand. The utility of attribute lists is based on observations of the type of messages sent by several different groupware systems. Most messages consist of single commands with a number of parameters or attributes associated with the command. Because these attributes often contain numbers and strings with special characters, parsing of the messages is a significant effort. Attribute lists allow encoding and working with such data in a straightforward manner.

3. Decoupled Flexible Registration. In GROUPKIT, the user registration system, in the form of the Registrar Clients is detached from the main application. Clearly, there is great benefit in separating registration concerns from the conference application. If separated, application developers can again concentrate on the functionality specific to their applications. The conference's only concern (much of which is handled automatically) is when users join or leave the conference. This makes it easier to build a variety of different registration systems. Provided the systems at some point resolve registration down to "newUser" and "userLeaving" messages, the application can easily accommodate a great diversity in registration schemes. The fact that the registration system is designed in a flexible way acts as a design affordance — developers are encouraged to build more flexible components or create new registration components to fit their own needs if the situation warrants. Of course, developers can rely on standard clients provided by the toolkit with no extra programming, if this is appropriate for the group.

Encourage Use

While providing appropriate time-saving features is necessary for a toolkit, it is also important to provide those features in such a way that the developer can clearly recognize the need for them. Well designed sets of features with obvious uses — design affordances — can also encourage developers to produce better programs, programs that include important human-centered features or are more robust. Certainly the best example in GROUPKIT is the overlays that encourage developers to include support for work surface activities, such as multiple cursors for gesturing, in their programs. Other interesting examples are attribute lists, which can result in a more robust system, and the toolkit documentation and examples, which can promote a particular style of use. Appealing to familiar concepts and general ease of use issues are also important considerations when encouraging developers to use toolkit features. Issues which arise in GROUPKIT include the use of familiar programming models and minimizing the use of inheritance.

1. Overlays. Overlays are a clear example of a toolkit design affordance. Developers can: recognize the use of overlays — supporting work surface activities; can understand conceptually how they work — operating on a familiar composition model; and because it is straightforward to apply them — typically a single line of code — are more likely to do so when the application would benefit from it. For such technically "non-essential" features such as gesturing, it is important to encourage developers to build them into programs when appropriate. Many early shared drawing programs, which would clearly benefit from the ability to gesture, did not implement gesturing because it was technically difficult. If a component were available to these developers that allowed a quick and easy implementation of gesturing, it is likely gesturing would have been included, arguably resulting in better programs. Thus one main strength of the overlay strategy is that it is not only clear what overlays do and how they work, but that it is very easy to add them to developed applications.

2. Attribute lists. Good programming practices would suggest extensive use of abstract data types. However, in practice it can be tempting to "hard-code" data structures rather than relying on higher-level constructs. Toolkits can and should encourage developers, by using design affordances, to choose higher-level structures whenever possible. The best example in GROUPKIT is the attribute lists. Developers are encouraged to "buy-in" to using attribute lists primarily because they allow data to be transmitted easily, special characters to be handled, and so on. As added benefits, however, using them results in simpler and more robust programs. Attribute lists remove the need for many simple hand-coded data structures (such as are often used for storage), and there is less concern for using uninitialized data structures — the method that returns a value given its attribute explicitly checks if the pair exists. As well, attribute lists encourage expansion. An early prototype of the Registrar used hard-coded lists for names, host numbers, etc. These have all been replaced by a single attribute list, where Registrar Clients can use the list in any way they choose without explicit support from the Registrar. Clients can easily attach any attribute to a conference or user, while in the earlier version adding extra attributes involved either changes to the Registrar or unconventional use of the available data structures. The Registrar code has been simplified, made more robust, yet at the same time is more powerful.

3. Documentation and examples. Documentation is an all-too-often neglected part of toolkit development. Good documentation describing the toolkit can highlight not only how to use particular components, but more important when to use them. GROUPKIT's documentation consists of not only a detailed reference manual explaining the different classes and how they relate to one another, but also a tutorial designed to show the appropriate uses of the components (Roseman 1993). A sample of both the tutorial and reference manual is included in Appendix A. Example programs included with the toolkit are valuable resources, and probably more likely to be used than even the best documentation. Good examples can highlight instances of good application design, which are sure to be remembered when similar new projects arise. The current GROUPKIT software distribution consists of several such sample programs, which are described in Chapter 5.

4. Familiar programming models. The toolkit should strive to build on previously learned concepts wherever possible, rather than forcing developers to learn new concepts. One way this familiarity is exploited in GROUPKIT is in the initial start up of applications. Single-user INTERVIEWS applications rely on a "Session" object to manage program-wide concerns. In GROUPKIT, this is extended so that now a "GroupSession" — a functional superset — manages concerns such as creating new Conference objects, establishing connections to the Coordinator that spawned it, handling GROUPKIT specific resources, and so on. The figures below show a standard INTERVIEWS main program (Figure 4.1) as compared with a GROUPKIT main program (Figure 4.2). Both programs proceed by creating an overall session, and then display a window containing a particular interface glyph. The glyphs ("HelloWorldGlyph" and "MultiUserHelloWorldGlyph") are created by the programmer to handle the graphical display and interaction for the application. Exploiting already familiar concepts can ease the developer's burdens in learning a new system.

5. Inheritance. Ousterhout (1991) makes the claim that object-oriented inheritance is a useful model to support interface toolkit developers, but not to support application developers who use the toolkit. Instead, he argues for composition of interfaces, creating larger pieces by directly combining smaller ones. This is supported not only by Ousterhout's TK toolkit, but also by the INTERVIEWS C++ toolkit, which relies heavily on composition as its primary model for building interfaces (Linton, Vlissides and Calder 1989).

Early prototypes of GROUPKIT used inheritance extensively for creating interfaces. For example, the functionality of the current ConferenceMonitor was gained by subclassing the generic Conference object. Messaging objects were subclassed from virtual messaging objects to send particular message types. This proved extremely awkward, particularly with the high overhead of subclassing in C++. The current scheme, where inheritance has been minimized when using the toolkit, has resulted in shorter and clearer implementations. Additionally, the result was conceptually closer to the basic INTERVIEWS model — inheritance is used to create new functionality, but existing components can be used without inheritance.

```
main(int argc, char** argv) {
  Session* session =              // overall program manager
       new Session("HelloWorld", argc, argv);
  session->run_window(            // start up a window ...
    new ApplicationWindow(        //     .. that we create
      new HelloWorldGlyph(        //     .. that contains this interface
        session->style() ) ) );
}
```

Figure 4.1. Standard INTERVIEWS main program.

```
main(int argc, char** argv) {
  GroupSession* session =         // overall groupware program manager
     new GroupSession("HelloWorld", argc, argv);
  session->run_window(                  // start up a window...
    new ApplicationWindow(              //     .. that we create
      new MultiUserHelloWorldGlyph(     //     .. with this interface
        session->style(),
        session->conference() ) ) );    // with a pointer to conference
                                        //    for communications etc.
}
```

Figure 4.2. Standard GROUPKIT main program.

Extensibility

Ideally, it should be possible to extend the functionality of objects in the system with minimal disruption to the developer. With careful design, object-oriented inheritance and polymorphism can provide this functionality. It is important to separate the interface to the class from the implementation details (Linton 1992) , which can be difficult in languages like C++ (Stroustrup 1986).

The interfaces for many of the GROUPKIT objects are designed with extension in mind. These include the main Conference, the ConnectionList, and attribute lists. An important consideration when constructing extensible components is keeping the basic component simple enough for use by the developer. This is illustrated in GROUPKIT by the annotation overlay. Finally, it is important to create general techniques in toolkits wherever possible, which provides the greatest promise of extensibility.

1. Conference. The current Conference object, which starts and maintains connections to other conferences, operates under the assumption of a fully replicated communications architecture. Under some circumstances, a centralized architecture may be beneficial. A "CentralizedConference" could be created based on the main Conference, to allow implementing a centralized architecture rather than a replicated architecture. While the implementation details would change, the developer would still use the Conference in the same way. Unnecessary details of the underlying architecture are kept hidden.

2. ConnectionList. Currently, the GROUPKIT ConnectionList object simulates a multi-cast by sending the same message over all socket connections, in the "toAll" method. This is because standard Unix systems do not have a true multi-cast facility. Developers already using the "toAll" method would be able to instantly take advantage of hardware multi-cast support with a single change to this method.

3. Attribute lists. Attribute lists currently use an ASCII string representation when transmitted over a socket. Without change to the interface, this could easily be changed to a binary transmission mode, which would increase efficiency by minimizing data conversions.

4. Complexity of overlays. As with many designs, the design of overlays involves many tradeoffs. Consider the bitmap annotation overlay, which presents a simple bitmap overlaying an entire application, so that users can annotate artifacts in the underlying application. However, what if these underlying artifacts move? In the simple overlay that has been implemented, the annotations remain in place and do not follow the changing artifact. Extra work would be required of the application developer to support movable annotations. While the current configuration is restricted because there is not even an option for movable annotations, its advantage is that application developers require only a single line of code to add the primitive annotation capabilities. Again, the "buy-in" is low, which encourages more use of the facility than the higher cost that would be incurred by providing a more powerful overlay that required more work to include in a program. Powerful features will likely not be used if the initial expense of including them is high.

While it is important that the initial cost to the developer to use a simple feature is kept low, it is also important that developers have the option to devote extra effort to achieve greater functionality. Though not yet implemented, a useful extension to the bitmap annotation overlay that would support attaching annotations to specific artifacts in the work surface is now described.

While the current implementation retains one large bitmap containing all annotations, an alternate approach is illustrated in Figure 4.3. Rather than one large bitmap, individual annotations or strokes — see (McCall, Moran, van Melle, Pedersen and Halasz 1992) — are kept as separate bitmaps, which can be moved independently around the work surface. In INTERVIEWS this could be done using a Page glyph, which is the basis for the multiple cursor overlay. Thus this would provide the ability to move annotations.

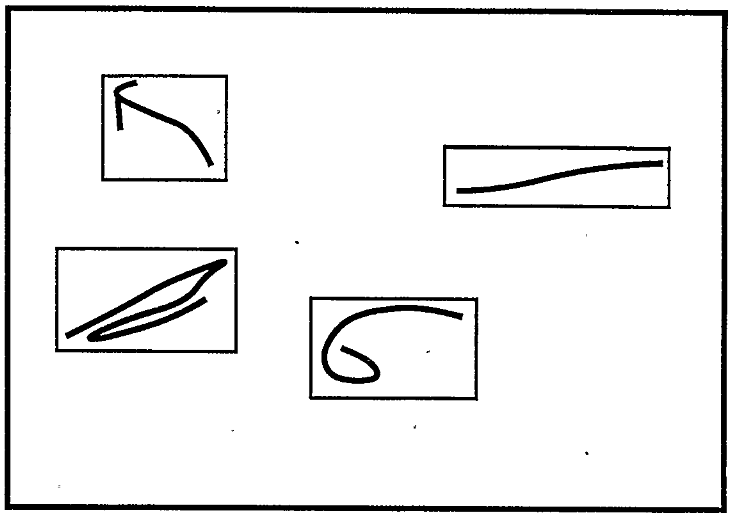


Figure 4.3. Bitmap annotations kept separate using a Page glyph. Here each annotation can be individually placed on the work surface.

The next step is to define a protocol whereby annotations, when first created, can ask the underlying application to locate artifacts near the annotation. A protocol would also be needed so that when application artifacts are moved the overlay is notified and can adjust the position of the annotations. This implementation would be a reasonable amount of effort for the developer, although this could be lessened if an object framework, such as that of GROUPDRAW (Greenberg, Roseman et al 1992), could be extended to support the necessary protocols. The important point is that developers are more likely to go to this effort once the initial idea of providing annotation capabilities has been accepted, rather than if the initial acceptance involves a large amount of initial work.

5. Generality of overlays. The strength of overlays is not in the particular two overlays that have been implemented, but in the promise of the overlay technique in general. In a toolkit, application developers should be provided with such techniques so that they can develop their own components, and not just be provided with end components. The recent interest in "shared feedback" (Dourish and Belloti 1992) provides examples of other components that could be provided with overlays. Shared feedback looks at providing more awareness to the user about other users' actions. Gesturing with multiple cursors is therefore one form of shared feedback. The SASSE word processor (Baecker, Nastos, Posner and Mawby 1993) suggests other components that could be used for shared feedback. First, a "gestalt view" provides a miniature version of the shared document, highlighting areas where other users are working. A "multi-user scrollbar" augments conventional scrollbars by identifying the positions of other collaborators, while still retaining the normal view of the document. Non-speech audio cues — see also (Gaver 1991a) — can be used to augment awareness of other users' actions. Finally, view sharing, view slaving, "lead and follow" techniques and others (Pendergast and Hayne 1992) are emerging that allow users to more closely monitor the work of other users over the shared work surface.

General components for all these functions could be implemented using an overlay, or more generally, composition technique, where an outer layer or overlay augments the behavior of the inner layer. This is analogous to the use of a "pane" in single-user toolkits, where an arbitrary large work surface is composed within a pane to permit scrolling of the work surface, without any work required by the developer of the inner work surface. A "group-aware pane" could extend this idea to provide the multi-user scrollbars or view slaving as general toolkit components, which could be easily applied to a wide variety of groupware applications.

Though the clear benefit of overlays is in "What You See Is What I See" or WYSIWIS situations, a number of issues arise where a "relaxed WYSIWIS" situation is desirable. Stefik et al. (1987) suggests four dimensions on which WYSIWIS can be relaxed. A potential area for future work might be considering how overlays could be applied to each of these dimensions. Some suggestions follow.

1. *Display space* (which objects are viewed?) — If particular objects in the underlying application are not to be viewed by others, mechanisms in the overlay could prevent annotations attached to those objects from being displayed as well.
2. *Display time* (when are views synchronized?) — Gestures, annotations, and other potential overlays could be designed to update information on remote displays at only periodic intervals rather than immediately.
3. *Subgroup population* (who shares view?) — Gesturing could be dependent on the subgroup, so that cursors of only subgroup members are visible, reducing clutter when larger groups are working.
4. *Congruence of view* (see same areas of view?) — The multi-user scrolling pane could implement multiple strategies for separating or combining views of participants, as in the examples by Pendergast and Hayne (1992) described above.

Flexibility

Building toolkit components inevitably involves a trade-off between flexibility and ease of use. Taking a flexible approach permits developers to build a wide variety of different systems, although implementation may be more difficult if few higher-level constructs are provided. In contrast, providing higher level components with less flexibility may result in some programs that are easier to build, whereas building other programs may involve abandoning the toolkit support entirely. Flexibility is an issue in three GROUPKIT areas, including communications, security, and particularly open protocols.

1. Communications. In GROUPKIT's communication facilities, the flexibility approach is preferred. The capability is provided to send messages to any user, yet few high-level constructs such as guaranteed message ordering over all sites are provided. For a domain such as groupware, this flexibility is important, as a wide variety of different applications could be constructed. Converging too quickly on higher level constructs may unduly restrict developers in new, unexpected situations. As well, it is possible to provide higher levels of support based on the primitives that are provided. These levels can include for example concurrency control, data encryption, or persistent objects.

2. Security. While open protocols permit any client using the protocol to interact with the controlled objects, this brings up the issue of malicious clients, and the associated security problems. For example, a malicious Registrar Client could easily delete users from conferences. In offering the flexibility, open protocols do open security risks. For several reasons this may not be particularly troublesome. Consider:

- In many organizations, social and organizational pressures will tend to prevent such abuses, because of the sense of community and trust present in many workgroups, or possibly the ramifications of such inappropriate actions. Security may be more of a concern in loose organizations such as the Internet, yet still there are several channels whereby problems with malicious users can be resolved.

- There is a certain cost to build any client, malicious or otherwise. Such an implementation requires a certain level of knowledge and some amount of time. This minimizes the possibility for accidental abuse — a novice accidentally typing some incorrect commands — yet in itself does not prevent experienced hackers from creating malicious systems.

- Finally, nothing actually prevents security measures from being programmed into the systems. Security may take the form of restricted access lists, password protection, or other such authentication and approval mechanisms. Open protocols do not prevent such mechanisms from being implemented, they simply do not require the mechanisms to be implemented.

3. Open protocols. The main advantage of the open protocol strategy is the flexibility gained. A wide variety of tools such as registration modules can be easily attached to the system, providing behaviors that were never anticipated by the original developers. More interestingly, open protocols allow several different controllers to interact, so that even within the same group different systems can be used. As discussed, this flexibility is important to accommodate different groups.

As another example of the flexibility possible with the open protocol approach, Brinck (1992) describes a set of inter-operable group drawing programs built using the RENDEZVOUS system. The programs, ranging from simplistic free-hand drawing to complex drawing systems with structured graphics and constraints, share a common underlying representation of the drawing surface. The difference between the programs is the particular set of drawing tools that operate on the drawing surface. This allows different users to work with different sets of tools, according to their needs and ability levels. Yet because the underlying representation of the drawing surface is the same, these different programs can work together effectively. This technique works because, as in open protocols, the drawing objects (controlled objects) are kept simple, while sets of tools (controllers) can be built and included in programs that interact with the drawing surface in a well-defined way (the open protocol).

Summary

This chapter has considered the underlying rationale for the design of many of the features described in Chapter 3. In doing so, many of the tradeoffs and issues that are necessarily considered when undertaking any design process have been made explicit. This rationale should prove useful for other developers who must confront similar issues in toolkit design projects.

The chapter focused on how toolkit functionality can be provided to best meet the needs of the application developer. Four areas — minimizing the developer's work, encouraging use of features, toolkit extensibility, and component flexibility — provided the focus for discussing some important issues.

Chapter 5

Evaluation

This chapter reflects on the design and implementation of GROUPKIT described in the previous chapters. The evaluation considers the GROUPKIT implementation as well as more general issues of the design requirements and user centered toolkit design.

Nature of Evaluation

A thorough evaluation of the design and implementation process is difficult at this point. Under the user-centered approach described in Chapter 1, any realistic evaluation must primarily take into account the experiences of toolkit users — groupware developers using the system to build real groupware applications. Unfortunately, as of this writing, only the first version of GROUPKIT — the result of several design iterations based solely on the toolkit developer's experiences and informal interaction with developers — has been released to the research community via the Internet. To provide an adequate basis for evaluation, feedback from these developers is necessary, both to evaluate and to further evolve the toolkit.

Having acknowledged the preliminary nature of any such evaluation, comments can be made about several areas of this work. The GROUPKIT implementation can be considered in terms of its coverage of the human and programmer-centered design principles, as well as some initial experiences in constructing simple groupware applications. Both the design requirements and the user-centered toolkit design approach itself can also be briefly considered.

GROUPKIT Implementation

The implementation of GROUPKIT, described in Chapters 3 and 4, can be considered in two ways. First, because the toolkit was designed in response to a series of design principles derived from human-centered and programmer-centered considerations, the implementation can be evaluated according to those principles. Second, usage experiences, currently limited to the toolkit developer, can be analyzed. Several simple groupware applications, such as sketchpads and group support tools, were constructed to help this analysis.

Coverage of design principles

The design principles from Chapter 2 serve as the basis for the GROUPKIT implementation. As mentioned in Chapter 3, time limitations restricted the set of principles covered by the implementation. Table 5.1 shows the mapping between the support in the current implementation and the design requirements. Table 5.2 provides a more detailed look at how GROUPKIT satisfies the requirements. The first four columns cover the toolkits described in Chapter 2 (RENDEZVOUS, MMCONF, CONFERENCE TOOLKIT, and LIZA). The fifth column (GK) summarizes the coverage of requirements by the version of GROUPKIT described in this thesis. Similarly, the last column (GK+) summarizes a "work in progress" version of GROUPKIT, discussed in the next chapter, which also includes prototypes of an object graphics layer and a shared terminal.

Usage experiences

Although several groupware developers around the world have begun experimenting with GROUPKIT, because of its recent availability the only known applications constructed using it have been written by the toolkit developer. These applications are primarily intended as simple demonstrations of toolkit features and are therefore not full-featured.

Requirement	Technique	Implemented support
Actions over work surface	overlays	multiple cursors, bitmap annotation
Structure group processes	open protocols	flexible registration
Integrate conventional work	shared terminals	in progress
Multiple distributed processes	communications infrastructure	separate registration, messaging and simulated multi-cast facilities
Shared graphics model	object layer	in progress

Table 5.1. Coverage of design requirements by GROUPKIT features.

Conference-Label displays the name of the conference in a window. It is about the smallest application possible, but provides fully replicated processes. The basic code for this application was provided in Figure 4.2 of Chapter 4.

Deck-Flip allows users to switch between several strings in a window. The current string selected is transmitted to others in the conference. This program illustrates message passing between processes.

Monitor allows users to browse the list of users in the conference and their attributes. This program demonstrates the use of the ConferenceMonitor facility to watch users joining and leaving the conference.

Brainstorm presents a simple group support tool where single-line "ideas" from users are anonymously transmitted to other conference users and compiled into a list containing all the group's ideas.

Vote allows users to call a vote of conference users on "Yes/No" questions, with the ongoing results being tallied by the vote-caller as others respond (see Figure 5.1).

Cursor-Demo illustrates the use of the Cursor overlay on top of an empty glyph. This shows the simplest case of adding a single overlay to an application.

Requirement	REND-VOUS	MM-CONF	CONF. T-KIT	LIZA	GK	GK+
Human-centered Requirements						
Actions over visual work surfaces						
convey process	+	o	o	o	o	+
simultaneity	+	+	o	o	o	+
modeless interface	.	?	o	o	o	o
common view	+	o	o	o	o	+
gestures	.	+	.	.	++	++
annotation	o	+	.	.	++	++
Flexible group processes						
floor control	+	++	+	?	o	++
registration	.	.	.	.	++	++
latecomers	.	.	.	.	.	+
Integration with conventional work						
non-computer artifacts	.	.	.	.	.	.
other communication	+	+	.	.	.	.
single-user apps	?	++	+	.	.	+
Programmer-centered requirements						
Multiple distributed processes						
conference management	+	+	+	+	++	++
communications infrastructure	+	+	++	+	+	+
persistent sessions	.	?	?	?	.	+
Shared graphics model						
shared visual objects	++	.	.	.	.	++
object concurrency	++	.	.	.	.	++
separate view	++	.	.	.	.	++

++	system provides good support
+	system provides support
o	system can handle but does not specifically support
.	system does not support
?	not clear if support is provided

Table 5.2. Detailed comparison of GROUPKIT and existing toolkits.

Group-Sketch uses the Cursor and Sketchpad overlays on top of an empty glyph or arbitrary image to create a simple group sketchpad (see Figure 3.15).

Share-Shell presents a Unix shell overlayed by the Cursor and Sketchpad overlays, and is capable of supporting the SHARE program (Greenberg 1990) which allows several users to share a single terminal, combining their input into a single stream and directing the output to each user's screen.

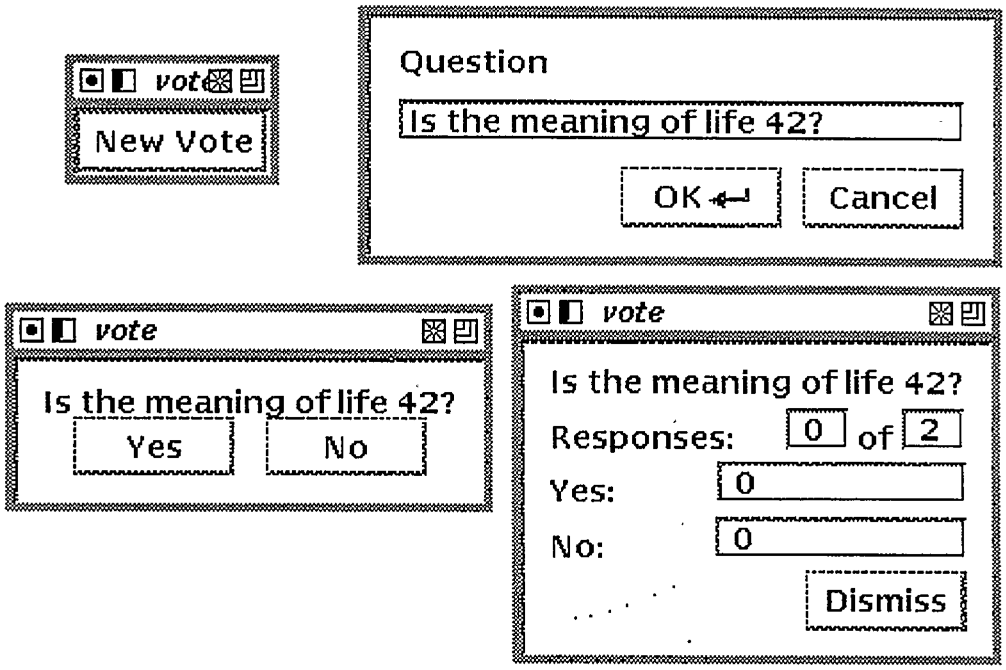
Draw-Lines is a simple program allowing users to draw and manipulate lines, which remain consistent on all users' screens. This program contains most of a general group object framework, adapted from GROUPDRAW (Greenberg, Roseman et al 1992).

Open-Registrar-Client is the "standard" registrar client, allowing users to create, join and leave conferences (see Figure 3.3, Chapter 3).

Master-Registrar-Client is a prototype of a registrar client suitable for a meeting facilitator, who can control the programs used by the meeting's users (see Figure 5.2, below).

Slave-Registrar-Client is a prototype registrar client suitable for users in a facilitated meeting.

Though the applications are merely demonstrations and are missing many normal features (saving files, cut and paste, printing, etc.) they are otherwise reasonable multi-user applications with interfaces supporting interface elements such as multiple windows, dialogs, and menus. For example, Figure 5.1 illustrates the "Vote" application, while Figure 5.2 illustrates the "Master-Registrar-Client," an alternative to the standard client illustrated in Figure 3.3 of Chapter 3.


vote

New Vote

Question

Is the meaning of life 42?

OK

Cancel

vote

Is the meaning of life 42?

Yes

No

vote

Is the meaning of life 42?

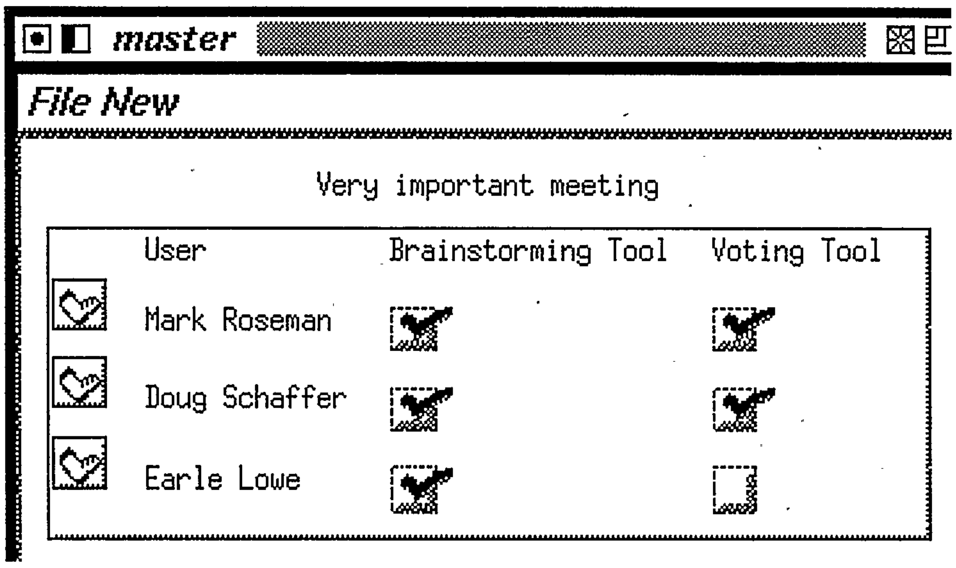
Responses: 0 of 2

Yes: 0

No: 0

Dismiss

Figure 5.1. Screen capture of the GROUPKIT "Vote" demonstration program. Shown are the initial window to start a new vote (top left); dialog box to specify the question to be voted on (top right); dialog box for vote initiator to record vote tally (bottom right); and dialog box allowing users to vote on the question (bottom left).


master

File New

Very important meeting

	User	Brainstorming Tool	Voting Tool
	Mark Roseman	✓	✓
	Doug Schaffer	✓	✓
	Earle Lowe	✓	☐

Figure 5.2. Screen capture of the "Master-Registrar-Client." This client is intended for use by a meeting facilitator. Other meeting participants appear along the left side of the window. The facilitator can start new applications, such as brainstorming or voting, by selecting them from the "New" menu. Users are added or deleted from applications by the facilitator using the check boxes. The facilitator may also remove a user from the entire meeting using the "boot" beside each name.

Program	Lines of Code
Conference-Label	51
Deck-Flip	210
Monitor	240
Brainstorm	212
Vote	442
Cursor-Demo	101
Group-Sketch	137
Share-Shell	66
Draw-Lines	1089
Registrar	328
Open-Registrar-Client	796
Master-Registrar-Client	800
Slave-Registrar-Client	357

Table 5.3. Size of GROUPKIT applications.

One informative measure is the size of each of these applications. Table 5.3 shows the number of lines of code contained in each of the programs. The lines of code here includes such thing as C++ header files, include lines, comments, blank lines as well as the code itself, so is perhaps somewhat inflated.

Reducing the amount of code need not necessarily reduce overall complexity or development time. Some qualitative observations about the sample applications are therefore also appropriate. First, much of the code in the programs is related to common single-user application concerns, particularly window and dialog construction, with small

Widget	Description
String Browser	browser allowing selection of a string from a list
Labeled Scroll List	a String Browser bundled together with a label and a scroll bar
Shell	a VT-100 terminal emulator, supporting a Unix shell process via pseudo-tty connections
Tabular	a composite widget made up of aligned rows and columns of arbitrary widgets

Table 5.4. Single-user widgets distributed with GROUPKIT.

amounts dealing specifically with "group" concerns. The "group" code tends to be task-specific at an appropriate level of abstraction, e.g. "send an idea" to other users in the brainstorming program is essentially an atomic action. The exception to this is the line drawing program, where much of the code deals with general concurrency issues and is not task-specific. This suggests that most of the generalities in that program should be embedded in the toolkit itself, not left for individual applications, an issue considered in the next chapter. Overall, the complexity of writing GROUPKIT programs appears to be kept low, that the decreased amount of code required has not resulted in an unacceptably increased complexity for writing each line.

Most of the programs were quick to develop — generally under two hours — yet this relied heavily on experience with the underlying INTERVIEWS toolkit. This is not surprising, as most of the code in the examples deals with interface components such as menus, dialogs, control panels, and other such widgets. The extensive need for such code prompted the creation of a number of new single-user widgets that have been released with the GROUPKIT distribution, as shown in Table 5.4. While some feedback from developers suggests that the learning curve for GROUPKIT experienced by someone knowledgeable in

INTERVIEWS is reasonably minimal, the curve that could be expected for an INTERVIEWS novice is very considerable. The lack of a strong knowledge of C++ and INTERVIEWS seems to be the primary difficulty in writing GROUPKIT programs.

Comparative Evaluation

How does development with GROUPKIT compare to development without using any groupware toolkit? The previous discussion suggests development using GROUPKIT is substantially easier. Table 5.5 gives the size of several groupware applications developed without the benefit of a toolkit. All these projects took at least several months of concentrated development time. Many problems were encountered during development that were tangential to the specific application at hand (Greenberg, Roseman et al 1992). These included difficulties with establishing communications, the need for registration, complex message parsing and dispatching, and dealing with multiple cursors.

Of course, these figures could be misleading, as the applications are more sophisticated than the GROUPKIT counterparts. Table 5.6 provides a better estimate. The figures, based on the sample applications described previously, highlight the effort required to replicate particular functions in three existing programs using GROUPKIT: a shared paint program (GROUPSKETCH), an object based drawing program (GROUPDRAW), and a shared Unix terminal (SHARE). For example, in order for the GROUPKIT sketching demo to compare with the functionally richer GROUPSKETCH application, the cursor overlay would need to be extended so that cursors change shape depending on the users' actions. Similarly, to compare the line drawing demo with GROUPDRAW, a number of menu commands for changing an object's coupling status would need to be added to the former. This comparison is confounded by differences in the underlying interface toolkits, as adding features with INTERVIEWS is often simpler than with toolkits used in the three applications.

The qualitative comments made in the previous section seem to apply as well; that despite the decreased amount of code, complexity of development per line of code was not substantially increased. In fact, the GROUPKIT based systems eliminated many confusing interdependencies, such as code for gesturing mixed with code for drawing. Use of GROUPKIT reduces many of the groupware difficulties; the apparent result is that developers can focus on the problems specific to their particular groupware application, reducing development time by very significant margins.

Program	Target Environment	Development Tool	Description	Lines of Code
GROUPSKETCH	Unix	C	Simple paint program	4500
GROUPDRAW	Macintosh	Think C / TCL	Object drawing program	8000
SHARE	Unix	C	Shared Unix terminal	7300
XGROUPSKETCH	Unix / X	C / XView	Fancy color paint program	5000
WSCRAWL	Unix / X	C / Motif	Deluxe color paint program	17500
SHDR	Unix / X	C / XView	Simple 2-user paint program	750

Table 5.5. Size of programs developed without use of a groupware toolkit.

System / Feature	Approximate Lines of Code	Estimated GROUPKIT Lines of Code
GROUPSKETCH		(based on Group-Sketch demo)
Socket setup and messaging	650	4
Registration	700	0
Window management	650	40 + *50 (menus etc.)*
Drawing and erasing	500	3 + *50 (add erasing to overlay)*
Load and save	150	*100*
Event handling	600	0
Caricatures	250	*110*
Multiple cursors	400	3 + *50 (cursor change w/ buttons)*
Miscellaneous	600	40
Total	**4500**	**450**
GROUPDRAW		(based on Draw-Lines demo)
Socket setup and messaging	1700	100
Registration	600	0
Window management	2500	100 + *300 (extend tools, menus)*
Generic object	900	850
Specific object types	700	*700*
Object interaction	400	150
Event handling	500	*50*
Multiple cursors	300	3
Miscellaneous	400	*100*
Total	**8000**	**2353**
SHARE		(based on Share-Shell demo)
Socket setup and messaging	1000	4 + *20 (send/receive typing msg)*
Registration	2200	0
Unix process input/output	200	0
Generic floor control	700	*300 (based on open protocols)*
Specific floor control policies	2600	*600 (subclass from generics)*
Miscellaneous	600	*50*
Total	**7300**	**974**

Table 5.6. Estimated replication of existing systems using GROUPKIT. Italics indicate estimates while Roman type shows actual figures from GROUPKIT sample applications.

Design Principles and User-Centered Design

The design principles presented in Chapter 2 are a useful set of core requirements for a groupware toolkit. They are clearly *not* intended as a complete set. As research continues into the use of CSCW applications, as developers continue to encounter problems in building groupware, and as novel groupware applications continue to appear, the set of design principles must continue to grow. It is important that growth continues not only on the programmer-centered side, which has been the emphasis of most toolkit work to date, but also on the human-centered side, which has largely been neglected. The design principles selected here are a good model because they focus on *both* aspects of groupware design.

The user-centered toolkit design approach developed in this thesis is harder to evaluate, based on the single case of GROUPKIT that itself is not complete. The approach was not strictly adhered to in the work described here due to time restrictions. Though some early feedback from developers was received, as a result of a poster presentation at CHI '92 (Roseman and Greenberg 1992a) and early drafts of a conference paper (Roseman and Greenberg 1992b), much more could have been done to solicit feedback from developers. Questionnaires on past and planned groupware development projects, familiar development tools, and other issues would provide much valuable feedback. Electronic discussion groups to discuss issues as they arise, or focus groups at conferences where groupware developers regularly gather would be useful sources of information. Toolkit development also might prove a fertile area for participatory design (Blomberg and Henderson 1990), where toolkit developers can collaborate with knowledgeable application developers already sharing a common vocabulary and approach to development.

The benefit of the user-centered toolkit design approach seems to be not that it presents anything novel, but instead articulates existing notions of good software architecture design. Designers of good toolkits likely follow a user-centered approach, consciously or not. By defining an explicit framework for toolkit developers to follow, a structured process is made available to ease toolkit design decisions. Whether the particular framework will result in better toolkits cannot now be shown, yet by defining such a framework, a starting point for discussion among toolkit developers is created, allowing the framework to evolve.

Critical Reflection

One of the goals of this thesis has been to provide aid and guidance for developers who will be engaged in future toolkit development efforts, to describe not just a generic path through the GROUPKIT development project, but to expose the high points as well as the roadblocks and shortcuts which altered the path. It is not the final destination that matters as much as the journey along the way.

One of the definite high points in the journey was the early identification of human-centered requirements as a legitimate basis for building a toolkit. Though human factors work for groupware applications was identified early on as critical to their success, little *explicit* effort was made by previous toolkit developers to help application developers cope with these issues. Some toolkit developers were including human-centered components, but did not explicitly recognize human-centered requirements as important — decidedly different from technical requirements perhaps yet equally valid as engineering goals. The identification of human-centered issues as requirements that can and should be incorporated into a toolkit is probably the major contribution of this thesis.

The high level design of the software is another high point. The design of the toolkit components provides a lot of power at a reasonable cost to the application developer. They address many high level issues and bundle them into useful packages for developers. The design does encapsulate enough of the underlying groupware issues so that developers can worry about the specifics of their particular applications.

Of considerable importance is the underlying portability of the design and the ideas it contains. Aside from the overlays, little in the design is heavily dependent on the chosen implementation platform. The overall design could be easily replicated on many different platforms. Regardless of whether developers embrace the actual GROUPKIT system, the design of the components provides a useful model for application and toolkit developers alike. These developers can benefit greatly from many of the toolkit issues discussed in detail in Chapter 4. Many of the detailed issues raised there — generality of components, use of inheritance, documentation and examples, security — can greatly influence the resulting software design.

The actual GROUPKIT implementation was more of a bumpy stretch of road with many unexpected curves. There was a great deal in the INTERVIEWS platform which helped to accelerate the system development, including a rich graphics model and true inheritance of widgets. Yet this choice of platform will undoubtedly restrict the target audience, minimizing actual use of the system and subsequent feedback. Support for INTERVIEWS has not advanced nearly as far as expected when this project began — slow and incomplete migration to the current release, a dearth of documentation for new users, and an out-of-date interface builder — coupled with an uncertain future have made the platform a less useful basis than desired. Though proposals to support INTERVIEWS from within the MIT X Consortium may yet address these issues, at present the lack of a complete widget set that can be reasonably learned and used by developers is an obstacle to successful adoption of the GROUPKIT platform.

The user-centered toolkit design methodology itself was not adhered to as much as it should have been. Though a useful idea in theory — again, not because of its novelty but its explicitness — it could have been much better applied in practice. This was largely a result of the ideas being made explicit relatively late in the project. User-centered toolkit design evolved out of the human and programmer-centered requirements, not the other way around. As discussed in the previous section, more efforts could have been made to directly solicit developer feedback early on in the process, augmenting the more indirect feedback received from studies of existing systems and their use.

Chapter 6

Concluding Remarks

This chapter reflects on the design and implementation of GROUPKIT described in the previous chapters. A number of interesting possibilities for future work are discussed, including support for shared graphics, high-level components such as text editing, asynchronous groupware, conventional work, and portability between systems. The thesis concludes with a brief review of the contributions of this research.

Ongoing and Future Work

The work presented in this thesis offers a useful framework for considering the design of toolkits for groupware application developers. Still, there are many areas left to be addressed. Some of this work has already been started, and is noted below.

Expanding Design and Implementation

Groupware development and CSCW itself are new disciplines. As the disciplines continue to evolve, new requirements for groupware will evolve, based both on the development of radically new systems and also on the results of field studies of those systems. The principles described in Chapter 2 present only the first step of an inherently evolutionary, iterative and open-ended process. To remain useful, the principles must evolve with the challenges of newer systems and human factors concerns.

On the implementation side, new implementation techniques must be found as more design requirements are discovered. While the components described in Chapter 3 are general enough to satisfy some new requirements, further techniques will be required. The design rationale in Chapter 4 can serve as a guide in designing these new techniques. New techniques and components will need to consider the issues of minimizing the developer's work, encouraging use, extensibility and flexibility to be most useful to groupware developers.

Shared Graphics

One design requirement unsatisfied by the current implementation of GROUPKIT is the provision of a shared graphics model as a general toolkit component. Such a model would provide the notion of a shared group object, encapsulating issues such as concurrency control, consistency, and separate object views. These are important but difficult issues to deal with, and providing toolkit level support would free the programmer from many such concerns.

The GROUPDRAW system (Greenberg, Roseman et al 1992) provides a shared graphical model that is suitable for inclusion in GROUPKIT. Many aspects of the model are consistent with the GROUPKIT design philosophies discussed in Chapter 4. Object-oriented inheritance is used extensively to provide a basic level of concurrency, yet permitting more complex and application-specific behaviors to be accommodated within the framework. The object-oriented design simplifies not only using standard toolkit components, but also building higher-level components. Finally, the communication support required by the model is simple message passing, of the sort provided by GROUPKIT.

Most of this work has already been completed. The "Draw-Lines" demonstration program contains most of the underlying GROUPDRAW framework. The message passing required was easily accommodated by GROUPKIT, and resulted in a great simplification of the GROUPDRAW code. Because the GROUPDRAW code relied on a high degree of separation between the underlying objects and the graphical views of those objects, the object level code was easy to extract. Porting GROUPDRAW from Think C (a subset of C++) to C++ was relatively trivial, and C++ features such as constructors and destructors resulted in improved clarity of the code. The higher level interface, providing views of the underlying shared objects as well as means for manipulating them was not ported, due to underlying differences between the Think Class Library used in GROUPDRAW and INTERVIEWS.

High-Level Components

Many single user interface toolkits provide high-level components such as shells, editors, canvases and dialogs that are composed out of simpler building blocks. The idea of providing high-level components to the developer transfers well to the groupware domain. The graphics model described above could be such a component, situated as a "layer" in the toolkit, based on the communications and graphics primitives already provided by GROUPKIT and INTERVIEWS (see Figure 6.1). The communications provided by GROUPKIT could support the message passing needs of the component, while INTERVIEWS provides support for the graphics primitives. An application based on the graphics component could also use the underlying services for its own particular needs.

Support for group text editing is another essential component. Work on the single-user text model in the INTERVIEWS "Doc" application — a "What You See Is What You Get" document editor — has provided useful abstractions such as separate text item and view objects, using inheritance to provide higher-level document objects (Calder and Linton 1992). This might usefully be combined with the group text models and interaction strategies now found in the SASSE editor (Baecker, Nastos et al 1993).

Other possible high-level components include group hypertext models, or components for decision making or facilitation. In all these cases, the goal would be to provide a general

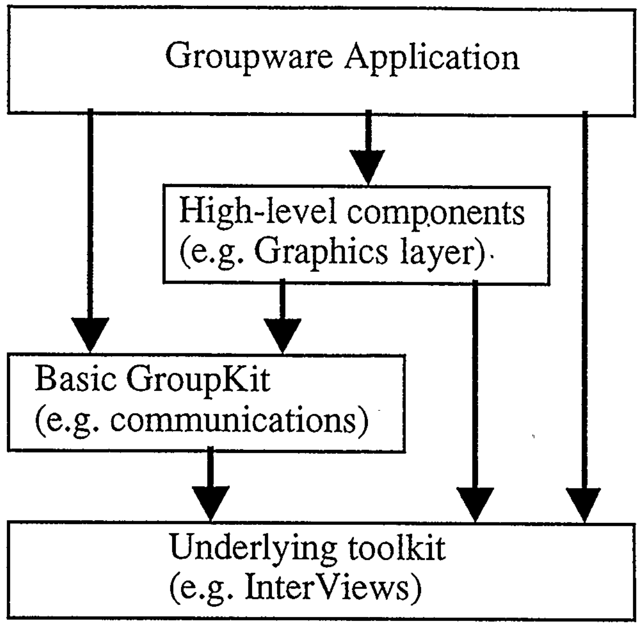


Figure 6.1. Layering of high-level toolkit support.

level of functionality in a toolkit layer, which may be customized as appropriate by the application developer.

Asynchronous Groupware and Everyday Work

The work described in this thesis considers real-time or synchronous groupware only. Ideally, a groupware toolkit should be able to support the construction of both synchronous and asynchronous groupware systems. A similar design process could be undertaken: study existing asynchronous groupware along with its associated human factors issues, derive design requirements, and generate techniques and toolkit components that can satisfy the design requirements. Transitions between synchronous and asynchronous systems — so-called semi-synchronous systems (Dourish and Belloti 1992) — present an interesting subset of groupware problems. Intelligent support for selecting appropriate communication channels, whether synchronous or asynchronous, is another problem worth investigating (Cockburn and Greenberg 1993). Finally, integration with conventional media such as the normal desktop, telephone, and increasingly video conferencing to supplement the computer based conferencing is an important area of investigation. The GROUPKIT registration mechanisms provide a useful framework for constructing support in these areas, by providing a policy-free mechanism to launch and terminate applications.

Portability

An important issue for groupware developers is cross-platform development, or designing applications that can interact across different types of hardware. While also an issue in single-user systems, it becomes increasingly important in work groups using heterogeneous computer equipment. While the current implementation has a high dependence on the platform (Unix, X-Windows, and INTERVIEWS), many of the underlying ideas could prove useful steps towards portability. The idea of separating objects from views decreases dependencies on particular graphics systems. The interfaces for many GROUPKIT classes abstract away from the hardware, so that the communications modules could be easily ported to other systems. Finally, open protocols suggest that by properly designing the communications protocols, many different interfaces — including those on other types of machines — can be accommodated.

One project currently being investigated is to provide a common text editing framework between Unix and the Macintosh, based on the text support provided by SASSE (Baecker, Nastos et al 1993). A goal of the project is to distill a set of requirements necessary for building portable groupware systems. While this work is still in the very preliminary stages, a number of directions are emerging. Such portability requires a high abstraction from the interface, which has not been a common characteristic of current groupware systems. Such abstraction requires the use of view separation as described previously, so that the interface initiates and reflects changes to an underlying object. While it seems likely that high level interfaces will be entirely hand-coded on particular machines, it is unclear how much code of the underlying objects may be completely portable across systems.

Summary of Contributions

This thesis has applied principles of user-centered design to the design of software toolkits, in particular for the domain of real-time or synchronous groupware conferencing systems. User-centered toolkit design advocates the following seven design steps:

1. Specify toolkit domain
2. Identify developers
3. Identify use of toolkit
4. Consider target applications
5. Design for proper use
6. Apply design affordances
7. Iterate design

To provide the support needed by the developers, a number of design requirements were derived based on existing groupware applications and human factors observations of these systems and work practices in general. Thus the requirements are based on both human-centered issues that directly affect groupware users, as well as technical or programmer-centered issues which primarily affect the groupware developer:

1. Human-Centered Issues:
    a. Support multi-user actions over a work surface like gesturing and annotation.
    b. Provide flexible structuring of processes like registration and floor control.
    c. Integrate with conventional work such as telephones and single-user software.

2. Programmer-Centered Issues:

   a. Support distributed processes for conference management and messaging.

   b. Provide a shared graphics model with concurrency and separate object views.

Based on the design requirements, the GROUPKIT system was constructed as a prototype groupware toolkit. Underlying GROUPKIT's design are three sets of techniques that serve to satisfy many of the derived design principles:

1. A *Communications Infrastructure* manages generic registration and communications needs, allowing developers to focus on their particular applications rather than developing a generic infrastructure.

2. *Overlays* provide generic work surface activities such as gesturing or annotation on top of particular applications, providing high-level groupware abstractions.

3. *Open protocols* provide a mechanism for implementing a wide variety of group processes, allowing developers to provide more flexible applications usable by a wider variety of groups.

Throughout the thesis, the design rationale behind the various components of the toolkit was made explicit, to capture many of the complex design decisions that must be dealt with in toolkit projects. This design rationale is important in considering toolkit design, and making it explicit will be a boon to toolkit developers in the future.

The result of the research is an effective design and implementation that developers can use as a basis for building groupware applications without concern for the myriad of difficulties that have traditionally plagued them.

Bibliography

Agha, G. (1986) *Actors: A Model of Concurrent Computation in Distributed Systems*, MIT Press, Cambridge, Massachusetts.

Ahuja, S. R., Ensor, J. R. and Lucco, S. E. (1990) "A comparison of applications sharing mechanisms in real-time desktop conferencing systems." In *Proceedings of the Conference on Office Information Systems (COIS '90)*, pp. 238-248.

Baecker, R., Nastos, D., Posner, I. and Mawby, K. (1993) "The User-Centered Iterative Design of Collaborative Writing Software." To appear in *Proceedings of InterCHI 93.*

Beaudouin-Lafon, M. and Karsenty, A. (1992) "Transparency and Awareness in a Real-Time Groupware System." In *Proceedings of UIST '92.*

Bier, E. A. and Freeman, S. (1991) "MMM: A User Interface Architecture for Shared Editors on a Single Screen." In *Proceedings of User Interface Software and Technology (UIST '91)*, pp. 79-86.

Blomberg, J. L. and Henderson, A. (1990) "Reflections on Participatory Design: Lessons from the Trillium Experience." In *Proceedings of the ACM CHI'90 Conference on Human Factors in Computing Systems*, pp. 353-360.

Bly, S. (1988) "A use of drawing surfaces in different collaborative settings." In *Proceedings of the Conference on Computer-Supported Cooperative Work (CSCW '88)*, pp. 250-256.

Bly, S. A. and Minneman, S. L. (1990) "Commune: A shared drawing surface." In *Proceedings of the Conference on Office Information Systems (COIS '90)*, pp. 184-192.

Bonfiglio, A., Malatesa, G. and Tisato, F. (1989) "Conference Toolkit: A framework for real-time conferencing." In *Proceedings of the 1st European Conference on Computer Supported Cooperative Work (EC-CSCW '89).*

Booch, G. (1990) *Object Oriented Design with Applications*, Benjamin/Cummings Publishing Company Inc., Redwood City, California.

Brinck, T. and Gomez, L. M. (1992) "A Collaborative Medium for the support of Conversational Props." In *Proc. of the Conference on Computer Supported Cooperative Work (CSCW '92)*, pp. 171-178.

Calder, P. and Linton, M. (1992) "The Object-Oriented Implementation of a Document Editor." In *Proceedings of OOPSLA '92.*

Chang, E. (1986) "Participant Systems." *Future Computing Systems*, **1**(3), pp. 253-270.

Chapanis, A. (1975) "Interactive human communication." *Scientific American*, **232**(3), pp. 36-42.

Chin, R. S. and Chanson, S. T. (1991) "Distributed Object-Based Programming Systems." *ACM Computing Surveys*, **23**(1), pp. 91-124.

Cockburn, A. and Greenberg, S. (1993) "Making Contact: Getting the Group Communicating with Groupware." Research Report 93/498/03, Department of Computer Science, University of Calgary, January 1993.

Crowley, T., Baker, E., Forsdick, H., Milazzo, P. and Tomlinson, R. (1990) "MMConf: An infrastructure for building shared applications." In *Proceedings of the Conference on Computer-Supported Cooperative Work (CSCW '90).*

Dewan, P. (1990) "A Tour of the Suite User Interface Software." In *Proceedings of the ACM Third Annual Symposium on User Interface Software and Technology (UIST '90)*, pp. 57-65.

Dewan, P. (1991) "Flexible user interface coupling in collaborative systems." In *Proceedings of the ACM CHI'91 Conference on Human Factors in Computing Systems*, pp. 41-48.

Dewan, P. and Choudhary, R. (1991) "Primitives for Programming Multi-User Interfaces." In *User Interface Software and Technology (UIST '91)*, pp. 69-78.

Dourish, P. and Belloti, V. (1992) "Awareness and Coordination in Shared Workspaces." In *Proc. of the Conference on Computer Supported Cooperative Work (CSCW '92)*, pp. 107-114.

Dubs, S. and Hayne, S. C. (1992) "Distributed Facilitation: A Concept Whose Time Has Come?" In *Proc. of the Conference on Computer Supported Cooperative Work (CSCW '92)*, pp. 314-321.

Dykstra, E. A. and Carasik, R. P. (1991) "Structure and support in cooperative environments: The Amsterdam Conversation Environment." *International Journal of Man Machine Studies*, **34**(3), pp. 419-434.

Ellis, C. A., Gibbs, S. J. and Rein, G. L. (1991) "Groupware: Some issues and experiences." *Communications of the ACM*, **34**(1).

Engelbart, D. and English, W. K. (1968) "A research center for augmenting human intellect." In *Proceedings of the Fall Joint Computer Conference*, pp. 395-410, San Francisco, Calif., AFIPS.

Ensor, J. R., Ahuja, S. R., Horn, D. N. and Lucco, S. E. (1988) "The Rapport Multimedia Conferencing System — A Software Overview." In *Proceedings of the 2nd IEEE Conference of Computer Workstations*, pp. 52-58.

Eveland, J. D. and Bikson, T. K. (1988) "Work group structures and computer support: A field experiment." In *Proceedings of the Conference on Computer-Supported Cooperative Work (CSCW '88)*, pp. 324-343.

Francik, E., Rudman, S. E., Cooper, D. and Levine, S. (1991) "Putting innovation to work: Adoption strategies for multimedia communication systems." *Communications of the ACM*, **34**(12), pp. 37-63.

Garfinkel, D., Gust, P., Lemon, M. and Lowder, S. (1989) "The SharedX Multi-user Interface User's Guide, Version 2.0." STL-TM-89-07, Hewlett-Packard Laboratories.

Gaver, W. (1991a) “Sound Support for Collaboration.” In *Proceedings of the 2nd European Conference on Computer Supported Cooperative Work (ECSCW '91).*

Gaver, W. (1991b) “Technology Affordances.” In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI '91)*, pp. 79-84.

Gibbs, S. J. (1989) “LIZA: An Extensible Groupware Toolkit.” In *Proceedings of the ACM CHI'89 Conference on Human Factors in Computing Systems*, pp. 29-35.

Gomoll, K. and Nicol, A. (1990) “Discussion of guidelines for user observation.” Technical Report, Apple Computer, Inc.

Greenberg, S. (1990) “Sharing views and interactions with single-user applications.” In *Proceedings of the Conference on Office Information Systems (COIS '90)*, pp. 227-237.

Greenberg, S. (1991) “Personalizable groupware: Accomodating individual roles and group differences.” In *Proceedings of the 2nd European Conference on Computer Supported Cooperative Work (EC-CSCW '91).*

Greenberg, S. and Bohnet, R. (1991) “GroupSketch: A multi-user sketchpad for geographically-distributed small groups.” In *Proceedings of Graphics Interface '91.*

Greenberg, S., Roseman, M., Webster, D. and Bohnet, R. (1992) “Human and Technical Factors of Distributed Group Drawing Tools.” *Interacting with Computers*, **4**(3), pp. 364-392.

Grief, I. (ed.) (1988) “Computer-Supported Cooperative Work: A Book of Readings.” , San Mateo, California, Morgan Kaufmann Publishers.

Group Technologies Inc. (1990) “Aspects: The First Simultaneous Conference Software for the Macintosh.” User Manual .

Grudin, J. (1989) “Why groupware applications fail: problems in design and evaluation.” *Office: Technology and People*, **4**(3), pp. 245-264.

Haake, J. M. and Wilson, B. (1992) "Supporting Collaborative Writing of Hyperdocuments in SEPIA." In *Proceedings of the Conference on Computer Supported Cooperative Work (CSCW '92)*, pp. 138-146.

Hughes, J. A., Randall, D. and Shapiro, D. (1992) "Faltering from Ethnography to Design." In *Proceedings of the Conference on Computer Supported Cooperative Work (CSCW '92)*, pp. 115-122.

Ishii, H. (1990) "TeamWorkStation: Towards a seamless shared space." In *Proceedings of the Conference on Computer-Supported Cooperative Work (CSCW '90)*, pp. 13-26.

Ishii, H., Kobayashi, M. and Grudin, J. (1992) "Integration of Inter-Personal Space and Shared Workspace: ClearBoard Design and Experiments." In *Proceedings of the Conference on Computer Supported Cooperative Work (CSCW '92)*, pp. 33-42.

Jay, A. (1976) "How to run a meeting." *Harvard Business Review*, **54**(2), pp. 43-57.

Johansen, R. (1988) *Groupware: Computer Support for Business Teams*, Macmillan Inc., New York.

Johnson-Lenz, P. and Johnson-Lenz, T. (1991) "Post-mechanistic groupware primitives: rhythms, boundaries and containers." *International Journal of Man Machine Studies*, **34**(3), pp. 385-418.

Killey, L. (1991) "ShrEdit 1.0: A Shared Editor for Apple Macintosh. User's Guide and Technical Description." , Cognitive Science and Machine Intelligence Laboratory, University of Michigan.

Kraut, R. E., Egido, C. and Galegher, J. (1990) "Patterns of contact and communication in scientific research collaborations." In *Intellectual Teamwork: Social Foundations of Cooperative Work*, pp. 149-172, J. Galegher, R. E. Kraut and C. Egido ed. Lawrence Erlbaum Associates.

Kuhn, T. S. (1962) *The Structure of scientific revolutions*, University of Chicago Press, Chicago.

Lakin, F. (1990) "Visual languages for cooperation: A performing medium approach to systems for cooperative work." In *Intellectual Teamwork: Social Foundations of Cooperative Work*, pp. 453-488, J. Galegher, R. E. Kraut and C. Egido ed. Lawrence Erlbaum Associates.

Lauwers, J. C. (1990) "Collaboration transparency in desktop teleconferencing environments." PhD Thesis, Stanford University, Computer Systems Laboratory.

Lauwers, J. C., Joseph, T. A., Lantz, K. A. and Romanow, A. L. (1990) "Replicated architectures for shared window systems: A critique." In *Proceedings of the Conference on Office Information Systems (COIS '90)*, pp. 249-260.

Linton, M. (1992) "Encapsulating a C++ Library." In *Proceedings of the USENIX C++ Conference*, pp. 57-66.

Linton, M. A., Calder, P. R. and Vlissides, J. M. (1988) "InterViews: A C++ Graphical Interface Toolkit." Research Report CSL-TR-88-358, Stanford University.

Linton, M. A., Vlissides, J. M. and Calder, P. R. (1989) "Composing User Interfaces with InterViews." *IEEE Computer*, **22**(2).

Liskov, B., Gruber, R., Johnson, P. and Shrira, L. (1991) "A Highly Available Object Repository for Use in a Heterogeneous Distributed System." In *Fourth International Workshop on Implementing Persistent Object Bases: Principles and Practice*, pp. 255-266, A. Dearle (ed).

Lu, I. and Mantei, M. (1991) "Idea Management in a Shared Drawing Tool." In *Proceedings of the 2nd European Conference on Computer Supported Cooperative Work (EC-CSCW '91)*.

Mantei, M. (1989) "Observation of Executives Using a Computer Supported Meeting Environment." *Decision Support Systems*, **5**, pp. 153-166.

McCall, K., Moran, T., van Melle, B., Pedersen, E. and Halasz, F. (1992) "Design Principles for Sharing in Tivoli, a Whiteboard Meeting-Support Tool." In *Workshop on Real Time Group Drawing and Writing Tools at CSCW '92*, Toronto, Canada.

Minneman, S. L. and Bly, S. A. (1991) "Managing a trois: A study of a multi-user drawing tool in distributed design work." In *Proceedings of the ACM CHI'91 Conference on Human Factors in Computing Systems*, pp. 217-224.

Muller, M. J. (1991) "PICTIVE — An Exploration in Participatory Design." In *Proceedings of the ACM CHI'91 Conference on Human Factors in Computing Systems*, pp. 225-231.

Mulligan, R. M., Altom, M. W. and Simkin, D. W. (1991) "User Interface Design in the Trenches: Some Tips on Shooting from the Hip." In *Proceedings of the ACM CHI'91 Conference on Human Factors in Computing Systems*, pp. 232-236.

Myers, B. and Rosson, M. (1992) "Survey on User Interface Programming." In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI '92)*, pp. 195-202.

Norman, D. (1988) *The Psychology of Everyday Things*, Basic Books, Inc., New York.

Norman, D. A. (1986) *User Centered System Design: New Perspectives on Human-Computer Interaction*, Lawrence Erlbaum Associates, Hillsdale, NJ.

Nunamaker, J. F., Dennis, A. R., Valacich, J. S., Vogel, D. R. and George, J. F. (1991) "Electronic meeting systems to support group work." *Communications of the ACM*, **34**(7), pp. 40-61.

Ousterhout, J. K. (1990) "Tcl: An Embeddable Command Language." In *Proceedings of the 1990 Winter USENIX Conference.*

Ousterhout, J. K. (1991) "An X11 Toolkit Based on the Tcl Language." In *Proceedings of the 1991 Winter USENIX Conference.*

Parrington, G. D. and Shrivastava, S. K. (1988) "Implementing Concurrency Control in Reliable Distributed Object-Oriented Systems." In *Proceedings of ECOOP '88*, pp. 233-249.

Patterson, J. F. (1991) "Comparing the programming demands of single-user and multi-user applications." In *Proceedings of the Fourth Annual Symposium on User Interface Software and Technology (UIST '91)*, pp. 87-91.

Patterson, J. F., Hill, R. D., Rohall, S. L. and Meeks, W. S. (1990) "Rendezvous: An architecture for synchronous multi-user applications." In *Proceedings of the Conference on Computer-Supported Cooperative Work (CSCW '90).*

Pendergast, M. (1992) "GroupGraphics: Prototype to Product." In *Workshop on Real Time Group Drawing and Writing Tools, CSCW '92*, Toronto, Ontario.

Pendergast, M. and Hayne, S. (1992) "Alleviating convergence problems in group support systems: The shared context approach." Working paper , Department of MIS, University of Calgary.

Rittel, H. and Webber, M. (1973) "Dilemmas in a general theory of planning." *Policy Sciences*, **4**, pp. 155-169.

Root, W. R. (1988) "Design of a multi-media vehicle for social browsing." In *Proceedings of the Conference on Computer-Supported Cooperative Work (CSCW '88)*, pp. 25-38.

Roseman, M. (1993) "GroupKit: User's Guide and Reference Manual." Technical Report 93/509/14, Dept. of Computer Science, University of Calgary.

Roseman, M. and Greenberg, S. (1992a) "GroupKit: A Groupware Toolkit (Interactive Poster)." In *ACM Conference on Human Factors in Computing Systems (CHI '92).*

Roseman, M. and Greenberg, S. (1992b) "GroupKit: A Groupware Toolkit for Building Real-Time Conferencing Applications." In *Proceedings of the Conference on Computer-Supported Cooperative Work (CSCW '92).*

Scrivener, S. A. R., Garner, S. W., Palmen, H. K., Smyth, M. G., Clark, S. M., Clarke, A. A., Connolly, J. H. and Schappo, A. (1992) "CSCW research using product design: Findings from proximal studies." Submitted to *Design Studies* .

Stefik, M., Bobrow, D. G., Foster, G., Lanning, S. and Tatar, D. (1987) "WYSIWIS revised: Early experiences with multiuser interfaces." *ACM Transactions on Office Information Systems*, **5**(2), pp. 147-167.

Stroustrup, B. (1986) *The C++ programming language*, Addison-Wesley, New York.

Tang, J. C. (1991) "Findings from observational studies of collaborative work." *International Journal of Man Machine Studies*, **34**(2), pp. 143-160.

Tang, J. C. and Minneman, S. L. (1990) "Videodraw: A video interface for collaborative drawing." In *Proceedings of the ACM CHI'90 Conference on Human Factors in Computing Systems*, pp. 313-320.

Tatar, D. G., Foster, G. and Bobrow, D. G. (1991) "Design for conversation: Lessons from Cognoter." *International Journal of Man Machine Studies*, **34**(2), pp. 185-210.

Vlissides, J. M. and Linton, M. A. (1989) "Unidraw: A Framework for Building Domain-Specific Graphical Editors." In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology.*

Appendix A

User's Guide and Reference Manual

This appendix contains a subset of the GROUPKIT User's Guide and Reference Manual, available in its entirety with the GROUPKIT software distribution or as:

Roseman, M. (1993) "GROUPKIT: User's Guide and Reference Manual." Technical Report 93/509/14, Dept. of Computer Science, University of Calgary.

The excerpt from the User's Manual discusses the creation of a very simple conference program, which does not even communicate with other conference program. This serves as a starting point for an example which is built up through the rest of the manual to add communications and overlays. The excerpt here discusses creating the various parts of the program, compiling it, integrating it with the GROUPKIT registration system, and running it.

The excerpt from the Reference Manual is part of a section on "Conference Support." The manual is structured so that related components are considered together, with a brief introduction explaining the relationships. Individual components are described with a complete list of methods contained in the component, often with reference to other components which use those methods, as well as examples of how the components are used by the application developer.

A Simple Example

About the Example

This section will start to build a GroupKit program. The example will be a multi-user version of the "patch" example contained in the InterViews distribution. The program uses an InterViews "deck" object to display one of three strings (cards) contained in the deck. Buttons for "previous" and "next" allow switching between the cards.

The multi-user version will broadcast changes to all users, so that when one user moves to a different card, other users will see the same card. The communications routines are detailed in the next section; this section describes how to get the basic single-user program up and running in the GroupKit framework.

Designing the ConferenceGlyph

The first step is designing the InterViews glyph which will manage the user interface. GroupKit programs subclass glyphs from the class ConferenceGlyph, which is an InterViews InputHandler containing a pointer to a Conference object. The Conference object allows the glyph to communicate with other conference users.

The glyph will contain the deck, a patch to redraw the deck when it changes, and buttons for previous and next. Callback routines for the buttons will be necessary, as well as a function "flip" which will be used by the callbacks. The declaration looks like this:

```
class FlipGlyph : public ConferenceGlyph {
public:
    FlipGlyph( WidgetKit*, Conference* );
private:
    void prev();
    void next();
    void flip(GlyphIndex cur);
    Patch* patch_;
    Deck* deck_;
};
```

The glyph will need to define callbacks for the buttons, which must be declared as in normal InterViews:

```
declareActionCallback(FlipGlyph)
implementActionCallback(FlipGlyph)
```

The constructor simply creates all the necessary components:

```
FlipGlyph::FlipGlyph(WidgetKit* kit, Conference* conf)
:
   ConferenceGlyph(nil, kit->style(), conf)
{
  const LayoutKit& layout = *LayoutKit::instance();
  deck_ = layout.deck(
    kit.label("Hi mom!"),
    kit.label("Big Bird"),
    kit.label("Oscar")
  );
  patch_ = new Patch(deck_);

  body(
    kit.inset_frame(
      layout.margin(
        layout.vbox(
          patch_,
          layout.vglue(5.0),
          layout.hbox(
            kit.push_button(
                   "Next",
                    new ActionCallback(FlipGlyph)
                           (this, &FlipGlyph::next)),
            layout.hglue(10.0),
            kit.push_button(
                    "Previous",
                     new ActionCallback(FlipGlyph)
                             (this, &FlipGlyph::prev))
          ),
        ),
        10.0
      )
    )
  );
}
```

Next, the callback functions, "next" and "prev" are defined, along with the routine "flip" which actually alters the deck:

```
void FlipGlyph::prev() {
  GlyphIndex cur = deck_->card() - 1;
  flip( cur == -1 ? deck_->count() - 1 : cur );
}

void FlipGlyph::next() {
  GlyphIndex cur = deck_->card() + 1;
  flip ( cur == deck_->count() ? 0 : cur );
}

void FlipGlyph::flip(GlyphIndex cur) {
  deck_->flip_to(cur);
  patch_->redraw();
}
```

The Main Program

The main program is similar to main programs in InterViews. The one change is that rather than instantiating a Session object, a "GroupSession" is instantiated. A GroupSession defines several GroupKit style attributes, as well as parsing command line arguments received by all GroupKit programs from the registration system. The GroupSession object uses this information to instantiate a Conference object, which is needed by the ConferenceGlyph subclass defined previously.

```
int main(int argc, char** argv) {
  GroupSession* session =
        new GroupSession("DeckFlip", argc, argv);
  session->run_window(
    new ApplicationWindow(
      new FlipGlyph(WidgetKit::instance(),
                    session->conference())
    )
  );
}
```

Finally, some include files and callback declarations are needed at the beginning of the program:

```
#include <IV-look/kit.h>
#include <InterViews/layout.h>
#include <InterViews/style.h>
#include <InterViews/action.h>
#include <InterViews/patch.h>
#include <InterViews/window.h>

#include "conference.h"
#include "groupsession.h"
#include "confglyph.h"
```

Compiling the Program

GroupKit uses Imakefiles to compile programs. Assuming the above program was stored in "deckeg.c", the following Imakefile would be used to compile it:

```
#ifdef InObjectCodeDir

Use_libInterViews()
MakeObjectFromSrc(deckeg)
GroupKitProgram(deckeg, deckeg.o)

#else

MakeInObjectCodeDir()

#endif
```

A script called "gkmkmf" should be installed, which works like the InterViews "ivmkmf" but also includes the necessary paths and Imakefile macros to easily build GroupKit programs. To compile the example use the following commands:

```
% gkmkmf
% make Makefiles
% make depend
% make
```

Informing the Registration System

GroupKit conference programs are not run directly by the user, but instead started by the registration system, which informs the conference program (through a number of command line arguments) about various things it needs to run, such as the location of the registration system (so that the conference can connect to the registration system and receive information about other users).

The registration system needs to know two things about each program that it can start. The first is the name (including path) of the executable programs which run the conference applications. The second is a brief description of the conference program, suitable for displaying to the user. These are specified through the standard X resources mechanism (i.e. .Xdefaults file). Assuming the deck flip executable was renamed to "deckeg" and placed in the "/home/grouplab/bin/SUN4" directory, the following lines could be added to the .Xdefaults file:

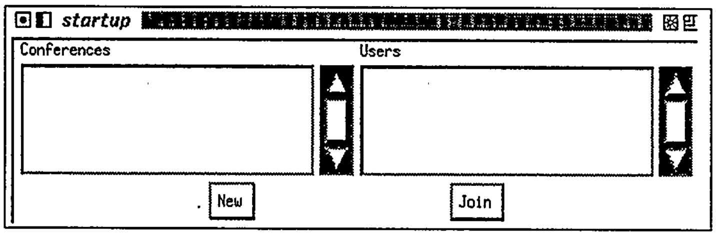
```
startup*GroupKitBinDir:        /home/grouplab/bin/SUN4
startup*conferenceTypes:       1
startup*conf1-desc:            Deck Flip Example
startup*conf1-prog:            deckeg
```

Here, "startup" is the name of the user registration program. Any number of conferences can be specified, by altering the "conferenceTypes" resource. Each program must have both a "conf<n>-desc" resource and also a "conf<n>-prog" resource.

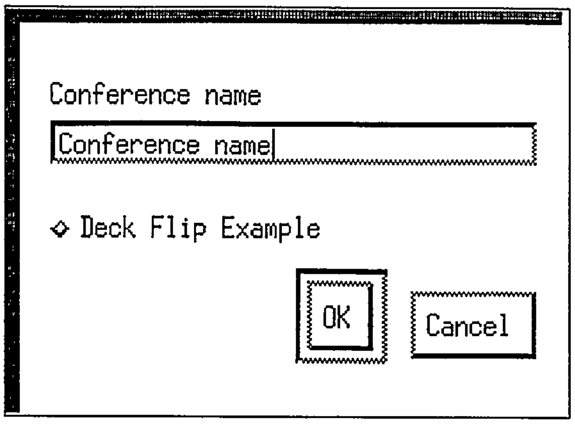
Running the Program

To run the program, first of all make sure the GroupKit registrar is running. If it is not running, start it up by typing "registrar &". Note that the registrar must run on a certain machine (specified by the "RegistrarHost" resource in the file "core-src/groupsession.c").

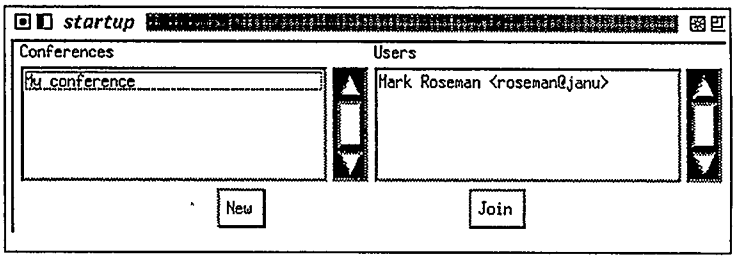
Next, run the program "startup", which provides a client to the registrar. This program expects to find the registrar at the host and port number identified by the RegistrarHost and RegistrarPort resources, but this can be overridden with the "host=<name>" and "port=<number>" command line options. This should bring up the window shown below. Note the appearance may be slightly different, depending on the default InterViews WidgetKit used (the screen dumps here show the monochrome WidgetKit).



Clicking the "New" button should bring up the dialog box below, asking you to create a new conference. The name of the conference can be entered in the field editor at the top, and the radio button for "Deck Flip Example" should be selected. If more conferences were specified in the .Xdefaults, more radio buttons would be in the dialog.



Clicking OK should start up the Deck Flip program in another window. The Conference list in the registrar client will be updated to include the newly created conference. Selecting the conference name will give a list of its users in the other list.



Other users may join the conference by selecting the conference name and pressing the "Join" button. This will bring up copies of the Deck Flip program on their screens, but right now the programs are behaving independently; the next section describes the routines necessary to communicate between the programs.

CONFERENCE SUPPORT

GroupKit applications are called conferences. Facilities are provided in GroupKit to minimize the work needed to maintain conferences. The center of this is the **CONFERENCE** object. It provides facilities for:

- accepting new users into the conference
- maintaining a list of users in the conference
- sending messages to one or more users in the conference
- setting up callbacks for messages received
- notifying other objects when users enter or leave the conference

Much of the application specific functionality tends to reside in the user interface portions of the application (in CONFERENCEGLYPHS). These glyphs will use the facilities provided by the CONFERENCE object for their communications needs. If these glyphs need to know about users entering or leaving the conference, they can be subclassed from **CONFERENCEMONITOR**, and register themselves with the CONFERENCE.

The CONFERENCE objects rely on the registration components for information about users joining and leaving the conference. The **COORDINATOR** serves as an interface between the registration components and one or more CONFERENCES. It will create one or more CONFERENCES (at the request of the registration system) that all share a common registration system.

As well, GroupKit applications require a number of parameters to control how they behave (e.g. names of conferences, host and port numbers of different components). These parameters come from X11 resources and command line arguments. The **GROUPSESSION** object augments the standard InterViews SESSION to handle GroupKit specific parameters.

Finally, a number of support classes help maintain information throughout GroupKit, both for internal use and transmission. An **AVPAIR** holds a single attribute, value pair, while an **ATTRIBUTELIST** holds a list of AVPAIRS. A table of ATTRIBUTELISTS (indexed by an integer id number) can be stored in a **ATTRLISTTABLE**, while tables of these tables (useful for holding all the users in all the conferences for example) can be stored in a **USERLISTTBL**.

CLASS: CONFERENCE
CATEGORY: CONFERENCE SUPPORT
SUPERCLASS: RPCPEER*

A CONFERENCE maintains connections to other users, as well as the user's COORDINATOR, serving as the center of communications for GroupKit applications.

Conference(char* name, const char* coord_host, int coord_port, int confnum);

Create a new CONFERENCE object, given the name of the conference, the host and port number of the COORDINATOR which initiated the CONFERENCE, and the id number of the CONFERENCE. The CONFERENCE initiates a CONNECTION to the COORDINATOR in order to receive registration information, such as announcements of new users. A CONFERENCE can also be created using the GROUPSESSION object.

void newUser(AttributeList* user_attrs);
void userLeaving(int id);

These routines are called whenever a new user joins the CONFERENCE, or an existing user leaves the CONFERENCE. The default behavior is to pass the notification on to a list of CONFERENCEMONITOR objects, which can take appropriate action, such as updating application data structures to reflect the new users.

boolean createReaderAndWriter(const char* host, int port);
void createReaderAndWriter(int fd);

These routines are used when initiating or accepting new socket connections (to the COORDINATOR and from other CONFERENCE objects respectively).

void info_callback(char* msg, class Connection* c);

This routine is used as the target of the INFOCONNECTION, interpreting information about the remote CONFERENCE user and storing it in the CONNECTION object.

void connectTo(char* msg);

This routine is activated as a callback from the COORDINATOR, and initiates a CONNECTION to a new CONFERENCE user.

void youAreID(char* msg);

This routine is activated as a callback from the COORDINATOR, and provides the CONFERENCE object with information such as the id number, etc. After this information is received, the NEWUSER method is called, so care should be taken so as not to "add" the local user twice.

void removeUser(char *msg);

This routine is activated as a callback from the COORDINATOR, and signifies a user is leaving. The default behavior is to call USERLEAVING.

void deleteConference(char *msg);

This routine is activated as a callback from the COORDINATOR, signifying that the CONFERENCE should be destroyed.

CLASS: CONFERENCEMONITOR
CATEGORY: CONFERENCE SUPPORT
SUPERCLASS: <NONE>

A CONFERENCEMONITOR can be notified when users enter or leave a CONFERENCE.

virtual void newUser(AttributeList* user_attrs) =0;
virtual void userLeaving(int id) = 0;

These methods are called when users enter or leave the CONFERENCE.

Notes:

Typically a CONFERENCEMONITOR is used as one of the base classes (along with a CONFERENCEGLYPH perhaps) for a new class that is interested in knowing when users enter or leave the CONFERENCE. To register a CONFERENCEMONITOR with a given CONFERENCE (from the CONFERENCEMONITOR constructor) the following code might be used:

```
conference->monitors()->append(this);
```

CLASS: COORDINATOR
CATEGORY: CONFERENCE SUPPORT
SUPERCLASS: RPCPEER*

A COORDINATOR acts as an intermediary between the registration system and one or more CONFERENCEs.

Coordinator(Style* style);

Create a new Coordinator.

virtual void createReaderAndWriter(int fd);

Accept a connection from one of the CONFERENCE objects we spawned. A callback is set up so the Conference can send us information to identify itself.

virtual boolean createConference(char* name, char* type, int id);

Create a new CONFERENCE, in response to a request from the registration mechanisms. The type of CONFERENCE to create is specified by the second parameter, which maps to a conference description resource. Resources are specified in the following manner:

```
*GroupKitBinDir:          <location of GroupKit binaries>
*conferenceTypes:         2
*conf1-desc:              Group Sketchpad
*conf1-prog:              gs
*conf2-desc:              Brainstorming Tool
conf2-prog:               bstorm
```

The command to start the conference is found by concatenating the `GroupKitBinDir` resource with the appropriate `conf*-prog` resource, determined by searching the `conf*-desc` resources for a match with the type parameter. Command line options are appended which specify the name and id number of the conference, as well as the local host and port number for the COORDINATOR (used so the CONFERENCE can connect to us when it is created). This command line is then used to spawn a new process to run the CONFERENCE. A sample command line might be:

```
/home/grouplab/bin/SUN4/bstorm -confname 'Ideas' -confid 23 -
     coordhost janu -coordport 1500
```

void deleteConference(int id);

Delete the specified CONFERENCE, by passing on the request to the appropriate CONFERENCE object.

void deleteUser(int conf-id, int user-id);

Delete the specfied user from the specfified CONFERENCE, by passing on the request to the appropriate CONFERENCE object.

void setLocalInfo(int conf-id, int user-id, char* userid, char* name);

Information for one of our CONFERENCES has been received, which we pass on to the CONFERENCE object itself.

void joinTo(int conf-id, int user-id, char* userid, char* name, char* host, int port);
void addrReq(char *s, class Connection*);
void addrResp(char *s);

The joinTo routine specifies that one of our Conferences should connect to the Conference run by the indicated user. However, the host and port number specify the address of the user's RegistrarClient, not the Conference itself (since the Registrar only stores information on the RegistrarClient). To get around this, the joinTo routine initiates a connection to the specified RegistrarClient to find out the Conference's host and port number. The RegistrarClient delegates this request to its attached Coordinator, specifically the addrReq method. The addrReq method in the remote Coordinator looks up the host and port number of the Conference and sends it back to the local Coordinator. This information is received by the addrResp routine, which then passes on all the required information to the Conference object to connect to the remote Conference.

RegistrarClient* rc_;

A pointer back to our RegistrarClient, mainly used to inform the RegistrarClient when one of our attached Conferences dies.

Notes:

Because the Conference objects are running as separate processes which need to connect to the Coordinator, there is some time between when a Conference is created and the Coordinator can send it messages. As a result, all messages to the Conference are queued up before the Conference has hooked up, and these are sent as soon as the Conference does hook up.

CLASS: GROUPSESSION
CATEGORY: CONFERENCE SUPPORT
SUPERCLASS: SESSION*

A GROUPSESSION is a SESSION which helps manage a GroupKit application.

GroupSession(const char* name, int& argc, charargv, const OptionDesc* = nil, const PropertyData* = nil);**

Create a new GROUPSESSION, specifying the name used for looking up resources, command line arguments, and any user-defined properties (resources) or options (mappings of command line options onto resources). The GROUPSESSION augments these with standard InterViews properties and options as well as GroupKit specific ones.

Conference* conference();

Create a new CONFERENCE object, based upon information which should be present in the command line arguments.

Usage example:

```
static PropertyData props[] = {
  { "*width", "750" },
  { "*height", "300" },
  { nil }
};

static OptionDesc options[] = {
  { "width=", "*width", OptionValueAfter },
  { "height=", "*height", OptionValueAfter },
  { nil }
};

int main(int argc, char** argv) {
  GroupSession* session =
              new GroupSession("BrainStorm", argc, argv, options, props);
  session->run_window(
    new ApplicationWindow(
      new BrainStormGlyph( session->style(), session->conference())));
}
```