A Collaborative Interface Agent for Lotus eSuite Mail

by

Ada Hoi-Fay Cheung

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Computer Science and Engineering

and

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

August 7, 1998

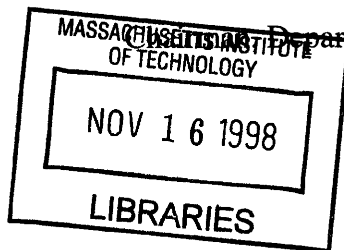
© Copyright 1998 Ada Hoi-Fay Cheung. All rights reserved.

The author hereby grants to M.I.T. permission to reproduce and
Distribute publicly paper and electronic copies of this thesis
and to grant others the right to do so.

Author____
Department of Electrical Engineering and Computer Science
August 7, 1998

Certified by____
Howard E. Shrobe
Thesis Supervisor

Accepted by____
Author C. Smith
Chairman, Department Committee on Graduate Theses


MASSACHUSETTS INSTITUTE
OF TECHNOLOGY
NOV 16 1998
LIBRARIES

preprocessor takes a Collagen library file as input, and based on the grammar specified, outputs a set of Java files, which are the recipes used by Collagen. With a preprocessor, we can easily expand and modify the language of the Collagen recipe library. In addition to saving implementation effort, it is a useful housekeeping tool in maintaining the consistency among recipes. Modifications can be made on the grammar specification, instead of modifying every single recipe.

4.5. Agent

The agent bean is responsible for decision making by choosing which communication or manipulation in the current agenda act to perform. Collagen provides a default agent with a selection strategy that simply chooses to perform the highest priority acts in the current agenda. In order to build an agent that can provide intelligence in a specific domain of interest, the selection strategy can be overridden in order to meet specific needs of different applications. For example, rule-based expert systems or neural nets can be used instead. Also, the agent developer can create a set of application-specific rules for prioritizing acts that can occur on the agenda. Moreover, he can experiment with a small number of rules for choosing other actions than the default collection. For example, if the user proposes that the agent perform an act that the agent cannot do, the agent might refuse and explain its refusal (e.g. "No, I cannot do that action.").

4.6. Adapter

The adapter bean is similar to an event adapter. An event adapter is the object between the event source and the event listener that adapts the source to the specific needs of the

listener. Notice that in Figure 4-1, there are two directions of events going through the adapter. This indicates that there are two pairs of event source and event listener. Collagen, or the discourse bean to be precise, is the listener of events fired by the application, while the application is the listener of events fired by the agent. Precisely speaking, the application and the agent each has a collection of event sources. Hence, it would be convenient to have a single instance of an adapter to handle events from multiple event sources, and forward the events to different methods to the corresponding listener. This can be achieved by using the Java Reflection API, which supports the analysis of Java objects and classes at runtime. This is the same technique that the JavaBeans *introspection* mechanism uses to determine the properties, events, and methods that are supported by a bean. In addition, the adapter acts as a translator. Application events (labeled as anEvent and anotherEvent) that go into the adapter from the application are translated to Collagen-understandable events. On the other hand, events coming from Collagen are also translated to application events.

4.7. Application

4.7.1. Java 1.1 Event Model

A target application that is built upon good design criteria will greatly simplify the implementation effort of connecting the application with Collagen. The event model in Java 1.1, as described in Section 4.2.1, is one of the good design criteria. External applications can easily register as listeners and receive events that notify manipulations on the user interface. In addition, the Java 1.1 event model facilitates the use of another good design criteria, the model-view-controller (MVC) paradigm.

A Collaborative Interface Agent for Lotus eSuite Mail

by

Ada Hoi-Fay Cheung

Submitted to the
Department of Electrical Engineering and Computer Science

August 7, 1998

In Partial Fulfillment of the Requirements for the Degree of
Bachelor of Science in Computer Science and Engineering
and Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

A collaborative interface agent for Lotus eSuite mail is designed and implemented. An application-independent collaboration manager, called Collagen, provides the standard mechanism of collaborative discourse between the user and agent. The technical challenge is to connect Collagen with eSuite mail by enabling the discourse bean to be notified of the user's manipulations on the application, as well as enabling the agent to manipulate the application. The actual decision making of what manipulation or communication to perform at different contexts is provided by a domain-dependent agent with application and user specific knowledge and rules. The ability of the collaborative interface agent to provide intelligent assistance is demonstrated in a sample scenario.

Thesis Supervisor: Howard E. Shrobe
Title: Associate Director, MIT Artificial Intelligence Laboratory

Thesis Supervisor: Candace L. Sidner
Title: Research Scientist, Lotus Development Corporation

Acknowledgments

I would like to give my sincere thanks to my Lotus supervisor Candy Sidner for her unfailing support and guidance throughout the project, and Charles Rich from Mitsubishi Electric Research Laboratory for his great technical insights and his effort on making Collagen available for the project. I would also like to extend my gratitude to my MIT supervisor Howard Shrobe for his time and effort on supervising the project. Next, I would also like to thank my grandmother, my parents, my sisters Fay and Denise, and Edmond for their love and care. Last but not least, I would like to thank the Lord Jesus Christ for His unfailing love and guidance throughout my life.

Contents

1. INTRODUCTION ---------- 10

2. COLLABORATIVE INTERFACE AGENT ---------- 12

3. COLLAGEN ---------- 14

3.1. Collaborative Discourse Theory ---------- 15

3.2. Discourse Interpretation ---------- 16

3.3. Discourse State ---------- 16

3.4. Discourse Generation ---------- 17

3.5. Basic Execution Cycle ---------- 17

4. CONNECTING COLLAGEN WITH AN APPLICATION ---------- 19

4.1. Interconnection ---------- 19

4.2. User ---------- 21
4.2.1. Manipulation ---------- 22
4.2.2. Communication ---------- 23

4.3. Collagen ---------- 23

4.4. Recipe Library ---------- 23

4.5. Agent ---------- 24

4.6. Adapter ---------- 24

4.7. Application ---------- 25
4.7.1. Java 1.1 Event Model ---------- 25
4.7.2. Model-View-Controller Paradigm ---------- 26
4.7.3. JavaBeans ---------- 27

4.8. Event Flow ---------- 28

5. ESUITE MAIL ---------- 30

5.1. Applets in eSuite Mail ---------- 31

5.2. InfoBus ---------- 32

5.3. User Interface ---------- 33

ActionBar -- 33
5.3.1 InfoCenter -- 34
5.3.2 Main Window -- 36

5.4. Comparison with the Ideal Application --38

6. CONNECTING COLLAGEN WITH ESUITE MAIL -------------------------------- 40

6.1. "Beanification" --41
6.1.1. InfoCenter -- 44
6.1.2. Main Window --- 45

6.2. Mail Adapter --49

7. KNOWLEDGE ABOUT ESUITE MAIL -- 51

7.1. Knowledge Engineering --51

7.2. Recipe Library for eSuite Mail Domain --52

7.3. Knowledge Engineering --53

7.4. Learning --55

8. DOMAIN-SPECIFIC AGENT FOR ESUITE MAIL -------------------------------- 57

8.1. Default Agent --57

8.2. eSuite Mail Agent --58

9. SAMPLE SCENARIO --- 62

9.1. The "Working on e-mail" Scenario --63

10. CONCLUSION -- 69

10.1. Summary of Work --69

10.2. Lessons Learned --70

10.3. Future Direction --71

A. USER INTERFACES OF ESUITE MAIL -- 73

A.1 User Interfaces in MailApplet --73

A.3 User Interfaces of ComposeMessageApplet ---74

A.3 User Interface of ReadMessageApplet --76

B. SEGMENTED INTERACTION HISTORY -- 77

REFERENCE -- 79

List of Figures

Figure 2-1: Collabortive Interface Agent Paradigm ---------- 13
Figure 3-1: Collagen Architecture ---------- 14
Figure 4-1: Collagen bean wiring diagram ---------- 20
Figure 4-2: Model-view-controller paradigm ---------- 26
Figure 5-1: ActionBar ---------- 34
Figure 5-2: Internal data flow between applet and InfoCenter via InfoBus ---------- 35
Figure 5-3: Main window ---------- 36
Figure 6-1: Applet and wrapper ---------- 41
Figure 6-2: Interactions among wrappers ---------- 43
Figure 6-3: Internal data flow among applet, InfoCenter and wrappers via the InfoBus ---------- 44
Figure 6-4: AWT component ---------- 46
Figure 6-5: Bongo widget ---------- 47
Figure 6-6: Event flow ---------- 49
Figure 7-1: Sample recipe ---------- 53
Figure 9-1: eSuite mail with agent ---------- 62
Figure 9-2: A set of top-level goals from the communication menu in the user home window -- 64
Figure 9-3: Agenda after user proposes filling in message ---------- 66
Figure 9-4: agent fills in cc: field ---------- 67
Figure A-1: Mailbox/Preview View – hierarchical folders shown ---------- 73
Figure A-2: Mailbox/Preview View – hierarchical folders hidden ---------- 74
Figure A-3: New Message Creation View ---------- 74
Figure A-4: Reply View ---------- 75
Figure A-5: ForwardView ---------- 75
Figure A-6: Read Message View ---------- 76
Figure B-1: Segmented interaction history of sample scenario ---------- 77

Table

Table 5-1: Features on the main window and ActionBar of each applet-----------------------------33

Chapter 1

Introduction

In a previous effort, a collaborative interface agent that operates on a custom-designed test application of travel scheduling was built. Part of the agent was built by an application-independent collaboration manager called Collagen, which is a collection of algorithms, data structures and specifications that application developers can easily use to add a collaborative interface agent to any application. Collagen was first implemented in Lisp, but has since been ported to Java. In this thesis, we report on using the new Java-version of Collagen to build a collaborative interface agent that collaborates with users of a commercial application, Lotus eSuite mail, to complete different tasks. By building a collaborative interface agent for Lotus eSuite mail, we have explored a new set of techniques and tools necessary for an agent to operate on a much more complicated, full commercial application. We expect that the same set of tools and techniques can be applied to build a collaborative interface agent for any application. The value of an agent is its ability to assist users in problem solving. We have built an intelligent agent with application and user specific knowledge and rules for deciding what act to perform in the e-mail domain. Based on a sample scenario, we have investigated the ability of the agent to provide intelligent assistance to users.

This thesis is divided into ten chapters. Chapter 2 defines what a collaborative interface agent is. Collagen is the collaboration manager of the collaborative interface

agent. Chapter 3 gives an overview of the architecture of Collagen and the collaborative discourse theory. Chapter 4 summarizes the general mechanisms for connecting Collagen to an application, as well as the techniques and tools involved. The next four chapters delve into the detail of building a collaborative interface agent for eSuite mail: Chapter 5 describes the architecture of eSuite mail; Chapter 6 discusses the technical detail of connecting Collagen with eSuite mail; Chapter 7 describes the knowledge acquisition process and the formal model of collaborative tasks being performed by both the user and agent in the eSuite mail domain; Chapter 8 summarizes the behavior of the agent that operates specifically on the eSuite mail domain. Chapter 9 evaluates the ability of the agent to provide intelligent assistance to users based on a sample scenario. Chapter 10 concludes this thesis.

Chapter 2

Collaborative Interface Agent

According to Maes and Wexelblat (1996), an *agent* is a computational system with the following properties to some extent:

- has goals, sensors, and effectors,
- decides autonomously which actions to take in the current situation to maximize progress toward its goals,
- learns or adapts to improve upon its effectiveness.

There are different kinds of agents. A software agent is an agent that assists users with computer-based tasks. For an application with a direct manipulation interface, the state of the objects in the application is graphically visible and modifiable. A user's manipulations on the interface can modify the state of the objects. An interface agent is a software agent that can affect the objects in a direct manipulation interface without explicit instruction from the user (Lieberman, 1997). As user interfaces are becoming more and more sophisticated and complicated, interface agents become important in providing assistance to users in operating an interface. One of the underlying assumptions is that a human-computer interface that embodies human discourse rules and conventions will be easier for people to learn and use than one that does not.

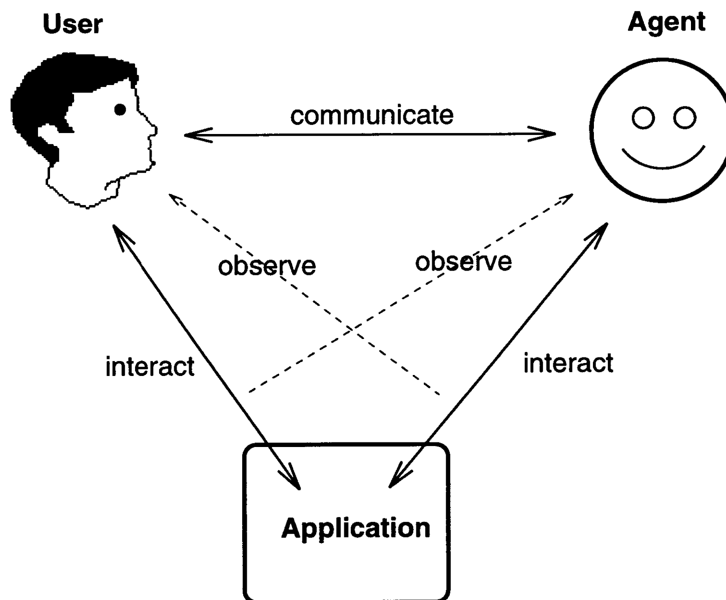


Figure 2-1: Collabortive Interface Agent Paradigm

Our version of interface agent, which we term a *collaborative interface agent*, is illustrated in Figure 2-1. It is a software agent that is able to communicate with and observe the actions of the human user. The relationship between the user and the agent is like that of two humans collaborating to complete a task. The agent can also interact directly with the application through the same graphical interface to perform a shared task. The user can observe the agent's interaction with the application. To add a collaborative interface agent for a specific application, part of the implementation is provided by an application-independent collaboration manager called Collagen. In addition, we also rely on domain-specific knowledge and rules that enable the agent to perform intelligent assistance. For example, the agent can help users by suggesting what to do next, summarizing the current state of the problem solving process, and performing delegated tasks.

Chapter 3

Collagen

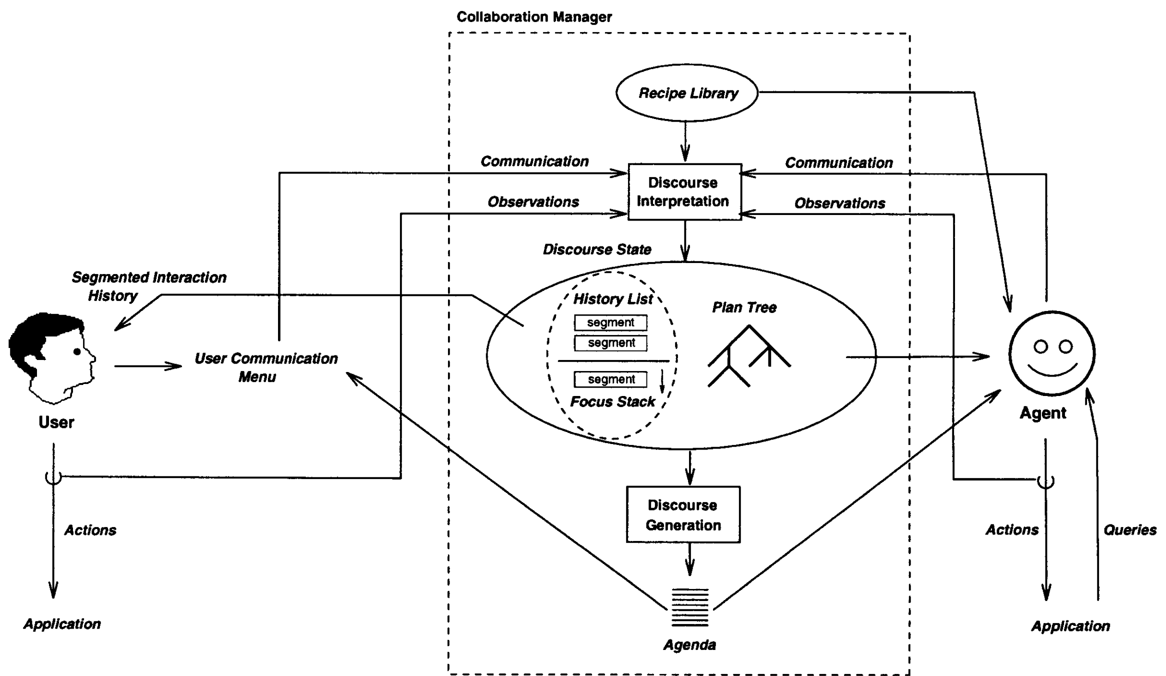


Figure 3-1: Collagen Architecture

Collagen embodies a set of conventions for a collaborative discourse. It acts as a collaboration manager between the user and the agent by providing standard mechanism for maintaining the flow and coherence of agent-user interaction. Figure 3-1 shows the architecture of Collagen. The first version of Collagen was developed in Lisp. Recently Collagen was ported to Java and now embraces the properties of being both application and platform independent.

In order to understand the architecture, we will first summarize the collaborative discourse theory on which Collagen is based. Then, we will discuss in greater detail the three central components of Collagen: discourse interpretation, discourse state, and discourse generation. Lastly, we will go through a basic execution cycle. This is a brief overview of the architecture of Collagen. For details, please refer to Rich and Sidner (1997).

3.1. Collaborative Discourse Theory

Here are some definitions of useful keywords in the context of collaborative discourse theory.

- Collaboration – a process in which two or more participants coordinate their actions toward achieving shared goals.

- Discourse – an extended communication between two or more participants in a shared context, such as a collaboration.

- SharedPlan – a formal representation (Grosz and Kraus, 1996) of the various aspects of the mental states of the collaborators: mutual beliefs about the goals and actions to be performed and the capabilities, intentions, and commitments of the participants.

- Discourse segment – a contiguous sequence of communicative actions that serve some higher level purpose. It is the building block of the natural hierarchical structure of discourse.

- Focus Stack – a representation that captures the shifting focus of attention in a discourse. Segments and subsegments in the flow of a collaborative discourse are pushed onto the focus stack when first created, and popped off the stack as the SharedPlan unfolds in the conversation. Sometimes communication by the user and the agent can force an interruption in the current segment, cause the current SharedPlan to be abandoned even though it is not complete, or cause a return to earlier segments.

Collaborative discourse theory (Grosz and Sidner, 1986; Grosz and Sidner, 1990) on which Collagen is based can be summarized in terms of three interrelated parts of discourse structure:

- intentional structure, which is formalized as partial SharedPlans.
- linguistic structures, which includes the hierarchical grouping of utterances into segments.
- attentional structure, which models the discourse participants' focus of attention and is captured by a focus stack of segments.

3.2. Discourse Interpretation

The discourse interpretation engine in Collagen uses an algorithm by Lockbaum (1995). It determines if the communication or observed manipulation act continues the current purpose, starts a discourse purpose that contributes the current one, or follows up on a previous discourse purpose. Section 3.5 describes this cycle in more detail.

3.3. Discourse State

The discourse state in Collagen is a concrete representation of the three parts of discourse structure described above. The discourse state is composed of a plan tree, a focus stack and a history list. The plan tree is an approximate representation of a partial SharedPlan. It consists of alternating act and recipe nodes. Both acts and recipes have bindings with constraints between them specified in their recipe library definition. An act node has a binding for each of its parameters. A non-primitive act node forks a recipe node. A recipe node has a binding for each step in the recipe.

A history list contains top-level segments that have been popped off the focus stack. As explained above, segments and subsegments are popped off the stack as the SharedPlan of the user is identified. These segments and subsegments are *closed* in the sense that no additional acts will be added to them. Conversely, segments and subsegments on the focus stack are called *open*. Segmented interaction history is a device for interactively displaying and manipulating the discourse state.

3.4. Discourse Generation

Discourse generation looks at the current focus stack and associated SharedPlan and produces a prioritized agenda of actions that would contribute to the current discourse segment purpose. The agenda contains possible, future communication and manipulation actions to be performed by either the user or agent that advance the current problem-solving process.

3.5. Basic Execution Cycle

The basic execution cycle of the architecture starts with the arrival of a communication or manipulation event from either the agent or the user at the discourse interpretation module. The interpretation module updates the discourse state, which then causes a new agenda of expected communication and manipulation acts to be computed by the discourse generation module. The agent may decide to select an entry in this new agenda for execution.

In Collagen, the only means of communication from the user to the agent is the communication menu. A subset of the agenda is displayed on the user communication

menu. The user is not allowed to make arbitrary communications, but only to select from communications expected by discourse interpretation. Hence, the capability to use any natural language utterance and the associated problems of understanding are avoided.

All of the internal data flow in Figure 3-1 takes place using Collagen's artificial discourse language. Whenever there is a need for the user to see information, such as displaying the user communication menu or the segmented interaction history, these internal representations are given an English gloss by simple string substitution in templates defined in the recipe library.

Chapter 4

Connecting Collagen with an Application

Although Collagen eases the implementation effort of a collaborative interface agent, a considerable application-specific implementation is required in order to connect Collagen with an application. The focus of the thesis will be to understand the techniques and tools necessary to connect Collagen with an external application, especially with a commercial application. The technical challenge will be for Collagen to be notified of the user's interactions with the interface of the application, as well as to perform actions on the interface as if it is the user. Also, a domain-specific agent will be built in order to provide intelligent assistance to users.

In this chapter, we will first discuss the proposed interconnection of the components for building the collaborative interface agent, followed by a general overview each of the main components.

4.1. Interconnection

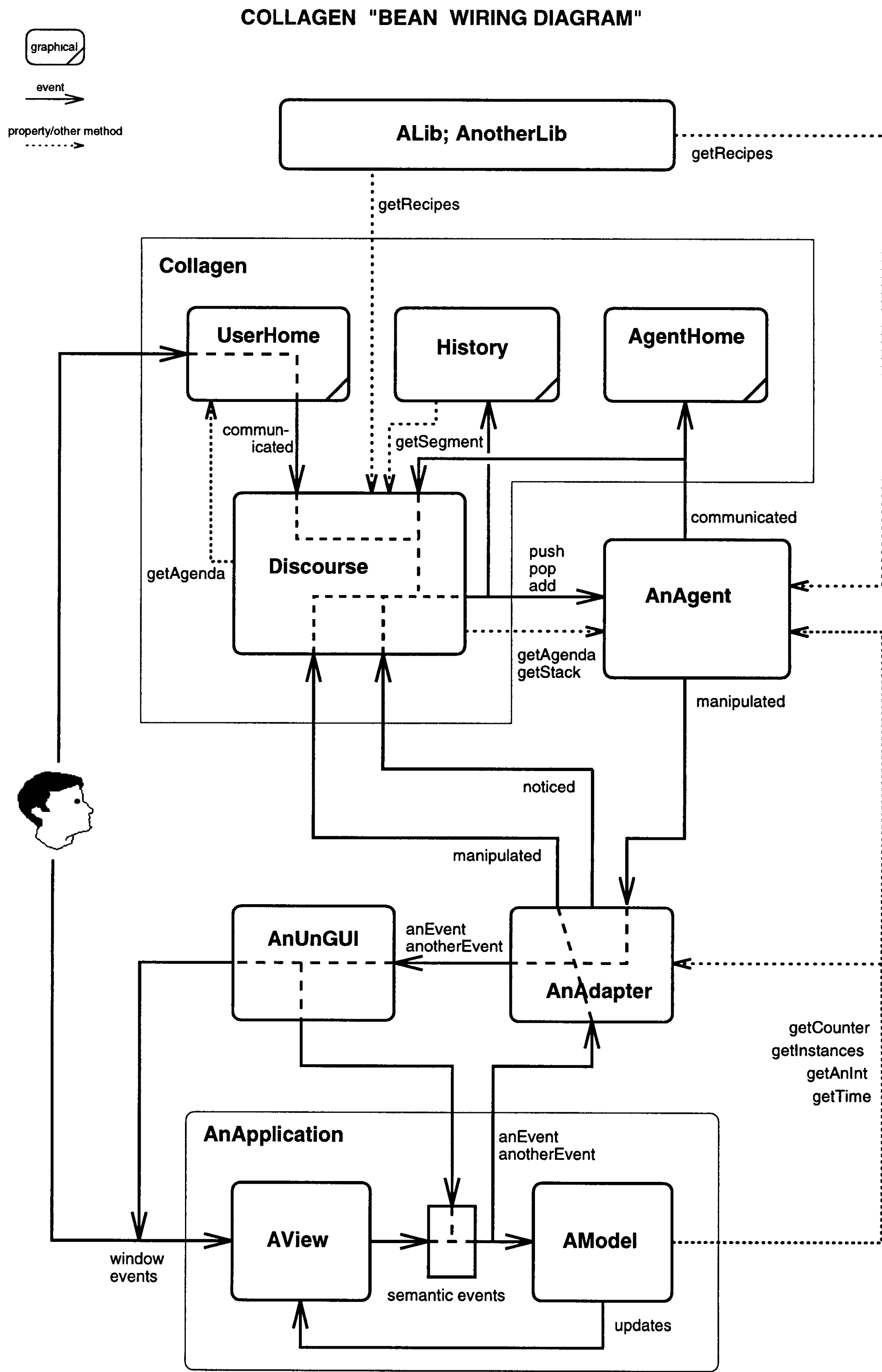


Figure 4-1: Collagen bean wiring diagram

Figure 4-1 shows the wiring diagram of Collagen. All components in the diagram are JavaBeans components, or simply called beans. The application is not necessarily a bean, but we assume it is for simplification. Notice the distinction between Java bean and JavaBeans. JavaBeans refers to Java's component architecture, which allows components built with Java to be used in graphical programming environments; while Java bean is a reusable software component that can be visually manipulated in builder tools. The graphical development environment allows components to be configured by specifying aspects of their visual appearance and the interactions between components.

The diagram is composed of six main modules, mainly the user (depicted by the face), Collagen, the recipe library (labeled as ALib; AnotherLib), the application (labeled as AnApplication), the adapter (labeled as AnAdapter), and the agent (labeled as AnAgent). The key module is the adapter, which is the bridge between Collagen (upper part of the diagram) and the application (the lower part of the diagram). To support asynchronous real-time interaction, the application and each home window run on a separate thread. Hence, the agent can ask a question while the user is operating on the interface.

4.2. User

The user can perform two types of human-computer interactions: manipulation and communication. Manipulation is the user's actions on an application such as mouse clicks and button presses. Communication is a means of expressing ideas between the user and the agent using a communication menu as described in the Section 3.5.

4.2.1. Manipulation

AWT, Abstract Windowing Toolkit, is a package in the Java API that defines a large collection of classes for building graphical user interfaces in Java. Graphical user interfaces built by AWT are event driven, meaning they spend much of their time idling, waiting for an event to occur; when an event occurs, an event handler responds to the event. In general, events are messages sent from one object to another, notifying the recipient that something interesting has happened. AWT provides an event model for handling events. In Java 1.1, the event model is characterized by two main components: *event sources* and *event listeners*. Event sources generate, or *fire*, events, which are received, or *handled*, by event listeners who have registered with the sources to be notified of the events. This is also known as the delegation event model as this approach involves delegation: the responsibility for handling an event generated by one object may belong to another object. JavaBeans architecture takes advantage of the event model. Beans are simply objects in the event model

Manipulations on the graphical user interface of a Java application (or applet) will generate AWT events. In Figure 4-1, the two modules with graphical user interface that allow direct manipulation are: the application and the user home window (labeled as UserHome). In addition, the segmented interaction history (labeled as History), which is presented whenever the user presses the History button on her home window, provides a means for users to explore the discourse state.

.

4.2.2. Communication

The user communication menu on the user home window and the agent home window (labeled as AgentHome) provides the means of communication between the user and the agent without natural language understanding. Communication from the agent to the user is achieved by printing English text in the agent's home window. Communication from the user to the agent is achieved by the user selecting from the communication menu.

4.3. Collagen

Collagen is described in detail in Chapter 3. It consists of the discourse bean, the user home window, the segmented interaction history, and the agent home window. Collagen provides application-independent components that can be reused for connecting with different applications.

4.4. Recipe Library

A recipe is a resource used to derive a sequence of steps to achieve a given goal. It is represented as a partially ordered sequence of act types (called steps) with constraints between them. The recipe library contains recipes indexed by their objective. There may be more than one recipe for each type of objective.

The recipe library is essentially the knowledge engine of the collaborative agent, and it is application specific. However, the underlying grammar of the recipes is application independent. Hence, our initial effort involved building a preprocessor using the Java Compiler Compiler (JavaCC), a tool that reads a grammar specification and converts it to a Java program that can recognize matches to the grammar. The

4.7.2. Model-View-Controller Paradigm

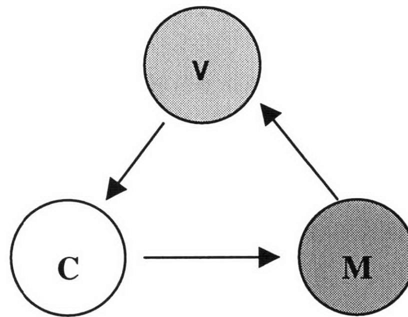


Figure 4-2: Model-view-controller paradigm

MVC, originated in Smalltalk, allows the construction of modular user interface that separates the interface (view) from the control elements (controller) and from the application code (model). The controller listens to user inputs on the view and translates them to changes of states in the model. The application, given the new state, notifies the interface to update its appearance.

The application illustrated in Figure 4-1 uses the MVC paradigm. Notice that the application is only separated into model (labeled as AModel) and view (labeled as AView). The controllers of the interface are not shown. They are considered implicit in the view since most GUI components include both the view and controller. The Java 1.1 delegation event model allows listeners to act as controllers handling events fired by the interface. This supports the view-controller relationship. Java also provides an observer-observable model that allows user interfaces be notified of the changes in the states of the

application programs. This supports the model-view relationship. The delegation model and observer-observable model together ease the implementation effort of the MVC framework in Java. By using the MVC paradigm, code is much more maintainable and easier to debug. Moreover, user interface components as well as the application can be reusable, which supports the current trend of Java technology with the advent of JavaBeans. This is what we want an application be able to do – to be reusable with a collaborative interface agent.

4.7.3. JavaBeans

The Java 1.1 event model also supports the JavaBeans architecture, which is an another good design criterion. JavaBeans is an extension of the event model for external components. In fact, all AWT components in Java 1.1 are simple Java beans. Java beans are simply objects in the event model, and they are designed for exporting properties, events, and methods. Therefore, if the application is a bean, then other components such as the discourse bean and the agent could easily receive events and access the application's exposed methods to affect the states of objects in the application. JavaBeans promotes reusability of software that can be readily plugged in with other components. The InfoBus, which will be discussed in detail in Chapter 5, defines a standard communication mechanism for interconnecting Java beans.

Since JavaBeans is still a relatively new technology, most Java commercial products are not Java beans. Although they probably have an internal event model, they may not have the mechanisms that allow interactions with external components.

Integrating such an application with Collagen could require that we modify the application to provide for interaction with Collagen.

4.8. Event Flow

In Figure 4-1, the arrows show the flow of different kinds of events within the application, from the application to Collagen, or from the application to the agent. When the user directly manipulates the interface, the controller object will receive a low-level window event that occurs via the delegation event mechanism. The controller is responsible for handling the event. It fires a semantic event to notify the application to change state. For example, the controller listens for low-level window events like mouseDown and fires semantic events like actionPerformed. The application, given the new state, notifies the interface to update its appearance via the observer-observable mechanism. This completes the event flow within the application.

There are two main directions of event flows between the application and outside components: the upward direction from the application to Collagen via the adapter and the downward direction from the agent to the application via the adapter. For the upward direction, after the model changes the state based on the semantic event, an event will be sent to the adapter. The adapter translates the application event to a Collagen-understandable event and fires it to the discourse bean. The discourse bean can therefore be notified of user's manipulations. For the downward direction, the agent fires a Collagen event to the adapter. The adapter translates the Collagen event to an application event and fires it to the application.

In essence, this is a multi-user system. Notice in Figure 4-1 that it is the responsibility of the application to fire semantic events regardless of where they come from. This way, it is guaranteed that there is a correct synchronization between manipulations by the agent and by the user. The little unlabeled box between AView and AModel is essentially the "synchronizer" that performs this role.

This illustrates an ideal situation of connecting Collagen with an application that is based on good design criteria. Unfortunately, eSuite mail holds almost none of the good design criteria for easy connection we have mentioned above. It is not a Java bean nor does it closely follow the MVC paradigm. Also, only part of the user interface uses the Java 1.1 event model. In the next chapter, we will present our design and implementation for connecting Collagen with eSuite mail. Readers can use this section for contrast and comparison.

Chapter 5

eSuite mail

The main technical contribution of this thesis is to connect Collagen with eSuite mail to build a collaborative interface agent. eSuite mail is one of the Java applets in Lotus eSuite WorkPlace project[1]. It is a lightweight[2] e-mail client designed for Internet environments. It supports standard messaging formats and open protocols. The current version of eSuite mail operates with the eSuite WorkPlace desktop, which is a launching platform and container for Java applets.

eSuite mail is a very complicated application. It is actually composed of several applets. In this chapter, we will first present an overview of the relationships among these applets. eSuite mail is based on two Java-related technologies called the InfoBus and the InfoCenter. To understand eSuite mail, we will first present a brief overview of these two underlying technologies. In this thesis, we are particularly focusing on the user interface of eSuite mail as we are building an agent that operates on eSuite mail UI. In Section 5.3, we will discuss the UI of eSuite mail in detail. Lastly, we will compare the eSuite mail with the ideal application mentioned in Section 4.7.

---

[1] Other applets in eSuite includes: calendar, address book, word processor, spreadsheet and presentation graphics.

[2] eSuite mail is said to be lightweight because it does not rely on user-interface code that is native to whatever operating system it is running on.

5.1. Applets in eSuite Mail

eSuite mail is composed of the several applets. Different applets operate within different Web pages. Communications among components on a Web page can be done through the Lotus InfoBus, a technology that will be discussed in the next chapter. Our work is involved with the following three applets that have graphical user interfaces: MailApplet, ReadMessageApplet, and ComposeMessageApplet.

The MailApplet is the main applet. This applet will be launched when eSuite mail gets started up. It is responsible for connecting to the mail IMAP/POP[3] server. It also manages the ReadMessageApplet and ComposeMessageApplet via local HTTP connection. It has a graphical user interface that displays the main mailbox view. There is only one instance of this applet per session.

When an e-mail message is selected by double clicking on it or by pressing the Open button on the ActionBar, the entire contents is requested from the IMAP/POP server. The ReadMessageApplet is responsible for displaying the InfoCenter ActionBar of the full message window. This full message window is a Web page generated at run time with an HTML frame to display the message body. There can be multiple instances of this applet per session.

The ComposeMessageApplet is responsible for providing the UI for composing, forwarding and replying an e-mail message. This applet resides in a Web page generated at run time. Part of this page is generated by reading another Web page on the server which contains the eSuite address book, local address source applet, and zero or more LDAPSource applets.

---

[3] IMAP (Internet Message Access Protocol) and POP (Post Office Protocol) are protocols designed to access mail stored on remote computers.

5.2. InfoBus

Internal communication as well as data sharing between applets are provided by the InfoBus. The Lotus InfoBus provides a standard communication mechanism for interconnecting Java beans or co-operating applets on a Web page to communicate data to one another. It defines a small number of interfaces between cooperating components, and specifies the protocol for use of those interfaces. The fundamental building block for data exchange is the "data item." InfoBus interfaces allow application designers to create "data flows" between cooperating components.

The protocols are based on the notion of an info bus. All components that implement the required interfaces can plug into the info bus. As a member of the bus, any component can exchange information with any other component in a structured way. Generally, the bus is asynchronous and is symmetric in the sense that no component may be considered the master of the bus. However, provisions are made in the protocol for a controlling component that can act as the bus master or arbitrator of bus conversations.

Components that make up the InfoBus application can be classified into three types:

- Data producers – components that generate and broadcast the data items onto the InfoBus
- Data consumers – components that are interested in receiving the data items
- Data controllers – an optional component that acts as the master of the bus and regulates the flow of data items between data producers and data consumers

5.3. User Interface

There are three applets in eSuite mail with graphical user interfaces: MailApplet, ReadMessageApplet and ComposeMessageApplet. Each user interface is composed of an ActionBar and a main window. The ActionBar is the menu bar at the bottom of the application. The window above the ActionBar is the main window. The UI of each applet is unique in its appearance and functionality. The main window of each applet has one or more appearances, which we call views. Appendix A shows pictures of all the views. The ActionBar also is context specific to the tasks that are to be performed. The following table lists the views available for the main window of each applet and the corresponding InfoCenter ActionBar options available.

Applet	Main Window	ActionBar
MailApplet	Main Mailbox/Preview View	New Message, Forward, Reply, Open, Delete, Move, Copy Text, Create, Help
ReadMessageApplet	Full Message View	New Message, Forward, Reply, Previous, Next, Delete, Move, Detach, Copy Text, Help
ComposeMessageApplet	New Message, Forward, & Reply View	Send, Draft, Cancel, Address, Attach, Cut, Copy, Paste, Properties, Bold, Italic, Help

Table 5-1: Features on the main window and ActionBar of each applet

The eSuite mail UI is a direct manipulation interface where the state of the application is graphically visible and modifiable by the user's interaction with the application UI. What makes eSuite mail a very complicated application is that the user interface is provided and controlled by two very different mechanisms. The InfoCenter

manages a rich collection of GUI components that provide the major functionality of eSuite mail through the ActionBar. The rest of the UI, that is, the main window, is controlled by a conglomeration of Java AWT components and Bongo widgets.

5.3.1 InfoCenter

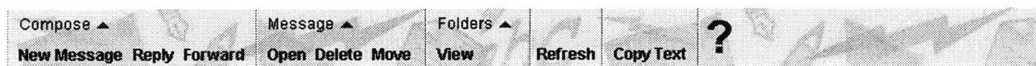


Figure 5-1: ActionBar

The InfoCenter provides the mechanism for adding a graphical user interface to eSuite mail. The InfoCenter user interface is comprised of an ActionBar, which contains a set of buttons and menus, and a set of widgets and property/command panels. The ActionBar is the menu-like bar at the bottom of the window.

The InfoCenter provides a substantial portion of the user interface functionality. eSuite applet actions most often are readily accessible with a single mouse click through the InfoCenter's ActionBar.

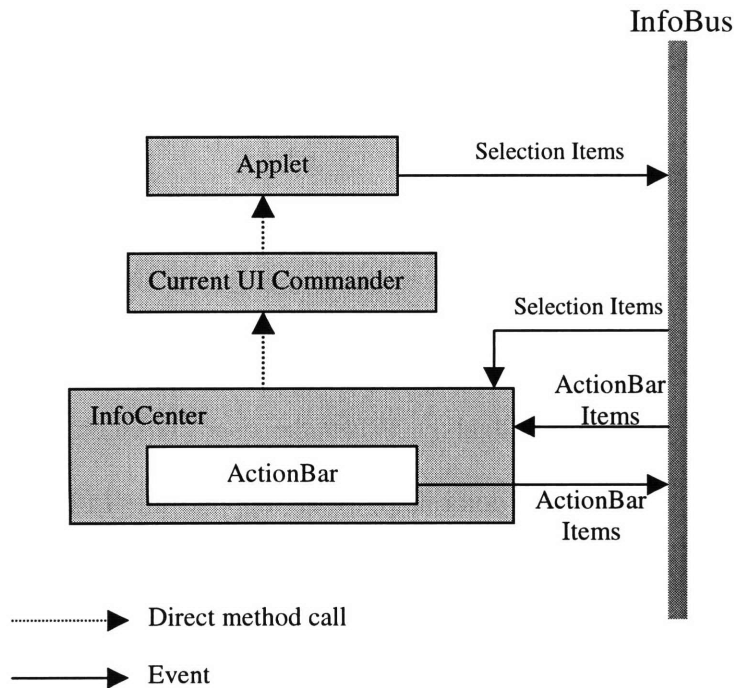


Figure 5-2: Internal data flow between applet and InfoCenter via InfoBus

The relationship between the InfoCenter and the applet is based on each applet communicating selection and user interface information to the InfoCenter, and the InfoCenter handling user manipulations by calling methods on specific interfaces on Applet objects. This relationship is adherent to the model-view-controller paradigm in which the applet is the model, the ActionBar is the view, and the InfoCenter is the controller. The InfoBus is used as the communication channel among the components. The basic choreography of this relationship is shown in Figure 5-3.

The ActionBar is context-sensitive to the state of the applet. When the state of the applet has changed, the applet notifies the ActionBar to update its appearance by announcing a selection via the InfoBus. The InfoCenter is a *data consumer* of the

InfoBus, and thus hears about the selection. The InfoCenter then queries the selection object for a user interface description. Based on the UI description, the InfoCenter displays an ActionBar.

When a user clicks on a button or menu item, the ActionBar, as a *data producer*, notifies the InfoCenter by communication via the InfoBus. If the action is of type PopUpMenu, QuickPick or Panel, InfoCenter will instruct the ActionBar to display additional UI (e.g. panels or popup widgets). When the user clicks on something that alters an applet property or sends a command to an applet, the InfoCenter calls a setProperty or doCommand method on a UI commander specified in the user interface description. This commander will subsequently make direct calls to the applet. If the applet determines that the current UI should change, it will announce a selection via the InfoBus and start the whole cycle over.

5.3.2 Main Window

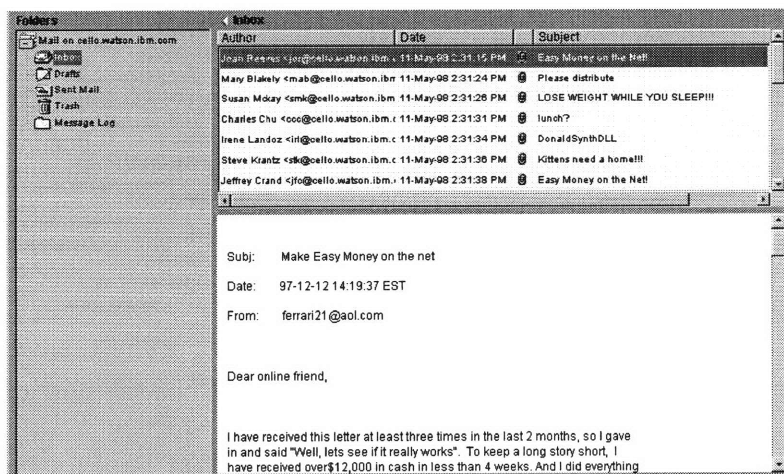


Figure 5-3: Main window

In Java, components are referred to as objects that have a graphical representation and that can interact with users. The UI of the main window is composed of a conglomeration of AWT components and Bongo widgets. AWT components are derived from the java.awt.Component class. Examples of AWT components include buttons, text fields, labels and scrollbars. Based on the Java 1.1 event model, components fire events that can be listened by other objects. To automatically receive the events, listeners need to register with a component by invoking one of the addXXListener methods depending on the type of events to be received.

A substantial amount of the UI components of the main window is Marimba's Bongo widgets. Marimba's Bongo is a visual interface builder for Java and is complementary to integrated development environments (IDEs). Bongo contains a rich set of predefined graphical widgets for visual control. User interfaces can be created simply by dragging and dropping these widgets. Behaviors can then be added to the interface by writing Java scripts. Bongo is built on the old Java 1.0 event model. Event handling in this model is based on inheritance. In order for a program to catch and process GUI events, it must subclass GUI components and override either action() or handleEvent() methods. Returning "true" from one of these overridden methods consumes the event so it is not processed further. Otherwise the event is propagated sequentially up the GUI hierarchy until either it is consumed or the root of the hierarchy is reached. The result of this model is that programs have essentially two choices for structuring their event-handling code:

- Each individual component can be subclassed to specifically handle its target events.

- All events for an entire hierarchy (or subset thereof) can be handled by a particular container.

5.4. Comparison with the Ideal Application

The ideal application mentioned in Section 4.8 is based on three good design criteria that facilitate connection with Collagen:

1. JavaBeans
2. Java 1.1 event model
3. Model-view-controller paradigm

In contrast to the ideal application mentioned, eSuite mail is not a Java bean. The user interface is complicated; the ActionBar and the main window is each provided and controlled by a different mechanism. The InfoCenter provides and controls the ActionBar. It possesses some good design criteria for easy connection with Collagen. The InfoCenter follows the MVC paradigm and the Java 1.1 event model.

AWT components and Bongo widgets provide and control the main window. It is not built upon MVC paradigm. Although the AWT components support the Java 1.1 event model that separates out the view from the control elements, there is no explicit model that separates the application code from the user interface. The Bongo widgets are even more hopeless as they are built on the old Java 1.0 event model. In Java 1.0, only a component derived from the Component class could handle events, and it had to be either the component in which the event occurred or a component above it in the component containment hierarchy. This makes it impossible for other objects to listen to events invoked by Bongo widgets without modifying the application code.

Fortunately, the major functionality of eSuite mail is concentrated in the InfoCenter. It is easy to observe actions and manipulate the UI of the InfoCenter, as it is built upon some good design criteria. In the next chapter, we will discuss in detail the actual design and implementation for connecting Collagen with eSuite mail to build a collaborative interface agent.

Chapter 6

Connecting Collagen with eSuite Mail

Referring to Figure 4-1, what we mean by connecting Collagen with eSuite mail is enabling the discourse bean to be notified of the user's manipulations on the application, as well as enabling the agent to manipulate the application as if it were a user. Technically, this means eSuite mail must be able to export events to notify Collagen of user's manipulations. Also, there must be publicly accessible methods and properties that allow Collagen to modify the state of eSuite mail. This way, the agent can manipulate on the application as if it is the user.

JavaBeans technology allows the property values of components to be changed through some type of visual interface, and their methods and events to be exposed so that the component can be manipulated by external components. eSuite mail is in the process of being converted to a Java bean-based applet in the eSuite DevPack[4] project, but the bean version is not available for our work. Thus, we cannot enjoy the ease of implementing a collaborative interface agent with eSuite mail that a fully "beanified" application can provide. This complicates our implementation effort, as there is no explicit mechanism for exporting events and exposing properties and methods to interested applications outside. In order to connect Collagen with eSuite mail, we first

[4] eSuite DevPack provides a comprehensive set of pre-built, pre-tested, reusable, cross-platform, integrated JavaBeans-based applets

need to "beanify" eSuite mail, that is to re-engineer eSuite mail to a reusable application with fully exposed properties and methods and exportable events.

In this chapter, we will describe the "beanification" process in detail. After re-engineering eSuite mail into a bean-like application, it is ready to connect with Collagen. The adapter is the central piece to make the connection. Section 6.2 will discuss the implementation of an adapter. The motive in building a collaborative interface agent for an application is to provide intelligent assistance to user. Section 6.3 will discuss our effort in building a more intelligent agent with application-specific knowledge to operate in the e-mail domain.

6.1. "Beanification"

In order to re-engineer eSuite mail into a bean, we need to find ways to expose the properties and methods, as well as to export events. The immediate design criterion is minimizing modification to the eSuite mail code.

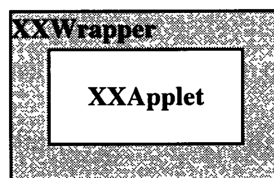


Figure 6-1: Applet and wrapper

Our solution to the problem is, instead of making changes at the low levels of the application, we wrap a new layer of code around the application. We call this new layer

of code a “wrapper.” The wrapper registers itself as listeners to events triggered by changes on different UI components of eSuite mail. In addition, the wrapper provides the mechanism to export these events. It acts as an event source and delivers events to any outside application that has registered as listeners. The wrapper contains references to various objects of the application and provides new public methods that are callable by outside applications to change the property values of these objects. Hence, outside applications can receive events and access the wrapper’s exposed methods to manipulate the state of the application. In other words, the wrapper is responsible for “beanifying” the application without significant changes to the code.

Referring to Section 5.1, there are three applets in eSuite mail with graphical user interface: MailApplet, ReadMessageApplet and ComposeMessageApplet. Each applet has a different UI serving different functions. Hence, we need a different wrapper for each applet. During run time, there is always a single instance of MailApplet, but there may exist multiple instances of ReadMessageApplet and ComposeMessageApplet as the user may be reading or composing multiple e-mails at any time. Each instance of an applet can be treated as a separate application to Collagen, and this can be very confusing as Collagen needs to dynamically keep track of which instance of the wrappers it is interacting with. Since there is only one instance of the MailWrapper, our design calls for Collagen to only interact with the MailApplet through the MailWrapper. Just like the MailApplet manages the ReadMessageApplet and the ComposeMessageApplet, the MailWrapper manages the other wrappers through simple method calls.

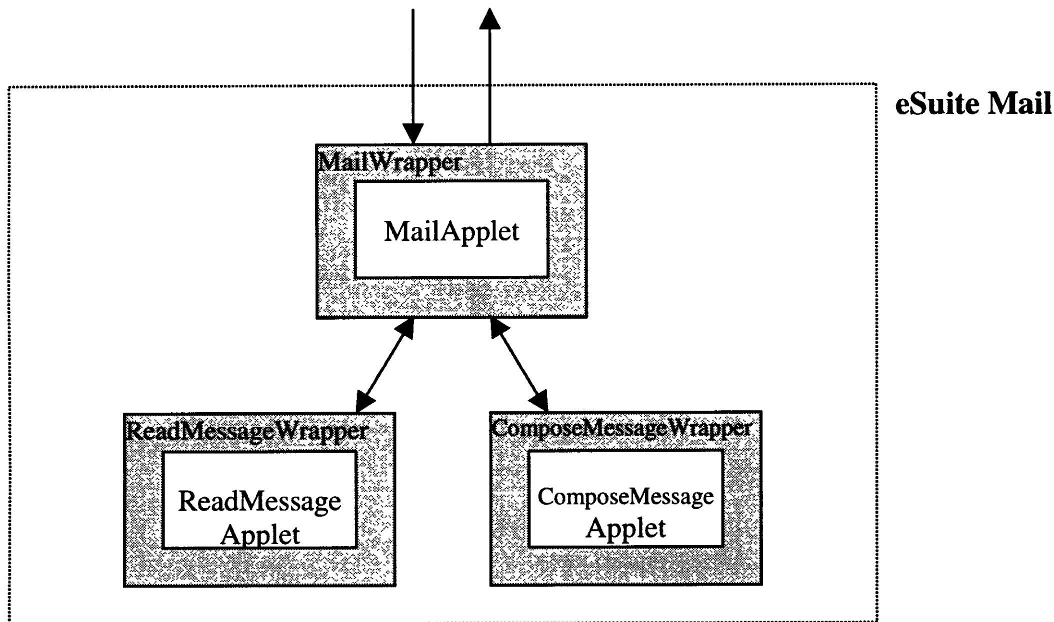


Figure 6-2: Interactions among wrappers

The agent and the discourse engine are the two outside components that want to receive events triggered by changes on the user interface of eSuite mail and to manipulate eSuite mail. As discussed in Section 6.3, the eSuite mail UI is provided and controlled by the InfoCenter and a conglomeration of AWT components and Bongo widgets in the main window. As the event handling mechanism of different UI components is very different from one another, each different UI component requires a different mechanism to export its events.

6.1.1. InfoCenter

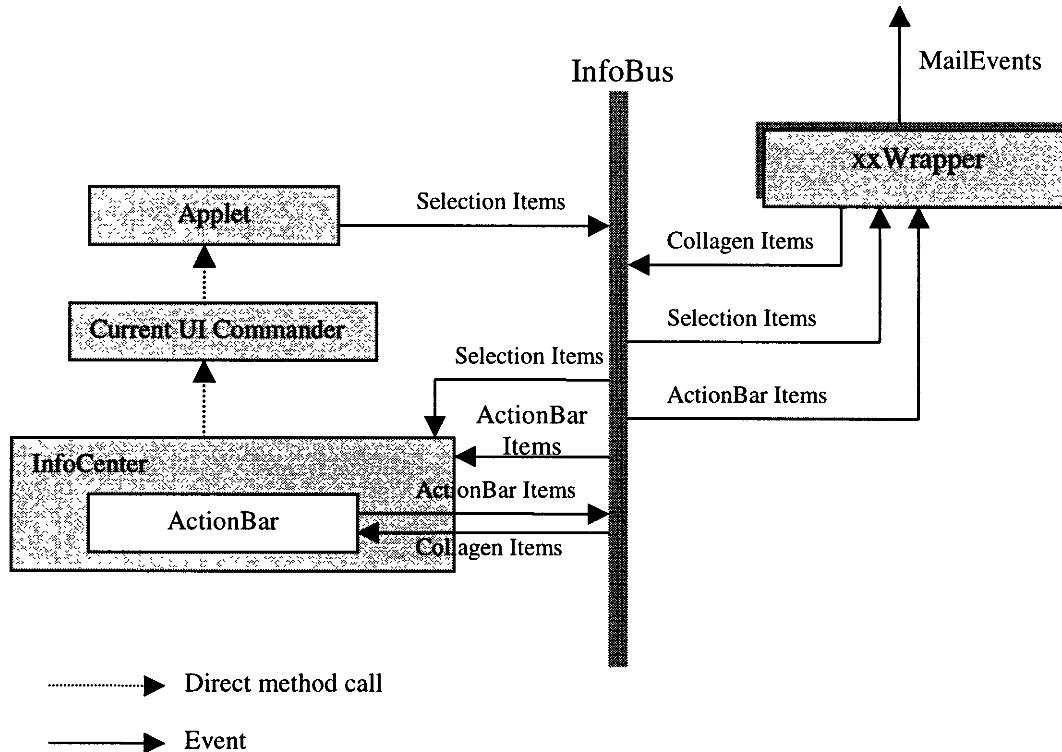


Figure 6-3: Internal data flow among applet, InfoCenter and wrappers via the InfoBus

6.1.1.1. User's Manipulations

The InfoBus is used as the communication channel to control and manage the UI provided by the InfoCenter. When an action occurs on the ActionBar, an ActionBar item is broadcast onto the InfoBus. Each applet in eSuite mail operates on a different web page; therefore, each applet communicates via a different InfoBus. By implementing the data consumer interface, wrappers can easily plug into their corresponding InfoBus and

receive ActionBar items. Hence, each wrapper can observe all user manipulations on its ActionBar such as menu selections or button clicks.

When an ActionBar item arrives, the wrapper translates it to a MailEvent for export. This way, events that are exported will be uniform in style, facilitating introspection. Remember our design requires that only the MailWrapper to interact with outside applications. Once a MailEvent is generated, a method on MailWrapper will be invoked to export the MailEvent to registered listeners.

6.1.1.2. Agent's Manipulations

Although the current implementation allows easy export of events of the ActionBar, it does not provide any mechanism for external applications to manipulate the ActionBar. The solution is to modify the ActionBar class so that it becomes a data consumer as well. Figure 6-3 shows the modification made. Wrappers, by implementing the data producer interface, can easily broadcast Collagen items onto the InfoBus. The ActionBar receives Collagen items from the InfoBus and behaves as if the direct manipulations were performed by the user but without actually changing the UI. The InfoCenter will either instruct the ActionBar to display additional UI, or instruct the applet to change state. This way, the agent can successfully *spoof* an action on InfoCenter.

6.1.2. Main Window

6.1.2.1. User's Manipulation

The UI of the main window is made up of AWT components and Bongo widgets. AWT components in eSuite mail support the Java 1.1 event model. The Java 1.1 event

model allows one component to register its interest in receiving events generated by another. When an event occurs, the interested component, or event listener, will be notified by having one of its methods invoked. This model readily supports the export of events by simply having external components register as event listeners.

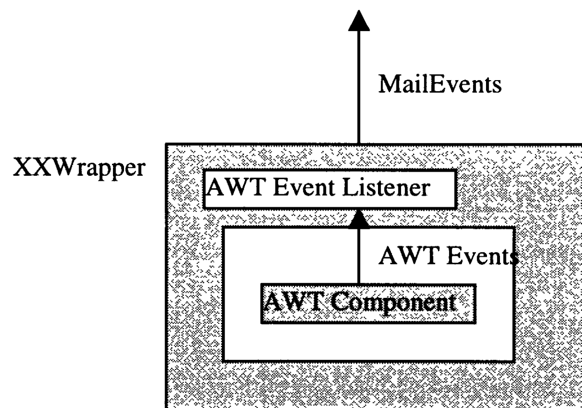


Figure 6-4: AWT component

In our design, the wrapper listens to all interesting events generated by the AWT components of the main window UI. The challenge is to get a reference to the various AWT components that fire interesting events. Every application is by necessity subclassed from java.awt.Container class. Containers are components that can contain other components. We can take advantage of the method getComponent in the Container class to find all of the components within a given container recursively. Precisely, each wrapper should get a reference to the top-level container, and recursively walk the hierarchy to find all of the herein-contained components. In this way, the wrapper should be able to register as event listeners and receive events from the AWT components. When an event arrives, the wrapper translates the event to a MailEvent for export.

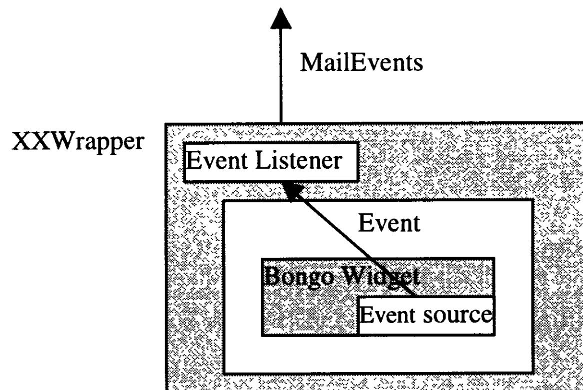


Figure 6-5: Bongo widget

Most of the components that make up the main window UI are Bongo widgets. Since they are built on the Java 1.0 event model, it is impossible for the wrapper to handle events that are generated unless we modify the source codes. We can modify the code in several different ways to export the events to the wrapper. We can have the component's handleEvent() method return false and pass the event up the containment hierarchy. We can convert the Java 1.0 part of the program to the 1.1 AWT API. This usually requires two steps:

1. Replace deprecated methods with their 1.1 equivalents.
2. Convert the program's event-handling code to use the new AWT event system.

These two approaches require a substantial amount of modification. Instead, we decided to add the Java 1.1 event handling mechanism to the component in which the event occurred. The component becomes an event source that allows other components to register as listeners. The component's handleEvent() not only handles events when they

occur, but also notifies listeners by invoking methods on them. This approach leaves the Java 1.0 event-handling codes intact. The wrapper can therefore register as listener of Bongo widgets and receive all interesting events.

6.1.2.2. Agent's Manipulations

The AWT components delegate a great deal of functionality to their peers. A peer is the platform-specific implementation of the corresponding AWT component. In other words, AWT components borrow native code from the platforms on which they run to do the real work of displaying and managing the components' behaviors. The advantage of this is that components have a familiar look and feel. For example, when you create such a program and run it under Windows, it has the appearance and behavior of a program written specifically for Windows. When you run the same program on a UNIX workstation, it runs just like any program written for UNIX. The disadvantage is that the AWT components do not respond to AWT events that might be used to *spoof* an action on the component. Hence, there is no easy way that allows the agent to manipulate the AWT components. The same problem applies to Bongo widgets.

We tackle this problem in a brute-force manner. The wrapper has references to all the AWT components and Bongo widgets. Our solution is to create a new public method for each manipulation that the agent wants to perform in the wrapper class. The new method will change the state of the application and instruct the UI to update its appearance based on the post-state of the manipulation if it was performed by the user.

6.2. Mail Adapter

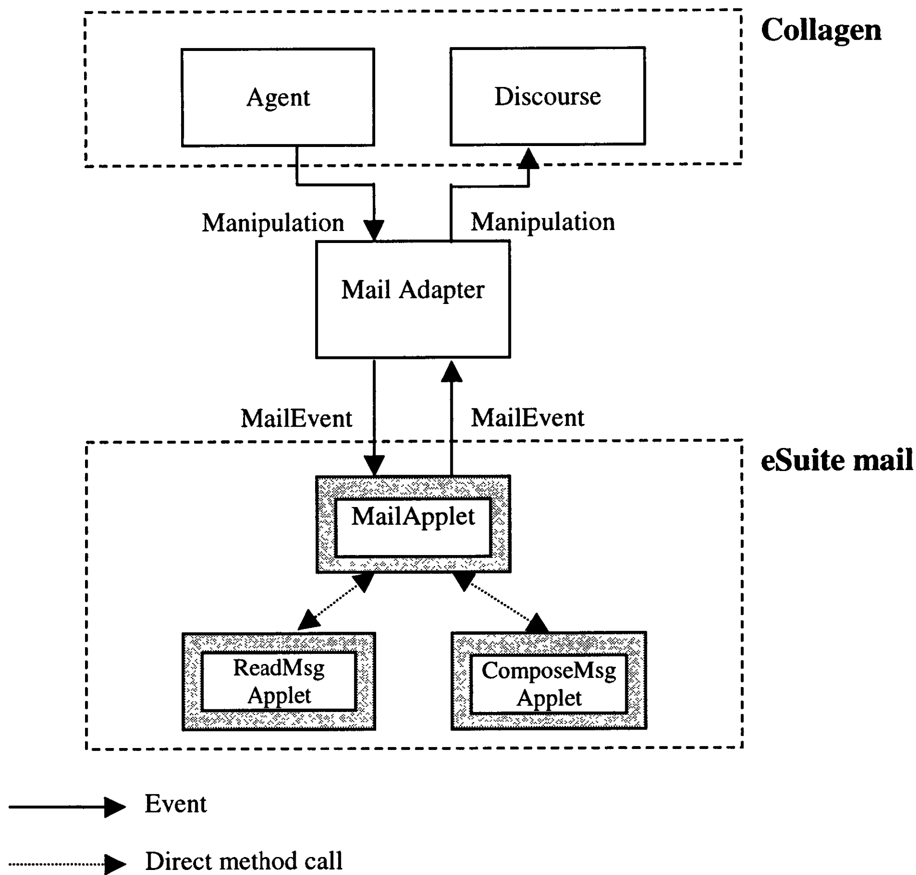


Figure 6-6: Event flow

An event adapter is the object between the event source and the event listener that adapts the source to the specific needs of the listener. As seen in Figure 6-6, the adapter for eSuite mail domain, or MailAdapter for short, regulates event passing among eSuite mail, Collagen and the agent. Events go in two different directions. For the upward direction, the MailAdapter regulates the events fired by eSuite mail. The MailAdapter is actually an event listener of MailEvent and an event source of Manipulation. When the

MailAdapter receives a MailEvent from eSuite mail, it translates the MailEvent into an event object subclassed from Manipulation. Manipulation is the customized event class used by Collagen. For example, when the user clicks on the Reply button on the ActionBar, the MailWrapper fires a MailEvent of type Reply_To_Sender to the MailAdapter. Given this MailEvent, the MailAdapter will create an instance of ReplyToSender subclassed from Manipulation, and fire it.

For the downward direction, the MailAdapter regulates the events fired by the agent. The MailAdapter is actually an event listener of Manipulation, and an event source of MailEvent. When the MailAdapter receives a Manipulation from the agent, it translates the Manipulation into a MailEvent and fires it. For example, when the agent wants to close the message that is currently displayed by the ReadMessageApplet, it fires a CloseMessage manipulation. The adapter translates the manipulation into a MailEvent. The MailWrapper, on receiving the MailEvent, calls the CloseMessage method on ReadMessageWrapper that instructs ReadMessageApplet to close the current message.

Chapter 7

Knowledge about eSuite Mail

7.1. Knowledge Engineering

In order to build an agent that can provide intelligent assistance, it is necessary to endow the agent with knowledge about the domain in which it is operating. Knowledge acquisition in building a collaborative interface agent takes the knowledge engineering approach. This requires a knowledge engineer to accrue a large amount of knowledge about the domain. The knowledge engineer then devises a set of goals and recipes for the domain.

Since we rely heavily on e-mail as a communication tool and use the e-mail client extensively, we can be considered as experts of the e-mail domain. However, relying only on our own knowledge is not enough. We are interested to know how others behave in the email domain, as different people may act differently. This may add novelty and variety to our knowledge. Researchers at Lotus have conducted experiments to observe human's behavior in the e-mail domain. Spoken data was collected from a set of human user and human wizard as computer dialogs. All dialogs were then transcribed and corrected. From the data, a Lotus researcher inferred a formal model of the collaborative tasks being performed by the agent and user in the e-mail domain. The model is complete enough to support collaboration with the user.

7.2. Recipe Library for eSuite Mail Domain

The recipe library is the knowledge engine of the collaborative interface agent that represents the task model. The recipe library for eSuite mail contains 24 recipes defined in terms of 16 different goals or action types. It is fairly small in size. If we were to cover the whole e-mail domain, the recipe library would be bigger. However, the library covers most of the typical user activities.

In Collagen, knowledge is represented by Sidner's artificial discourse language (1994). The most fundamental concept in Collagen's internal representation is an act. Acts can be divided into two kinds: manipulation and communication (see Section 4.2.1, 4.2.2). For manipulation acts in the eSuite mail domain, the simplest kind is *basic manipulation act* that corresponds one-to-one with single button event in the interface. Examples of basic manipulation act include ReplyToSender, Forward, and Send.

There are more complicated manipulation acts that define different goals. As described in Section 4.4, a recipe is a resource used to derive a sequence of act types (steps) with bindings and constraints between them that achieve a certain goal. There are 5 top-level manipulation act types: DoMyEmail, ManageEmailDocs, ReadAMessage, SearchEmail, and SendEmail. Each top-level act defines a top-level goal. These top-level acts are nested in structure since goals are made up of subgoals. Moreover, there may be more than one recipe for each top-level act as there are often more than one way to achieve a given goal. For example, there are 5 recipes for ManageMailDocs: ManageEmailByPutInFolder, ManageEmailByArchiveIt, ManageEmailByDeleteIt, ManageEmailByFileIt, ManageEmailByPrintIt. Recipes are written in a Java-like

declarative language that is preprocessed into Java code. Below shows the DoMyEmail recipe:

```
public recipe DoMyEmailRecipe achieves DoMyEmail {
  step ReadAMessage read;
  optional Respond respond;
  optional ManageMessage manage;
  bindings {
    achieves.message = read.message;
  }
  constraints {
    read.message = respond.incoming;
    read.message = manage.message;
    read precedes respond;
    respond precedes manage;
  }
```

Figure 7-1: Sample recipe

In the Collagen framework, there are four main kinds of communication acts: Propose.Should (propose an act), Propose.Identity (propose the identity of parameters), Propose.Who (propose who should perform an act), Propose.RecipeFor (propose an recipe for a task). Besides these four main kinds of communication, there are "ok" and "no" that are used by the user to respond to agent's proposal. Communication is limited in the current version of Java Collagen as only Propose.Should, "ok" and "no" are available for our work.

7.3. Knowledge Engineering

Through collaboration, the agent essentially shares a common goal with the user to complete a task. Hence, the recipe library should contain a complete set of goals and recipes that covers as much of the domain as possible. This is a drawback of this

approach, as it requires the knowledge engineer to collect a large amount of knowledge about the domain. Personalized assistance can be made possible if the agent knows specific knowledge about the habits of individual users. This is important especially in highly personalized domains such as e-mail, where users often behave very differently from one another. However, this complicates the task of the knowledge engineer, as it is often hard to collect knowledge about the user without generating overhead to the user. In order to preserve the value of the agent to the user, the overhead generated by knowledge collection must be minimized. We have two approaches to the problem. Both approaches essentially compile a user profile, from which the agent can learn typical things about the user.

In the first approach, knowledge about the user is collected by having new users fill out a form. For example, the form can ask users to specify:

1. whether by default the user would like to get copies of his e-mail or want to be asked before sending an e-mail,
2. whether by the default the user would like to include its contact information appended at the end of an e-mail,
3. whether the user would like to be reminded of unread e-mails and who they are from,
4. e-mail addresses of people whom a user considers to be significant. The user can be reminded in the future if there is any new incoming e-mail or unread e-mail from these "significant" people.

Implementation of this approach is simple, as knowledge about the user is explicitly collected within the same block of time. However, this is a blunt way to collect

knowledge as it creates an extra chore to the user, who may find this a nuisance. This may depreciate the value of the agent to the user.

The second approach is a remedy to the previous approach. Knowledge collection is done at the time the user is doing the task. Instead of filling in the form at one time, the user fills in the form on the fly. For example, the first time the user proposes working on an e-mail, the agent could ask if there are any people who the user thinks are critical to get e-mail from. Another example is that the first time the user fills in a message, the agent could ask if the user wants to append his contact information at the end of the e-mail. This approach is feasible because Collagen has contextual information so that the form-filling process can take place in the middle of a conversation. This is a more natural way to collect knowledge from the user. We would like to pursue this approach in the future. For the thesis, we assume the agent already has a user profile built in.

We must bear in mind the two approaches only capture the big picture about the user, meaning the typical things about the user, but they are not solutions to getting the agent to know the user in all subtle details.

7.4. Learning

An alternative approach for knowledge acquisition is learning. In this approach, the agent gains knowledge by observing user's actions over a period of time and learns about the recurrent patterns. This offers a high level of personalization, as the knowledge learned by the agent is different from person to person. There are tradeoffs to this flexible approach. Most learning agents have a slow learning curve and they require a

sufficient number of examples before they can make accurate predictions. Moreover, agent has no knowledge to deal with completely new situations.

Chapter 8

Domain-Specific Agent for eSuite Mail

Referring to Figure 4-1, most of the components for building the collaborative interface agent for eSuite mail are already discussed in previous chapters: Collagen in Chapter 3, eSuite mail and the MailAdapter in Chapter 5 and 6, the recipe library in Chapter 7. The only component left for discussion is the domain-specific agent, or MailAgent for short. It collaborates with the user to complete different tasks in the eSuite mail domain by communicating with the user and manipulating the application. Technically, it decides which possible, future communication or manipulation acts in the current agenda to execute in order to advance the current problem-solving process.

The MailAgent is built upon the default agent provided by Collagen. In this chapter, we will first summarize the agent's default behavior that can be applied to any domain. Then, we will discuss the MailAgent that is especially designed for clever decision-making in the eSuite mail domain. The rationale behind building the MailAgent rather than relying on the default agent alone is that the MailAgent is more clever and useful as it holds specific knowledge about the application and user.

8.1. Default Agent

The behavior of the default agent is summarized as follow:

- It has a selection strategy that simply chooses to perform the highest priority act in the current agenda. Acts are pre-prioritized in the recipes.
- It assumes all the user's communication turns are one utterance long and immediately responds.
- It performs primitive authorized acts without asking for user's permission when it is its turn, but will first ask for permission through proposal (Propose.Should) for other acts. By default, all communications are authorized acts and all manipulations are unauthorized. The agent developer can override any library definition to compile a new list of authorized acts.
- It asks for permission through proposal to perform optional steps in the recipes.
- It interprets "ok" as the user ending his turn.
- It responds to "no" by popping (stopping) the current segment if any.
- It asks for the missing parameters if the current act is not fully instantiated.

8.2. eSuite Mail Agent

The default agent is only helpful in providing the general functioning of an agent. It is impossible to build a default agent that could make clever decision at all circumstances for every domain. Sometimes, the user may even find the default behavior awkward. In light of this, decision-making in different domains is based on different if-then rules for prioritizing and selecting an act from the current agenda to execute. This ensures that the agent could make the best decision in executing the most suitable communication or manipulation in the problem-solving process.

We have generally found that the eSuite mail domain is very simple. This is due to several reasons. First, the interface of eSuite mail is user-friendly and most functionality is available by a single mouse-click on the ActionBar. Second, e-mail client is such a popular application nowadays that many people use it on a daily basis. Since there is nothing tricky in the domain, most people know the application so well that they can be considered as experts. What makes the domain interesting is that it is highly personal. The agent will be most useful in providing assistance that is customized for a specific user's needs. Hence, the domain of interest should be both application and user specific. The following summarizes the domain-specific behavior of MailAgent that we have implemented:

- If an act is both optional and unauthorized, the default agent will uninterestingly ask twice for permission to perform the action through making a proposal. This makes collaboration awkward. The MailAgent suppresses the second proposal.
- The MailAgent knows the user. In the next chapter, we will demonstrate the capability of the MailAgent to perform personal assistance. For example, assume the user has the habit of including his e-mail address in the cc: list. Since the fill-in-CC act has a missing parameter, which is the text to be filled in, the default agent is unable to perform the act even when it is its turn. With the user profile, the MailAgent can automatically fill in the missing parameter and perform the act.
- For another example, assume the user never deletes or moves the e-mail (i.e. ManageMessages step from DoMyEmail recipe) after reading and responding an e-mail message. The MailAgent knows that this optional ManageMessage step is not desired by the user, so it can skip this optional step and just propose the next act.

- The default selection strategy chooses to perform the highest priority acts in the current agenda. Acts can be considered as pre-prioritized in the library. The agent can override the default choice of action by re-prioritizing acts according to the user's preference. For example, assume the user has the habit of replying to the sender with history appended. In the recipe library, the manipulation act reply-to-sender has a higher priority than reply-to-sender-with-history. Knowing the user's habit, the agent will propose reply-to-sender-with-history first.
- The agent can query the application and communicate information implicit in the application state the user. Since the AgentHome becomes part of the user interface, the ability of the agent to communicate additional information about the application state to the user actually enrich the functionality of the application. For example, when the user performs the open-message act to read the message, the agent will check against the application to find out the number of unread messages in the Inbox and then communicate the result to the user. eSuite mail does not have the capability to remind users of unread messages, and the MailAgent, knowing the needs of the user, provides this extra functionality to the user.

We would like to include the following behavior in the MailAgent. However, due to limitations of the application, it cannot be implemented.

- We would like to experiment with a small number of rules for choosing other actions than the default collection. If the user proposes the agent to perform an act that the agent cannot do, the agent will refuse and explain its refusal (e.g. "No, I cannot do that action"). One such act is accessing mail when the server is down. Since it is not

possible for the eSuite mail application to check server readiness, this rule cannot be implemented.

Chapter 9

Sample Scenario

In this chapter, we will use a sample scenario to demonstrate the ability of a collaborative interface agent to provide intelligent assistance to the user.

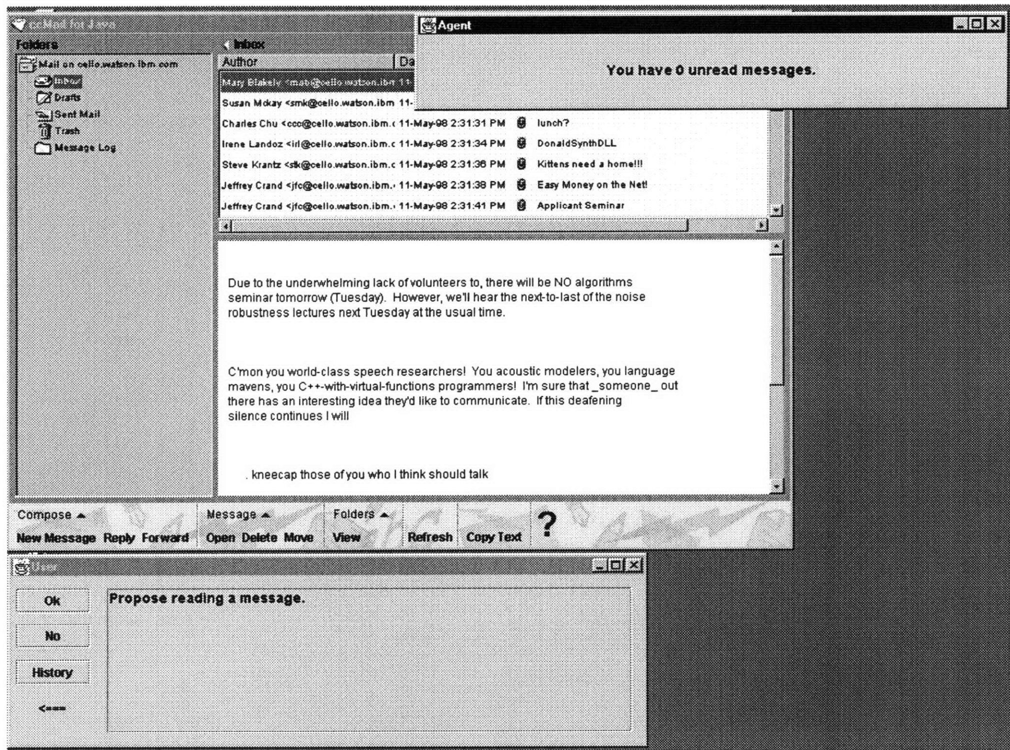


Figure 9-1: eSuite mail with agent

Figure 9-1 shows the user interface of eSuite mail with the user home window at the lower left-hand corner and the agent home window at the upper right-hand corner. Communication from the agent to the user is achieved by printing English text in the agent's home window. Communication from the user to the agent is achieved by the user selecting from the communication menu, which can be expanded by clicking the arrow in the user home window.

9.1. The "Working on e-mail" Scenario

In this scenario, the user is going to perform the usual routine of reading, replying and managing an e-mail. We give this routine a general name: "working on e-mail." The DoMyEmail recipe (refer to Figure 7-1) models this routine.

Even in performing simple task like this, we can often get distracted. In this simple scenario, we will demonstrate the ability of the user to make intelligent decisions and help the user to stay focused on what needs to be decided next to push the task forward. We will also show the agent performing delegated tasks. As mentioned earlier, the agent has both application and user specific knowledge. For this scenario, the agent knows the user's e-mail address and is aware that the user likes to receives a copy of the e-mail that she has sent.

The interaction begins with the user choosing one of the default top-level goals displayed on the current communication menu in the user home window. Notice that the user could just start working on her own, but instead she proposes working on an e-mail together with the agent.

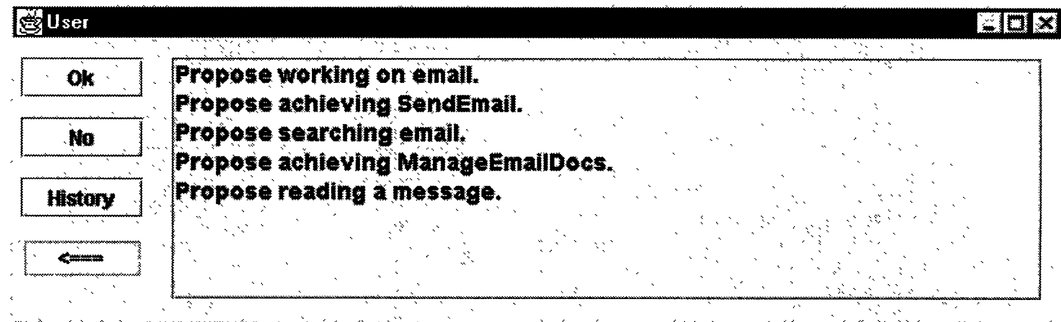


Figure 9-2: A set of top-level goals from the communication menu in the user home window

At this time, the agent reminds the user the number of unread messages in the Inbox.

You have 0 unread messages

This is an example that demonstrates the agent's ability to communicate additional information about the application to the user. The user now clicks "Ok" in her home window to signal the end of the current segment. The agent then takes the initiative and proposes:

Propose reading a message

This is an example of intelligent assistance in which the agent helps the user focus on what needs to be decided next in order to push the task forward. The user is free either to respond to the agent's proposal or to ignore it and proceed on her own. If she chooses to respond, the following options will be presented in her communication menu:

Propose selecting a message
Propose examining a message

Instead, the user clicks on the mailbox view to select a message from Steve Krantz. At this time, the agent "observes" the selection via the wiring provided by the adapter (refer

to Section 6.2). The user then clicks "Ok", signaling the agent to take control. The agent immediately proposes:

Propose examining message from Steve Krantz <stk@cello.watson.ibm.com> about Kittens need a home!!!

Again, the user is free either to respond the agent's proposal or to ignore it and proceed on her own. This time, the user clicks "Ok", signaling the agent to proceed with its proposal. The act with highest priority on the agenda is the manipulation for opening the message from Steve Krantz. Since this is an authorized act, the agent immediately opens the message for the user. This starts up the ReadMessageApplet and the selected message is displayed on the full message view. This is an example of the agent's ability to perform a delegated task in the interface as if it is the user. The user again clicks "Ok" to signal the agent to take control. The agent immediately reacts:

Propose using message from Steve Krantz <stk@cello.watson.ibm.com> about Kittens need a home!!!

The user only wants to read the message and has no plan to use it in any other ways such as searching for key words and visiting a URL. The user rejects the proposal by clicking "No" in her home window. Collagen reacts to this response by popping (stopping) the current segment. The user immediately proposes to have the agent close the message:

Close message from Steve Krantz <stk@cello.watson.ibm.com> about Kittens need a home!!!

The agent then closes the message. This completes the ReadingAMessage step from the DoMyEmail recipe.

Next, the user decides to let the agent lead the collaboration. She clicks "Ok" to stimulate the agent's response. The agent takes the lead:

Propose responding to message from Steve Krantz <stk@cello.watson.ibm.com> about Kittens need a home!!!

The user proceeds with the current task by clicking on the Reply button. This starts up the ComposeMessageApplet with the main window showing the reply view. The user then asks the agent fill in the message for her by selecting the following from the communication menu:

Fill in this message

Now the agent decides what its next best action is given the current context. The following acts are on the agenda:

Fill in the body of this message.
"Propose filling in the body of this message."
Fill in the subject of this message.
"Propose filling in the subject of this message."
Fill in the recipients of this message.
"Propose filling in the recipients of this message."
Fill in the CC field of this message.
"Propose filling in the CC field of this message."
Fill in the BCC field of this message.
"Propose filling in the BCC field of this message."
"Propose searching this message."
"Propose finishing this message."
Cancel this message.
"Propose canceling this message."
Send this message.
"Propose sending this message."
Save draft of this message.
"Propose saving draft of this message."

Figure 9-3: Agenda after user proposes filling in message

Note that each act in the agenda is either a communication, represented by an English gloss in quotes "...", or the description of an application-level manipulation. By default fill-in-body has the highest priority and is the agent's default choice of action. However, the agent, knowing the user likes to receive a copy of the e-mail she has sent out, decides to execute fill-in-CC act. Notice that the agent automatically fills in the text parameter of

the fill-in-CC act with user's e-mail address (adac@mit.edu). This is an example of intelligent assistance in which the agent re-prioritizes the agenda and chooses the best act to perform given the current context. The agent immediately fills in the cc: list with adac@mit.edu.

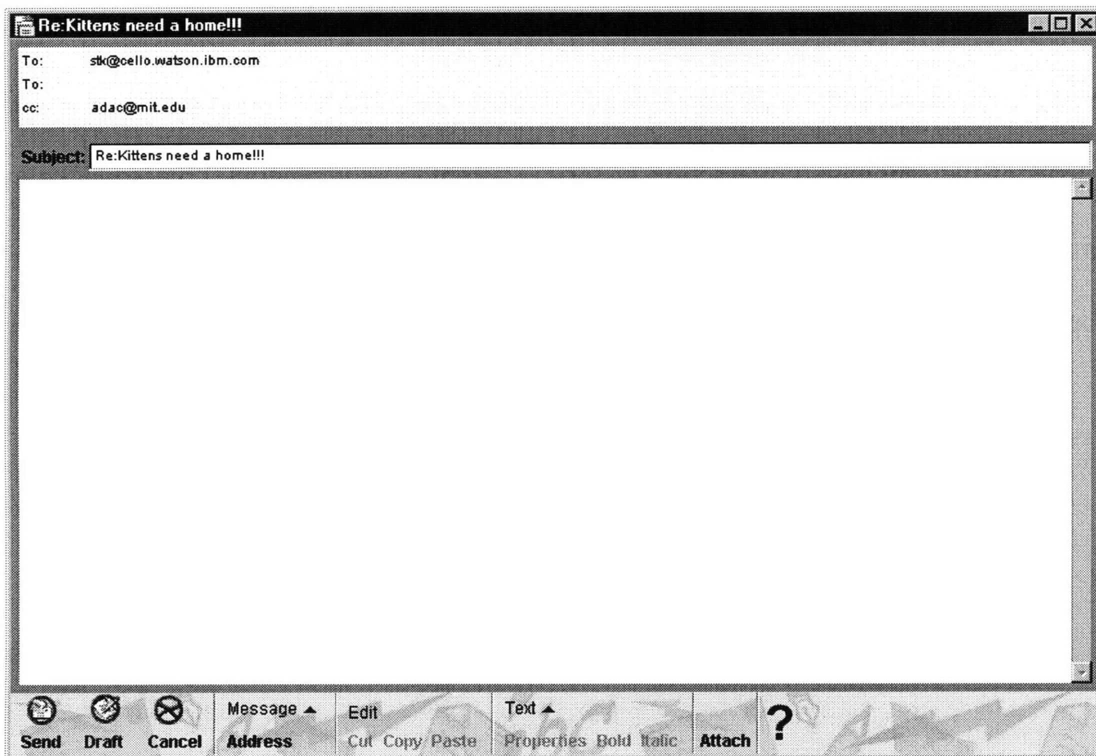


Figure 9-4: agent fills in cc: field

At the end, the user decides not to send the e-mail and clicks the Cancel button on the ActionBar. This completes the Respond step from the DoMyEmail recipe.

The user clicks "Ok", and the agent immediately proposes the next optional step in the recipe:

Propose managing message from Steve Krantz <stk@cello.watson.ibm.com> about Kittens need a home!!!

The user decides to delete the e-mail and clicks the Delete button on the ActionBar. Because the agent knows that deleting is one means of managing a message, it recognizes the manage-message step is done and that the overall DoMyEmail recipe is completed. This concludes the scenario.

Appendix B shows the segmented interaction history of this scenario. Clicking the History button on the user home window causes the History window to appear. The contents of the segmented interaction history is automatically generated by Collagen as a printout of data structures built up during the scenario by the algorithms described in Chapter 3. Refer to Appendix B for details about the segmented interaction history.

Chapter 10

Conclusion

With increasing complexity of user interfaces of commercial application, there is a growing need of clever interface agents to assist users in their problem solving process. In this thesis, we have proven the feasibility of using Collagen to implement a collaborative interface agent for Lotus eSuite mail. We have also demonstrated the agent's ability to provide intelligent assistance through some sample scenarios. In this chapter, we will conclude the thesis with a summary of work, a description of the lessons learned from this project, and ideas for future work.

10.1. Summary of Work

Through the implementation of a collaborative interface agent for eSuite mail, we have identified a set of tools and techniques necessary to build such an agent that operates on a commercial application. In our work, Collagen provides the standard mechanism for collaborative discourse between the agent and the user. The technical contribution of our work is to connect Collagen with the application by enabling the discourse bean to be notified of the user's manipulations on the application, as well as enabling the agent to manipulate the application. This requires the application to behave like a Java bean, with exportable events and publicly accessible methods and properties that allow external control of the application state. If the application is not a Java bean as in the case of

eSuite mail, it is necessary to "beannify" the application through building a wrapper outside the application. Because the interface of eSuite mail relies three different event handling mechanisms: InfoBus and InfoCenter, the Java inheritance event model, and Java delegation event model, we have demonstrated a fairly complete set of techniques required for reengineering an application into a Java bean. After the application is "beannified," we have explained how the adapter actually provides the connection between Collagen and the application.

Besides correct wiring, the agent should be endowed with knowledge about the domain it is operating on in order to support collaboration between the user and agent. We have described our effort in collecting knowledge to build a recipe library that represents the formal model of the collaborative tasks being performed by the agent and user in the eSuite mail domain. The value of an agent is its ability to assist user in problem solving. The actual decision making of what manipulation or communication to perform in different domains is provided by an intelligent agent with application and user specific knowledge and rules. We have demonstrated the ability of the collaborative interface agent to provide intelligent assistance to the user in eSuite mail domain through a sample scenario. The value of the agent lies in its clever decision-making that is customized to the current context, and its ability to help the user focus on what needs to be decided next in order to push the task forward.

10.2. Lessons Learned

Several lessons are learned during the development of the collaborative interface agent for eSuite mail.

1. An agent developer should be aware of some subtle modeling issues. The set of events that Collagen needs to know about and import should be a subset of events that the application would like to export. Also, the developer should decide which detailed events should be imported to Collagen. Listening to too many events will create an overhead and slow the application.
2. There are three design criteria that promote easy connection with Collagen: (1) model-view-controller paradigm, (2) Java 1.1 delegation event model, (3) JavaBeans technology. These good design criteria should conjunct with those that promote reusability of application. With the advent of Java and JavaBeans, the idea of reusable components extends to the application level.
3. The sample scenario suggests that using Collagen to apply human collaborative discourse principles to software agents can lead to agents that are natural and easy to collaborate with. In a highly personal domain like e-mail, it is necessary for the agent to have user-specific knowledge so that it can make clever decision that is suitable for user's need. The agent is also valuable in helping the user focus on what needs to be decided next in order to push the task forward. The agent acts as a personal assistant, helping the user to get organized and keeping the user on focus.

10.3. Future Direction

We have made future plans to extend Collagen. They include:

1. Improve on the method for collecting user-specific knowledge. Knowledge collection about the user can be done in the middle of a conversation so that the user will not feel interrupted.

2. Integrate eSuite mail with eSuite address book. This will add a rich dimension of user customization that can be explored by the agent developer. The agent can provide help to the user in new ways such as answering the question who is adac@mit.edu by looking up the address book.
3. Include plan recognition capabilities. User's goal can be determined by going up the hierarchy of the plan tree.
4. Use speech to replace the text-based communication between the user and agent. The Lotus and IBM Research expect to undertake this approach.
5. Build collaborative interface agents for other applets in the eSuite project.

Appendix A User Interfaces of eSuite Mail

The following figures show the user interfaces of the three applets in eSuite Mail. Each applet has one or more views.

A.1 User Interfaces in MailApplet

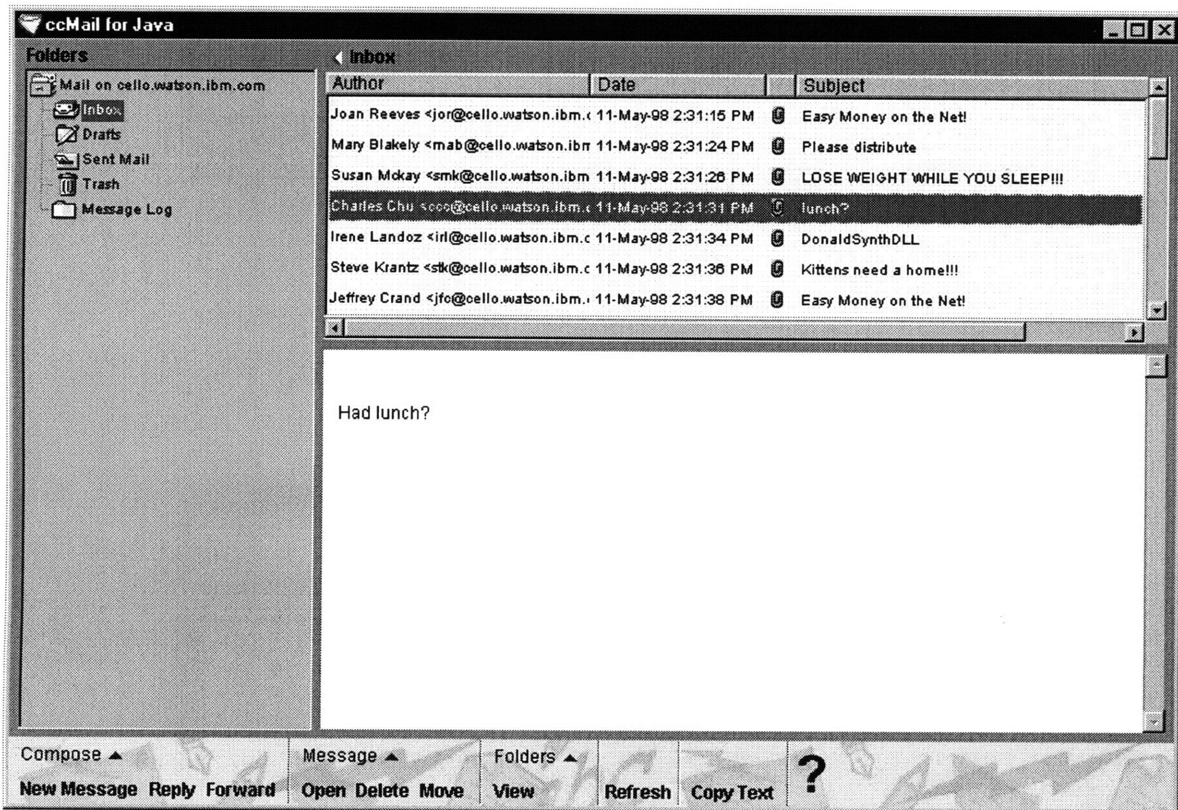


Figure A-1: Mailbox/Preview View – hierarchical folders shown

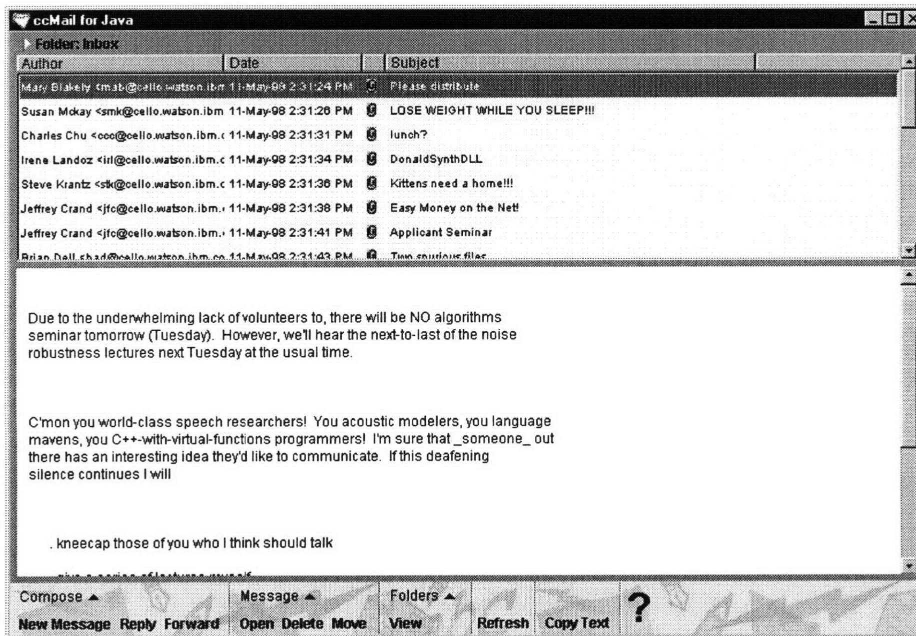


Figure A-2: Mailbox/Preview View – hierarchical folders hidden

A.3 User Interfaces of ComposeMessageApplet

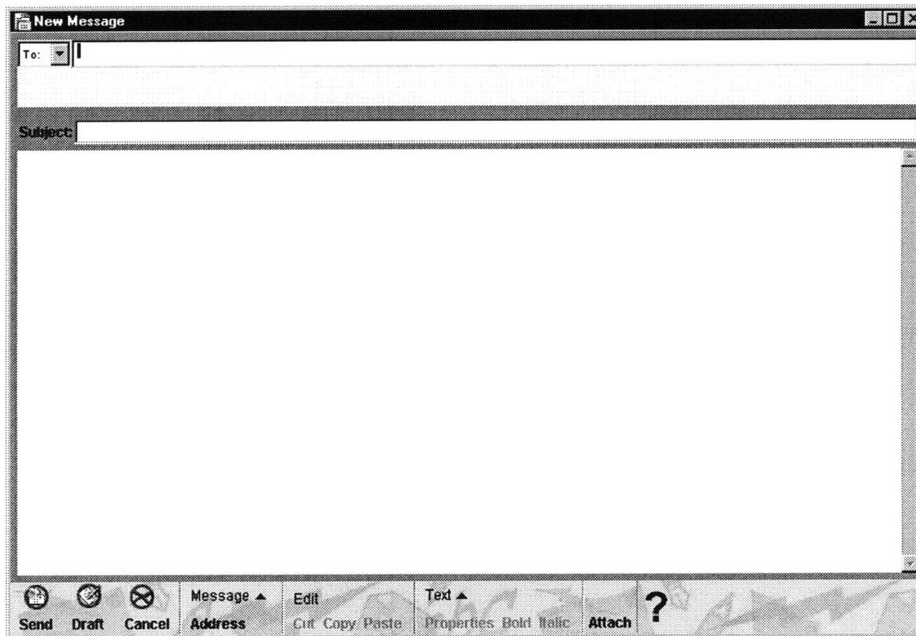


Figure A-3: New Message Creation View

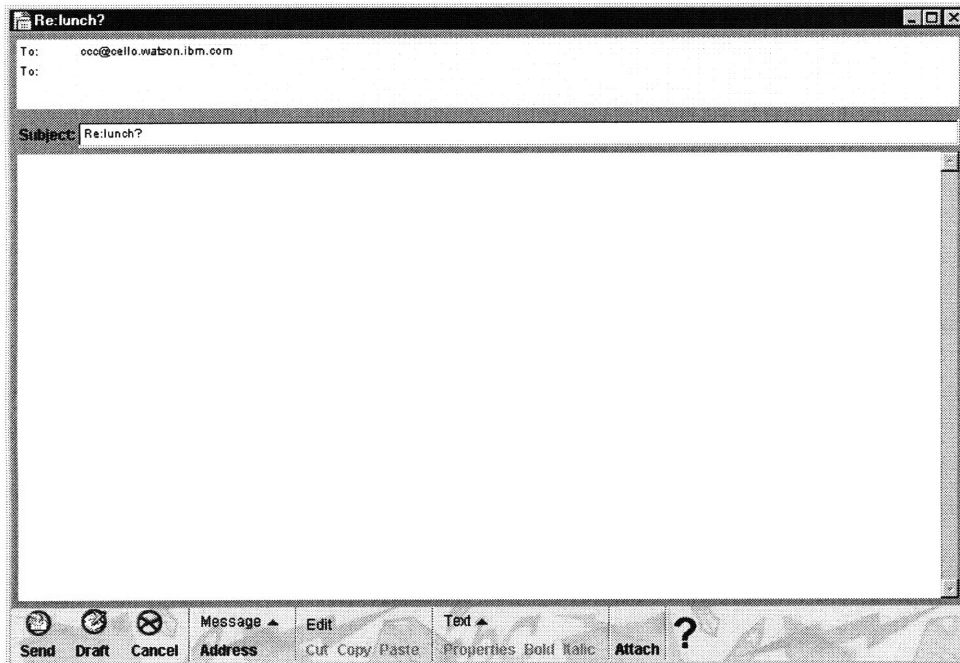


Figure A-4: Reply View

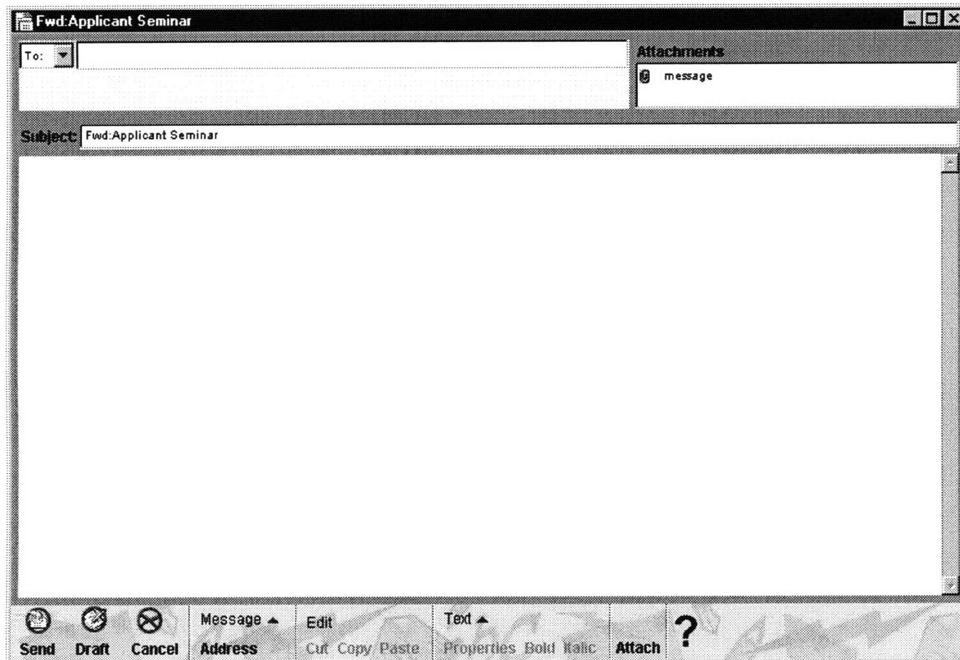


Figure A-5: ForwardView

A.3 User Interface of ReadMessageApplet

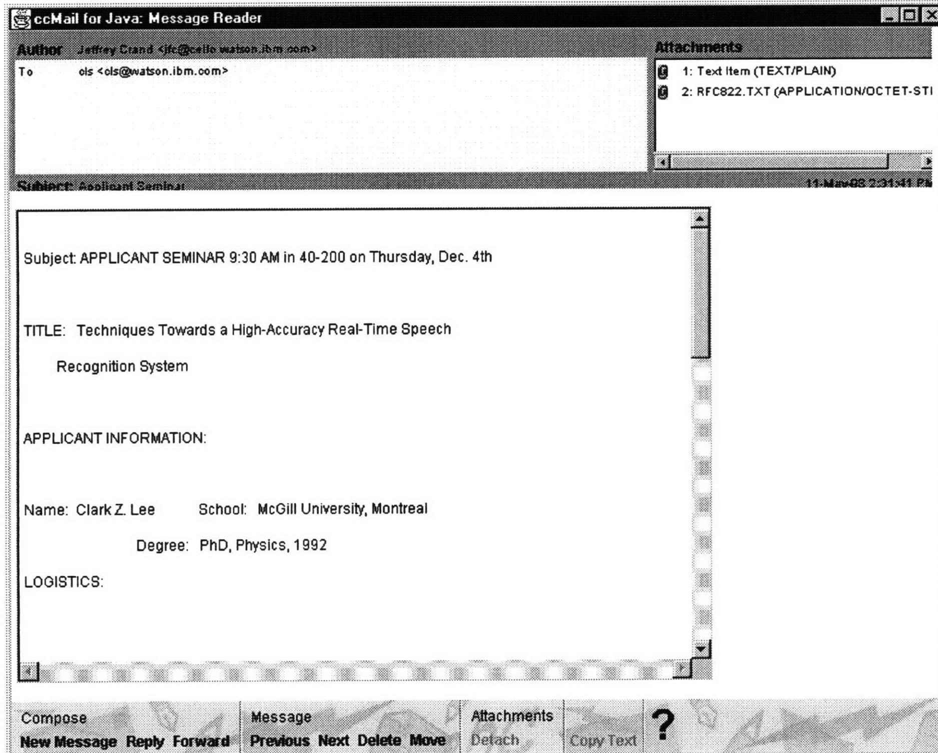


Figure A-6: Read Message View

Appendix B Segmented Interaction History

```
// Collagen 0.9.1  (JDK 1.1.6) Wed Jul 22 16:52:55 EDT 1998
Working on email.
 USER:   "Propose working on email."
 AGENT: "You have 0 unread messages."
 USER:   "Ok."
 Done reading message from Steve Krantz <stk@cello.watson.ibm.com> about Kittens need a home!!!  by
opening a window.
   AGENT: "Propose reading a message."
   USER:   Select message from Steve Krantz <stk@cello.watson.ibm.com> about Kittens need a home!!!
   USER:   "Ok."
   Done examining message from Steve Krantz <stk@cello.watson.ibm.com> about Kittens need a home!!!
     AGENT: "Propose examining message from Steve Krantz <stk@cello.watson.ibm.com> about Kittens
need a home!!! "
     USER:   "Ok."
     AGENT: Open message from Steve Krantz <stk@cello.watson.ibm.com> about Kittens need a home!!!
     USER:   "Ok."
     Stopped using message from Steve Krantz <stk@cello.watson.ibm.com> about Kittens need a home!!!
       AGENT: "Propose using message from Steve Krantz <stk@cello.watson.ibm.com> about Kittens need
a home!!! "
       USER:   "No."
     Done agent closing message from Steve Krantz <stk@cello.watson.ibm.com> about Kittens need a
home!!!
       USER:   "Close message from Steve Krantz <stk@cello.watson.ibm.com> about Kittens need a
home!!! "
       AGENT: Close message from Steve Krantz <stk@cello.watson.ibm.com> about Kittens need a home!!!
     USER:   "Ok."
 Done responding to message from Steve Krantz <stk@cello.watson.ibm.com> about Kittens need a
home!!!  with this message by replying.
   AGENT: "Propose responding to message from Steve Krantz <stk@cello.watson.ibm.com> about Kittens
need a home!!!"
   USER:   Reply to message from Steve Krantz <stk@cello.watson.ibm.com> about Kittens need a home!!!
with this message.
   Done agent filling in this message.
     USER:   "Fill in this message."
     AGENT: Fill in the CC field of this message with adac@mit.edu.
   USER:   Cancel this message.
   USER:   "Ok."
 Managing message from Steve Krantz <stk@cello.watson.ibm.com> about Kittens need a home!!! .
   AGENT: "Propose managing message from Steve Krantz <stk@cello.watson.ibm.com> about Kittens
need a home!!! ."
   USER:   Delete message from Steve Krantz <stk@cello.watson.ibm.com> about Kittens need a home!!!
```

Figure B-1: Segmented interaction history of sample scenario

Each act in the history is either a communication or the description of an application-level manipulation. Both the user and agent can perform each act.

Indentation in the history indicates the nesting of collaborative goals. The phrase at the start of the indent indicates the goal being achieved by that segment. The goal of an open segment is glossed with a present participle, such as “doing.” Closed segments are glossed starting with “done” or “stopped.” Remaining unexecuted steps of a recipe are glossed starting with “expect.” Notice that our scenario ends after completing the Respond step. The segment corresponds to the goal Respond is closed. The optional step Manage remains unexecuted and it is glossed with “expect.”

Reference

Colan, M. (1998). InfoBus 1.1 Specification. URL http://java.sun.com/beans/infobus/

Englander, R. (1997). Developing Java Beans. Sebastopol, CA: O'Reilly & Associates, Inc.

Flanagan, D. (1997). Java in a Nutshell. Sebastopol, CA: O'Reilly & Associates, Inc.

Geary, D. (1997). Graphic Java Mastering the AWT. Mountain View, CA: Sun Microsystems Press.

Grosz, B. & Sidner, C. (1986). Attention, Intentions, and the Structure of Discourse. Computational Linguistic (pp. 175-204).

Grosz, B. & Sidner, C. (1990). Plans for Discourse. Intentions and Communication (pp. 417-444). Cambridge, MA: MIT Press.

Lashkari, Y., Maes, P., & Metral, M. (1994). Collaborative Interface Agent. Proceedings of AAAI '94 Conference. Seattle,Washington.

Lieberman, H. (1998, January). Intergrating User Interface Agents with Conventional Applications. ACM Conference on Intelligent User Interfaces. San Francisco, CA.

Lieberman, H. (1997, March). Autonomous Interface Agent. ACM Conference on Human-Computer Interface [CHI-97]. Atlanta, GA.

Lochbaum, K.E. (1995). The Use of Knowledge Preconditions in Language Processing. Proceedings of 14th International Joint Conference of Artificial Intelligence (pp. 1260-1266). Montreal, Canada.

Maes, P., & Wexelblat, A. (1996). Interface Agent. Proceedings of the CHI '96 conference companion on Human factors in computing systems: Common Ground (pp. 369-370).

Rich, C., & Sidner, C. (1997). Adding a Collaborative Agent to Graphical User Interfaces. Proceedings of the ACM Symposium on User Interface Software and Technology (pp. 21-30).

Rich, C., & Sidner, C. (1998). Collagen: A Collaboration Manager of Collaborative Interface Agent. Submitted for publication to UMUAI. Will appear in Vol. 7, No. 3-4, 1998.

Rich, C., & Sidner, C. (1997b). Segmented Interaction History in a Collaborative Interface Agent. Proceedings of International Conference on Intelligent User Interfaces (pp. 23-30). Orlando, FL.