Building Groupware on THYME

Seth Landsman[1] and Richard Alterman[2]

[1] The MITRE Corporation, Bedford MA 01730, USA,
landsman@mitre.org
[2] Brandeis University, Waltham MA 02454, USA,
alterman@cs.brandeis.edu

Abstract. The study of collaboration within a community of users, as
it is mediated by a computer application, requires the construction of
groupware applications to mediate the collaboration. It also requires the
capability to analyze the collaboration as it is mediated by the computer
application. As the understanding of collaboration within a group un-
folds, ideally, the application can be quickly adopted to fit the changing
requirements of the group. Whereas existing toolkits for building group-
ware applications provide support for building production-level group-
ware, they do not provide the mechanism for analysis and rapid devel-
opment of new applications, which is necessary for the study of collabo-
ration.

This paper presents an engineering methodology and a set of key re-
quirements for building groupware applications that enable the analy-
sis of collaboration as mediated by these applications. We describe the
THYME framework and show how it can help groupware developers to
build applications rapidly and successfully. We also detail how a one
semester Human Computer Interaction class used the THYME frame-
work to construct synchronous groupware applications in a short amount
of time and with great success.

1 Introduction

Electronic collaboration with remote clients or colleagues is of growing impor-
tance in doing business. Key personnel are often traveling and company offices
are spread throughout the world. Groupware is situated to be a key aspect of
how organizations operate. However, in order for collaboration to become per-
vasive, the tasks that they enable must be understood, as must the needs of the
community of users who will incorporate the groupware applications into their
tasks.

As part of the practice of building groupware applications for study, we have
developed a concise lifecycle that describes how a collaborative is designed, built,
analyzed, redesigned, reconstructed and reanalyzed. This methodology is a revi-
sion of the spiral model [2] of software development, which includes explicit user
testing and analysis as part of the iteration.

The actions involved in the development of a groupware application are illus-
trated in Figure 1. The *Designer* is given a set of application requirements and

produces an application design. In the first round of development, this design is often very high-level, and can be built with primarily off-the-shelf components.
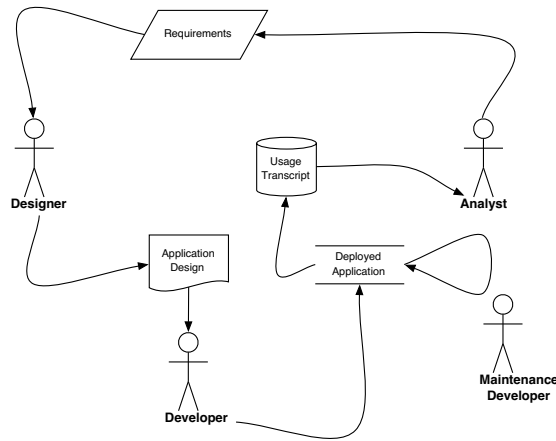

Fig. 1. Lifecycle of a THYME application

From the designer's requirements the *Developer* creates a deployable application. The deployed application is constructed to generate a usage transcript, which is analyzable via a set of existing or generated tools. A *Maintenance Developer*, who may or may not be the application developer, handles minor changes to the application such as bug fixes and other discovered issues.

Based on the analysis of the application transcript, the *Analyst* determines what changes to the application may need to be made. For groupware applications, the analysis of the mediated interaction among the users is at least as important to the redesign of the application as the interface work of the individual user. The changes are composed into a new set of requirements. These requirements are handed to the designer, who continues the cycle through the next round.

In building our framework, we took into consideration these requirements and our past experiences in building groupware (an example being the VesselWorld project [12]). Our goal was to be able to run through several iterations of the lifecycle, improving the usability and fitness of the application to the users and the task after each cycle.

To accomplish implement this proposed lifecycle, several capabilities must exist:

1. Development time must be reduced. Applications need to be constructed and modified with great velocity. This calls for a technique that reduces errors and limits interdependency in the code.
2. The application needs to generate a transcript of its use.

3. The transcript needs to have an associated toolset with it in order to analyze the transcript. These tools, such as tools to playback the transcript or collect aggregate data, must be already built or inexpensive to build; otherwise too much time will be spent developing these, adding time to each lifecycle iteration. THYME contains both transcript collection facilities that transparently build a complete transcript of use, and tools and tool generators to aid in the analysis of the transcript. The details of how a playback tool is generated is covered in another paper [11].
4. The application should be able to leverage off-the-shelf components wherever possible for common tasks, both for user-oriented tasks, such as chatting or using a shared whiteboard, and infrastructure level tasks, such as discovery and messaging. It also needs to provide facilities for reusing existing components.

This paper presents the THYME groupware framework. THYME is a flexible, component-oriented [20] architecture for building groupware applications from reusable, tailorable, and analyzable components. With this framework, groupware applications can be built, validated, and deployed quickly. The design principles behind the framework encourage the construction of reusable components that can be tailored to fit a specific needs, and, thereby, allows new groupware interactions to be codified in future groupware applications. A further consequence is that existing groupware components and interactions can be modified, sometimes subtly and sometimes significantly, to fit the needs of a specific groupware application. Groupware components in THYME also produce transcripts of their interaction with the user, allowing the user's interaction with the application to be later analyzed through, for example, ethnographic and discourse analysis techniques. These transcripts are used to perform playback-based on quantitative analysis, as well as other types of quantitative and qualitative analysis [4].

In this work, we will also discuss how THYME has been used in the classroom, specifically how a Human-Computer Interaction class made use of THYME to implement a same-time / different-place [3] groupware application as their term project. These projects were implemented over the course of a 28-day period, a feat that twelve out of fourteen groups successfully accomplished. Based on an analysis of their work, we have drawn conclusions as to the use of THYME and the adoption of its programming model. This figure is contrasted to a previous Human Computer Interaction class. This class had 49 days to implement their projects, and produced fewer complete projects.

2 The THYME Framework

To enable the proposed lifecycle for building groupware applications we constructed the THYME framework. THYME is a component-oriented framework for building groupware applications that are analyzable, rapidly constructed, and modifiable.

THYME is loosely based on the JavaBeans [6] programming model, similar to other frameworks such as Wren [15] and FlexiBeans [19]. Similar to FlexiBeans, THYME applications communicate via a peer-to-peer architecture, although can emulate a client-server relationship when a groupware application is best suited for such a design. THYME is designed to be more light-weight than Wren or FlexiBeans; components are not repository-based, instead they are instantiated and communicate through a messaging architecture, as described below.

Applications built using THYME are composites based around reusable, minimally interacting components. This architecture is in contrast to, for example, GroupKit [18], which is based around monolithic applications. GroupKit provides substantial communication primitives and shared information mechanisms, which greatly reduce the initial development of an application and provide greater development simplicity. However, modifications to the application may be more expensive.

THYME can also be contrasted to replicated architectures, like DistView [17] and DISCIPLE [14]. Whereas these frameworks provide a direct and simple method of collaboration, by creating a replicated task environment for each member of the collaboration, they trade-off the flexibility required for some types of complex collaboration. THYME explicitly supports the relaxed what-you-see-is-what-I-see (WYSIWIS) types of applications, where replicated architectures generally do not.

To ground the discussion of the framework, take the example of a *chat room*, arguably the most pervasive form of collaboration current in use. The THYME chat room, as depicted in Figure 2, allows the communication of textual information between multiple users in an open, symmetric, synchronous fashion. The basic layout exposes two major areas of interaction: the shared incoming chat view (top) and the outgoing chat view (bottom).

Fig. 2. The chat room

A set of components form the chat room component collection. Figure 3 shows the minimal set of components that exist in the chat room and the ones required from the larger THYME infrastructure.
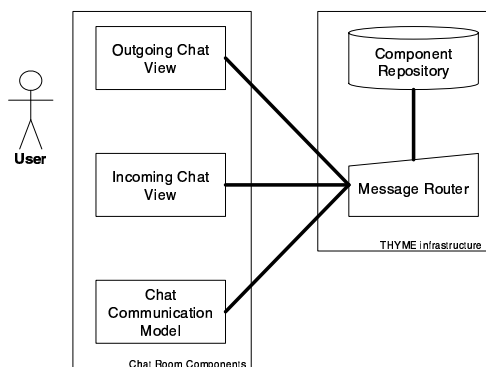
Fig. 3. Chat room components

The three custom components that form the basis of the chat room are relatively simple. They communicate through a custom message associated with the component collection called the Chat Communication Message, which contains the sender of the message, the type of activity the message encodes, and activity-specific information, such as a chat utterance. The message passing properties of the components allow reasoning about their interaction. If a THYME component has a limited set of messages it reacts to, then its interaction with other components can be clearly shown. As discussed later in this section, embedding the chat room component collection in an application is simple, straightforward, and reliable.

The remainder of this section details the THYME framework, using the chat room example as the running example.

2.1 Distributed Component Model

Within the THYME component model, there are five types of objects, with each object in a THYME application being classifiable into one of these types. Figure 4, based on the chat room system described above, shows the two major types of functional objects that exist within an application, the *component* and the *service*. Components interact with each other to perform the activity of the application. Services are components that are part of the underlying THYME infrastructure and provide resources and basic functionality for the components.

Figure 5, again based on the chat room, shows the three other types of THYME objects, the *message*, *data*, and *identifier*. Messages are used to communicate between components. They package a set of data that is passed between components, which the component uses to alter its underlying state, replace its internal state, or interpret as a request. The identifier is used to represent a local or remote component, being resolved into a component reference at run-time by a component resolver service.
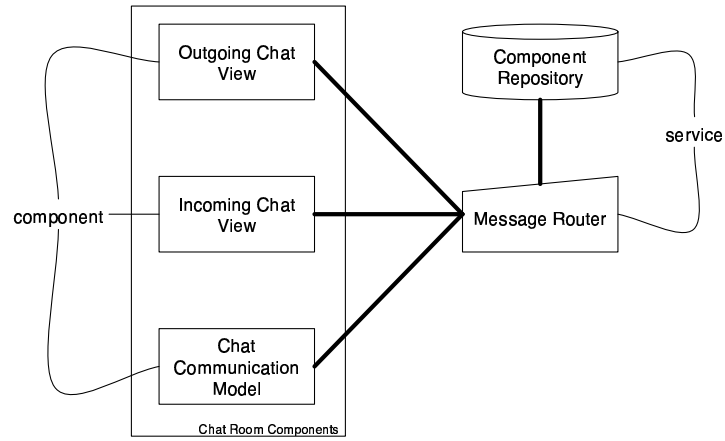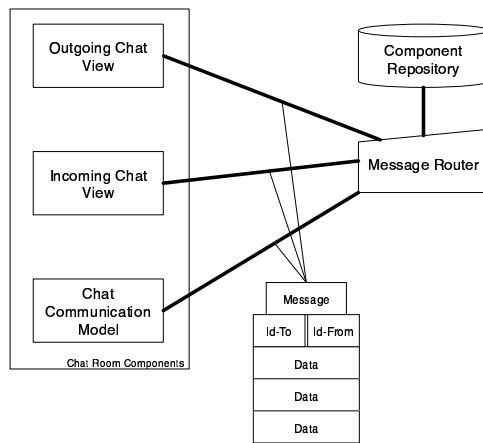
Fig. 4. Components and services

Fig. 5. Messages, data, and identifiers

Components may also be grouped into composite components, which inherit the properties of the union of their constituent parts. A complete groupware application is one such example of a composite component. The composite component is defined as $COMPOSITE = \{C_1, C_2, \ldots, C_n\}$. A composite component can be part of another composite component. The chat room is often used as a composite component that consists of the set:

$\{INCOMING\text{-}CHAT\text{-}VIEW, OUTGOING\text{-}CHAT\text{-}VIEW,$
$CHAT\text{-}COMMUNICATION\text{-}MODEL\}$

The interaction between components is restricted by the following rules:

1. Identifiers are long lived, component references are short lived. A caller should never hold on to a component reference past its immediate use, meaning past sending a message or set of messages. In any case, the component reference should not be used outside of the current method scope. The general pattern for using a component is to dereferencing the component identifier to a component reference, using that reference, and discarding it. When sending a message to a component, the message is addressed with the identifier of the receiving component. The mapping of identifier to component may be updated during run-time, so it is assumed that the mapping between the identifier and component is transient, and should be treated as such.
2. Components should interact via messages whenever possible. Services can be interacted with directly or via messages.
3. Components should be able to handle unexpected messages and data. Assume that other components can be hostile, compromised or just buggy. In these cases, the component should not fail or end up with a corrupted state. Dropping a message because it is incorrect or unexpected is correct behavior. Similarly, a component should be able to continue to act properly if it does not receive a timely response from a message sent to another component.

Components interact through messages. There is a mapping between a component identifier and a component reference. The component repository service is used to retrieve the component reference for a given identifier.

A component is altered by the processing of a message. Succinctly, given a component C and message M, $C(M) \rightarrow C'$. C' is a product of the set of data contained in the message, the state of the component, and the action contained in the message. The component processes each data object individually, but how the data affects the component's internal state is shaped by each of these factors.

The state of a component is the product of the basis component, C_0, and an ordered list of all messages that have been applied to the component. In the previous example, $C(M) = C'$ is also written as $C_n(M_{(n+1)}) = C_{(n+1)}$. The component C after the third message ($C(M_3)$) is actually $C_0(M_1) \rightarrow C(M_2) \rightarrow C(M_3)$.

The state of a composite component is defined as the state of all of its constituent composite components. A message M_n applied to a composite component is distributed to all of its constituent components, resulting in $COMPOSITE(M_n)$, which is also written as $\{C_1(M_n), C_2(M_n), \ldots, C_N(M_n)\}$.

Orthogonality The property of *orthogonality* refers to whether or not two components can directly interact. A component has a set of message types that it accepts, and a set that it produces. If two components do not have any overlap in these sets, they are said to be *orthogonal* to each other. The production and acceptance sets of a composite component is the union of the production and acceptance sets from all constituent components, respectively. If two composite components have production and acceptance sets that do not overlap, they are also considered orthogonal.

Orthogonality does not replace due diligence of the developer in verifying correct behavior of the system. Two components that are orthogonal may still interact through a common second component. The orthogonality relation does, however, provide guidance to the developer in determining what interactions are necessary to test based on the application's dependency graph.

2.2 Message Routing

Messages are sent between THYME components, similar to Smalltalk [1] or Objective-C [16], not remote procedure call (RPC)-based, like CORBA [5] or Java RMI [7]. A THYME message is sent from component to component through the use of a service called the *message router*, similar to the CORBA Object Request Broker (ORB). However, the routers that THYME uses to communicate between components is lighter-weight and used to handle communication within a single address space, as well as between components that exist within separate address spaces. THYME messaging, also like Smalltalk and Objective-C, is blind, in that a message can be addressed to any component without knowledge of whether or not the component can accept the message. In the case where a component receives a message it cannot process, it is dropped silently. RPC solutions require some knowledge of the methods available on remote objects, trading off the *a priori* knowledge of a component in favor of being able to use the programming language method call syntax to communicate with remote objects.

Messages are passed between components via message routing. The routing process ensures that the correct component or set of components receive a message and are given the opportunity to apply the message to their state. When a component sends a message, it is addressed with a set of component identifiers. This message is then passed to the message router. The message router determines how to route the message to the appropriate set of components. In the non-network scenario, this process involves resolving the set of component identifiers into a set of component references. The message is then delivered to the resulting set of component references.

Network Routing Network routing is an extension to the basic model of component routing, and adds the capability for components to communicate across different process spaces, resulting in inter-process and inter-host message passing.

In the THYME network model all components are contained within a specific service, called a *node*. The node provides the infrastructure for the general management and communication needs of a component. By default, in a system that runs on a single host and in a single process space, all objects are contained within a single, default node. When multiple process spaces are taken into consideration, multiple nodes exist. A node is differentiated from another node via its component identifier, called its $NODE\text{-}ID$. When messaging spans nodes, the node identifier is also used in the targeted component's component identifier.

All components that share direct access to a message router are defined as local and are grouped inside of a node. When sending a message to a component, the message router determines if the message is locally addressed. If it is local, the message is delivered normally via the message router. If the message is not locally addressed, the message router will establish a connection to the foreign message router identified in the component identifier. The foreign message router is sent the message, and goes through the same process, determining locality between the receiving message router and the receiving component, and delivering or re-sending as appropriate.

An example of routing between two nodes in the chat room application can be seen in Figure 6. The node on the bottom of the screen containing a component that ultimately sends out a new chat message. Within the node, the message is routed from the outgoing chat view to the chat communication model. The chat communication model addresses the message to the remote chat communication model. The local message router realizes that this target is on a remote node and sends the message to the foreign message router. Once the foreign message router realizes that the message is targeted to a component local to it, the message router obtains a component reference and delivers the message to the model. The model then dispatches a message to the incoming chat view so that the utterance contained in the original message can be displayed. Again, the message router obtains a reference to the incoming chat view and delivers the message.

2.3 THYME Component Library

The THYME framework provides two sets of groupware capabilities, shared groupware widgets and groupware component collections. Groupware widgets are shared versions of common user interface components, such as lists and tables. Groupware component collections are implementations of common groupware metaphors like shared surfaces, applications, and the chat room.

This section shows an example of both types of capabilities.

The Shared and Co-Present Scroll Pane Widgets The shared scroll pane is an example of a strict WYSIWIS widget. This widget provides a UI object that extends `javax.swing.JScrollPane`. When a widget of this type moves its scrollbar (technically, when an `adjustmentValueChanged()` event is fired), a `SharedScrollPaneActionMessage` is constructed and sent. This message informs all connected components to change their scrollbar positions, ensuring that
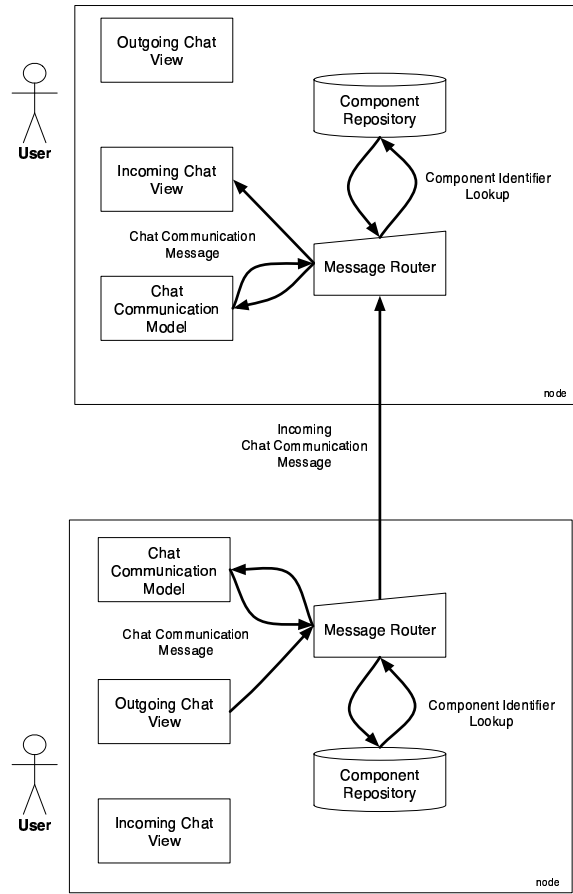
Fig. 6. A non-local routing example in the chat room

all shared scroll panes are showing the same portion of their scroll pane view at all times.

The co-present scroll pane is a relaxed WYSIWIS form of the shared scroll pane. Instead of having scroll pane position be replicated in each UI object, the user is informed of the position of other users' scroll panes through marks that are placed on the scrollbar. When a change is made to the co-present scroll pane widget, the same SharedScrollPaneActionMessage is sent. The co-present and shared scroll panes can interoperate.

The Shared Whiteboard Component The shared whiteboard presents a shared surface and set of artifacts that can be placed and altered on the canvas. The artifacts on the canvas are replicated on all canvases that are attached to the same room.

The component collection contains two view components, one for the palette of artifact types and one for the canvas. The palette is populated at run time with the list of available artifacts and canvas manipulated actions (such as DELETE and SELECT). The canvas responds to mouse actions, based on where the mouse is clicked, the status of the palette, and the internal status of the canvas. For example, if the palette has an oval artifact selected, the canvas has nothing selected, and the mouse action is a click and drag, an oval will start to be drawn. However, the palette has the SELECT action selected, the mouse is clicked within an already drawn oval and the mouse action is a click and drag, the already drawn oval will be resized on the canvas. The basic shared whiteboard is shown in Figure 7.
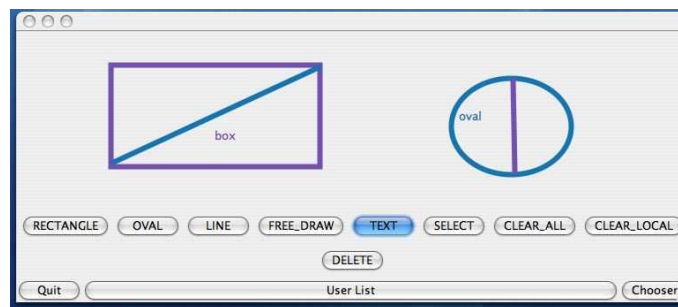

Fig. 7. The shared whiteboard

This component collection is a mostly-strict WYSIWIS groupware application. All artifacts that are done being drawn are replicated on every user's canvas. When an artifact is being drawn or being changed, it is updated at intervals on other client's canvases, while being updated continuously on the canvas of the client that is updating the artifact.

The whiteboard information is, ultimately, positional. A major externality that this component is vulnerable to involves the size and shape of the whiteboard dimensions. If one user has a much larger display than another, then some artifacts may be drawn off-screen of the other user.

3 THYME in Use

The THYME framework has been used in a variety of applications [13] [10]. This section discusses the details of how a class in Human Computer Interaction was given a term project that required teams of students to implement a same-time / different-place groupware application using the THYME framework. The class was divided into teams of three or four students. There were fourteen teams in total. At the beginning of the semester, a schedule was given, shown in Figure 8. One feature of this schedule is that the class had only 28 days to implement their projects. A key point is that the students designed their system without any knowledge of the THYME framework, sample THYME applications, or any knowledge of the THYME capabilities.

---

14 days Description of system, users and tasks. This task required the teams to interview some sample users of their proposed application and design sample scenarios of the application's use.

21 days Initial design. In this task the teams designed the interface, presented story boards of its use, and performed a GOMS [8] analysis of a subset of the proposed interface. The interface was also presented to sample users to get their comments and impressions. During this period, the class did not have access to THYME or its capabilities.

28 days Prototype implementation. At the beginning of this period, the THYME manual [9], initial instruction, and source code was given out. The THYME source code included the complete implementation of the THYME framework and implementations of sample applications that showed how the components could be combined into working groupware applications. The class did not have access to the THYME framework before this period. In addition to a working prototype, the teams produced user documentation for their applications. During this time period, three teaching assistants held approximately six hours per week of office hours, which were used by some, but not all, teams.

21 days Usability testing and redesign. This task required the teams to have their user population make use of the application. The teams collected transcripts of these sessions, which were later analyzed to identify areas of problematic coordination. This analysis led to a proposal document that described how the application could be changed to overcome collaboration problems encountered in testing.

Fig. 8. Term project schedule

During the prototype implementation stage, the class was given access to the THYME framework, the shared whiteboard, and the chat room. They had

access to both the class library and the source code. They were also given a template project, a simple THYME application that showed how to embed both the shared whiteboard and chat room components in a single application.

Of the fourteen teams, twelve teams completed applications that were able to be tested. A usable application was defined as one that was sufficient to obtain user feedback regarding its appropriateness to the task and generate a transcript of use.

As a point of comparison, a similar class was taught in the Fall semester of 1999, when the THYME framework was not available, but some comparable sample groupware code was distributed. In this previous class, the teams were given 49 days to implement their applications, 21 more days than the 2002 class. Nevertheless, the previous class had significantly fewer usable applications, roughly half of the teams in that class produced usable applications.

3.1 Resulting Projects

Each project implemented by the Fall 2002 class showcases some of the different types of applications that can be constructed using THYME. This section details a subset of the implemented projects and how the framework was used to implement their groupware design.

ORA The **O**nline **R**esearch **A**ssistant is an application that allows a more experienced researcher (such as a librarian) to help another researcher locate information on the World Wide Web.

This application was built using the shared whiteboard, chat room, and shared browser components and made use of the shared scrollpane widget. The shared whiteboard was modified to be used as a glass pane on top of the browser. The browser and glass pane were put inside of a shared scrollpane. The chat room assigned a different color to each user, synchronizing the choice of color with the shared whiteboard. The chat room was also modified to show "emoticons" in the incoming chat view. No new messages were added. All these changes were cosmetic, altering properties of the components, but not altering their interaction.

An example of this application in use can be seen in Figure 9. In this example interaction, two users are collaboratively working to find a specific reference using the ORA tool. Through the use of the overlaid shared whiteboard, the collaborator who found the appropriate reference can direct the other participant to it.

The ORA client application is defined by the set

$$\{\{INCOMING\text{-}CHAT\text{-}VIEW', OUTGOING\text{-}CHAT\text{-}VIEW, CHAT\text{-}MODEL\}, SHARED\text{-}WHITEBOARD\text{-}COMPONENT\text{-}COLLECTION, SHARED\text{-}BROWSER\text{-}COMPONENT\text{-}COLLECTION\}$$

Where the $INCOMING\text{-}CHAT\text{-}VIEW'$ is a modified version of the original $INCOMING\text{-}CHAT\text{-}VIEW$ that supports user-defined colors and emoticons.
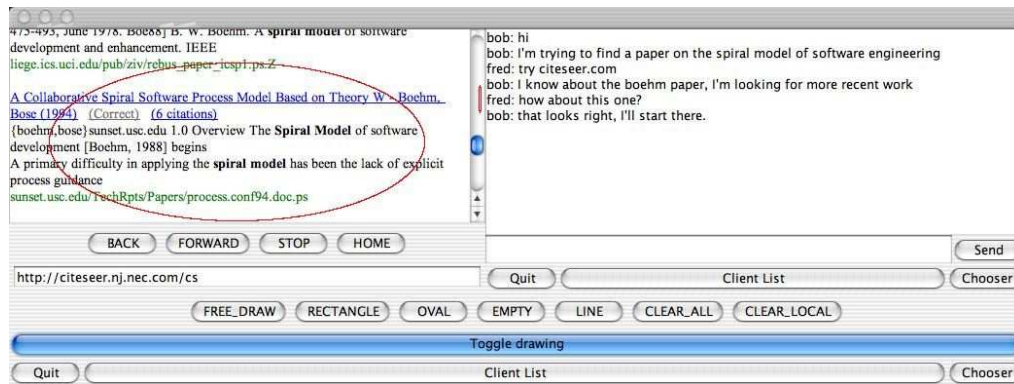
Fig. 9. Screenshot of the online research assistant

SALSA The **S**upplementary **A**cademic **L**earning **S**ystem - **A**lpha application provides a way for an instructor to lecture to geographically distributed students and for those students to interact with the instructor. SALSA added two related concepts to the THYME framework, floor control [3] and classes of users.

In SALSA there were two different classes of users, the instructor and the student. The instructor regulates the floor control of the system. He could give permission for a student to speak, decide who the next student to speak would be, and revoke permission at any point. During a typical session, the instructor would lecture and a student would "virtually" raise his hand. When the instructor was ready to accept questions or comments, he would transfer control to a user of his choice. Only one person could affect the chat room or shared whiteboard at a time.

RA Scheduler The RA Scheduler is a groupware application to facilitate the scheduling of Resident Assistant office hours at a university. The RA Scheduler team used the chat room in implementing their own shared scheduling calendar by implementing a series of canned messages, listened for by their chat client component. When those messages were received, the chat client would parse them and pass them to another component, a technique we refer to later as *hijacking* of the component set. This effect was accomplished by modifying the IncomingChatView (and only the IncomingChatView) to display their graphical calendar interface and to send canned messages when the calendar is acted upon. The message router, acting as a broker for the messages, blindly passes them on to the other hijacked views, which parse the payload of the message and update their view of the calendar appropriately.

4 Conclusions

The THYME framework allows developers to quickly build and rebuild groupware applications to fit the needs of their task domain and their users. This

paper showed the underlying component model behind the THYME application and provided evidence of how a class of novices to the area of groupware development could quickly adopt THYME for their needs. In addition to the component model and basic groupware features, such as synchronous communication and the chat room, THYME also has a rich groupware component library, which was touched upon here. Additionally, THYME is being used as an active development platform. The ORA application was adapted for experiments that took place at the University of Massachusetts School of Management, and other applications are currently being built and used in experimentation [13].

The use of THYME in this classroom setting was driven by the need to show that analyzable groupware could be developed quickly given the appropriate set of tools. The breadth of groupware applications developed, sampled in this paper, show the potential of the THYME framework and its simple, but not simplistic, model for developing groupware and component library that can be adopted quickly. Additionally, as covered elsewhere [11], the complete transcript of use provides by the THYME components can be used by analysis tools that enable the detailed understanding a community of users' practice as mediated by the groupware application.

Introducing a new framework and component model to Computer Science students is hard. Component-oriented programming is not usually taught at the undergraduate level and many of the techniques that make it an effective model of development take time and effort to perfect. However, by observing how a component model, such as THYME, is used over the course of the semester, we can hope to gain some insight as to how to make the transition easier and more complete, thereby broadening the applicability of new, and potentially better, models of groupware development.

5   Acknowledgments

References

1. The smalltalk programming language. http://smalltalk.org.
2. Barry Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 21(5):61–72, 1988.
3. Clarence A. Ellis, Simon J. Gibbs, and Gail Rein. Groupware: some issues and experiences. *Communications of the ACM*, 34(1):39–58, 1991.

4. Alexander Feinman and Richard Alterman. Discourse analysis techniques for modeling group interaction. In *Ninth Internation Conference on User Modeling*, 2003.

5. Object Management Group. The common object request broker: Architecture and specification. Technical report, 1995.

6. JavaSoft. The javabeans component architecture, 2003. http://java.sun.com/products/javabeans/.

7. Javasoft. RMI, 2003. http://java.sun.com/products/jdk/rmi/.

8. Bonnie E. John and David E. Kieras. The GOMS family of user interface analysis techniques: comparison and contrast. *ACM Transactions on Computer-Human Interaction*, 3(4):320–351, 1996.

9. Seth Landsman. The Tiny THYMEr, a manual for using the THYME framework. Technical Report TR-02-231, Dept of Computer Science, Brandeis University, 2002.

10. Seth Landsman. *A Software Lifecycle for Building Groupware Applications: Building Groupw are On THYME*. PhD thesis, Brandeis University, 2006.

11. Seth Landsman and Richard Alterman. Using transcription and replay in analysis of groupware applications. Technical Report CS-05-259, Brandeis University, 2005.

12. Seth Landsman, Richard Alterman, Alexander Feinman, and Joshua Introne. Vesselworld and ADAPTIVE. Technical Report TR-01-213, Dept of Computer Science, Brandeis University, 2001. Presented as a demonstration at *Computer Support Cooperative Work 2000*.

13. Johann Ari Larusson and Richard Alterman. Integrating collaborative technology into the interdisciplinary classroom. In Preperation.

14. Wen Li, Weicong Wang, and Ivan Marsic. Collaboration transparency in the disciple framework. In *Proceedings of Group 1999*, 1999.

15. Chris Lüer and David S. Rosenblum. WREN - an environment for component-based development. In *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 207–217. ACM Press, 2001.

16. Lewis J. Pinson and Richard S. Wiener. *Objective-C: Object-Oriented Programming Techniques*. Addison-Wesley Pub Co, 1991.

17. Atul Prakash and Hyong Sop Shim. Distview: Support for building efficient collaborative applications using replicated objects. *Proceedings of Computer Supported Collaborative Work*, 1994.

18. Mark Roseman and Saul Greenberg. Groupkit: A groupware toolkit for building real-time conferencing applications. In *Proceedings of CSCW 92*, 1992.

19. Oliver Stiemerling, Ralph Hinken, and Armin B. Cremers. Distributed component-based tailorability for CSCW applications. In *ISADS*, pages 345–352, 1999.

20. Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison Wesley, 1997.