An Architectural Design of A Toolkit for Synchronous Groupware Applications

Nobuhisa Yoda and Brad A. Myers

16 December 1994
CMU-CS-94-226

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This report is a revised version of a Masters thesis submitted to the Information Networking Institute, Carnegie Mellon University, as a partial fulfillment of requirements for a Master of Science degree.
Also appears as Information Networking Institute Technical Report CMU-INI-TR-1994-7 and Human Computer Interaction InstituteTechnical Report CMU-HCII-94-109.

Abstract

The central question studied in this thesis is how to design high-level reusable components for developing synchronous groupware applications. A modern user interface requires the separation of applications into two components, namely application and graphical user interface. An effective architecture for synchronous, multiple-user groupware applications requires a separation of one more component from the application: the management of the session and shared data. This component provides a framework that manages the collaborative user session, shared data in an abstraction form, and floor control. This thesis calls this separation "coordination independence." This thesis demonstrates the feasibility of the architecture by describing an experimental system, called TALISMAN, using Garnet. TALISMAN is implemented as a single-process, centralized system with multiple remote graphical displays. The system which features this separation relies upon object inheritance and the constraint satisfaction mechanism in the underlying object system. The architecture also makes the conversion of a single-user application into a multiple-user application relatively easy.

This research was sponsored by NCCOSC under Contract No. N66001-94-C-6037, ARPA Order No. B326. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NCCOSC, ARPA, or the U.S. Government.

Keywords: Groupware, toolkit, software architecture, human-interface, graphical toolkit

PREFACE AND ACKNOWLEDGMENTS

This research was done in conjunction with the overseas international staff education program sponsored by Toshiba Corporation, Japan. I thank the corporation and especially the Toshiba Multimedia Engineering Laboratory for the generous support. I would also like to thank my advisor, Dr. Brad A. Myers for extensive guidance in achieving a success throughout this thesis research. I thank Andy Mickish and Richard McDaniel for technical advice on the Garnet system. Finally, I would like to thank my wife, Naoko, for all the support.

LIST OF ABBREVIATIONS

CSCW Computer Supported Cooperative Work

Garnet Generating an Amalgam of Realtime, Novel Editors and Toolkits

TALISMAN A Toolkit and Application Layer Interface for Synchronous Multiple-user Access over Network

CHAPTER 1.

INTRODUCTION AND MOTIVATIONS

Recent advancements in information networking technologies have enabled people at a large distance to communicate more rapidly and efficiently than ever. Vast opportunities exist for people to collaborate by using groupware. Communication among people over computer networks includes both asynchronous and synchronous interactions. Most commercial applications, including electronic mail and workflow management systems, are message-based asynchronous groupware because people collaborate at different times. A communication delay is permissible in such asynchronous groupware. Various applications as well as their development kits have been commercially introduced to help users customize asynchronous groupware systems.

Synchronized interactions, on the other hand, require a real-time exchange of information among session participants with some form of floor control. Examples of floor control policies include people "taking turns" by alternating input, and simultaneous editing where everyone can edit at the same time. Possible application domains include window sharing, text-based real-time conferencing, telephony, and multiple-player games. Currently, however, applications for synchronized communications are often primitive and low-level from the perspectives of software extendibility and reusability. Although there have been various papers addressing low-level issues of toolkit and language implementations, a software architecture that features a high-level, reusable framework is still under research. It is often the case that synchronous groupware is tailor-made for one particular program. Extending such groupware applications is very difficult because the typical system architecture is not designed for different applications. The availability of reusable system components would help improve the efficiency of software development.

An effective architecture with reusable components for synchronous groupware applications requires a separation of one more component from the application: the management of the session and shared data. This component provides a framework that manages the collaborative user session, shared data in an abstraction form, and floor control. This thesis calls this separation "coordination independence."

TALISMAN is a prototype system that provides reusable components based on the centralized data management configuration. Through TALISMAN, this thesis attempts to demonstrate that coordination independence, in addition to dialogue independence, is a useful concept to help develop synchronous groupware applications. TALISMAN is written in LISP using the Garnet [13] user interface development environment.

CHAPTER 2.

A BRIEF SURVEY OF APPROACHES TO THE PROBLEM

2.1. Screen Sharing

As Kohlert, et. al. state [10], there have been numerous attempts to design and implement "multi-user interface toolkits" for synchronous groupware applications. There are several different approaches to design these toolkits. One of the approaches is to share contents of the entire screen display among people. While the shared screen may contain more than one window, there is usually no floor control or conference management features. Examples of such an approach include NLS and MBLink systems[10]. The screen sharing is accomplished by implementing special mechanisms to capture and distributed screen images at a low level in the system.

This approach does not require application programs to be aware of the screen being shared by multiple users. The system can replicate screen images without making any changes to the application code.

2.2. Window Sharing

Another approach is to share windows among users without modifying original applications. The window sharing approach is characterized by using special mechanisms within the multiple-window system to handle drawing objects in replicated windows as well as to propagate input events across replicated windows. Shared X from HP[3] is an example of this approach. Shared X not only allows displaying replicated windows across network, but also accepts user inputs from multiple hosts to the window. The window sharing approach has been the most popular among commercial applications because users can share windows from any applications. On the other hand, the systems based on this approach cannot provide the fine floor control based on application-specific rules because the input control mechanism is embedded within a low layer of the window system.

2.3. Collaboration-aware Window Sharing

A window sharing system such as Art Windows [10] features specialized window management and input event propagation policies to share windows, while the application is also aware of the window sharing feature of the underlying window system. The Groupkit system demonstrates a centralized user registration mechanism under a distributed processing environment [5, 6, 16]. After registering a user in the central registrar, the system maintains the system state consistency by peer-to-peer application process communication with application-dependent communication protocols. The Groupkit system also uses specialized low-level mechanisms, such as an overlay of windows, to provide a collaborative environment for multiple users.

2.4. Data Sharing Through Dialogue Independence

An approach which seems promising attempts to separate application-specific components from user-interface components at the system architecture-level [10]. The key concept behind this approach is data sharing based on dialogue independence, where the user-dialogue component is separate from the application-specific, computational component

[1]. Systems such as Rendezvous [8, 9, 15] and LIZA [2] are examples of this approach. The LIZA toolkit uses a server-client structure, where a single server maintains the global state information and multiple clients are responsible for the user interface [3]. LIZA extends the Suite toolkit to modify a single-user application to a simultaneous multi-session application.

The Rendezvous system architecture is based on an object-oriented model. The system uses an "Abstraction-Link-View (ALV)" model [9] to relate abstract information with multiple views for individual users. According to the model, an application designer must classify all the information as either abstract data or view data. The abstract data is shared among multiple users. The designer also must declare how abstract data and view data are converted to each other. A link is an entity that connects two graphical objects by determining one's characteristics in terms of another's. The system also supports a special type of link called the "Tree Maintenance Link" to define a relationship between an abstraction object and a view object.

While this dialogue-independent architecture is very effective for multi-user applications, we believe that the two-component architecture is not sufficient to provide highly reusable components for synchronous groupware applications. The problem with the two-component architecture for groupware systems is it does not separate the multiple-user coordination management from the application. The coordination management component provides services for floor control, telepointing, and also controls how and when to update abstract data and views. Systems based on the two-component architecture may suffer in changing the session and shared data management functions because the shared abstract data component is embedded in the application-dependent code.

To provide a high-level architecture with reusable components, we need to enable software developers to reuse not only the graphics dialogue components but also the user coordination aspects such as floor control, telepointing, and updating the abstraction and views. We term this separation mechanism "coordination independence." An effective high-level architecture for synchronous groupware requires both dialogue independence and coordination independence.

2.5. Appropriate Interface Levels for Network Communications

In order to facilitate collaborative work among multiple users, the synchronous groupware system must provide data communication services across the network. Different design approaches have resulted in different layers as the entry point for network communications. There are two major approaches to provide networking services. First, the groupware toolkit itself provides network communications. For example, Groupkit [16] features peer-to-peer inter-process communications across the network to pass all of the groupwork-related information, including the user registrations and application-specific messages. The other method is to use the server-client networking services of the underlying window system. For example, Rendezvous uses the X Window System [9], and the data sent across the network are the X protocol messages between the X server in the user terminals and the host on which the groupware application and toolkit reside.

The pros and cons of the two approaches depend on how the system attempts to strike the balance between the system performance and the support for heterogeneous system platforms. The former approach can be advantageous in terms of the system performance, because the communication data format of the groupware application can be optimized at the groupware toolkit level to assure high efficiency. Systems with the former approach can achieve a better communication-overhead than those with the latter approach because

the information sent across network is likely to be shorter, more abstract than the information sent across network by the window systems. Moreover, systems with the former approach can reduce the number of messages by using broadcast messages.

On the other hand, the latter approach can be more reasonable if the groupware emphasizes the heterogeneous platform use without facing the burden of maintaining software for different platforms. For example, one of the unique strength of the X Window System is that the X Protocol is machine-independent. There are many X servers available across various system platforms, encompassing both workstations and personal computers. By using the latter approach, the groupware system can appreciate the strength of the X Window System. The groupware toolkit can be implemented on only one platform, and still support collaborative sessions across different machines. A simpler system maintenance is another benefit of this approach because the groupware toolkit needs to be maintained only on one platform.

A serious drawback of the latter approach is a possible high communication overhead caused by the X Protocol messages transmissions. The problem becomes significant especially if the groupware application is graphics-intensive. For example, a communication overhead problem may arise when a groupware application displays movements of mouse cursors of other users on one user's window. To notify movements of the mouse cursors, at least one X protocol message that contains mouse cursor coordinates must travel through the network whenever a user moves a mouse at a remote location. Moreover, if the remote mouse cursor is displayed at multiple locations, separate X protocol messages must travel through the network because the X protocol does not have any message broadcasting capability.

The present implementation of TALISMAN is based on the latter approach. All the communications between hosts are done at the X protocol level. All the data copying among groupware toolkit components occurs within the object system supported by Garnet in a single host.

CHAPTER 3.

FUNCTION REQUIREMENTS AND ASSUMPTIONS

3.1. Requirements

Previous studies have listed some functional requirements for synchronous groupware. Roseman et. al.[16] summarized technical requirements for a groupware toolkits into the following two sets with three specific aspects in each set. These requirements, which were derived empirically, assume the groupware system consists of distributed programs that execute concurrently.

Technical support of distributed processes
Conference Management
Communications Infrastructure
Persistent Sessions
Technical support of a graphics model
Shared Visual Objects
Object Concurrency Control
Separate View from Representation

Table 1. Roseman's requirements for a groupware toolkit

Roseman et. al. categorized the requirements into two areas: the distributed process management and the graphical object management. The former area concerns coordinating multiple users while the latter deals with sharing views across a network. This classification is useful for deciding requirements for TALISMAN.

Since our aim is to provide a toolkit for the groupwork environment where each view represents a user-dependent interpretation of the shared abstraction data, we isolate application-dependent requirements from the system requirements. For instance, suppose two users share data such that one user uses a scroll-bar widget while the other uses a text box widget to view the data. The process of one participant changing a value by scrolling a square box in a scroll-bar widget is difficult, if not impossible, to convey to another participant where the value is displayed just as a number. The two widgets add different semantics for manipulating the data. There seems to be a need for a level of abstraction higher than the widgets to show the process of changing values. Such a level of abstraction may depend on the context of the application. Thus, conveying the process of creating artifacts to express ideas is not always possible in a groupware toolkit if the process of creating the artifacts differs in one participant from another.

By examining the requirements shown by the past studies, we propose the following requirements for a toolkit that provides reusable components for synchronous groupware applications.

- Isolate components for the view-dependent processing from components for both the collaboration management and for the application-dependent computation;
- Provide an underlying mechanism to efficiently transmit messages among views and the conference management to change and update views;
- Distinguish the globally propagated events from the events that are local to respective views;
- Localize view-specific events within a view;
- Provide an underlying mechanism to control input permissions for users;
- Provide a mechanism to process late comers / early leavers of a groupwork session;
- Allow gesture communications by displaying participants' telepointers;
- Minimize the overhead caused by redrawing views;
- Not hinder the capabilities provided by the underlying graphical user interface system;
- Be capable of using the existing widgets;
- Take advantage of the capabilities of the existing underlying system.

3.2. Assumptions

In developing TALISMAN, there are several important assumptions that were made. These assumptions were needed to both simplify the system design and to satisfy the time constraints imposed on the research.

TALISMAN assumes that the applications based on this architecture will be used in conjunction with other communication means such as voice over telephone lines. The gesture communications provided by telepointers are intended to supplement the voice communications.

The present study assumes that users of the system work in a reasonably constructive manner. Users are in general assumed not to concurrently input to the system, especially on the graphical objects that correspond to the same abstraction data. In any groupware system that allows concurrent user inputs, data consistency is an important issue. Ideally, the system should update the state of data fast enough so that more than one user input can never occur at once. In practice, however, both the latency of network data communication and the overhead of updating the system state can allow multiple users to input at once. As a compromise, the system should take only the first input at a particular state of data, and reject the rest of the inputs until the system completes updating the state. Use of the voice communication may solve such a problem by negotiating before randomly changing the state of the shared data.

TALISMAN also makes the assumption that the size of the shared abstraction data is reasonably small, and the data copying among objects does not cause significant overhead. Moreover, the structure of the shared abstraction data and the structures used in respective views are similar enough to avoid a large overhead in transforming data between a view and the shared abstraction data.

TALISMAN further assumes that users who collaborate by using the system are not concerned about eavesdropping and hostile intrusions from outside. In order to support a secure collaboration environment, the system would need to provide security mechanisms such as user authentication and encryption of information transmitted across network. Although the present experimental implementation is not required to demonstrate such

features, the system should ideally be designed to show that the system can later incorporate modules that add these capabilities.

CHAPTER 4.

ARCHITECTURE OF THE TALISMAN GROUPWARE TOOLKIT

4.1. An Overview

Figure 1 depicts a conceptual architecture of our system. This structure extends the idea of the balanced control architecture for human-computer interface management from Hartson and Hix [7] to synchronous groupware applications. From the perspective of a low-level user interface system, the control structure provides an event-based, asynchronous dialogue mechanism. From the higher level view-point of the groupware system, however, the structure provides the balanced control architecture of a synchronous dialogue coordination system, where the user coordination mechanism is explicitly separated from the application-specific computation. The architecture provides high levels of application interface abstraction, the notion that Lane [11] claims to be essential to satisfying requirements for user interface adaptability across devices.

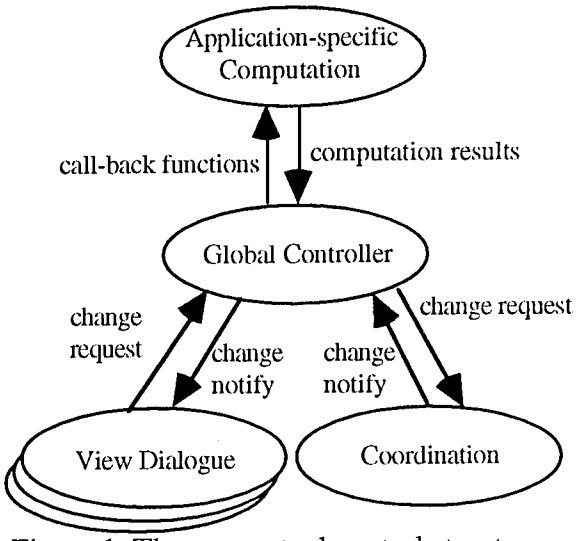


Figure 1. The conceptual control structure.

The entire system consists of four components shown in Figure 1: the View Dialogue, the Coordination, the Application-specific Computation, and the Global Controller. This control structure makes programming groupware easier because all but the Application-specific Computations provide reusable functions as described below.

The View Dialogue component makes sure that the state of each view is consistent with the shared abstract data by relaying user inputs from the underlying graphical interface system to the Coordination component, and by updating the graphical display based on the abstract data. The Application-specific Computation component contains all of the code specific to the application. The Global Controller component provides the underlying mechanisms to facilitate communication among the other three components.

For example, using a Tic-tac-toe game, the View Dialogue component manages graphical objects for each player, and displays the Tic-tac-toe game board with nine cells and game

pieces. The Coordination component stores the states of the nine cells as the shared abstract data, and controls which player can play a piece. When a player places a game piece on the board, the View Dialogue component requests the Global Controller component to notify the Coordination component to update the state of the corresponding cell in the shared abstract data. The Global Controller component receives from the View Dialogue component where a player wants to place a game piece. It then invokes an application-specific Tic-tac-toe rule function to determine the state of the game. This includes code to compute whether a player is placing a piece at a valid place based on the states of the nine cells. If it is a valid move by the player, the Global Controller component notifies the Coordination component to update the state of the shared game cells.

The TALISMAN prototype implementation based on the Garnet system takes advantage of the prototype-instance model, the Garnet invalidate demon mechanism, and the one-way constraint mechanism. The prototype-instance model ensures that values of every object are automatically copied to their instances. The Garnet invalidate demon mechanism enables the system to trigger a function when there are changes in an object's value. The one-way constraint mechanism enables an object to declare its relationship to other objects, and maintains the relationship as the values of other objects change [13, 14].

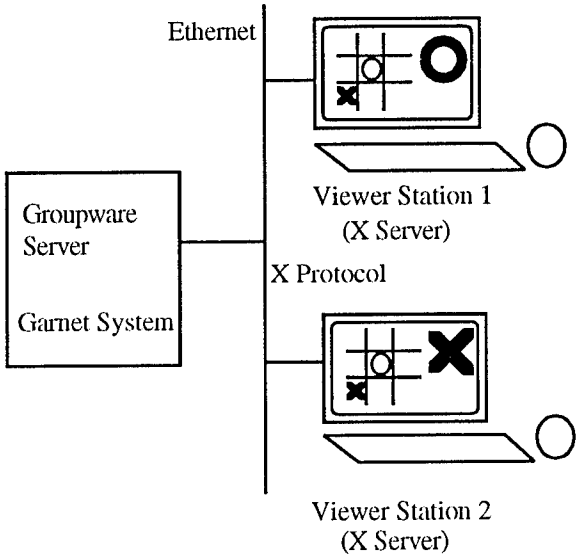


Figure 2. A system configuration for the prototype implementation.

TALISMAN was implemented as a multi-threaded centralized system where all the session management is performed in one process. It has one dedicated thread for each remote host to receive user input events from the host. The conventional X protocol messages are transmitted across a computer network. Figure 2 shows a conceptualized system configuration for a Tic-tac-toe game example.

4.2. The View Dialogue

The View Dialogue component provides a reusable toolkit to display application data in different views for each user. Figure 3 describes the internal architecture of the component. This component consists of the view management and its underlying graphical user-interface system. Functions provided by the view management include data format

conversions between shared data and graphical views, as well as updating views when data changes.

Each instantiation of a view creates an instance of the built-in View Dialogue object. The object maintains a list of graphical objects that are used to show the view. A benefit of this design is such that each view representation is completely independent from other viewers. Any redrawing operation of the graphical objects for a view representation is contained within the view, and does not cause the objects for other views to redraw.

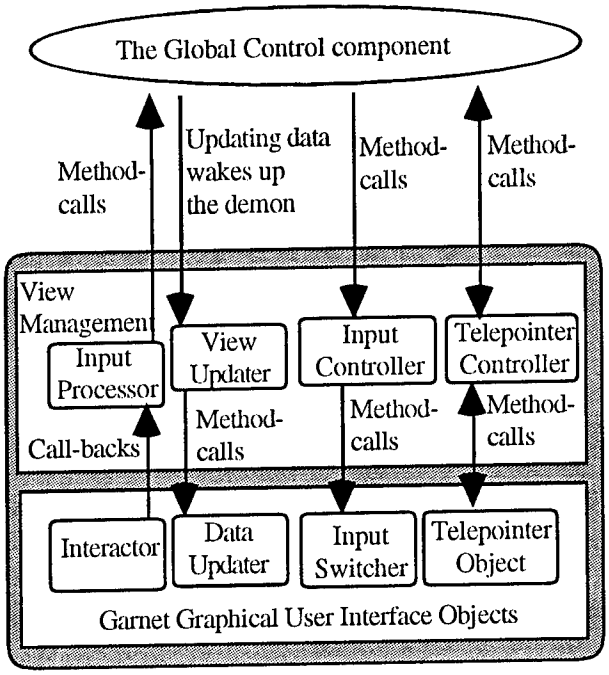


Figure 3. A processing model of the View Dialogue

In addition to the list of objects, the view management contains the following four subcomponents: Input Processor, View Updater, Input Controller, and Telepointer Controller. The Input Processor accepts user input events from the Interactors [12, 13] of a corresponding Garnet graphical object, and sends a request to the Global Control component to notify the Coordination component to update the shared abstract data.

After the Coordination component updates the shared abstract data, the notification propagates to all view instances. The View Updater accepts update-view notifications from the Global Control component, and changes values of graphical objects according to the value of the shared data by calling the Data Updater of the corresponding graphical objects. The Input Controller sets user input permissions of the view by notifying the Input Switchers in all the corresponding objects in the View Dialogue component. The Telepointer Controller creates and deletes the telepointing graphical object.

As shown in Figure 4, a view may be an aggregate of multiple graphical objects. The View Dialogue component provides a variety of graphical "widgets" for synchronous, multiple-user sessions. For example, there is a generalized aggregate graphical tool to create cells that a viewer can use for games such as Tic-Tac-Toe and chess. Presently, the widget

library contains following widgets: scroll-bars, radio-buttons, gauge meters, text boxes, cells, and a standardized graphical editor that provides cut, copy, and paste operations.

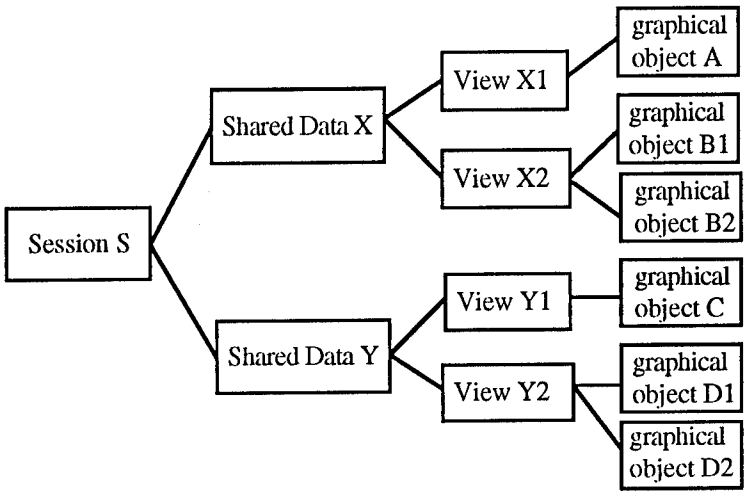


Figure 4. Relationship of abstraction, view, and graphic objects.
A session can contain multiple shared data, each of which feature multiple views. Views for multiple shared data may be viewed in a single window.

In order to implement efficient run-time communication among these sub-components, the Data Updater and the Input Switcher of the widgets respectively provide uniform external interfaces. In the Garnet system, updating value representations of graphical widgets means setting values of different widget component names and types, depending upon types of widgets. For example, a scroll-bar widget takes an integer while a radio-button widget takes an enumerated data type. The Data Updater assumes the task of converting a data type used for the abstraction data to the data type used in specific widgets. The Data Updater effectively isolates widget-specific information from the view management. If the abstraction data is an integer type, the Data Updater for a text box widget converts the integer into a character string, and sets the string to an appropriate widget component. The Data Updater has a widget-specific mechanism.

The Input Switcher provides a level of convenience similar to the Data Updater but from the input control perspective. The Input Switcher controls whether to allow user inputs on a particular widget. Since a name of widget component that sets user input control varies across different widget types, the Input Switcher provides an interface at a higher abstract level to set input state of widgets with two parameters: name of a widget and an input state of the widget. The Input Switcher has a simple mechanism that sets a boolean value to a widget-specific component.

In the present TALISMAN implementation, the Input Switcher takes only one boolean input parameter by which the view management can specify the current user input status for a specific widget. Consequently, application designers can easily change the selection of widgets without affecting the rest of the system. Moreover, TALISMAN can accommodate any existing Garnet graphical widgets and objects simply by adding the Data Updater and the Input Switcher interfaces to them.

4.3. The Coordination Component

The Coordination component contains modules for collaborative session management, shared abstract data management, floor control, and telepointer control. The component provides high-level, reusable functions in all four areas.

As shown in Figure 5, the component contains four subcomponents: the Session Manager, the Shared Data Manager, the Floor Controller, and the Telepointing Manager. The Session Manager controls the beginning and ending of a collaborative session. It also provides services so users can join and leave an on-going session. The Data Manager accepts requests for changes to shared data, and updates the shared abstract data. It then notifies the Global Control component of the new state. For example, in a Tic-tac-toe game, when the user clicks on a square, the View Dialogue component detects the user input event, and determines in which cell the user wants to place the game piece. The View Dialogue then tells the Global Control component which cell needs to update its state. After calling a Tic-tac-toe rule function to validate the move, the Global Control component requests the Data Manager to update the shared abstract data. After updating the shared abstract data, the Data Manager notifies the Global Control component about the change of the data. The Global Control component invokes a Tic-tac-toe rule function to determine whether one player has won or the game is tie. The Global Control component relays its result to the Data manager. When someone has won or it is a tie, the Session Manager is notified by the Data Manager that the session should end. The Session Manager requests the Floor Controller to terminate all the user input permissions.

The shared data is managed in an abstract form. In TALISMAN, the form of the abstract data is not necessarily identical to the structure used to represent views. In the case of the Tic-tac-toe game, the shared abstract data is simply a list of nine integers, each integer representing a status of one of the nine cells. On the other hand, for a shared drawing tool where all viewers share replicated graphical drawings, a shared abstract data can be a list of names of equivalent graphical objects among viewers. Reusable functions are available to manage different types of shared abstract data.

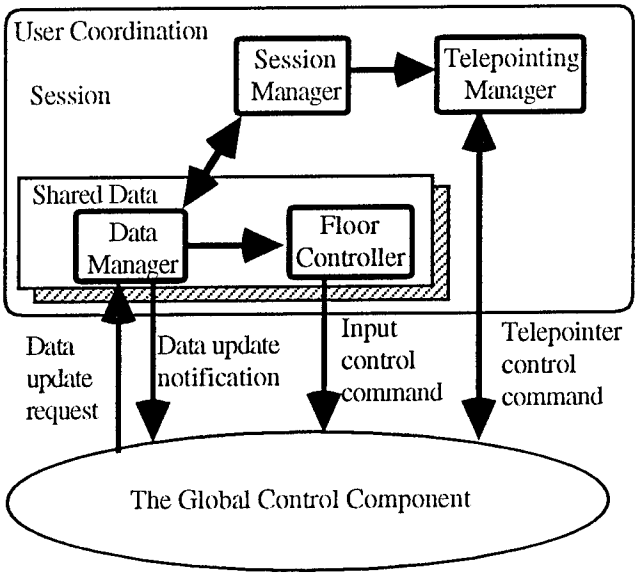


Figure 5. The architecture of the Coordination component

The system instantiates one Floor Controller for each instance of shared data. The component controls the floor by notifying the Input Controller subcomponent of each View instance to set whether its viewer is allowed to input data. The Input Control subcomponent in turn notifies the Input Switcher subcomponent of the View Dialogue component to change input permission status of specific graphic objects. Examples of the provided options for the initial settings for floor control include: allowing only the first registered user to input, allowing all initial users to input, allowing a predetermined number of users to input, and allowing no one to input. Examples of reusable functions for subsequent floor control include "taking turns" (alternating input permission among users), granting exclusive floor rights to one user all the time, and enforcing user priority levels in determining the turns. The idea of providing the interchangeable floor control mechanisms has the same spirit as the floor control protocols in the SHARE system [5, 16].

The Telepointing Manager is responsible for displaying telepointers in windows so all users can see where each other's cursors are. The telepointing manager interfaces with the View Dialogue component to display the actual telepointer objects. When a user is registered in a collaborative session, the system creates a telepointer object for the user. The object consists of a graphic object to represent the user's telepointer and an Interactor to capture movements of the user's cursor in the window. In order to display the user's telepointer image on corresponding windows of other users, the system creates an instance of the telepointer object, and adds the instances as a component to each of the other users' window. Due to the automatic inheritance of values from prototype to instance, the remote telepointer moves whenever the viewer moves his telepointer. The prototype-instance model of the Garnet system makes such a remote telepointing mechanism very easy to implement.

A unique feature of this design is that the entire mechanism for the remote telepointing is embedded in the system, and is hidden from the application. The system only requires the application designer to declare whether and how a viewer's telepointer should be shown to other viewers. There are four options for each user concerning remote telepointers:

- whether or not to display telepointers of other users on a user's window;
- whether or not to allow other users to display a user's telepointer.

From the Garnet perspective, the telepointers are implemented by a two-point interactor that is always running within application's windows to get the mouse cursor positions. The present implementation takes advantage of the multiple priority-level feature of Garnet. Interactors for all the telepointing graphical objects wait and run at a special priority level, which is higher than all other graphical objects. By setting the interactor for the telepointing graphical objects to also pass input events to lower priority-level interactors, the interactors for the telepointers always receive input events, and then the other interactors still receive input events when needed. As a result, the remote telepointers move even when another interactor is running. This feature is necessary to implement gesture communication while one user is drawing graphical objects in shared drawing applications.

4.4. The Application-specific Computation Component

The Application-specific Computation component provides specialized code required to execute the application system. Examples in a game application such as a chess include validating piece movements and determining the end of a game based on the status of cells.

Presently the component provides call-back functions to the Global Control component for following operations:

- pre- and post-processing of a collaborative session;

- pre- and post-processing of abstract data-updates;
- pre- and post-processing of changing a session floor (i.e. changing turns).

In addition to the above call-back functions, all the reusable components are customizable to specific application needs. For example, if an application requires some unique computations to determine when to allow data inputs, an application developer can supply a function to the User Coordination component.

4.5. The Global Controller Component

The Global Controller represents a core underlying mechanism for inter-component communication. A typical control sequence is as follows:

- wait for and receive a user input from the View Dialogue;
- invoke Application-specific Computational functions to analyze the input;
- invoke Coordination functions to update shared data and user states;
- provide feedback for the updated state to the View Dialogue component.

Since the sequence is similar across different application domains, the Global Controller is a reusable component.

4.6. Data Transmission

In any synchronous groupware, the communication overhead between a view and the shared data management is a deep concern. The system must maximize the efficiency of the data transmission. One of the main causes of the overhead is transforming data types during the transmission. To reduce marshaling and de-marshaling required for transmitting the shared data, all communications among the four components use the data format of the shared data in the TALISMAN architecture. This strategy eliminates the data type conversion between a view data and the shared data type at the Coordination component side. For example, when a view is sending the data update request, its View Dialogue component transforms the data into the data type used for the shared abstraction data before sending it to the Global Control component. Similarly, when all the views are notified of data updates, each View Dialogue component converts the abstract data type into a view-specific data type to update its view. The Data Updater in the View Dialogue further converts data types if necessary to set new values in specific widgets. In both cases, the View Dialogue component is responsible for the data type conversions.

The uniform data format during the data transmission also simplifies the Coordination component because the component never has to be aware of data types used in views. Since there are not independent intermediary data converters, how a view treats the shared abstraction data is completely encapsulated within the view. All the views however must know the data format used to store the shared abstraction data.

CHAPTER 5.

IMPLEMENTATION ISSUES

5.1. Centralized Data Management

In the present TALISMAN implementation, a user input does not directly update the view. Only the view Update Notice messages cause the view to update its data representation. Figure 6 depicts a typical situation of centralized data communications. When View 1 detects a user input, it transforms the input data into the Update Request of the shared abstraction data. View 1 then requests the Coordination component to update the shared abstraction data. After the Coordination component updates shared abstraction data, the Update Notify message is received by all the views, including View 1. This causes the views to update their view representations.

An inherent problem with this scheme is the possible time lag between the time that a user inputs, and the views update themselves. The TALISMAN implementation attempts to minimize the delay by relying upon the invalidate-demon feature of the object system in Garnet to propagate the Update Notice messages. The invalidate-demon is normally used in Garnet for the purpose that requires efficient processing, namely to trigger automatic redraw of graphical objects [14]. The TALISMAN implementation takes advantage of this Garnet feature since it is optimized for efficiency.

5.2. Object Inheritance Properties and the Invalidate Demon

Synchronizing updates between the shared abstract data and the views require that there is a two-way link between an instantiation of the shared data and a corresponding view. In one direction, we use call-back functions. In the other direction, we use an "invalidate demon" mechanism combined with the prototype-instance inheritance property.

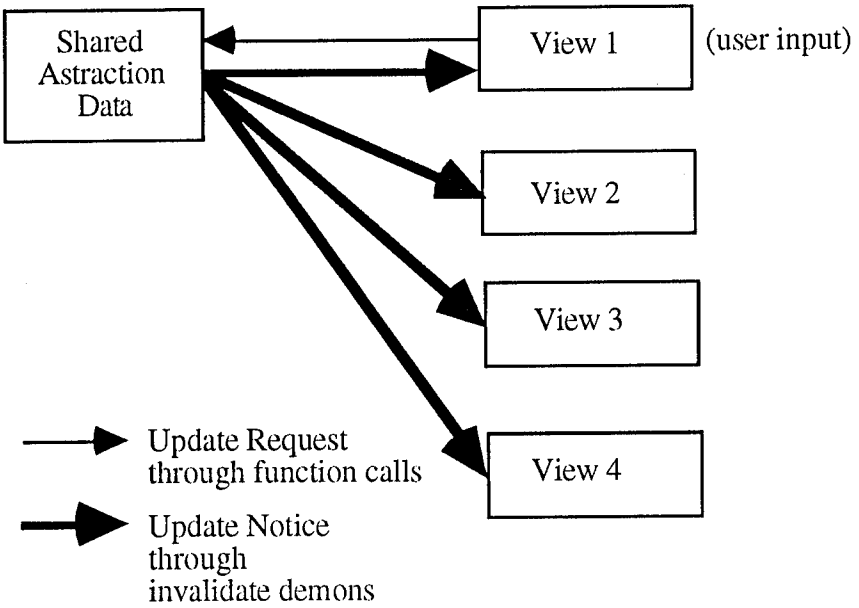


Figure 6. Update data flow diagram
View sends a data update request; all the views are then notified.

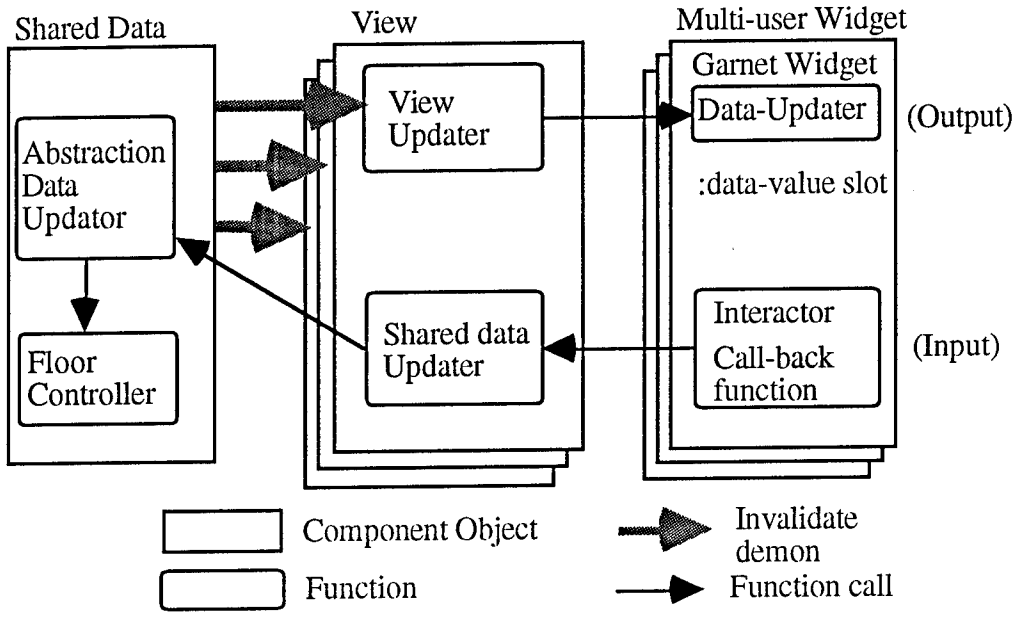


Figure 7. Message flow diagram

To reflect a change of a user's view to the corresponding shared abstract data, the system uses call-back functions. The Interactor invokes an appropriate call-back function whenever a user clicks on the game board to place a game piece. The call-back function converts the view-specific format of data to the format of the shared data, and calls a notifier function to update the shared data. These functions are provided by TALISMAN.

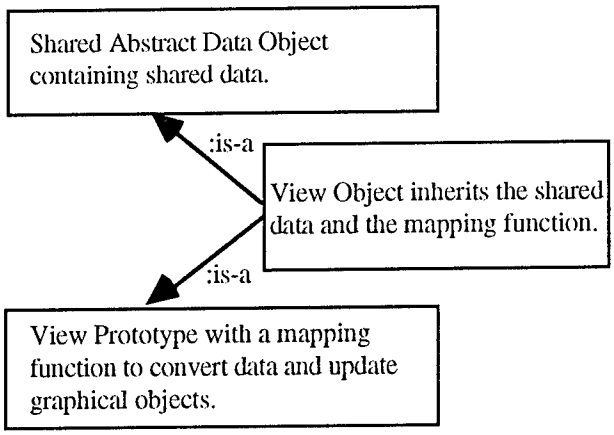


Figure 8. Multiple-inheritance of the View object

To update views of all users based upon the shared abstract data, the system uses the "invalidate demon," which is inherited to each view object from the view prototype. All view objects are instances of both the shared data prototype and the view prototype. Since each view can have a different representation of the data, the data conversion between the shared abstract data and the view can be different in each view. By making each instantiation of view object multiply inherit both the shared data object and the view object, the shared data is inherited to each view instantiation automatically. Any updating activity

on a shared data invokes the demon functions in all corresponding view objects. Subsequently, each demon function converts the data format from the shared abstract data, and updates a view data. Again, the demon functions are provided by TALISMAN. (See Figure 8.)

The benefit of multiple-inheritance is not only to use the "invalidate mechanism" for data-update propagation, but also to confine the process of updating a view within the View Dialogue component. Because the view object inherits from the shared data value, the component does not need to access the Coordination component to retrieve the updated value of the shared data. The view object refers to the data value locally, and calls appropriate Garnet graphical objects to update their values.

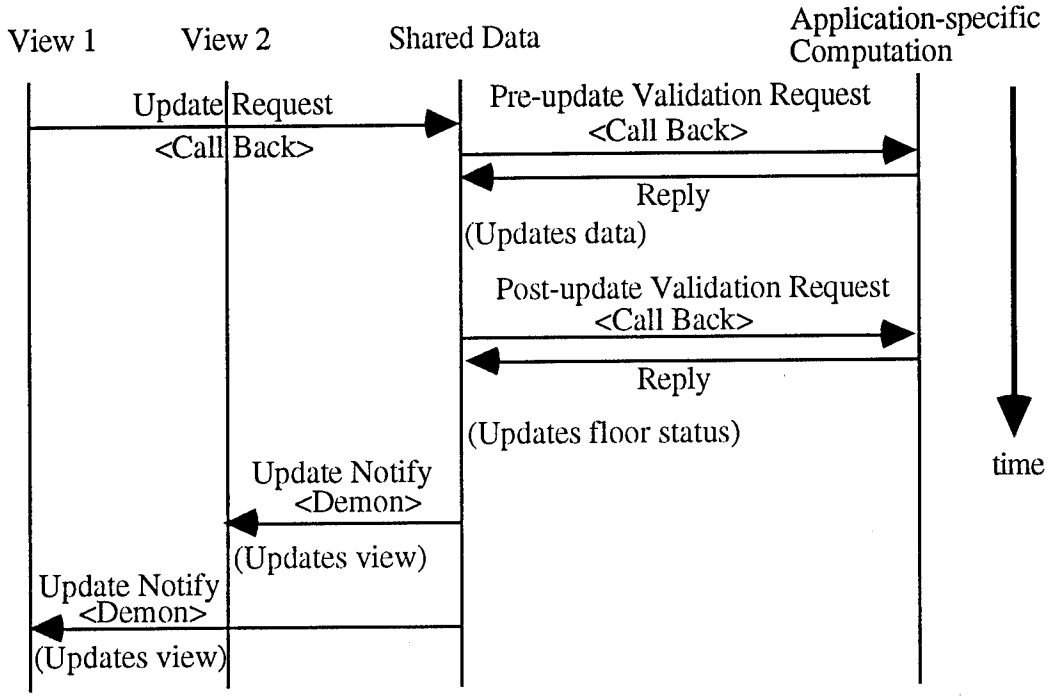


Figure 9. Example of a typical message timing chart while updating data. The Update Notify is sent implicitly by the invalidate demons. The format of data in the transmitted messages is equivalent to the format of the shared abstraction data. The order of sending the Update Notify messages is arbitrary.

This implementation uses the call-back mechanism to detect and process changes in the views' representations. If a constraint system allows values of objects to have constraints from states of more than one object, unlike Garnet's, then the object constraint mechanism could be used instead. In the present object system for Garnet, only one formula can be set in one slot. Moreover, a user of the object system in some cases cannot even set one formula to a slot because one "hidden" formula is already set for Garnet's internal use. Such cases include a data value slot of a Garnet graphical widget. If you could establish a constraint so that the abstract data depends on values of all the view objects, then you would not need to use demons. The system would then update the shared abstract data whenever one of the views change the value.

5.4. Designing the Format of the Shared Abstraction Data

As Hill states[9], the format of the shared abstraction data is critical to ensure that synchronous groupware will work. And, designing the *right* format is very difficult. The shared abstraction data must contain all the necessary information for all the views to represent the data. While the Rendezvous system [9] eased the format design effort by suggesting that the shared data format reflects the data structure seen in the real world, the TALISMAN architecture takes a different approach. The TALISMAN system encourages that the data be as abstract as possible, and not retain physical semantics. For example, in the Tic-Tac-Toe game application, a groupware designer does not have to retain the semantics of sharing a three-by-three rectangular cells with circle and cross pieces. Instead, the designer can see the cells as a list of nine integers each of which expressing a state of the corresponding cell. Construction of game board semantics is the responsibility of each view of the game.

This approach of designing the shared abstract data could result in developing very interesting applications. For example, one could design an interactive entertainment application where multiple players can play with one another while each player thinks they are playing completely different games. For example, each of three players could be playing a slot machine, a roulette, and a card game respectively, and all of them are actually playing against one another. This strange yet fun game is possible because all of the players have the same objective: to get a certain combination of numbers. All the semantics of the shared numbers are value-added at the view level.

CHAPTER 6.

DEMONSTRATIONS OF SYNCHRONOUS GROUPWARE SUPPORT

6.1. Multiple-Views on Shared Data

6.1.1. Description

As the most primitive demonstration of the system, a simple program was written that displays a single shared abstract number in four different view representations. The demo displays four windows at different hosts across network. Each of the windows displays a different type of graphical widget to represent the value of the shared data. The four types of the widgets are scroll bar, gauge meter, radio-buttons, and text box. The system can display these windows on separate screens. Users can change the value of the shared data by manipulating the graphical widgets in any of the windows. However, the system takes a user input from one window at a time, and changes the input window every time that a user changes the value. Effectively, the system enforces that the users to take turns. For this simple example, there is no application-specific computation. A change made by an authorized user is immediately reflected to all other users. A sample window image is shown in Figure 10. Four graphical widgets are a scrollbar, a gauge, a radio-box and a text box.

6.1.2. Design

To make the design of the system as simple as possible, the shared abstraction data for this demonstration has an integer number type. Figure 11 depicts relationships between the shared abstraction data and the views. Four views refer to an integer stored in the shared abstraction data object. For the scroll-bar and the gauge widgets, the shared data value is directly set to the respective widgets to update view representations. If a widget requires a data type different from the shared abstraction data, views provide a type converter to convert the data type. In the case of the text box widget, the type converter converts an integer type to a string type before setting the value to the widget. Similarly, the type converter that is associated with the radio-button widget converts the integer type of the shared data value to an enumerated data type before setting the data to the widget.

A benefit of this design is such that the programmers can use the existing Garnet graphical widgets and objects to display the view-dependent values without any internal modifications. The only change required for these widgets is to add type converter routines to transform the data type (an integer) used in the shared abstraction data into the data type used by the widgets. Programming languages such as LISP and C provide convenient functions to convert a type from one to another. The data types for the widgets in the views are determined by the corresponding original Garnet widgets.

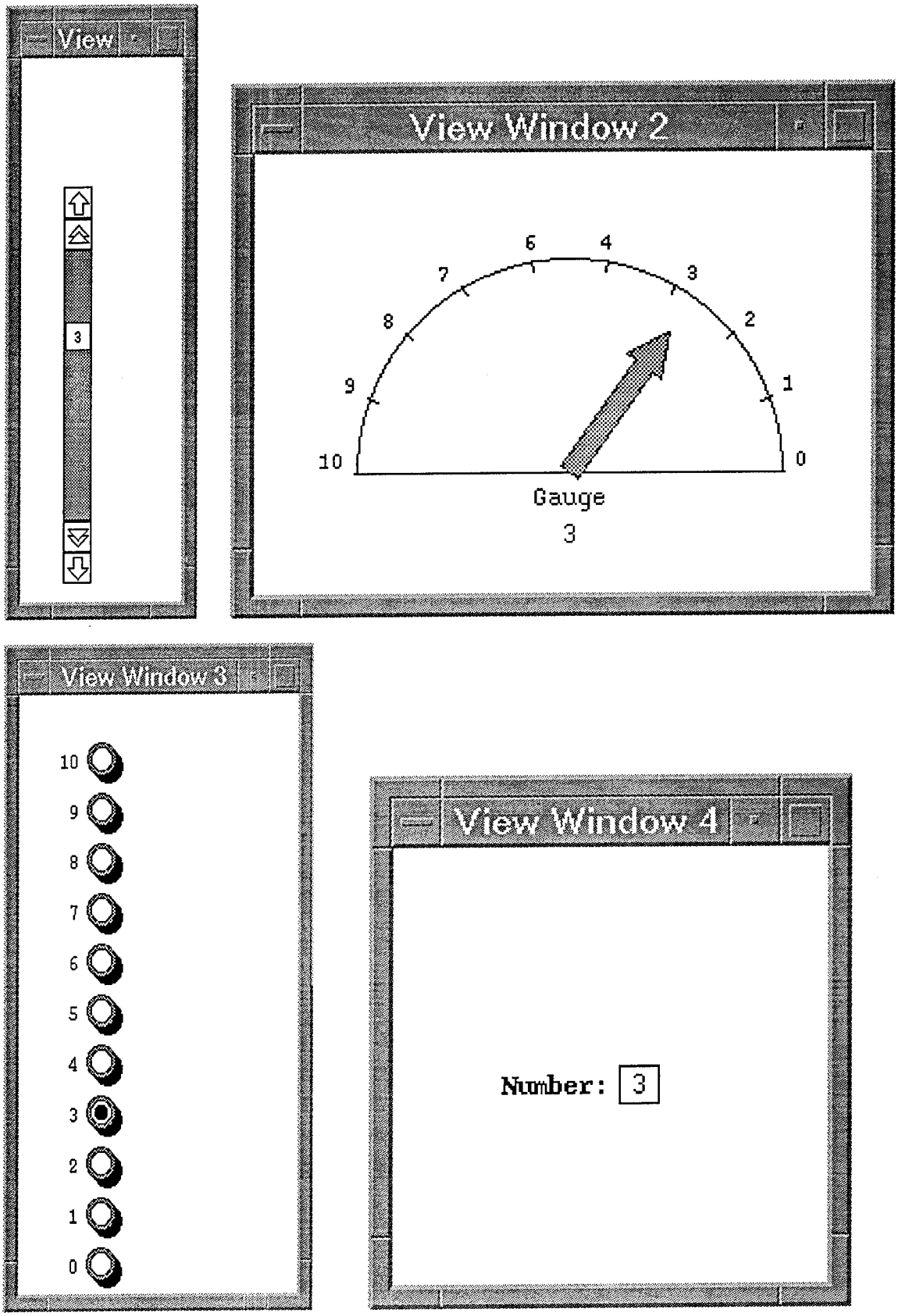


Figure 10. Window Images of the Multiple View on Shared Data Application

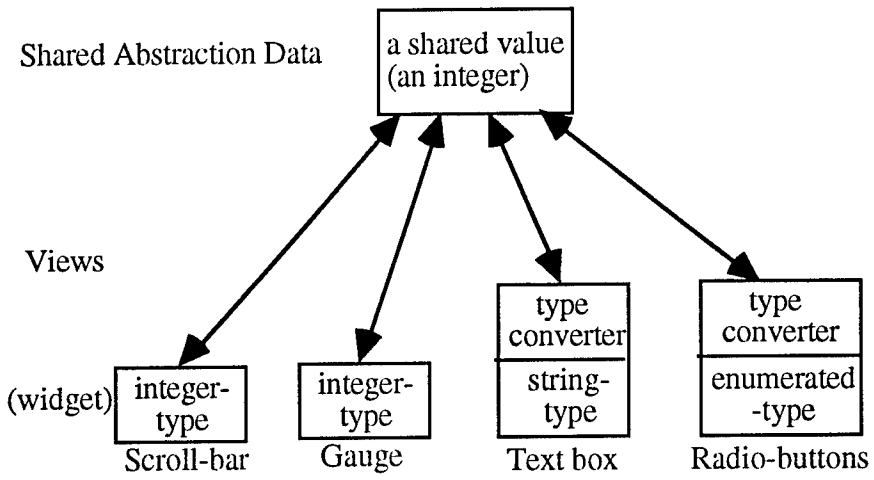


Figure 11. Shared Abstraction Data and View representations for the multiple-view representation.

6.2. The Tic-Tac-Toe Game

6.2.1. Description

The Tic-Tac-Toe Game application allows two players at remote locations to play a tic-tac-toe game, while observers may also watch the game without interfering with the players. The game consists of one window for each player or observer. The contents of the window is customizable to a particular view. For example, one viewer may display the tic-tac-toe game pieces as circular and cross shapes, while another viewer may observe the pieces as the colors of the cells. Only the players are allowed to place pieces on the game board. Observers cannot place any pieces on the board. The system enforces that the two players take turns.

In addition to the multiple-view capability of the TALISMAN, the Tic-Tac-Toe game demonstrates the following three aspects of the synchronous groupware toolkits. First, it requires the floor control mechanism; two players take turns. Second, the game application demonstrates the clear separation of the application-specific computation from the toolkit by using game rule functions to detect the illegal piece placements as well as the end of the game. Third, the game demonstrates user gestures through telepointing. The telepointers of all users are displayed on all the other windows.

6.2.2. Design

The Tic-Tac-Toe application displays one window for each player, and a third window for a viewer who just observes the game. The shared data is a list of nine integers, each representing a status of one of nine cells on the board. (See Figure 12.) All other information about the representation of the game, including player identifications and uit and reset buttons, are viewer dependent. Figure 13 shows a sample window for a player.

A Structure of Shared Abstruction Data

A1	A2	A3	B1	B2	B3	C1	C2	C3

A1	B1	C1
A2	B2	C2
A3	B3	C3

Board View

Figure 12. Shared Abstract data and View Data for a Tic-tac-toe application. A combination of a letter and a number indicates a code name for a corresponding cell and a list element.

The application uses reusable functions extensively to minimize the programming required for the application. First, the Tic-tac-toe application uses a built-in widget to graphically represent a 3-by-3 cell for a game board. Next, the application uses a reusable function to set a floor control policy, which in this case is to alternate the input between two users after a player makes a move. Finally, the application uses the telepointing mechanism to show the telepointers of all of the players.

There are two application-specific computations in the Tic-tac-toe application. One is to validate moves made by the players. This function is invoked every time one player places a piece by clicking on a square of the game board. A player cannot place a piece in a cell unless the cell is empty. The other application-specific computation is to determine the end of a game. A method called "check-session" looks at the shared data, and decides whether the game is over. The parameter for the function is the user who most recently modified the shared data, and returns True if the game should end. This function is called after an update to the abstract data. That is, the game status is monitored whenever a player places a piece on a cell. When the end of the game is announced by the function, the system no longer allows any players to place pieces in the cells.

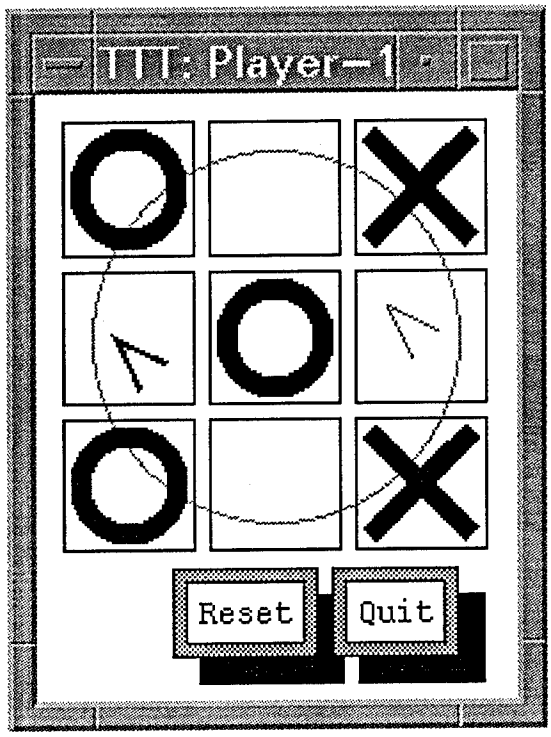


Figure 13. A Tic-Tac-Toe game. The telepointers for the opponent and one spectator are represented by the side-ways "V"s. The large thin circle reminds the player that this player places circle pieces. In actual screen, windows are colored to indicate whether it is the player's turn to place a piece.

Observers of the game can be easily added by instantiating another view with read-only permission to the game. All the participants, including observers, can remotely display telepointers on screens of the other participants.

6.3. Shared Drawing Tool: GarnetDraw+

6.3.1. Description

We have also modified a single-user graphics drawing tool, called GarnetDraw, to be a shared drawing application. The tool consists of a drawing window, pull-down menus, a palette of the shapes that can be drawn, and object attributes palettes for color and line style. (See Figure 14.) Only the drawing window is shared among users. The menus and palettes may differ among users. This sample application demonstrates the capability of TALISMAN to differentiate the shared data management from the view-dependent data management. Moreover, this application requires more complex shared abstraction data structure than the previous Tic-tac-toe application. The shared drawing tool requires the system to maintain replicated graphical objects across multiple displays. In addition to the attributes of graphical objects, the system must maintain such parameters as the drawing order and creation / deletion of the objects. Finally, the most significant difference of this sample application from the Tic-tac-toe game is that this application must support concurrent manipulations of graphical objects by users. Because this application must support concurrent user inputs, maintaining the data state consistency across views becomes very difficult.

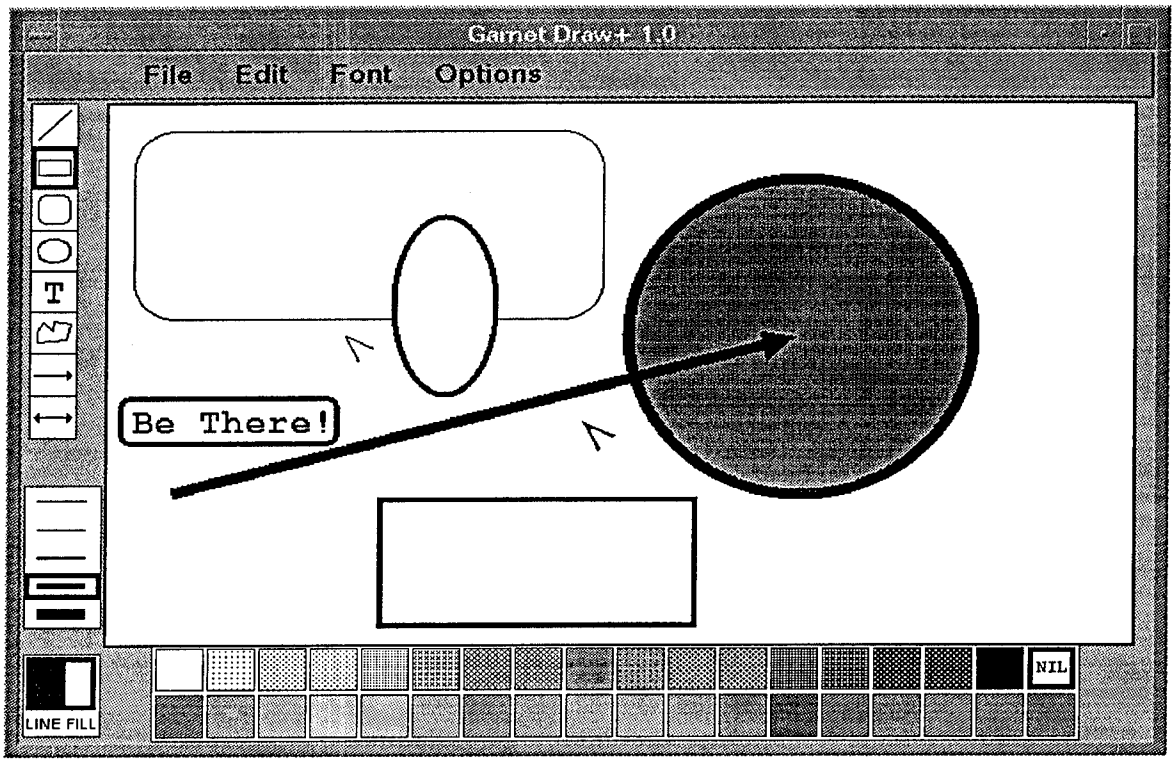


Figure 14. GarnetDraw Application for Multiple Users

6.3.2. Design

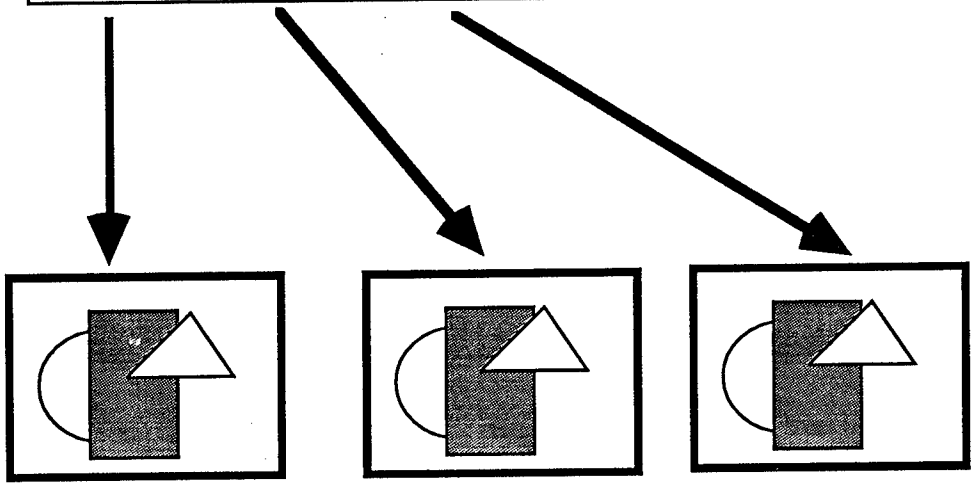
In our sample implementation, the shared abstract data takes the form of a list of the corresponding graphical objects used in different viewers. Thus, for each graphical object in one view, there is a corresponding object in the other view, and the abstract data contains a list of these associations. As users draw more graphical objects, the list grows. Each combination also contains an identifier for the user who created or last modified the graphical object. When the View Dialogue component updates the respective views, each view copies attributes of the most-recently-changed graphical object to its own object.

Because this is a modification of a single-user application, it is important to minimize the changes necessary to the original flow of the program. Nevertheless, there are several major modifications. First, an array of the viewer objects was added to support an arbitrary number of users. These objects contain all the view-dependent information, including viewer window identifiers and view-specific interactors. Second, the user input notification functions were added to notify for creating, modifying, and deleting graphical objects to the Coordination component. Third, a function was added to update a view based on the changes made to the shared abstraction data. This function is triggered by the invalidate demon. The demon code resides in the TALISMAN toolkit, and not in the application. Moreover, the multiple-user widgets are created for the standard-edit widgets to notify the shared data abstraction whenever a user attempts to edit a graphical object in the view.

Since the data structure does not denote the order of drawing graphics, this design does not consider supporting late comers to the shared drawing session. Whenever a viewer modifies a graphical object, the entry that contains the object is moved to the beginning of the list. When a viewer creates an object, a new entry is created at the top of the list. In the above example of Figure 15, the Viewer C just moved the circular object to the bottom of the drawing, and the result is replicated to all other viewers. When the invalidate demons invokes view updater functions, these functions refer to the first combination of the object names in the updated shared list. After identifying the last modified object and the object corresponding to its own view in the combination, the function copies all the graphical attributes of the last modified object to the view's object. When the Command column contains either ToTop or ToBottom command type, then each view moves its object to the appropriate order for drawing.

Shared Data List

Object #	Object #	Object #	Command	Last Modified
Circle-1	Circle-2	Circle-3	ToBottom	Viewer C
Trian-1	Trian-2	Trian-3	NIL	Viewer A
Rect-1	Rect-2	Rect-3	NIL	Viewer B



Viewer A Viewer B Viewer C

Figure 15. Shared Data and Replicated Views

This shared drawing tool allows participating users to draw graphical objects concurrently. There is no floor control to enforce a sequence of inputs by multiple users. There are two cases of concurrent operations. First is the case where more than one user operate on different graphical object. This case includes different users creating new graphical objects at the same time. The system must reflect all the changes made by users, including creating new objects. The TALISMAN system can support this because the system guarantees to invoke the abstraction data update demon exactly once for each view per each abstraction data update operation. A second case is multiple users modifying the same objects at the same. Although a use of the application in conjunction with telephones and other communication methods may avoid such a situation, such a situation is certainly possible to occur. Because of the exactly-once guarantee of the demon invocation, all the views

always have the latest changes made to an object. For example, if two users try to resize a same rectangle, whoever finishes the resize operation later determines the final size of the rectangle.

6.4. Evaluations

6.4.1. Scope of Evaluations in This Study

The most reliable method of evaluating a toolkit is to build as many applications as possible, and determine how the toolkit actually has reduced the development time and modification time of the application software. Unfortunately, the present study does not have any full-scale evaluation of the model because of the time constraint of this thesis. As the minimal evaluation of the toolkit, three sample applications have been developed to gain some idea of how much this toolkit actually contributes to the groupware development.

6.4.2. Reduction of Code using TALISMAN

By providing all the graphical widgets as a widget library, toolkits can reduce the amount of coding for the application program. We compare two programs to do the single task of Figure 10; one does not use codes provided by TALISMAN and the other does. Result shows that the number of functions written in the application program decreased. For example using the Multiple-Views sample program, the number decreased from 17 to 7. The number of customized schema explicitly created in the application program decreased from three to none. The application programmer has to create the following instances using TALISMAN: an instance of a window schema, an instance of a schema to manage the shared abstraction data, one instance of a schema to store the shared abstraction data, four instances of the view schema, and one instance of each of the four widgets. TALISMAN, as well as Garnet, provides all the prototypes.

In the Tic-Tac-Toe game application, the number of lines of code for the game rule functions accounts for 15% of the entire Tic-tac-toe application code. The rest of the application code for the game is mainly to create instances of windows, player labels and game pieces by using Garnet graphical objects.

6.4.3. Converting A Single-User Application

Another way of measuring effectiveness of a groupware toolkit is to test the ease of converting a single-user application to a groupware application. We have found from this experience that the difficulties of converting a single-user application to a groupware one are no more than the difficulties of converting a data structure-oriented program to an objected-oriented paradigm, because architectural components for the TALISMAN system are easily identified once an application program is designed from an object-oriented perspective. The hardest part of creating this sample application was indeed isolating the code used for view-dependent graphical widget management from the code used for the shared work space. As in typical program source codes, there were too much hacking using global variables. The first task of modifying the code was to eliminate most of the global variables to encapsulate one component from another. While we did not make any changes in the code with respect to the way each user interacts with the system, we added code to maintain consistency across replicated views. The types of the actions by each user are creating and deleting of an object, changing in the location, changing the color and the line width of the objects, and moving an object to top and bottom.

We have compared a single-user, one window version of sample applications to the same application based on TALISMAN for multiple users. For example using the Tic-Tac-Toe game, the TALISMAN version added an extra 30% of code to make the application multiple-user purposes. The amount of code for the GarnetDraw application increased by 16%. The increase in the code comes from adding application-specific mechanisms to copy and create drawing objects based on states of the abstraction data.

6.4.4. Limit of the TALISMAN Model

The group drawing application has also shown that the TALISMAN cannot provide reasonable responses when the size of the abstraction data becomes very large. Moreover, significant performance degradation occurs when more than three users operate simultaneously. This performance problem results not only from the high overhead of processing multiple views in TALISMAN but also from the communication overhead at the X Protocol level.

Moreover, we are yet to investigate how much complexity in the structure of the abstraction data the TALISMAN model can tolerate. What is the logical limit of the model? The present implementation of the model implicitly assumes that each abstraction object is independent of one another. The implementation does not allow a state of one abstraction object to affect that of another abstraction object. If a state of an abstraction object is dependent on one another, a mechanism to manage abstraction object update notifications becomes more complex. An update of one abstraction object needs to be forwarded not only to its corresponding View Dialogue components but also to other abstract objects that depend on the state of the abstraction object.

6.4.5. Extensibility and Reusability of the Application Software

While prototyping the system, we strived to make transforming one application to another conceptually similar application relatively easy. For example, transforming the Tic-tac-toe game to a chess game basically requires the following four modifications: First, the function to check for the end of the game must be modified according to the rules of chess. Second, new Garnet graphical objects need to be created to replace the circle and cross labels. Third, the cell size must be changed from three by three to eight by eight. The shared abstract data type can be simply changed to be a list of 64 integers. Fourth, application-dependent rules such as chess-piece movements and stale mate detection must be defined.

There are two key issues here. First, all the underlying components, including the floor control, the management of the shared data, and the remote telepointing mechanism, are reusable for the new application without any modifications. Also, the TALISMAN architecture clarifies what functions need to be customized to a particular application.

CHAPTER 7.

FUTURE WORK

7.1. Introduction

While the present work has successfully demonstrated the feasibility of the high-level architecture for a synchronous groupware application, more research is necessary to investigate the robustness and portability of the TALISMAN architecture. This chapter describes the possible future work for this thesis from two perspectives: engineering enhancement and fundamental research. The engineering enhancement issues concern immediate engineering work. Their common purpose is to extend this thesis work by further testing groupware applications based on TALISMAN, as well as to refine the TALISMAN architecture where necessary in the short-term. The fundamental research issues, on the other hand, address long term studies necessary to understand how toolkits can help create synchronous groupware applications

7.2. Engineering Enhancement

There are several engineering enhancement issues. All of the issues have a common purpose, which is to test effectiveness of the TALISMAN architecture with more groupware applications.

Our most immediate plan is to implement TALISMAN using a new user interface system that is a successor to the Garnet system. The most apparent difference of the new underlying system from Garnet is that the new system is written the C++ language, whereas Garnet has been written in LISP. Accordingly, we will implement our new TALISMAN in C++. We will benefit from the C++ implementation because the C++ language features better access to the operating system and to the network than LISP. Implementing in C++ paves the road for long-term investigations. For example, the new system will allow us to have design options for both central and distributed processing models for a future research. The present TALISMAN is based on the central architecture simply because the LISP language does not provide flexible mechanisms for inter-process communications across the network.

The Garnet successor will also provides new features that TALISMAN may take advantage of. For example, the new object system allows us to set more than one formula in a slot to define constraints among objects. The Garnet system only allows us to have one formula in a slot. By using this new feature, we may be able to establish a constraint so that the abstract data depends on values of all the view objects. This can effectively eliminate uses of demons. The constraint system will update the shared abstract data whenever one of the views changes the value. Figure 16 sketches the idea of using a constraint mechanism to maintain states of a view and its corresponding graphical representations.

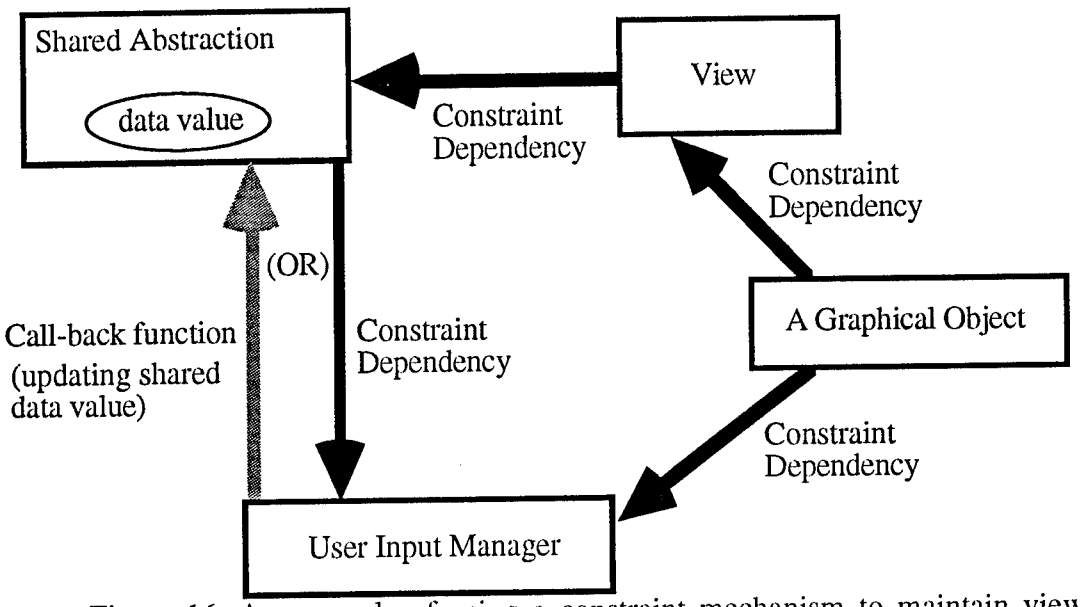


Figure 16. An example of using a constraint mechanism to maintain view and graphical objects.

Using the new, more capable successor system, we need to evaluate TALISMAN with more groupware applications. We need to build more sample applications to more clearly determine limitations of the TALISMAN architecture. The task of building applications is twofold. First is to develop more toolkit libraries for TALISMAN. The libraries include widgets for different floor control policies, as well as abstraction prototypes for a bigger variety of shared data types. Presently, there are only three widgets for floor control policies and two prototypes for abstraction objects. Second, we need to evaluate the system by using more complex structures for shared abstraction data. One of examples of more complex shared objects is playing cards, each of which has two sides. In card games, different card players have different views of card stacks on the table. There are cards that must be seen only by their owner, and there are other cards that are seen by all the players. There are also cases where one card changes its state between the two. We need to investigate how moving the objects between private and shared status will make the management of shared abstraction data more complex. We are yet to investigate whether the TALISMAN architecture can sustain more complex data management.

7.3. Fundamental Research

There are at least three issues that require a long-term, fundamental research. First, we need to investigate whether the TALISMAN architecture is robust enough to work in the distributed processing environment. Under such an environment, components of TALISMAN are located in different computers across network. The distributed implementation may contribute to a better performance through load balancing among computers as well as through an optimization in message protocols. The present TALISMAN implementation has no control over optimizing networking protocols because the standard X-protocol messages travel through the network. Meanwhile, one of the key issues in implementing TALISMAN in the distributed processing environment is to synchronize concurrent operations of users working together. Some form of an event synchronization mechanism is required to maintain states of the collaborating users consistent. It is yet to determine whether the abstraction level of the TALISMAN

architecture presented in this thesis is appropriate if we distribute the architecture components across the network.

Another long term agenda is to design and implement a high-level rapid prototyping environment of synchronous groupware applications based on the TALISMAN architecture. The prototyping environment should include interactive human-interaction designing tools and simulation tools to rapidly test the design. These tools extend the notion of the current user-interface builder tools to design multi-user interactions. In addition to laying out graphical user interface objects, such groupware building tools would allow designers to specify an abstract data representation, view representations, and how multiple users should interact with one another (e.g. floor controls).

Since almost all of today's software applications categorize themselves as single-user applications, it will be very useful to provide groupware design tools that aid software engineers to convert these single-user applications to multiple-user applications efficiently. This thesis has found that the conversion process of a single-user application to a multi-user application in TALISMAN is similar to that of redesigning a program based on the object-oriented approach. Once the application is constructed in an object-oriented manner, it becomes relatively easy to add multiple-user mechanisms to it. The conversion tools should facilitate application designers in separating modules for view-dependent processing from modules for managing shared data. Moreover, such tools should be able to help design structures for shared abstraction data objects.

CHAPTER 8.

CONCLUSION

8.1. Introduction

This thesis has demonstrated that the concept of coordination independence in addition to dialogue independence is very useful to efficiently design synchronous groupware applications. This chapter describes contributions made by this thesis from two perspectives, namely technology contributions and system-level contributions. The former perspective concerns contributions made by specific mechanisms within the TALISMAN architecture on such issues as floor controls and multi-user coordinations. System-level contributions, on the other hand, describe contributions made by this thesis at a more abstract level.

8.2. Technology Contributions

Technologically, this thesis has made the following three contributions. First, this thesis has demonstrated that the concept of coordination independence along with dialogue independence is useful to design a toolkit for synchronous groupware applications. The coordination independence includes separations of both floor control and remote telepointing mechanisms from application codes. There is no need to rewrite floor control policies if TALISMAN supports desired policies in reusable components.

Second, this thesis has demonstrated usefulness of demons and object inheritance for controlling multiple views. These object system mechanisms provide efficient propagation of update events of states. Moreover, we have found that object constraint mechanisms are extremely useful for managing remote telepointing objects.

Third, TALISMAN provides several different types of reusable components for synchronous groupware applications. TALISMAN separates an application into the following four components: View Dialogue, Coordination, Application-specific Computation, and Global Controls. All the components except the application-specific computations provide reusable functions and objects for different applications. These reusable functions and objects include floor control, telepointing, and shared abstract data manipulations. The reusable components are beneficial for following purposes:

- floor (user input) controls;
- floor initializations;
- remote telepointing management;
- data type conversions by views;
- data communications between the shared abstraction data object and views.

8.3. System-level Contributions

This thesis has made a system-level contribution by presenting an architecture with high-level, reusable components. This thesis also has demonstrated that these reusable components help create synchronous groupware applications efficiently. The central idea behind the design is to combine the dialogue independence and the coordination independence. Designing an architecture for synchronous groupware applications requires a careful consideration of both dialogue independence and user coordination independence.

Application developers can reduce the amount of programming by using such reusable functions.

This thesis also contributes at the system-level by providing some guidelines for converting a single-user application into a groupware application. This thesis has found that a system designer must identify and isolate global data processing from view-dependent processing in a single-user application. The design approach was found to be similar to object-oriented design process. All the processes that concern one viewer must be encapsulated within the viewer, while all the information that is shared among viewers must be represented as an abstraction object. The key issues of efficiently converting a single-user application to a multiple-user application include avoiding excessive global variables and encapsulating information within objects that use the information. Moreover, the concept of object inheritance is very useful to reduce the amount of coding for different views. By representing common features of views as a prototype object, an application developer only needs to add view-specific codes by inheriting the common features from the prototype object.

REFERENCES

1. Cordy, J. R., Hill, R.D. Singh, G., Vander Zanden, B. Report of the "Linguistic Support" Working Group. *Languages for Developing User Interfaces,* Jones and Bartlett Publishers, Boston, 1992, Chapter 21.

2. Dewan, P., Choundhary, R. Primitives for Programming Multi-User Interfaces. In *Proceedings of the UIST '91,* ACM, New York, 1991, pp. 69-78.

3. Gust, P. Shared X: X in a distributed group work environment. Unpublished paper presented at the Second Annual X Technical Conference, January, 1988.

4. Gibbs, S. J. LIZA: An Extensible Groupware Toolkit. In *CHI '89 Proceedings,* New York, 1989, pp. 29-36.

5. Greenberg, S. Personalizable Groupware: Accommodating Individual Roles and Group Differences. In *Proceedings of the 2nd European Conference on Computer Supported Cooperative Work (EC-CSCW '91),* Klewar Press, Amsterdam, 1991.

6. Greenberg, S., Roseman, M., Webster, D., Bohnet, R., Issues and Experiences Designing and Implementing Two Group Drawing Tools, *Proceedings of the 25th Annual Hawaii International Conference on the System Sciences,* 1992, 139-150.

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

8. Hill, R. D. Languages for the Construction of Multi-User Multi-Media Synchronous (MUMMS) Applications. *Languages for Developing User Interfaces,* Jones and Bartlett Publishers, Boston, 1992, Chapter 9.

9. Hill, R. D. The Abstraction-Link-View Paradigm: Using Constraints to Connect User Interfaces to Applications, In *Proceedings of CHI'92,* 1992, 335-342.

10. Kohlert, D., Rodham, K., Olsen, D., Implementing a Graphical Multi-user Interface Toolkit, *Software - Practice and Experience, 23,* 9 (1993), 981-999.

11. Lane, T. G. A Design Space and Design Rules for User Interface Software Architecture. *Carnegie Mellon University Technical Report CMU-CS-90-176.*

12. Myers, B. A. Encapsulating Interactive Behaviors. In *CHI '89 Proceedings,* New York, 1989, pp. 319-324.

13. Myers, B. A., et. al. The Garnet Reference Manuals. *Carnegie Mellon University Computer Science Technical Report CMU-CS-90-117-R4,* October 1993.

14. Myers, B., Giuse, D., Vander Zanden, B., Declarative Programming in a Prototype-Instance System: Object-Oriented Programming Without Writing Methods. *SIGPLAN Notices, 27,* 10 (1992), 184-200.

15. Patterson, John F. Comparing the Programming Demands of Single-User and Multi-User Applications. In *Proceedings of the UIST '91,* ACM, New York, 1991, pp. 87-94.

16. Roseman, M., Design of a Real-Time Groupware Toolkit, Masters Thesis, The University of Calgary, 1993.

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890