EMOO - Experimental MOO - A Virtual Environment Framework

by

Min Wu

Thesis
Submitted in partial fulfillment of the requirements for
the Degree of Master of Science (Computer Science)

Acadia University
Fall Convocation 2000

TABLE OF CONTENTS

TABLE OF FIGURES

ABSTRACT

EMOO – an Experimental MOO – is an innovative implementation of a virtual environment based on MOO (MUD object-oriented) concepts, which provides a user-friendly interface to a virtual environment. It addresses collaborative work, particularly in geographically dispersed teams and allows users to subscribe to events and to be notified automatically when the event occurs.

The focus of this thesis project is on conceptual and design issues of EMOO and the comparison of implementation of a similar system in Java and Smalltalk.

ACKNOWLEDGEMENTS

I would like to thank all who helped during the development of this thesis and the underlying project. Their suggestions, assistance with the project, reviewing of the thesis and overall guidance and support are sincerely appreciated. This includes Dr. Ivan Tomek, Dr Rick Giles, Guang Yang, David Murphy, Krista Yetman, and Marlene Jones.

In particular, I would like to thank my supervisor, Dr. Ivan Tomek, for his consistency of instruction and his consideration. It is Dr. Tomek who led me into the object world and taught me OO (Object-Oriented) thinking, OO analysis and OO design. I could not have done it without him.

I also want to thank to Dr. Peter Hitchcock, my external examiner, Dr. Leslie Oliver, my internal examiner, Dr. Daniel Silver, acting director, and Dr. Anna Migliarisi, chair, for examining my thesis.

DEDICATION

To all the people who love me and are my beloved.

Chapter 1 Preface

1.1 Introduction

The concept of a networked Virtual Environment (VE) that emulates selected features of the physical world has been known and popular at least since the late 1970's when the first game-oriented MUD (Multi-User Dungeons) [Haynes, 1998] was developed. The concept has since evolved and modern VEs are used not only in game-playing but also for socialization and education. Lately, as software systems become larger and larger, software development often involves geographically dispersed teams of developers. In order to explore the concept of using a virtual environment to support such design activities, we designed and implemented several projects from Jersey MOO, through MUM (Multi-Universe MOO), to the subject of this thesis, a project named EMOO (Experimental MOO). Each of them has different implementation and characteristics, and a detailed description will be given in Chapter 2.

EMOO is a VE based on MOO (MUD object-oriented) concepts. As other VEs, it is a client-server application, but unlike other VEs, it places heavy emphasis on the client to perform as much processing as possible to minimize server and network load. Another distinguishing feature is that it allows its users to subscribe to events and be notified when they occur.

EMOO also allows its users to create an arbitrary number of universes. The management and access to individual network universes is controlled by a metaserver, which is aware

of all currently accessible universes. Client operation is based on tools that have graphical interfaces to provide access to EMOO's general functionality.

The implementation language of EMOO is Java. We have previously used Smalltalk to implement a similar environment called MUM (Multi-Universe MOO). MUM is a pilot project that has many innovative features, but its design is complicated, which affects its performance. EMOO uses different design to address this problem.

In the rest of this thesis, I will describe EMOO design and compare it with MUM with the goal of comparing the implementation of these two major applications in the two different languages.

1.2 Organization of the thesis

The thesis consists of nine chapters.

- Chapter 1 Preface gives a brief introduction of EMOO and an overview of the thesis.
- Chapter 2 Background Information introduces basic concepts of MUD and MOO, their brief history, and a description of research activities in this area.
- Chapter 3 EMOO Development Technologies explains the technologies and the development environment used to implement EMOO and the reason for using these technologies.
- Chapter 4 System Specification describes EMOO design goals and functions, using UML to describe high-level use cases.

- Chapter 5 Design Overview outlines EMOO's overall architecture and principles of operation.
- Chapter 6 Detailed Design gives a detailed description of individual EMOO subsystems and their responsibilities, and the main instance variables and methods of the main classes.
- Chapter 7 Environment Comparisons compares EMOO with the similar system MUM. It also compares Java with Smalltalk based on implementation the similar systems.
- Chapter 8 Using EMOO uses screenshots to illustrate step by step how EMOO works and how to use it.
- Chapter 9 Conclusion summarizes EMOO current state and suggests possible future work.
- Glossary gives the definitions and description of selected terms.
- Bibliography lists all the references used in the thesis.

Chapter 2 Background Information

2.1 What is a Virtual Environment?

A virtual environment (VE) is a software application that emulates physical world with a multitude of navigable places, portable objects, and communicating autonomous entities including human users and software agents. Several paradigms have been used to implement a VE framework, including MUDs, MOOs, digital worlds, virtual reality, collaborative buildings, and mixed paradigms. Among them, MUDs and MOOs are used most, and their technologies are most mature [Tomek, 2000].

2.2 What is a MUD/MOO?

MUD is an acronym of Multiple User Dimension, Multiple User Dungeon, or Multiple User Dialogue. MUD is a virtual world designed inside a mputer allowing a user to extend the environment, socialize, accomplish goals, solve puzzles, add new areas to the world, and generally have a lot of fun. MUCK, MUSH, MUX, DGD, YAMA, MUSE, MUG, Ogham and MOOS are variants of MUDs [Okstate]. MOO stands for MUD Object Oriented. It is a MUD implemented in an Object-Oriented language.

MUDs (including MOOs) are client-server applications. They started appearing about twenty years ago as an Internet-based version of a popular role-playing fantasy game called Dungeons and Dragons. The essence of a MUD is a text-based environment including entities such as rooms, castles, caves, and so on, representing a simulated world

or universe in which participants can move around (navigate), communicate with one another, create, destroy, pick up and drop simulated objects, use them, and carry them around.

The MUD server typically holds a database representing a virtual representation of a physical space organized into "rooms". A room corresponds to a place where objects and characters representing users or software agents may be located. One of the essential features of MUDs is that they allow users to communicate. The primary means of communication within MUDs is by talking to characters representing other people who are located in the same room.

2.3 Brief history

Roy Trubshaw and Richard Bartle developed the first MUD known as MUD1 in 1979 using assembly language programming [MUDline]. The original version merely allowed a user (player) to move about in a virtual location, but later versions provided for more variation including objects and commands that could be modified online or offline.

As the Internet became popular and extensively used, Jim Aspnes wrote TinyMUD Original in August 1989, the first of the 'tiny' family of MUDs. TinyMUD was a simple, user-extensible multi-user game that was available to anyone on the Internet who knew the address and port number of the server. It was created for players to reside in, navigate, converse and build virtual worlds together. It kept the entire database in

memory rather than on hard disk and was much faster. The design assumed that the database would not grow too large.

With the experience of using and developing tiny MUDs, Stephen White created the first MOO in May 1990. Compared to MUDs, it had greater flexibility in object manipulation, and was more realistic in terms of what objects could do by allowing actions to be attached to objects, rooms and players. In a MOO, every conceptual object was also an object in the implementation sense and had a unique identifier, as well as other properties such as a name, current location, description, owner, etc.

After the creation of the first MOO, Pavel Curtis created LambdaMOO in October 1990, a popular form of MOO. It was derived from Stephen White's MOO. LambdaMOO was a network-accessible, multi-user, programmable, interactive system, well-suited to the construction of text-based adventure games, conferencing systems, and other collaborative software. It used its own programming language called Lambda MOO to allow users to create their own objects.

2.4 How do MUDs and MOOs work and what are their functions?

The traditional MUD interface is command-based. In order to use a MUD, a user must have a "character" (also called an "avatar") on the server and supply a password to login to the MUD as that character. Once the connection has been opened, all commands entered by the user are perceived to come from the character. When the connection closes, the state of the character, including its location, possessions, etc., is usually

preserved by the server. When the user types a command such as "look", the command is sent to the MUD server, the server processes the command according to a stored dictionary, and sends the result back to the client who displays the result on the screen. In some cases multiple-clients are notified [Haynes, 1998].

Though the first MUD was game-oriented, the concept has since evolved and modern MUDs are used not only for game-playing but also for socialization, education and collaborative work. More details will be provided in the following sections.

2.5 Recent research in this area

2.5.1 LambdaMOO

The currently most popular implementation of MUDs for "serious mudding" in educational and collaborative uses is LambdaMOO [Lambda]. Lambda is a special-purpose object-oriented programming language that allows its users to set up a MOO server with a universe of places and objects, define a group of users with the authority to modify and extend the universe, and authorize others to use the universe in more or less restrictive ways. Figure 2-1 is a screenshot of a LambdaMOO connection.

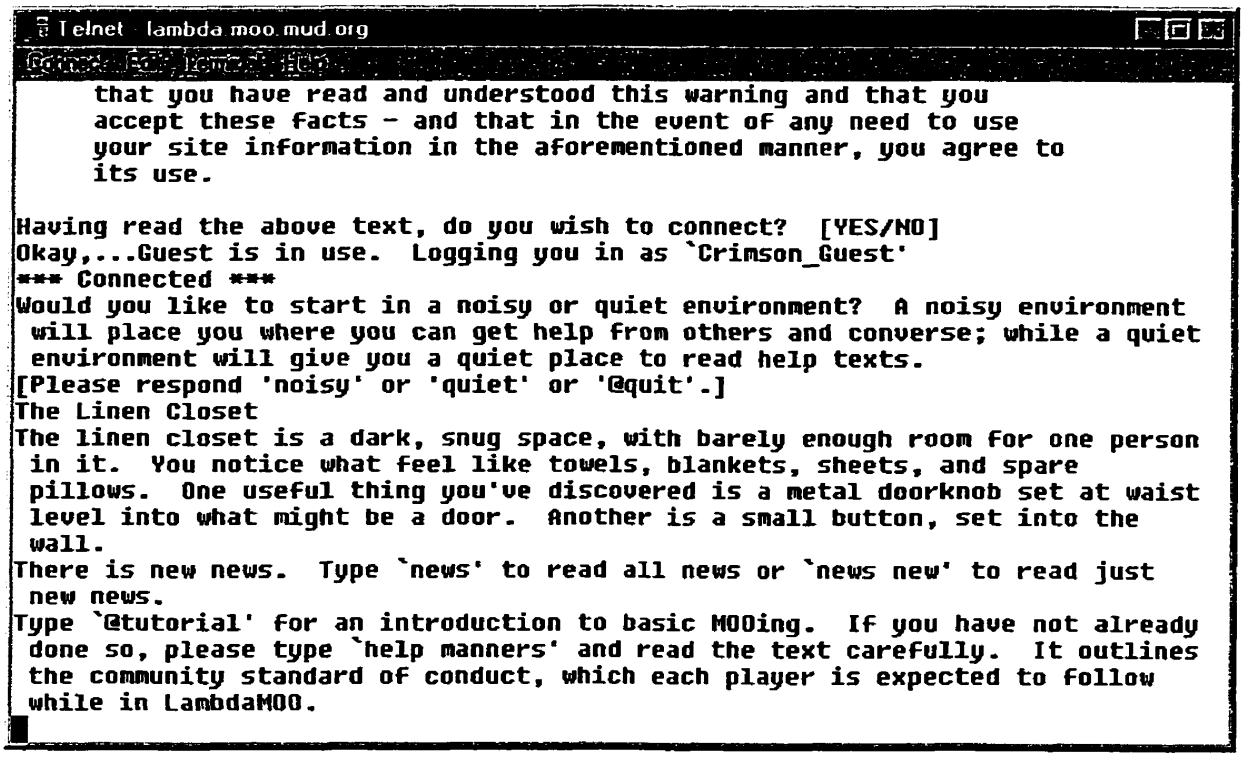

Figure 2-1. LambdaMOO's connection interface

2.5.2 Jersey MOO

Most MUDs and MOOs, including LambdaMOO, use Telnet for their user interfaces. More recent implementations started using GUI and web browsers [Lingua MOO], while still mostly retaining command-based operations. The user interface of TWUMOO [TWUMOO] is a typical example (Figure 2-2) of such an interface. A different approach was used by Jersey MOO.

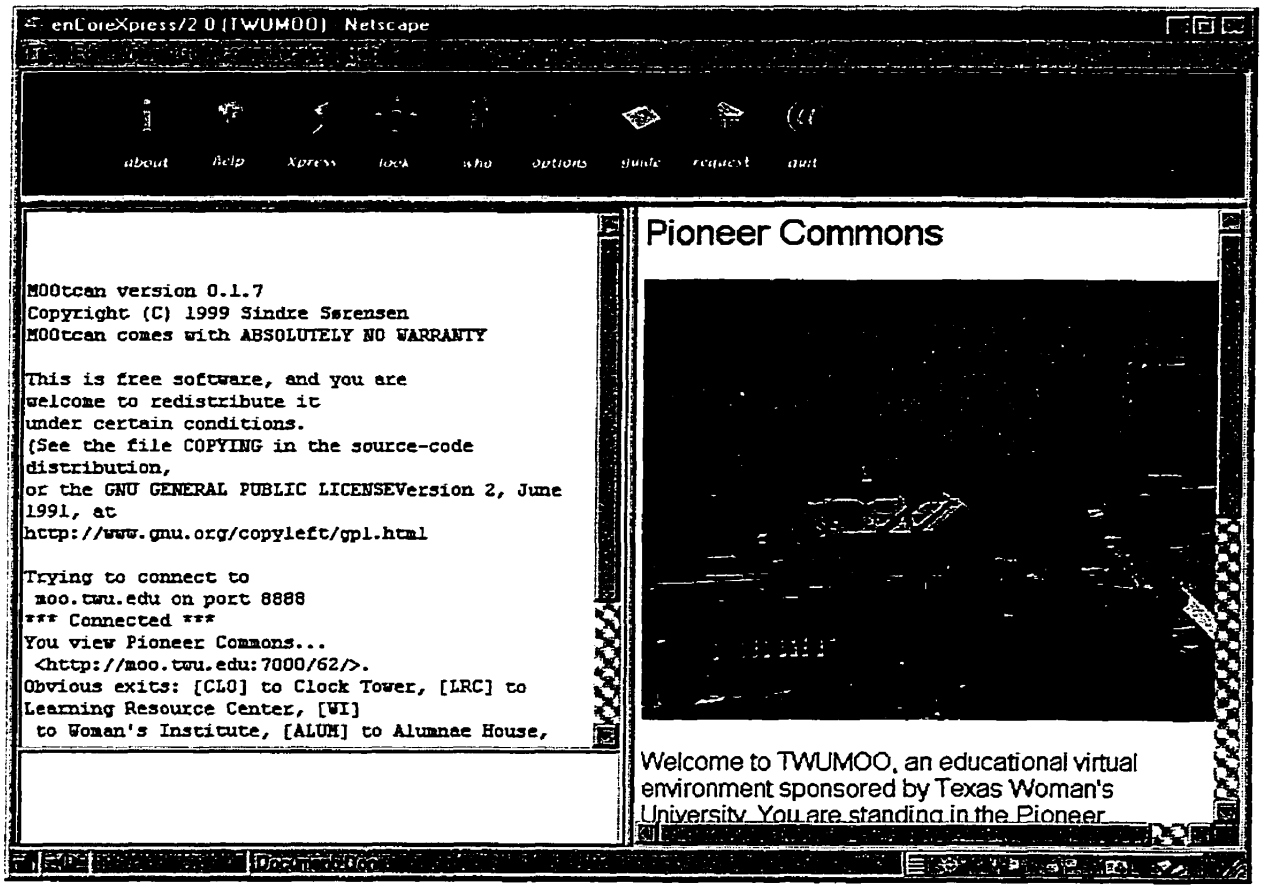

Figure 2-2. Portion of TWUMOO user interface

The basis of Jersey MOO was developed by Object Technology International Inc. (OTI) and extended at Acadia University. It introduced mouse-click-based operation to save their users from having to remember many MUD commands. It was used to explore the concept of using a virtual environment to support cooperation within and among software development teams working on one or several projects, at the same physical location or geographically dispersed, at the same time or asynchronously [Tomek, 1999]. Jersey MOO consisted of a Smalltalk server and a web user interface implemented with Java applets (Figure 2-4). Because all execution occurs on the server and the server executes Smalltalk messages, all communication from the client to the server is in terms of

Smalltalk messages, transmitted over the network as ASCII text. Since Jersey MOO uses Java applets as user interface, it does not require users to remember and type commands. Instead, command execution is based on clicking command names. This action then produces a command template, which is completed by the user and sent to the server for execution. In addition, certain operations are implemented fully by mouse clicks and do not require any additional user interaction. Communication and navigation have their special interface. Figure 2-3 shows Jersey's main user interface.

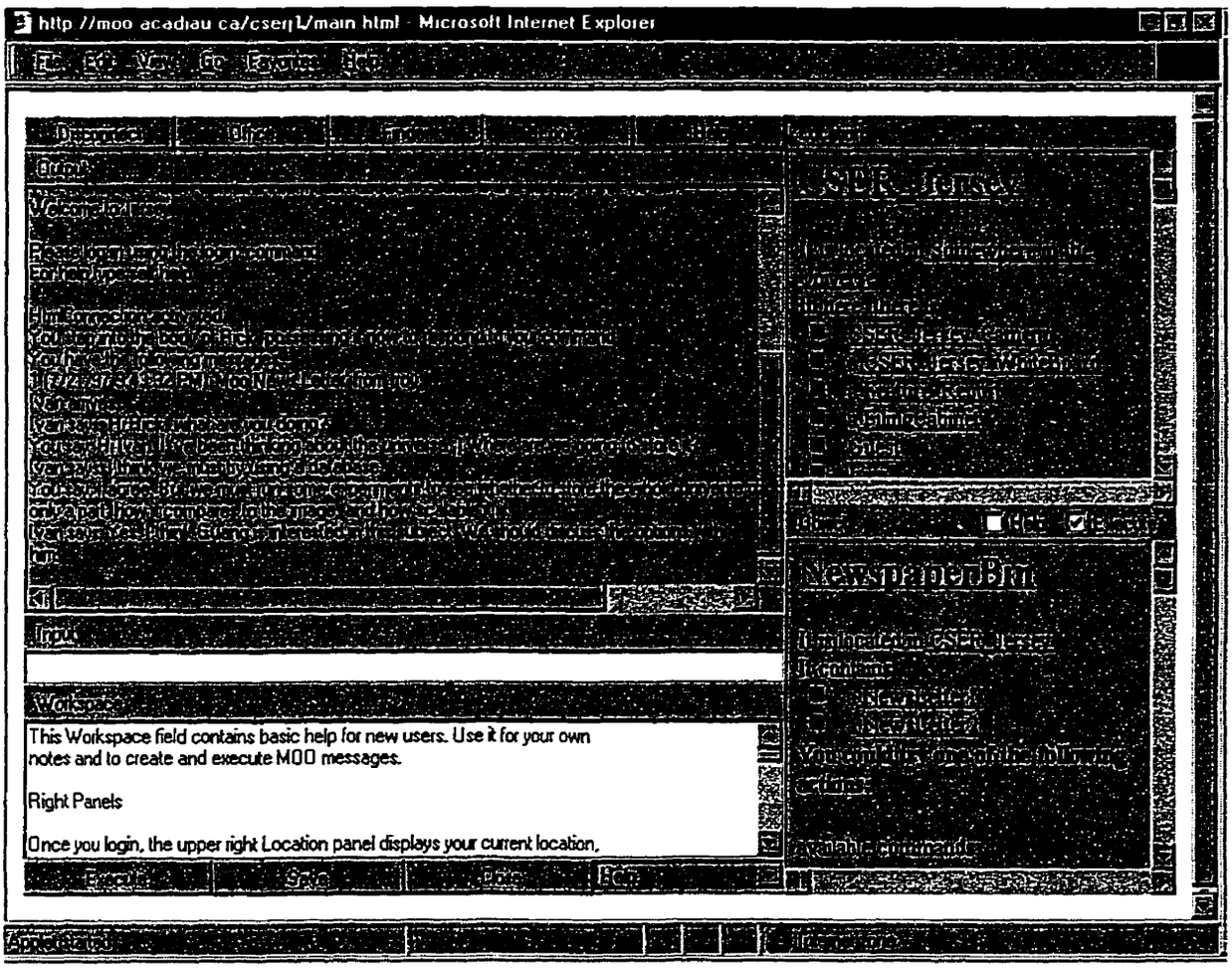

Figure 2-3. Jersey's main user interface

2.5.3 MUM

After some experimentation with Jersey, the Acadia University group concluded that its design makes it difficult to implement certain features that are necessary in a virtual environment useful for collaborative work and education. For example, keeping users informed about some actions in the universe in which they are interested is difficult to implement and requires the use of Jersey agents. Moreover, although Jersey removes the need to remember commands, the user interface is still not quite as easy to use as a window customized for a specific task. Instead of having to fill in a template, the user should only need to enter clearly identified values and click a button to execute the command. Because of these shortcoming of Jersey, it was decided to implement a new MOO with these and other features from scratch and the result is MUM – Multi-Universe MOO [Tomek, 1999].

MUM provides a user-friendly interface to a virtual environment, so that the user does not need to remember and type commands. It addresses collaboration, particularly in geographically dispersed teams, by allowing users to subscribe to events and to be notified automatically when the event occurs. It also allows users to create any number of universes, and move their agents with their possessions from one universe to another. MUM uses event-driven operation to implement these features, that is, every action in MUM is an event, and every object in the environment operates solely on the principle of response to events. The implementation language is Smalltalk, which allows very easy modification of events at run time. Figures 2-5 and 2-6 showing the MUM Launcher and MUM UniTool are two examples of a MUM user interface.

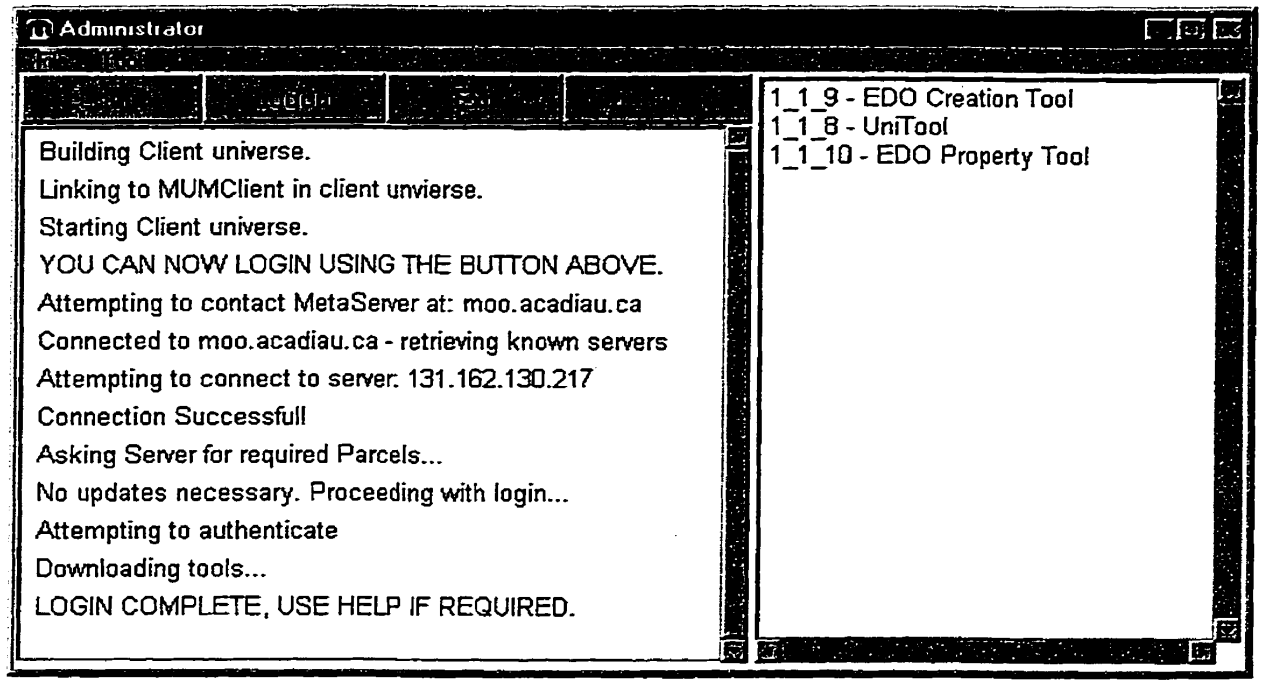

Figure 2-4. MUM Launcher

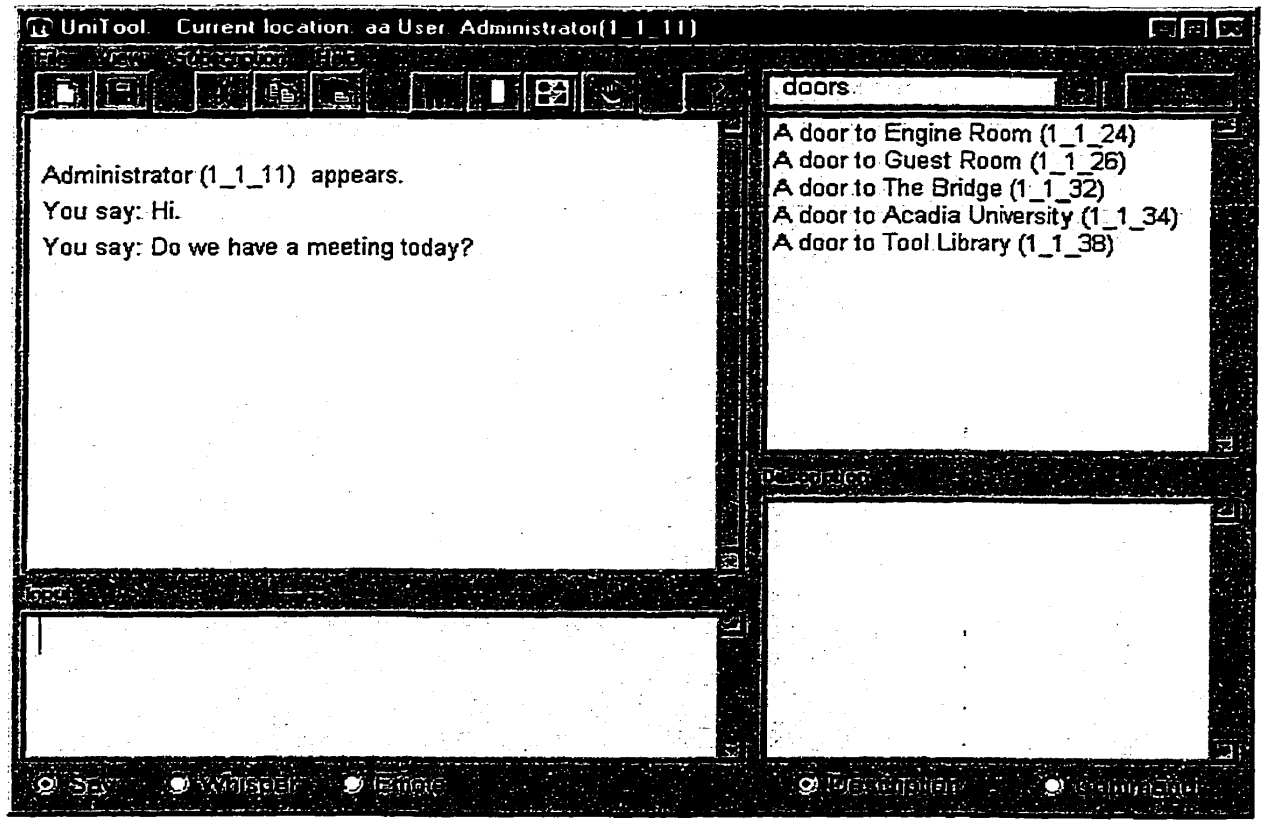

Figure 2-5. MUM UniTool

2.6 Conclusion

My involvement with MOOs began with MUM. Though MUM is a pilot project using state-of-art methodology and technologies, it still needs to be improved with new approaches and designs. Thus I chose an implementation of a new design of the MUM as my thesis project and implemented a MOO called EMOO (Experimental MOO). I used Java as the programming language to explore a different technology and for comparison with Smalltalk. The rest of the thesis details my approach and draws conclusion based on the experiences of MUM and EMOO.

Chapter 3 EMOO Development Technology

3.1 Overview

This chapter describes the technologies and development environment used to implement EMOO, and explains why they were used.

EMOO is a client-server system based on the distributed object model. It uses Java (Java 2™) [JavaSoft] as the programming language with Remote Method Invocation (RMI) to implement network functions and the distributed environment. JBuilder 3 [Borland] is the Java Integrated Development Environment (IDE) used during development. Unified Modeling Language (UML) was used during system analysis and design, and Rational Rose 2000 Evaluation Edition [Rational] was used as UML design tool.

3.2 Why Distributed Objects?

The term "distributed objects" is used to describe interacting objects residing on different computers scattered across a network. Any object can reside anywhere on the network, and applications can interact with these objects exactly as they do with local objects.

Distributed object technology must be related to client-server applications. The traditional client/server model simplified the development and maintenance of complex applications by separating centralized systems into client and server components that were easily developed and maintained. The server components provide service and

information and clients access the information or make requests on the server. But many client/server solutions merely divide one application into two parts, so it still remains difficult to build, maintain, and extend mission-critical client/server applications. Using this approach, developers sometimes have to create the same functionality over and over again which makes code reuse difficult. Eventually, a proliferation of similar modules must be updated and maintained separately and a change to one module must be propagated to similar modules throughout the system.

Distributed object technology fundamentally changes all these concepts. With the help of the powerful communications infrastructure, distributed objects complement today's client-server applications with self-managing objects that can cooperate across different networks and operating systems. The distributed-object computing model makes it easier to distribute data and function freely and transparently. Programming objects can run on remote hosts as well as local hosts. In large complicated applications, distributed object technology can save a lot of design overhead, and make a large distributed system easier to maintain [Farley, 1998].

3.3 Why Java?

All our previous work with networked Virtual Environments (VE) used Smalltalk. One of the reasons for using Java to implement a VE was to acquire new VE experience. Another was to compare development in these two major object-oriented languages. (The comparison of two implementations in Java and Smalltalk will be given in Chapter 6.)

Java is also currently very popular and it is becoming the dominant mainstream object-oriented language. Which technical features make Java so attractive [Meier, 1999]?

- Java is an object-oriented programming language with a set of supporting technologies, such as RMI, JDBC (Java Database Connectivity), JNDI (Java Naming and Directory Interface), JINI, and so on.
- It is both compiled and interpreted. Java source compiles into bytecodes looking a lot like conventional machine language. Bytecodes are executed by a Java Virtual Machine (JVM).
- It is designed to be portable and platform independent.
- Web browsers can interpret Java.
- It is very dynamic. Java's dynamic loading makes it possible to load classes incrementally into a virtual machine as it executes.

Java's features make it very suitable for developing a VE, because all these features are important and needed to implement a VE. Moreover, more and more technologies are available in Java, so it is very useful for VE's evolution.

3.4 Why RMI?

There are several ways to support network communications, including basic socket communications, RMI remote objects, DCOM (Distributed Component Object Model) remote objects and CORBA (Common Object Request Broker Architecture) remote objects. The following sections explain why we chose RMI rather than one of the alternative technologies.

3.4.1 RMI

Remote Method Invocation (RMI) is a Java native scheme for distributed objects [RMI]. It provides a way for client and server applications to invoke methods across a distributed network of clients and servers running the Java Virtual Machine (Figure 3-1). The RMI API (Application Programming Interface) allows programmers to access a remote server object from a client program by making simple method calls on the server object. To transfer objects on the network, the RMI API uses the Serialization API to wrap (marshal) and unwrap (unmarshal) the objects. To marshal an object, the Serialization API converts the object to a stream of bytes, and to unmarshal an object, the Serialization API converts a stream of bytes back into an object [Morrison, 1997].

The RMI architecture is based on the principle of stubs and skeletons. The stub, loaded on the client station, plays the role of the server's proxy. A remote method call is performed through the stub, and the server's location is made transparent to the developer as a call to a remote method has the same syntax as a call to a local method. The server-side counterpart of the stub is the skeleton, which behaves like the client for the server.
A working RMI system is composed of the following parts (Figure 3-1):

1. Interface definitions for the remote services, which extend the *Remote* interface.
2. Implementations of the remote services, which extend class *UnicastRemoteObject* or use the *exportObject()* method to link into RMI.
3. Stub and skeleton files, which are generated by the RMI compiler *rmic* that runs on the remote service implementation class file.

4. RMIRegistry, which is an RMI Naming service that allows clients to find the remote services.
5. A Server hosting the remote services, using the *Naming.rebind()* method to make a remote call to the RMI registry on the local host.
6. A client program that needs the remote services, using the *Naming.lookup()* method to look up the remote object by name in the remote host's registry.

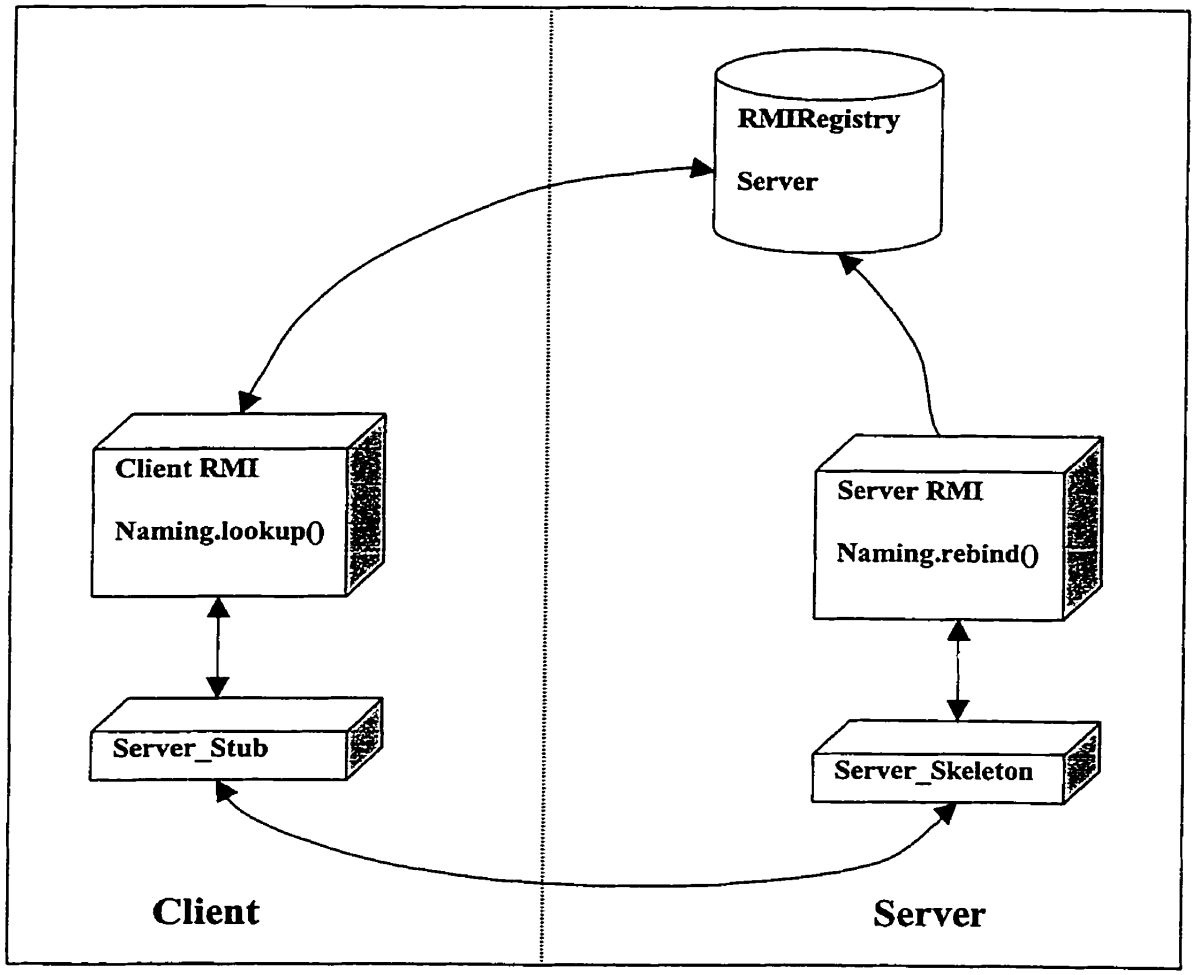

Figure 3-1. Interaction between a server, a client and the RMIRegistry

3.4.2 Sockets

Sockets are a mechanism for implementing low-level network connections [Stevens, 1990]. Socket-based applications are usually less expensive, more flexible, faster, but using sockets to communicate between client and server can be cumbersome because of all the details that must be taken care of. The most complex task when using sockets is that it is very difficult to send an object as a parameter, or receive one as a return value. Since only primitive values can be sent across the socket stream, sending objects via a socket connection may require extracting the object's attributes, marshaling and transmitting them, and vice versa to receive return objects.

In comparison with sockets, RMI has the advantage of a higher-level Java networking API that provides transparent object access in a distributed environment. It encapsulates all of the network-level details of remote method calls. Its object serialization system provides a way to send and receive objects over the network as primitive values [Farley, 1998].

Another disadvantage of sockets is that a program using sockets is larger and more complex than one using RMI. On the other hand, sockets may be faster. This is analogous to the choice of programming languages: A program written in assembly language can usually outperform one written in C or Java. However, an assembly program is much harder to write, is likely to contain more errors, and is more difficult to maintain than a program written in a high-level language.

3.4.3 DCOM

Distributed Component Object Model (DCOM) is Microsoft's technology for distributed object architectures [DCOM]. DCOM is an extension of Microsoft's Component Object Model (COM), formerly called Network OLE, that forms the basic architecture of the Windows operating system. The main objective of COM is to permit independent development of software components that can intercommunicate, regardless of languages or functions. DCOM extends the functionality of COM by allowing components to communicate remotely via LAN, WAN, or the Internet, but it is only well suited for Microsoft-centric environments. If any operating systems other than Microsoft NT and Windows are required, DCOM is not appropriate.

RMI, as a Java-centric scheme for distributed objects, has all of the benefits of Java. In particular, RMI system is platform independent, so it is suited for any operating systems and environment.

3.4.4 CORBA

The Common Object Request Broker Architecture (CORBA), developed by the Object Management Group (OMG), enables invocations of methods on distributed objects residing anywhere on a network, just as if they were local objects [CORBA]. Unlike RMI, which is Java-centric, CORBA is designed to be language-independent and allows objects on the client side to make requests objects on the server side without any prior knowledge of where those objects reside, what language they are implemented in, and which operating system they are running on. According to [Whatis], "The essential

concept of CORBA is the Object Request Broker (ORB). ORB support network of clients and servers on different computers means that a client program (which may itself be an object) can request services from a server program or objects without having to understand where the server is in a distributed network or what the interface to the server program looks like. To make requests or return replies between the ORBs, programs use the General Inter-ORB Protocol (GIOP) and, for the Internet, its Internet Inter-ORB Protocol (IIOP). IIOP maps GIOP requests and replies to the Internet's Transmission Control Protocol (TCP) layer in each computer". CORBA is the best for large distribute systems with legacy services implemented in many languages, but it is much more complicated than RMI. It has a rich, extensive family of standards and interfaces, and defines remote objects using a separate language IDL (Interface Definition Language) for language independence.

Compared with CORBA, RMI not only simplifies distributed application development for Java system, but also has the additional benefit of automatic distributed garbage collection. Since EMOO is completely implemented in Java, there is no need to link it to other systems implemented in other languages or to migrate to other languages in the future. RMI is the best choice for EMOO [Morrison, 1997].

3.5 Why JBuilder?

There are many Java IDEs for development of Java applications, such as JBuilder [Borland], Visual Age for Java [IBMVJ], Visual Café [Symantec] and Visual J++

[MicrosoftVJ]. Each IDE has its own features, so different people may choose different IDEs. We have decided to use JBuilder, which is described next.

JBuilder, developed by Borland Company, is a group of very productive visual development tools for creating high-performance, platform-independent applications using Java. It is designed for all levels of development projects, ranging from applets and applications that require networked database connectivity to client/server and enterprise-wide, distributed multi-tier computing solutions. The JBuilder3 open environment supports Pure Java, JavaBeans, Enterprise JavaBeans, Servlets, JDK 1.1, Java 2, JFC/Swing, RMI, CORBA, JDBC, ODBC, and all major corporate database servers. JBuilder also provides developers with a flexible open architecture to incorporate new JDKs, third-party tools, add-ins, and JavaBean components [Borland].

Using JBuilder's "Two-Way tool" technology the developer can easily design elegant GUIs by placing widgets on the screen, and automatically converting them into code. GUI design becomes very fast and the designer can concentrate on implementing system functions. Using JBuilder's powerful Debugger one can rapidly find a variety of errors: The debugger can execute program step by step, trace and watch values of all variables via setting breakpoint. JBuilder's online help and method tips are also convenient: The programmer does not need to lookup and remember which methods an object knows. In response to typing the name of an object, JBuilder pops up a list with all the methods the object knows.

3.6 Why UML?

According to [Booch, 1999], "The Unified Modeling Language (UML) is a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system". UML was originally conceived by Rational Software Corporation and three of the most prominent methodologists in the information systems and technology industry, Grady Booch, James Rumbaugh, and Ivar Jacobson (often called "Three Amigos"). When UML was created, Rational established the UML Partners consortium with several organizations willing to dedicate resources and work to UML, including Digital Equipment Corp., HP, i-Logix, IntelliCorp, IBM, ICON Computing, MCI Systemhouse, Microsoft, Oracle, Rational Software, TI, and Unisys. This collaboration made UML well defined, expressive, powerful, and generally applicable. Finally, UML has been submitted to and approved by the Object Management Group (OMG) as a standard. UML graphical representations can be used as the basis of communication between software developers, so all the developers of a system can understand what the responsibilities of their parts of the system are during analysis and design phase [Fowler, 1997].

UML is a notation for expressing object-oriented analysis and design (OOA/OOD) in the form of diagrams. There are nine kinds of diagrams in UML to visualize a system from different perspectives: Class diagram, Object diagram, Use case diagram, Sequence diagram, Collaboration diagram, Statechart diagram, Activity diagram, Component diagram and Deployment diagram.

We used a variety of UML notations to specify EMOO before beginning implementation, including use case diagrams showing the uses of the system, class diagrams showing the relationship between classes, sequence diagrams showing the activities in the system, and the deployment diagram showing the architecture of the system. Some of these diagrams are included later in this thesis. UML can help to analyze the needs correctly, define a precise high-level design, record design decisions, communicate with others, and minimize errors and misunderstanding.

3.7 Why Rational Rose?

There are many UML tools, such as Rational Rose [Rational], Visual UML [Visual Object], System Architect [Popkin], Visio [MicrosoftVS], and TogetherJ [Together]. Some of them are simple drawing tools, others are complex CASE (Computer-Aided Software Engineering) tools. Rational Rose is currently most widely used. It provides power for visual modeling, component-based development and round-trip engineering (transforming UML notations into code, and vice versa), along with support for the UML [Rational].

Compared with pencil and paper documents, Rational Rose can quickly and conveniently create, modify, and save UML diagrams, and provide feedback on elementary mistakes such as syntactically incorrect diagrams.

3.8 Conclusion

Integrating all the above technologies and tools, EMOO can be designed and implemented relatively quickly with a reasonable assurance of quality.

Chapter 4 System Specification

This chapter describes EMOO features and functionality from a user's point of view. Since EMOO is a re-implementation and extension of previous project called MUM, most EMOO system specification came from MUM.

4.1 An overview of EMOO

EMOO is a virtual environment with high quality customizable and extendible user interfaces in which the client does as much work as possible to alleviate the load on the server and the network. It also allows users to register their interest (subscribe) in events occurring in the emulated universe and obtain automatic notification when these events occur.

4.2 Basic Principles

EMOO has the following basic MOO features:

- Support for multiple users: EMOO allows any number of users to use the program at the same time and perform typical MOO operations such as communication, navigation, and contribution to the evolution of the environment.
- Network-based operation: Users connect their clients to a server running a universe on the Internet.

- Client-server architecture: EMOO consists of two main parts: Servers animating "universes" in which all avatars, tools and places exist and interact, and clients, through which users control their avatars.
- Support for extendibility: Objects that make up the environment are constructed from templates defined as classes in a programming language. New templates can be created at run-time and instantiated at any time.
- Persistence: A running universe can be saved in a file and reloaded.

EMOO has the following additional features:

- Event-driven operation combined with message passing: Objects understand both messages and events. Operations that users may be interested in are implemented as events. This is the extension of EMOO over MUM.
- Events can be subscribed to: User avatars and any other objects can subscribe to events. When the event occurs, all subscribers are notified.
- Minimization of server and network load: The client does as much work as possible to alleviate the load on the server and the network.
- Multi-Threaded Design: The event dispatcher/event handler in the environment runs in its own thread. Thus, each object's operation is separate from the operation of other objects, and if an operation fails it does not cause the whole system to crash.
- Powerful client-side user interfaces: Users interaction with a universe is via a GUI, which provides access to EMOO's general functionality. Users only need to enter clearly identified values and click a button to execute an operation.

- Multiple interconnected universes: Users can create any number of universes. The management of multiple connected universes is mediated by a metaserver.
- Off-line operation: Users can operate EMOO without being connected to the network. If a user chooses the "Local" mode, a local universe is created and the user is connected to it.

4.3 EMOO: High-level Use Cases

This section first lists the most important categories of operations, and then subdivides these categories into individual operations.

4.3.1 Major Use Case Categories

The major usages of EMOO are summarized in Figure 4-1.

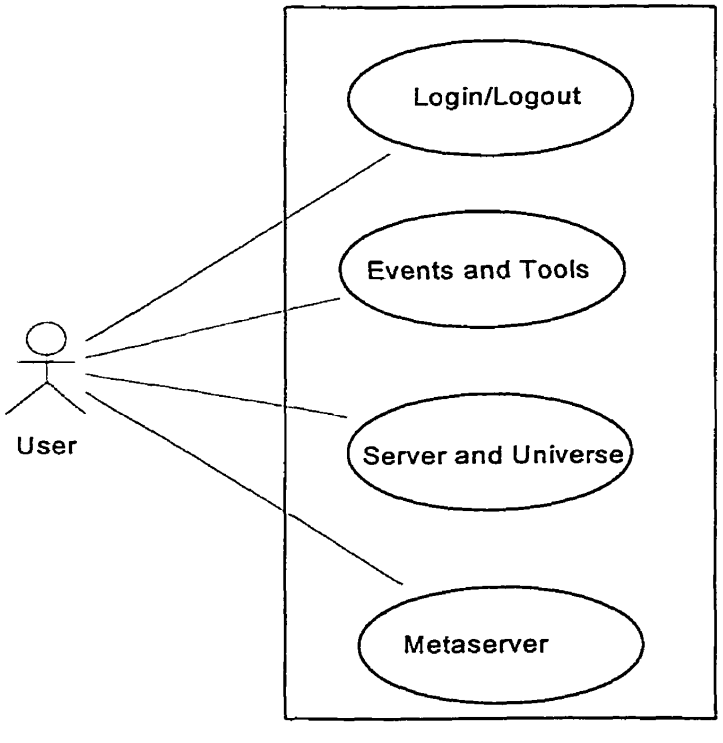

Figure 4-1. Major use case diagram

4.3.2 Category 1 – Login and Logout

This category describes events and actions related to user logins and logouts. Figure 4-2 depicts the use cases graphically and the following sections provide detailed descriptions.

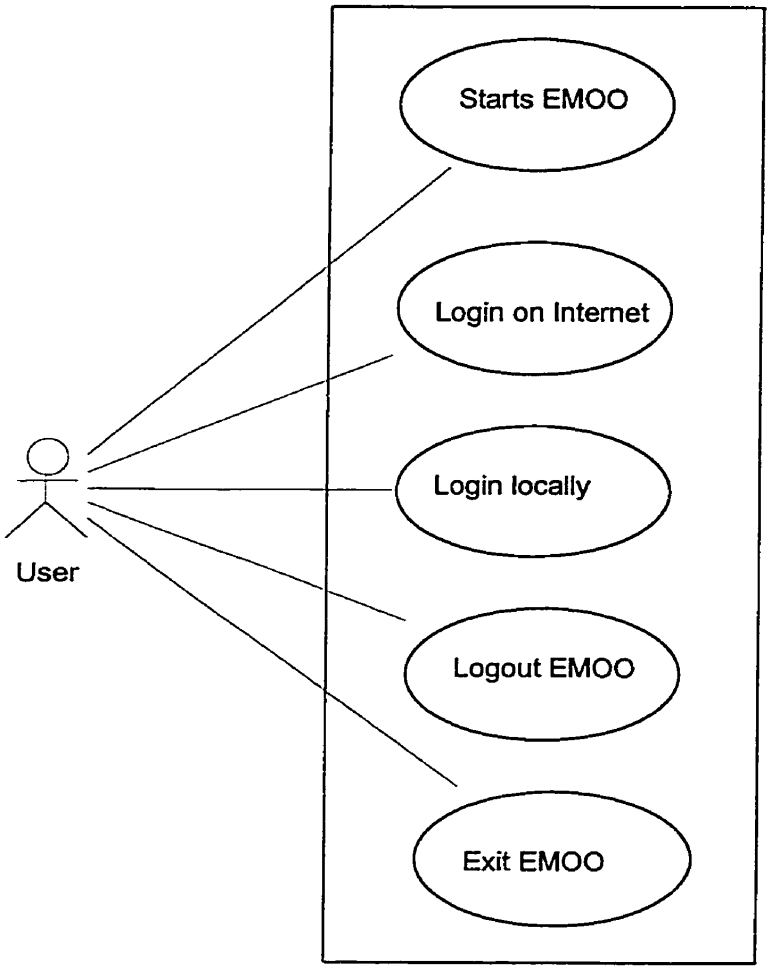

Figure 4-2. Login/Logout use case diagram

4.3.2.1 User starts EMOO.

1. User types "java EMOOLauncher" on command line.
2. Program opens EMOO launcher window.

4.3.2.2 User logs in on Internet as registered user (administrator or agent) or guest.

1. User clicks Remote radio button and clicks Login.
2. Program pops up a list with all available universes.

3. User chooses a universe and clicks Select button.
4. Program requests user name and password.
5. User enters user name and password (if required), and clicks OK.
6. Program displays whether user logged in successfully or not.
7. User opens UniTool. If the user was on EMOO before, he or she is located in the last location if possible. If the user was never logged in or if the last location is inaccessible, he or she is located in the entry room.

4.3.2.3 User logs in locally as registered user (administrator or agent) or guest.

1. User clicks Local radio button and clicks Login.
2. Program starts local server and requests user name and password.
3. User enters user name and password (if required), clicks OK.
4. Program displays whether user logged in successfully or not.
5. User opens UniTool with user located in the last local location (if any) or the entry room.

4.3.2.4 User logs out.

1. User clicks Logout in launcher.
2. Program records current location of avatar.
3. Program disconnects EMOO launcher from the server.

4.3.2.5 User exits.

1. User clicks Exit in launcher.
2. Program requests confirmation.
3. User confirms.
4. Program disconnects EMOO launcher from the server and closes the launcher.

4.3.3 Category 2 – Events and Tools

This category describes the sequence of events and actions related to the use of EMOO tools, command execution and event handling.

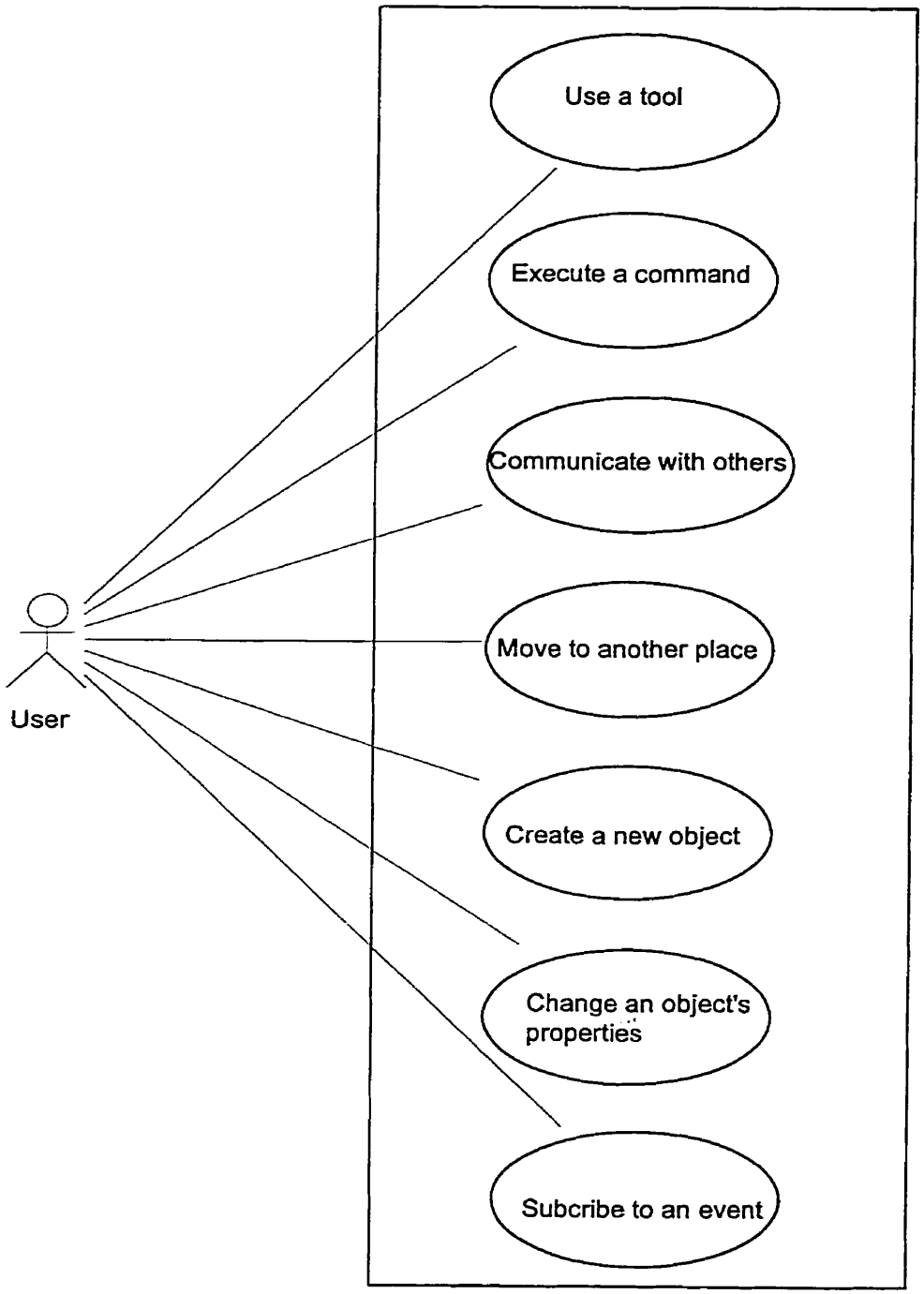

Figure 4-3. Events/Tools use case diagram

4.3.3.1 User opens a tool listed in EMOO launcher window.

1. User selects a tool from the Tool menu in launcher window.
2. Program opens the selected tool's user interface.

4.3.3.2 User executes a command of an object displayed in UniTool.

1. User selects an object.
2. User selects the command from the Action menu in UniTool window.
3. Program executes command.

4.3.3.3 User subscribes to an event for a selected object using UniTool.

1. User selects Events from Subscribe menu.
2. Program displays list of names of subscribable events in UniTool.
3. User selects an event and clicks OK button.
4. Program subscribes user to the event and notifies him or her when the subscribed event occurs.

* Unsubscribing is similar to subscribing but the list now displays all subscribed events that can be unsubscribed.

4.3.3.4 User executes the "say" command in the UniTool.

1. User selects the "say" radio button, and enters text to be communicated followed by <Enter> in the input field.
2. Program displays originator's name (as "John says: " in other occupants' window, or as "You say: " in the originator's window) and the text in the UniTool of all users currently in the same location.

4.3.3.5 User executes the "whisper" command in the UniTool.

1. User selects the "whisper" radio button, chooses the target person from the list of occupants, and enters text to be communicated followed by <Enter> in the input field.
2. Program displays originator's name (as "John whispers to " in the target person window or as "You whisper to " in the originator's window) and the text in the UniTool of these two users who are whispering.

4.3.3.6 User moves to another location.

1. User selects Doors in the ComboBox in UniTool.
2. Program lists all doors in the current location.
3. User selects a destination from the door list and clicks the Go button.
4. Program transfers user's avatar to the selected place.
5. Program updates the list of occupants in the original location and in the new location and notifies all affected users.

4.3.3.7 User creates a new place.

1. User opens the Creation tool from the launcher.
2. User builds a new place and enters its properties.
3. Program creates a new place with a door to the current place and updates the door list of all occupants of the place.

4.3.3.8 User creates a new object in a place.

1. User opens the Creation tool from launcher.
2. User selects the type of an object to create.
3. User inputs properties of a new object and clicks Create button.
4. Program creates the object and updates the contents of this place and notifies all affected users.

4.3.3.9 User changes an object's properties.

1. User opens the Property tool from launcher.
2. User chooses an object from a list.
3. User changes properties and clicks Accept button.
4. Program attempts to change the object's properties and displays whether the operation is successful or not. This operation may be unsuccessful if the user is not the owner of the object.

4.3.4 Category 3 – Server and Universe

This category describes events and actions that occur when an administrator starts or shuts down a server and manages a universe.

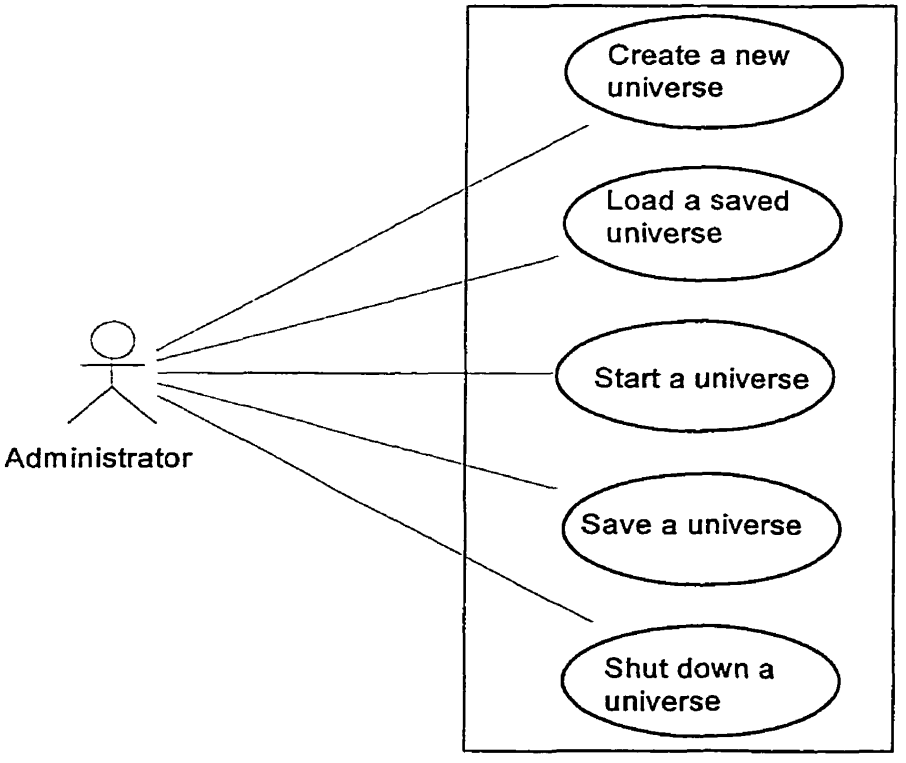

Figure 4-4. Server/Universe use case diagram

4.3.4.1 Administrator creates a new universe and starts the server.

1. Administrator types "java EMOOUniverseManager" on command line.
2. Program opens EMOO Universe window.
3. Administrator selects "New" or "New Universe" from menu.
4. Administrator enters a name for the new universe.
5. Administrator clicks Start button.
6. Program creates a new universe and starts the server.
7. Program contacts a metaserver and changes the status of the metaserver's list.

4.3.4.2 Administrator loads a saved universe and starts the server.

1. Administrator types "java EMOOUniverseManager" on command line.
2. Program opens EMOO Universe window.
3. Administrator selects "Load" or "Load Universe" from menu.
4. Administrator enters the name of the saved universe.
5. Administrator clicks Start button.
6. Program loads the universe and starts the server.
7. Program contacts a metaserver and changes the status of the metaserver's list.

4.3.4.3 Administrator saves a universe.

1. Administrator clicks on the Save button in the EMOO Universe UI.
2. Program opens a dialog window.
3. Administrator enters a file name for the universe and clicks the OK button.
4. Program saves the universe.

4.3.4.4 Administrator shuts a server down.

1. Administrator clicks Stop button in the EMOO Universe window.

2. Program shuts down the server.

3. Program changes the status of metaserver's list.

4.3.5 Category 4 – Metaserver

This category describes the events and actions related to the management of a metaserver.

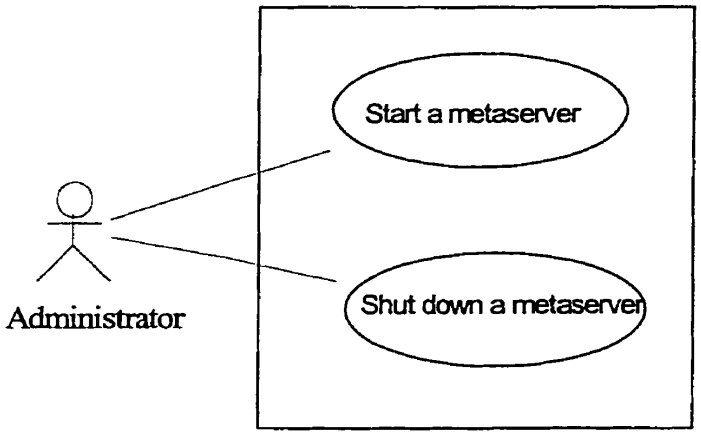

Figure 4-5. Metaserver use case diagram

4.3.5.1 Administrator starts a metaserver.

1. Administrator types "java EMOOMetaServerManager" on command line.

2. Program opens EMOO Metaserver window.

3. Administrator clicks Start button.

4. Program starts a metaserver.

4.3.5.2 Administrator shuts a metaserver down.

1. Administrator clicks Stop button in the EMOO Metaserver window.

2. Program shuts a metaserver down.

Chapter 5 Design Overview

This chapter gives a general overview description of EMOO design.

5.1 The big picture

Figure 5-1 shows the main parts of EMOO architecture. Users can contact a metaserver and connect to universes using their clients via network.

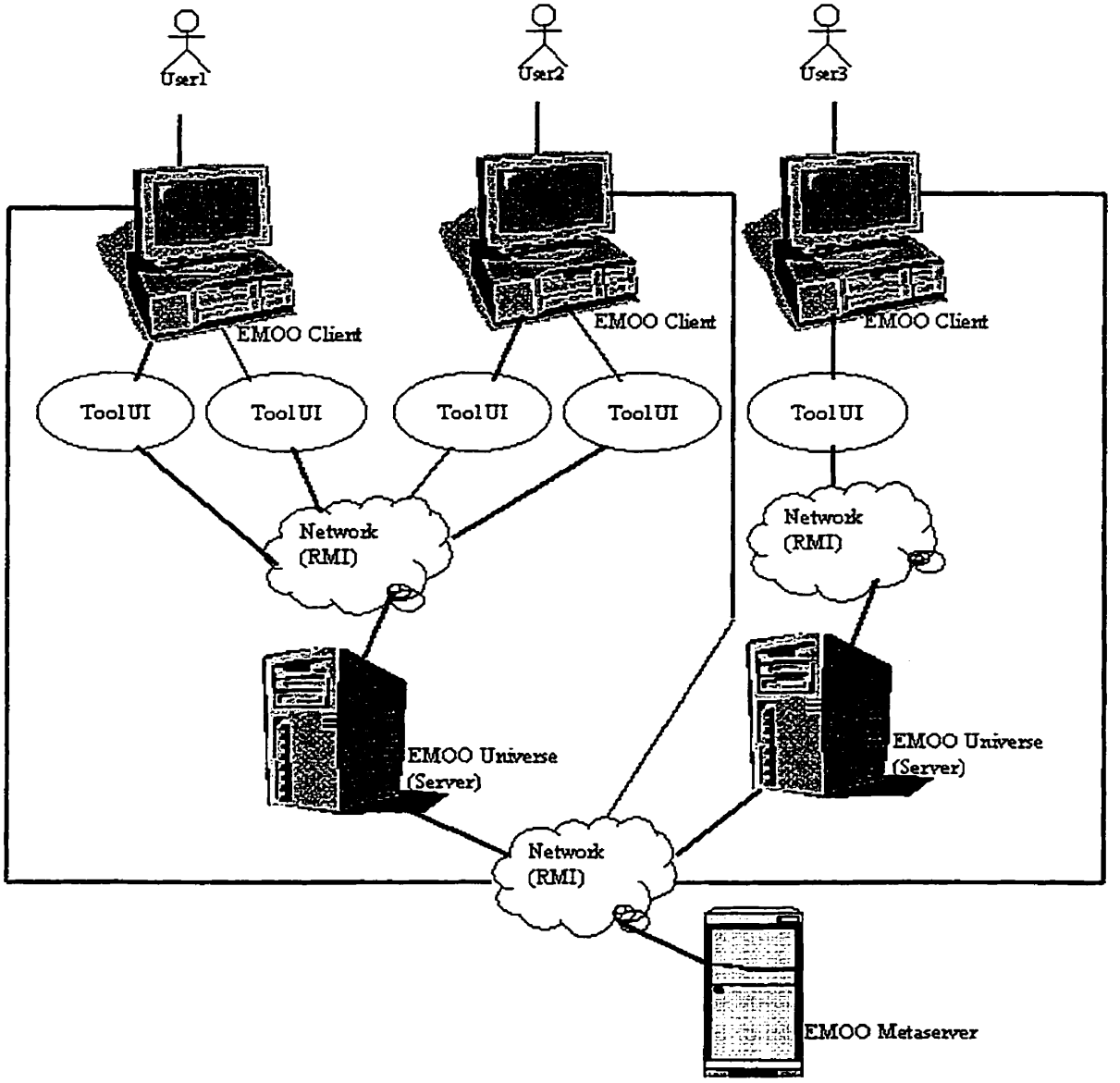

Figure 5-1. EMOO big picture

5.2 Overall Architecture

The high-level UML deployment diagram (Figure 5-2) shows the layers of EMOO architecture.

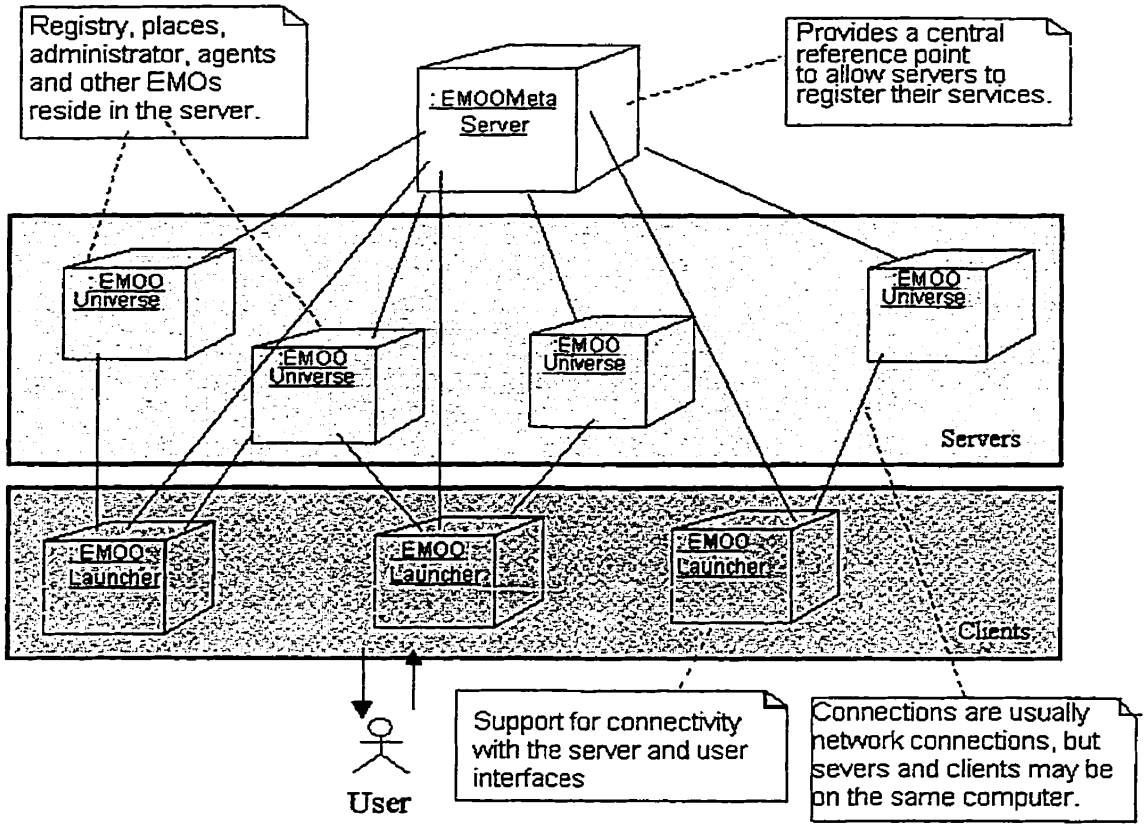

Figure 5-2. EMOO's overall architecture

As all MOOs, EMOO is a client-server application but has one additional layer – metaserver, which will be explained later.

An EMOO system may consist of several EMOO universes and one or more EMOO metaserver. Metaservers keep track of available universes all over the network as well as their status such as IP address, name and running state. When a universe is created, a

record is added to the server list of the metaserver and whenever the universe starts or stops the record is changed correspondingly. Universes are the places where EMOO objects reside. A running universe accepts client connection requests submitted by clients via EMOO launchers.

To clarify EMOO design, the following sections, introduce the main concepts and principles of operation, and define the terminology. The details are presented in Chapter6.

5.3 Principles of operation

When a user starts EMOO from his or her client machine and connects to the Internet, EMOO first contacts a metaserver. The metaserver returns a list of all universes with information about those universes that are currently connected, including their IP addresses. The user then selects a running universe and logs in, which establishes a direct connection to the selected universe. The selected universe checks whether the user is authorized to enter. After login the user is ready to function in the universe.

Objects in EMOO understand both messages and events. Only some messages that users may be interested in are implemented in the form of events. Most interactions between objects in EMOO are handled directly by messages, that is to say, an object communicates with other objects by sending messages. Message passing is efficient because messages are automatically handled by Java system, but it is difficult to implement message subscribing.

Some messages that an object understands such as an avatar "go" message, may be important to users who may want to know when the message is sent to the object. These messages are implemented via events. Events are easy to trace because they are processed by event handlers. Events also make it easy to implement operations consisting of complex sequences possibly involving asynchronism (This is currently unavailable in EMOO).

Each event has a name, a receiver and parameters. The event name identifies the type of the event, the event receiver is the id of the object that will receive the event, and parameters contain necessary information to execute the event. Each universe has a single central event dispatcher that is responsible for dispatching events. When dispatching an event, the dispatcher creates an event handler and binds it with the receiver objects. Event handlers are actually Java threads. Each EMOO object has an event dictionary that contains the definition of all events it understands. Event handlers get event definitions from event dictionaries, translate events into messages, and send them to target object. An event handler thread ends when the event is completed. Compared to message passing, it takes longer time to execute events and requires more complicated code.

To keep track of all EMOs (EMOO Objects, see Figure 6-3) in the universe, the universe has a registry containing a hashtable matching EMOs and their unique ids. All references to EMOs in the operation of the universe including messages and events, and in the communication between the client and the server are in terms of these ids.

The interaction between the user and an object may take one of two forms. One possibility is that the EMO has a specialized interface, the other is that it does not. In the first case, users communicate with the object via the interface and do not have to use any commands. If an object does not have its own specialized user interface or if the interface cannot be used for a particular operation, the user must communicate with the object via commands. This is implemented in a somewhat similar way as in Jersey but with a better user interface: The user selects an object in the UniTool, and chooses a command from the Action menu (see Figure 5-3). Clicking a command causes UniTool to send the command to the corresponding object. At present, EMOO commands are very simple that do not require parameters. If they did, a dialog window could be generated on the fly as in MUM.

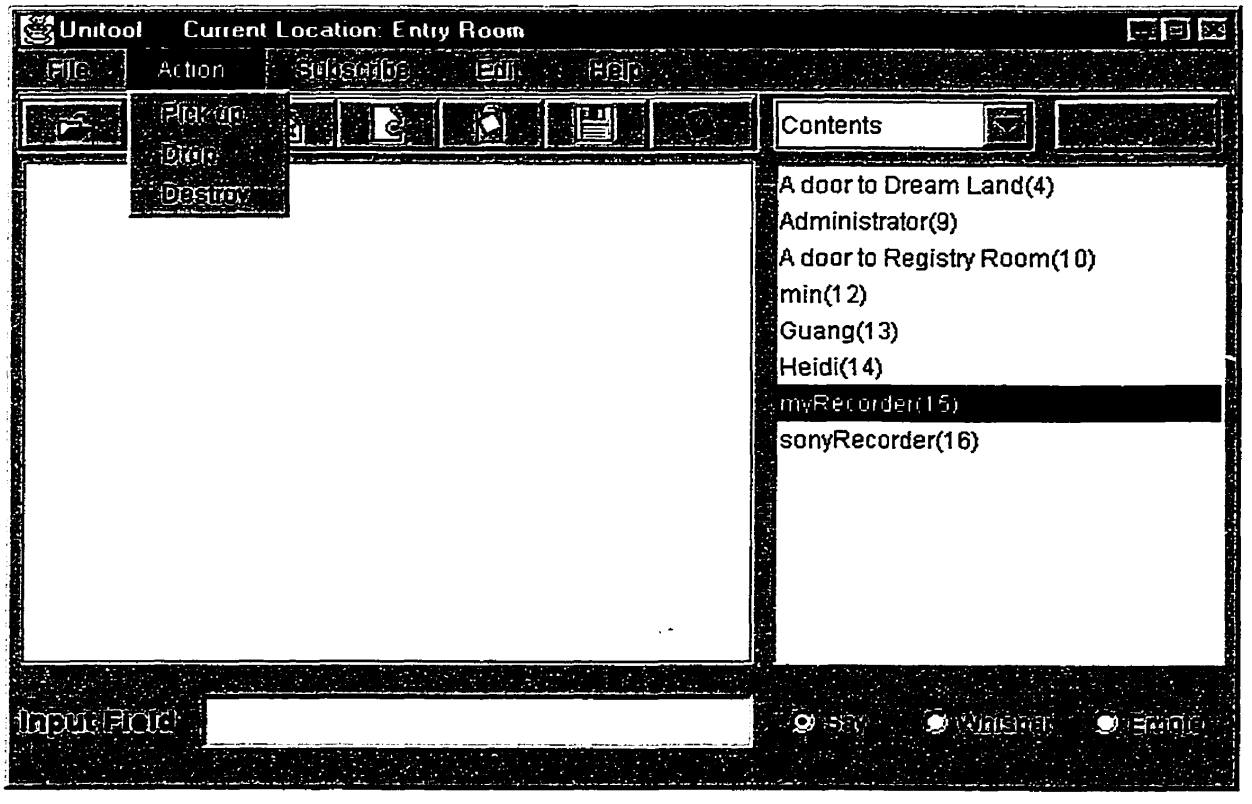

Figure 5-3. Executing a command on a selected object

After this general overview of EMOO design, Chapter 6 will describe the individual subsystems in more detail.

Chapter 6 Detailed Design

This chapter explains the details of EMOO subsystems and their classes and methods.

6.1 Network Layer

6.1.1 Overview

The network layer implements communication between clients and servers using Remote Method Invocation technology. When a user connects to a server, the server first checks its "clients" table to find whether this user is already connected to this server. If a user with that name is already logged in, it does not allow another log in with the same name. If the user name is available, the server checks its "agents" table to find whether this user has an avatar on the server and if there is one, it performs password authentication. If the user has no avatar, the server creates a new avatar, and adds a new record to its "agents" table. If the user connects to his avatar successfully, the server adds a new record in "clients" table. When the user logs out, the server removes the user's record from the corresponding table. The login operation is described by the UML activity diagram in Figure 6-1.

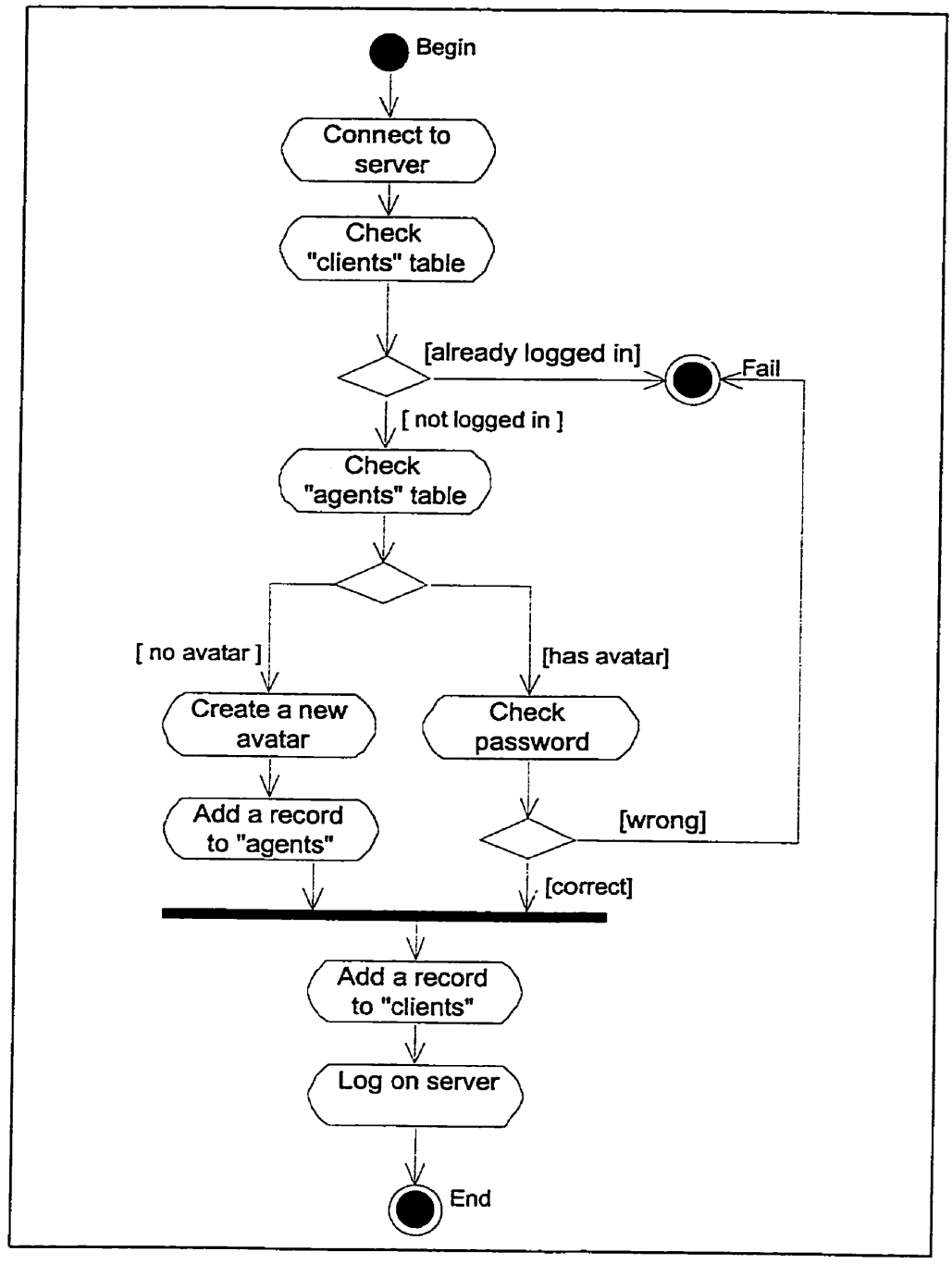

Figure 6-1. Activity diagram of login operation

6.1.2 Classes and Interfaces

The network layer defines three interfaces: RMIMetaServerInterface, RMIServerInterface and RMIClientInterface. They all extend the Remote interface in RMI package and are implemented by classes RMIMetaServer, RMIServer and

RMIClient. Classes EMOOMetaServerManager, EMOOUniverseManager and EMOOLauncher are used to manage metaservers, universes and clients. The relationships between these classes and interfaces are shown in Figure 6-2:

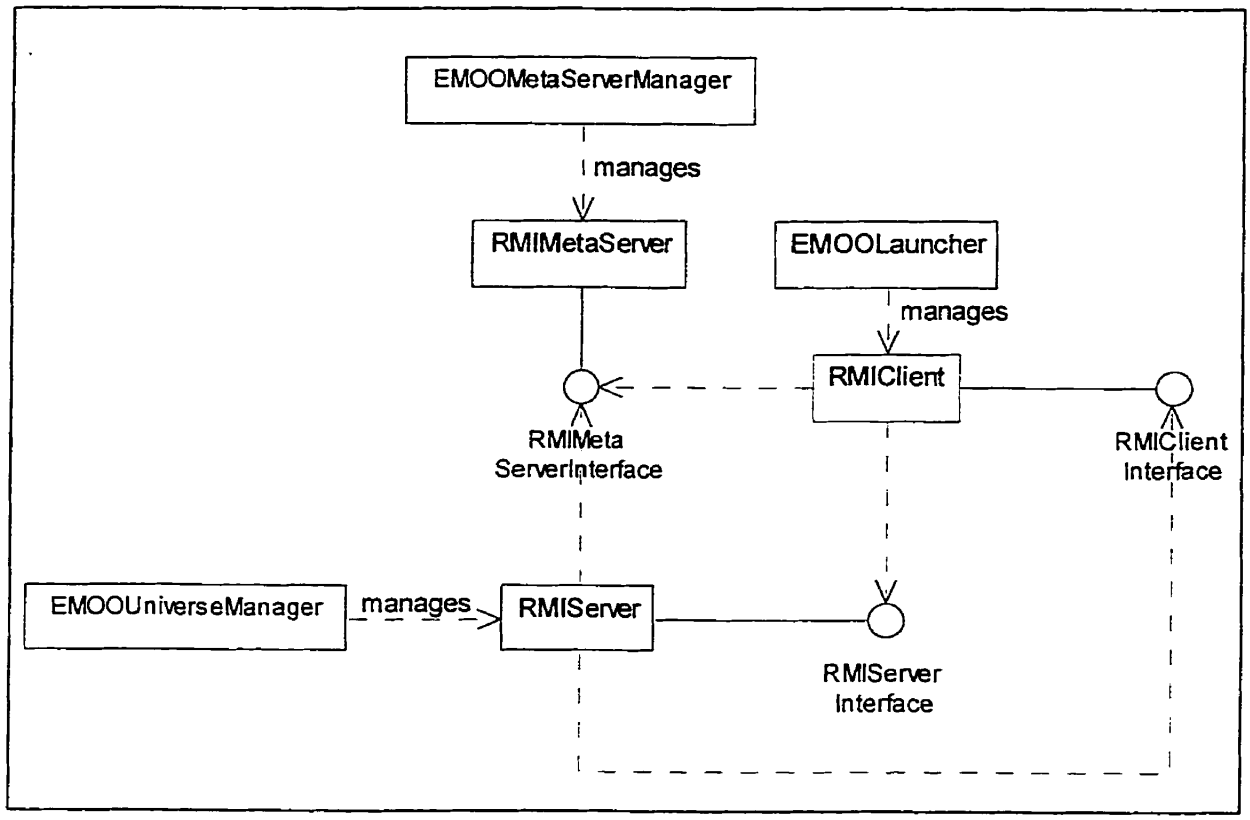

Figure 6-2. Network Layer class diagram

The following is a brief description of these and other essential classes. We start with a description of domain classes.

RMIMetaServer

Responsible for providing information on available universes.

Superclass: UnicastRemoteObject

Main Instance Variables:

serverList <Vector> records the information of all universes that register with it

Main Methods:

serverRegistry(String serverName,String ipAdd) registers server on the serverList

updateList(String serverName,String ipAdd,String status) updates server status in serverList

destroy(String serverName,String ipAdd) deletes a server from serverList

RMIServer

Responsible for mediating the communication between users and their avatars.

Superclass: UnicastRemoteObject

Main Instance Variables:

clients <Hashtable> records usernames with their RMI connection

agents <Hashtable> records usernames with their corresponding avatars

Main Methods:

login(RMIClientInterface c, String userName, String password) receives clients login

logout(String userName) lets clients logout

broadcast(String target, String msg) sends information to clients

RMIClient

User's client responsible for connecting to a metaserver to find all available universes and connecting to a certain universe.

Superclass: UnicastRemoteObject

Main Instance Variables:

userName <String> client login name

userPassword	<String>	client login password
server	<RMIServerInterface>	reference to a RMIServerInterface

Main Methods:

connect(String host)	connects to server
disconnect()	disconnects from sever
notify(String msg)	receives information coming from server

The remaining components of the network layer are as follows:

EMOOMetaServerManager

A metaserver manager with GUI responsible for starting and stopping a metaserver.

Superclass: JFrame

Main Instance Variables:

metaServer	< RMIMetaServer>	referenee to a metaserver

Main Methods:

start()	starts a metaserver
stop()	stops a metaserver

EMOOUniverseManager

A universe manager with GUI responsible for starting and stopping a server, and responsible for creating, loading and saving a universe.

Superclass: JFrame

Main Instance Variables:

currentUniverse	<Universe>	reference to the currently running universe

Main Methods:

newUniverse()	creates a new universe
loadUniverse()	loads a universe from a saved universe file
saveUniverse()	saves a universe into a file
startUniverse(Universe aUniverse)	starts a server that hosts the universe

EMOOLauncher

Instantiates a client, and provides a tool for users to operate in EMOO system. Additional details will be explained later.

Superclass: JFrame

Main Instance Variables:

client	<RMIClient>	reference to a client
currentServer	< ServerIdentity>	reference to a currently running server

Main Methods:

getServerList()	gets a list of available servers
remoteLogin()	logs in a remote universe
localLogin()	logs in a local universe

6.2 Universes

6.2.1 Overview

Universes themselves are EMOs just like the objects that reside in them. This is because making universes operate on the same principle as everything else provides uniformity of design and operation, and allows users to subscribe to events. When a universe is created,

several fundamental EMOs are also created automatically. These include the following ones: a "Registry Room" with a registry that is responsible for instantiating EMOs and holding records of all objects in the universe, an "Engine Room" with an avatar representing the administrator, normally the creator of the universe. Once the universe is created, users may start expanding it by adding new users, new places, new objects, and new tools. A universe class diagram is shown in Figure 6-3.

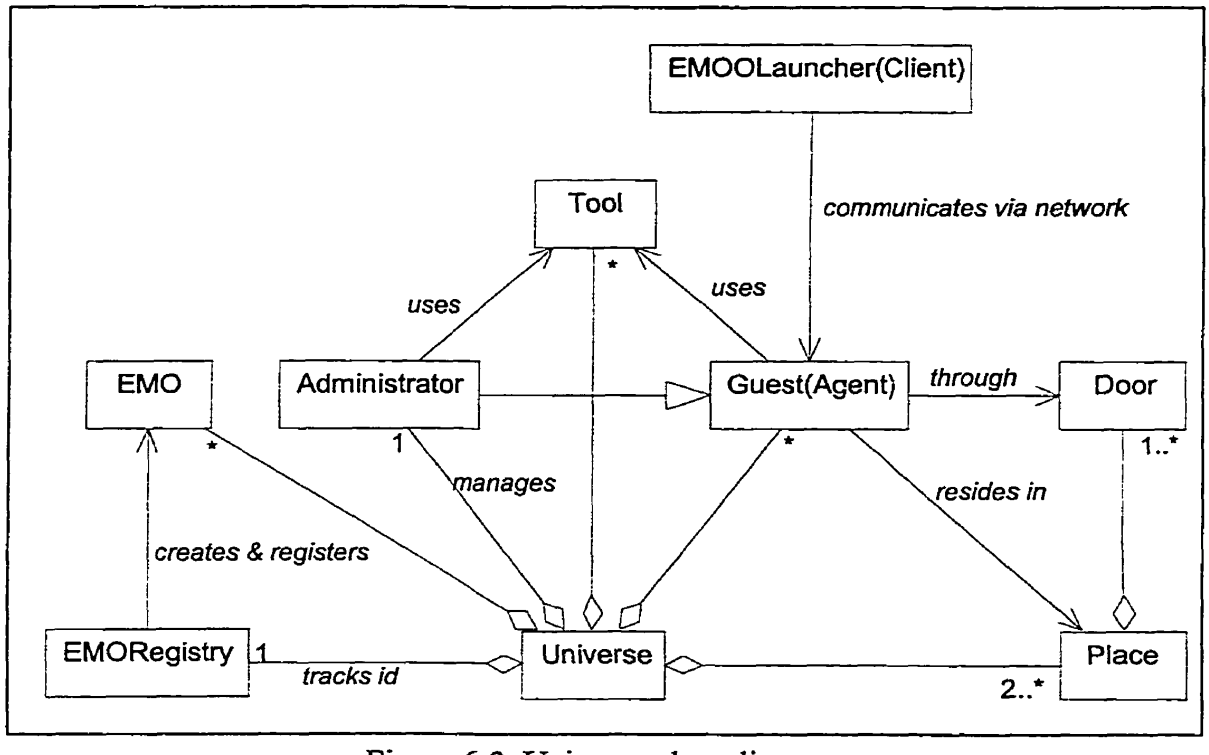

Figure 6-3. Universe class diagram

6.2.2 Classes

EMO

EMO is an acronym for EMOO Object. This is the abstract super class of all objects in EMOO. Its main instance variables and methods and generalization relationship with other EMOs are described in the diagram Figure 6-4.

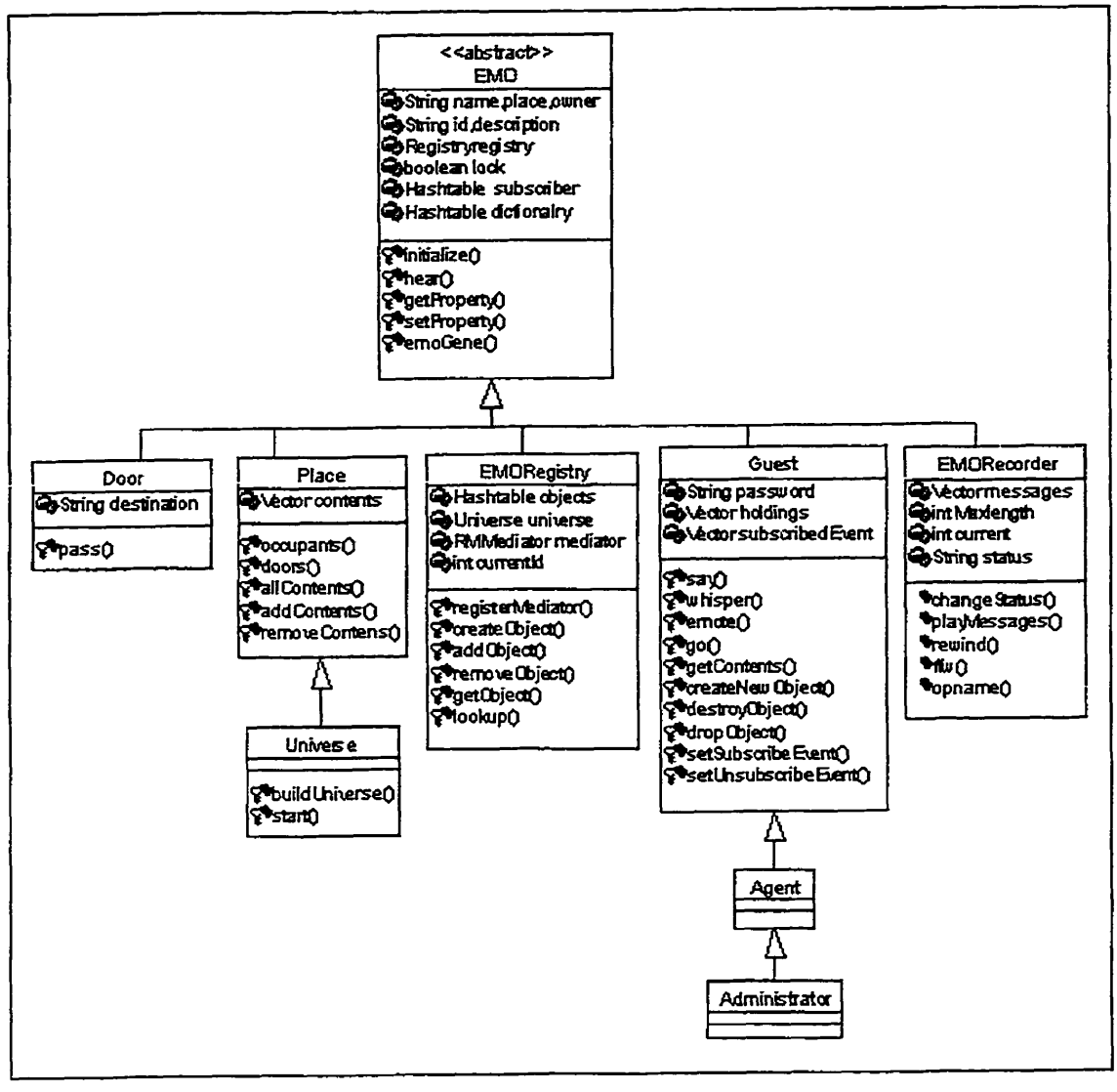

Figure 6-4. EMO class diagram

Superclass: Object

Main Instance Variables:

id	<String>	id of the EMO
name	<String>	name of the EMO
place	<String>	location of the EMO
owner	<String>	id of the EMO's owner (initially the EMO's creator)
lock	<Boolean>	permission to be changed by other EMOs
description	<String>	description of the EMO
subscribers	<Hashtable>	subscribers to a certain event
dictionary	<Hashtable>	maps events to corresponding messages
registry	<EMORegistry>	reference to a universe's registry

Main Methods:

initialize()	initializes the values of instance variables
hear(String s)	receives notification messages
getProperty()	returns an EMO's changeable properties
setProperty(Hashtable p)	sets new property values
emoGene()	returns an EMO's gene

Note: An EMO's gene is a table that contains essential information representing the EMO in a compact way. It is used when the universe is being saved in a file or loaded from a file. The EMO can be rebuilt from its gene.

EMORegistry

Maintains a list of all EMOO objects that exist in the EMOO universe. Responsible for creating new EMO objects.

Superclass: EMO

Main Instance Variables:

objects	<Hashtable>	id -> object associations of all objects in the universe
universe	<Universe>	place where the registry and other EMOs reside
server	<RMIServer>	reference to the server of the universe
currentId	<int>	id of the next new object

Main Methods:

startRMI()	starts the RMIServer of the universe
createObject(String className, String objectName, String place, String description)	creates a new EMO object
addObject(String id, EMO o)	adds a new object to the "objects" table
removeObject(String id)	removes an object from the "objects" table
getObject(String id)	returns an object specified by its id
lookup(String name)	returns an object specified by its name
objectToGene()	converts an object in the registry to its gene
geneToObject(Hashtable gene)	reconstructs an object from its gene

Place

Represents a place in the EMOO universe.

Superclass: EMO

Main Instance Variables:

contents	<Vector>	contains ids of all objects in this place

Main Methods:

occupants()	returns all avatars in this place
doors()	returns all doors in this place
allContents()	returns all objects in this place
addContents(String id)	adds an object to this place
removeContents(String id)	removes an object from this place

Universe

Subclass of Place, contains all EMOO objects. Responsible for building constructs such as Registry Room, Entry Room.

Superclass: Place

Main Instance Variables:

no instance variable in this class.

Main Methods:

buildUniverse()	builds fundamental EMOs when a universe is first created

Door

Represents a uni-directional door in a place. Used by avatars to go to another place.

Superclass: EMO

Main Instance Variables:

destination	<String>	place to which this door leads

Main Methods:

pass(String guestId)	passes an avatar to the destination place

Guest (Agent, Administrator)

Guest, Agent and Administrator define avatars representing users in a universe. They perform actions requested by users. Agent and Administrator are subclasses of Guest. The difference between Guest, Agent and Administrator is that they have different authorities. Administrator has the highest authority and can perform all operations, including creation of agents and guests. Agents can perform all operations except for creating agents and guests. Guests can not create and destroy agents, guest, or objects.

Superclass: EMO

Main Instance Variables:

password	<String>	password of the user connecting to this avatar
holdings	<Vector>	ids of objects held by the avatar
subscribedEvent	<Vector>	all events that the avatar is subscribed to

Main Methods:

say(String s)	implements "say" operation
whisper(String toName, String msg)	implements "whisper" operation
emote(String s)	implements "emote" operation
go(String door)	implements "go" operation
createNewObject(String className, String objectName, String description)	creates a new object
destroy(String id)	executes "destroy" command

dropObject(String id)	executes "drop" command
holdObject(String id)	executes "hold" command
getEvents(String id)	gets events that can be subscribed to
setSubscribeEvent(String eventName)	subscribes to an event
setUnsubscribeEvent(SubscriberIdentity si)	unsubscribes a subscribed event

6.3 Events and Event Handling

6.3.1 Overview

EMOO events are objects representing action requests. They are handled by EMOO event handling mechanism. The universe contains one central event dispatcher, which has an event queue, and is responsible for extracting events from it and activating event handlers to execute them. Each EMO has a "dictionary" table that translates events to corresponding messages. Using Java Reflection API, event handler can execute an event by dynamically invoking methods corresponding to their names at runtime. The relationship between classes in this subsystem is described in Figure 6-5:

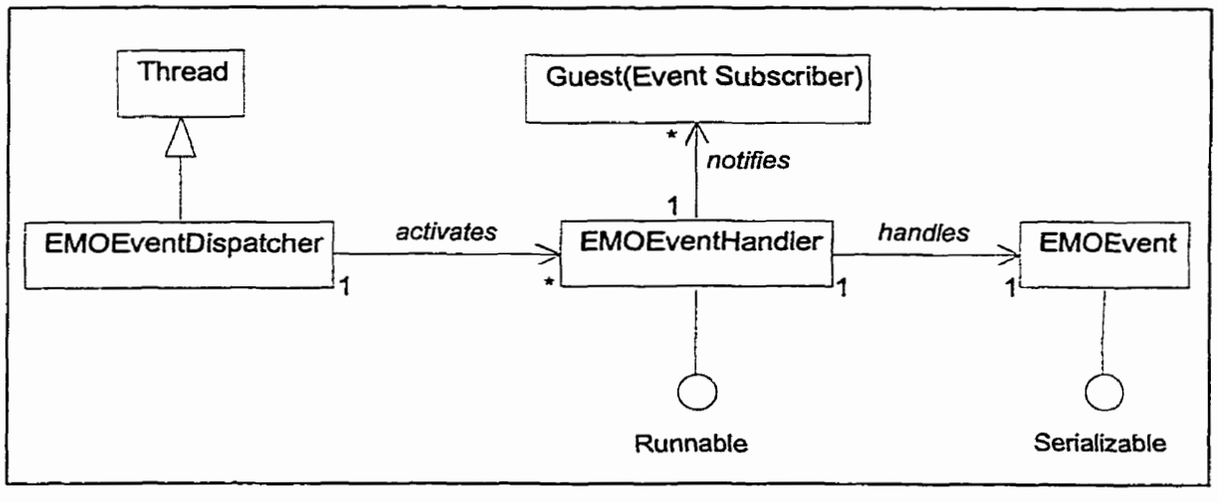

Figure 6-5. Event handling class diagram

6.3.2 Classes

EMOEvent

Represents an event that can be subscribed to. In order to be able to transport event objects on the network, it implements java.io.Serializable interface.

Superclass: Object

Main Instance Variables:

name	<String>	name of event
target	<String>	object that the event is sent to
parameters	<Vector>	parameters necessary to execute the event

Main Methods:

setParameters(Vector v)	sets parameters needed to execute the event

EMOEventDispatcher

Responsible for dispatching the events in its event queue and activating EventHandlers to execute them. It is subclass of Thread class to implement multithreaded control, thus it has its own thread to perform its operation independently without disturbing any other running application.

Superclass: Thread

Main Instance Variables:

eventQueue	<Vector>	stores events needed to be handled

Main Methods:

addEvent(EMOEvent e)	adds incoming event to eventQueue
dispatch()	dispatches the events in its event queue and

	forkes EventHandler to executing events.
run()	executes "dispatch" while thread is running

EMOEventHandler

Responsible for handling EMO events and notifying subscribers. It implements java.lang.Runnable interface, so that it can be runnable within a "dispatch" thread.

Superclass: Object

Main Instance Variables:

event	<EMOEvent>	event handled by the event handler

Main Methods:

run()	handles EMO event and notifies subscribers when it is executed

6.3.3 Event Handling and Message Passing

EMOO events are handled by EMOO's event dispatcher and event handler. Messages do not use the dispatcher and event handler. The following sequence diagrams show the implementation on two examples. Figure 6-6 describes the operation of event "go". Figure 6-7 describes the operation of message "say".

6.3.3.1 Operation of events

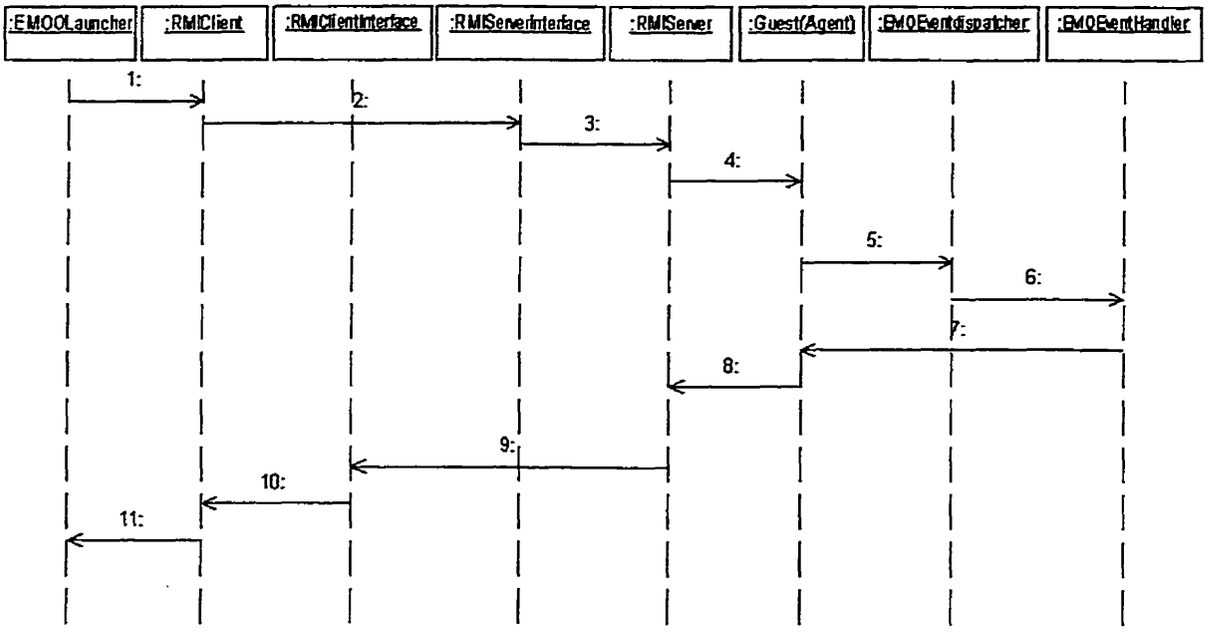

Figure 6-6. Sequence diagram of the "go" event

Simplified details of individual steps of the "go" event are as follows:

1. User chooses a door in the door list and clicks the "Go" button on EMOOLauncher. The "go" message is sent to RMIClient.
2. RMIClient invokes the "go" method on the remote server through RMIServerInterface.
3. RMIServer executes RMIServerInterface's "go" method.
4. RMIServer sends the "go" message to the corresponding agent.
5. Agent sends the "goEvent" to the EMOEventdispatcher.
6. EMOEventDispatcher dispatches the "goEvent" to EMOEventHandler.
7. EMOEventHandler dynamically invokes agent "go" method to handle "goEvent" and sends a notification to subscribers. (This is not shown due to lack of space.)

8. Agent executes the "go" operation and returns a feedback message to RMIServer
9. RMIServer invokes the "update" method on the remote client through RMIClientInterface.
10. RMIClient executes RMIClientInterface's "update" method.
11. EMOOLauncher UI receives the update message and displays it.

6.3.3.2 Operation of messages

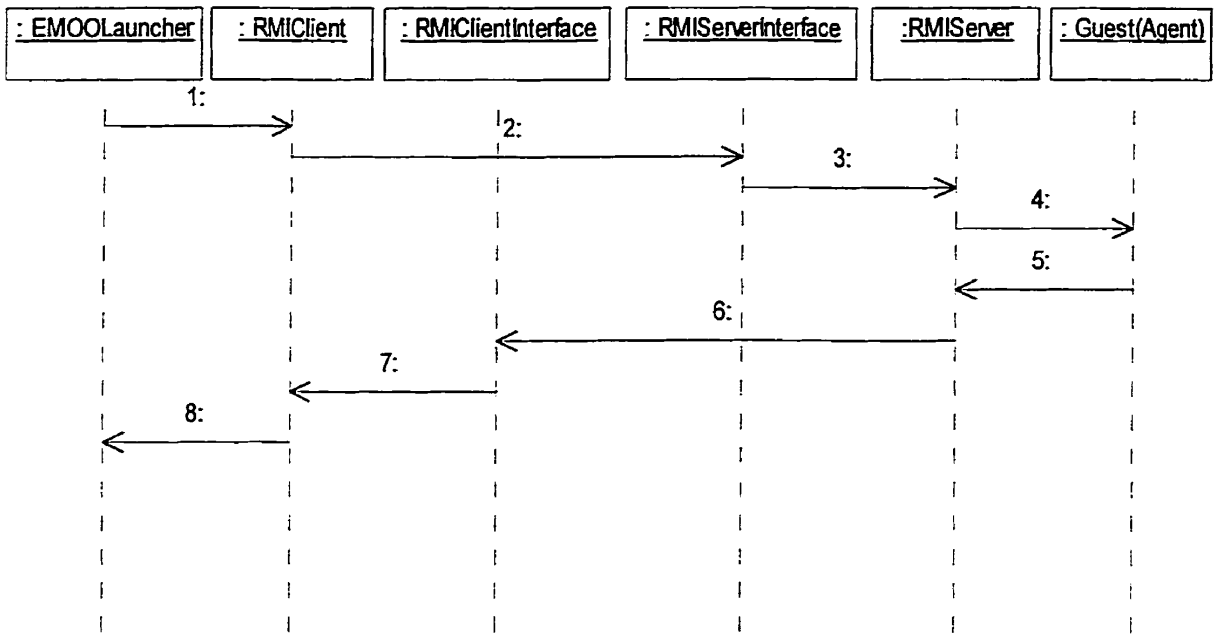

Figure 6-7. Sequence diagram of the "say" message

Details of individual steps in the execution of the "say" message are as follows:

1. User selects "say" radio button on EMOOLauncher UI, inputs a string, and presses return. The "say" message is sent to RMIClient.
2. RMIClient invokes the "say" method on the remote server through RMIServerInterface.

3. RMIServer executes RMIServerInterface's "say" method.
4. RMIServer invokes the "say" method of the corresponding agent.
5. Agent executes the "say" operation and returns the result of "say" to RMIServer.
6. RMIServer invokes the "broadcast" method on the remote client to broadcast the result of "say" through RMIClientInterface.
7. RMIClient executes RMIClientInterface's "broadcast" method.
8. EMOOLauncher receives the broadcast message and displays it.

6.4 Tools

6.4.1 Overview

Tools are special objects whose main purpose is to provide client-side user interfaces. With a tool, users do not need to remember any commands. All operations are executed by clicking appropriate widgets on the UI. Users can add new tools, as they want. Currently, EMOO implements the following tools.

6.4.2 Basic Tools

EMOOLauncher

Allows the user to login, both locally and via Internet, and logout. It also provides access to other EMOO tools. See Figure 8-8 in Chapter 8 for a screenshot of the interface.

UniTool

Provides basic communication and navigation functions. Has a communication part with input and output areas, an entity list area showing objects, doors, occupants and personal

holdings, commands understood by the selected object, and an objects' subscribable events. See Figure 8-14 in Chapter 8 for a screenshot of the interface.

CreationTool

Allows the user to create a new instance of an existing object template and to define its properties. See Figure 8-12 in Chapter 8 for a screenshot of the interface.

PropertyTool

Displays an object's properties and allows the user to edit them if the user is authorized to do so. See Figure 8-13 in Chapter 8 for a screenshot of the interface.

EMORecorder

Records and plays what avatars say and emote in a place. See Figure 8-17 in Chapter 8 for a screenshot of the interface.

Chapter 7 Environment Comparisons

The first part of this chapter compares EMOO with its predecessor MUM [Tomek, 1999]. The second part compares the implementation languages Java and Smalltalk in light of our EMOO and MUM experience.

7.1 EMOO vs. MUM

EMOO is a successor of MUM and their functionality and design have many similarities. There are, however, also differences, mainly in design. This section discusses both the similarities and the differences.

7.1.1 Similarities

Both EMOO and MUM have a metaserver with a registry of all server universes, their status, and Internet addresses. They both support multiple universes and are easily extended. By contacting a metaserver users can select which universe to connect or go to.

As client-server applications, EMOO and MUM do as much work as they can on client side to reduce the server load. Client side applications include tools. Tools are responsible for handling user input and convert it to proper messages or commands understood by objects in server universes. These messages or commands are sent to the objects via network connection. Tools also display information received from the server.

Tools provide friendly user interfaces. Users do not need to type a command or message directly, so there is no need to remember commands and type them in again and again. By selecting items in lists or clicking buttons they can do almost everything that traditional MUD/MOO users can do. The friendly user interfaces of EMOO and MUM make it easier and more interesting to explore the universes. Although the number of available tools is still limited, the concept of a tool provides unlimited possibilities.

Both EMOO and MUM have event objects representing action request. Events are handled by event handler, and can be subscribed and unsubscribed by users. The event handler works in its own thread so that one object's operation will not interfere with other objects' operations.

7.1.2 Differences

As a new design and implementation based on some of MUM's ideas, EMOO has made some improvements, and used some different design and implementation approaches. The most obvious difference is that EMOO is implemented in Java, whereas MUM is implemented in Smalltalk. A detailed comparison of Java and Smalltalk is given in next section.

MUM uses fully event-driven operation and all objects in MUM communicate with others by sending events. Using this mechanism, it is easier to trace events sent by any object. This enables programmers to find bugs in the program quickly. Another advantage of event-driven operation is that users can express their interests in any object

in MUM. However, fully event-driven operation leads to event class explosion: there are more than two hundred event classes in MUM, and some of them have no methods, just provide a uniform format to match the event-processing mechanism of event handlers, and help pass information from the originator to a distant target. This bloats the number of classes, makes code hard to read and difficult to reuse. Another disadvantage of fully event-driven operation is that it creates an overhead in execution. Event handlers consume much more CPU time than message passing, because execution of one event may require hundreds of messages.

In EMOO, objects understand both messages and events. Only those operations that users may be interested in are implemented in the form of events and can be subscribed. Thus, EMOO's code is very succinct and does not require so many event classes. Most of the communication between objects is done by sending messages, so operation is more efficient. Using this hybrid mechanism, EMOO can keep most of MUM's good features, improve efficiency and simplify implementation.

Each MUM object has its own event handler and event process. The handler works independently without disturbing any other objects, so if there is an exception, only the object that caused the exception will be affected. This feature enhances the usability of MUM and makes it less error-prone. However, it leads to too many event handler objects and too many processes running at the same time, which increases system resource consumption and slows down execution.

In EMOO, a central event dispatcher works as a post office and is responsible for sending events to destination objects. All outgoing events are sent to the dispatcher for

redirection. When dispatching an event to the destination object, the event dispatcher forks a thread and creates an event handler object to handle this event. Using this event handling mechanism, EMOO can have good and stable performance without too many handler objects and running processes.

On the client side, each MUM tool has a tool base that is responsible for translating messages to events and vise versa. Although tool bases do not take any server time to do these translations, developers have to write corresponding tool base code.
In EMOO, tools communicate with objects in remote universes directly via RMI, which greatly simplifies developers' work and provides an efficient way for cooperation between tools and objects.

In MUM, the client contains a small universe that is constructed every time the client is opened. This client-side universe is the environment in which the components that make up the client communicate. This client-side universe is necessary because MUM is fully event-driven. EMOO is not fully event-driven, and there is no need for a client-side universe.

Events in MUM not only can be subscribed to, but also can be suspended and resumed. This makes it easy to implement asynchronous activities with complex state transitions. The MUM event handler is implemented as a finite state automaton (FSA) interpreting a state diagram describing all possible ways in which the event can be executed, so that the

execution of an event can be suspended at any state and resumed. EMOO does not use FSA mechanism, and EMOO events can not be suspended.

In MUM, users can travel from one universe to another with their possessions. Users can also download tools residing in the server universe when using them if they do not have corresponding code. These functions greatly extend the usability of MUM. EMOO also could but currently does not implement these functions.

7.2 Java vs. Smalltalk

The comparison of Java and Smalltalk is a sensitive topic [Chimu], partially because Java poses a greater challenge to Smalltalk than any other language. The following comparison is based on some objective data and my subjective evaluation as a programmer with non-trivial but limited experience in both Java and Smalltalk. The comparison here is based on VisualWorks 3.0 [VisualWorks] that was used to implement MUM and Java 2 that was used to implement EMOO.

7.2.1 About Java and Smalltalk

Java was developed by a team led by James Gosling at Sun Microsystems in the early 90s [Horton, 1999]. It was originally designed for writing programs for small computers embedded in consumer electronics appliances, such as microwave ovens. It is a new object-oriented programming language, synthesized from several existing languages, so someone [Chimu] said that "Java is a light-statically-typed, simple version of Smalltalk with the syntax of the C family."

The Figure 7-1 shows the influences that shaped Java:

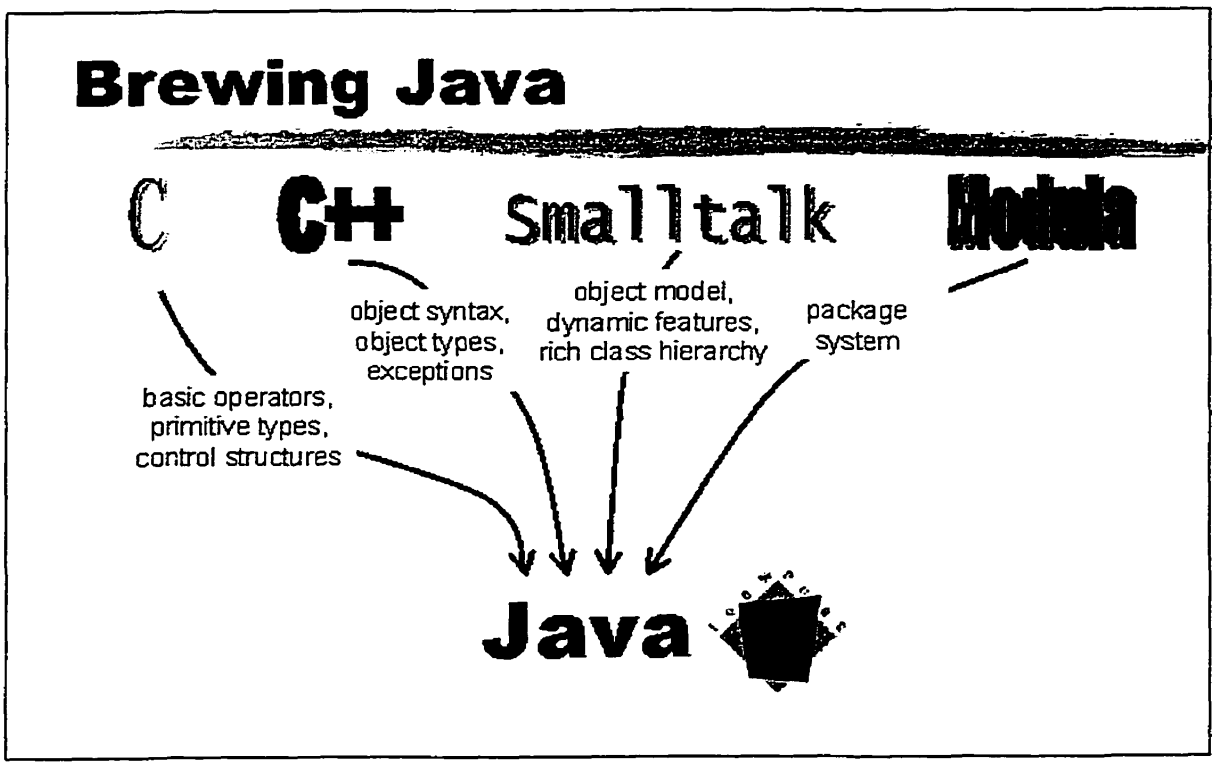

Figure 7-1. What is Java? [Badros]

Smalltalk [Smalltalk] was developed at Xerox's Palo Alto Research Center over a period of ten years between 1970 and 1980. It was originally designed for the Dynabook project, a vision of an inexpensive notebook-sized personal computer with the power to handle all information-related needs, so that it could be used both by professionals and by adults and children without any prior knowledge of computers. It is a pure object-oriented programming language integrated with a multi-windowed development environment [Winston, 1997].

Java is the latest widely used object-oriented language. Smalltalk is among the purest object-oriented language, and it introduced many of the current object-oriented concepts, so it is no wonder that there are many similarities between it and Java.

7.2.2 Similarities between Java and Smalltalk

Both Java and Smalltalk are object-oriented languages. Classes are the abstraction of objects. Attributes of a class are described in class (static) variables and instance variables. Behaviors of a class are implemented in methods and executed by sending a message (invoking a method). Both languages provide single inheritance, but subclasses can redefine inherited behavior (override methods) [Naughton, 1999], [Sharp, 1997].

Both Java and Smalltalk are portable. In order to implement platform-independence, most implementation of both languages translate source code into bytecodes that are then executed by a virtual machine. Although the nature of the bytecodes is somewhat different, the two languages have the same execution model.

In the area of memory management, both Java and Smalltalk provide automatic garbage collection to retrieve instances that are no longer referenced, and in this way memory is released.

7.2.3 Differences between Java and Smalltalk

OO thinking

Everything in Smalltalk is an object and this makes design clean and eliminates errors caused by working around the language. Java has eight primitive data types that are not objects and special wrappers need to be used to encapsulate them into objects. This complicates programming because some entities understand methods, and some don't. This makes applications more difficult to maintain and understand.

Typing

Java is statically typed, which means that users must declare an object's type before using it, but programmers can bypass this feature by casting that allows users to assign a value of one type to a variable of another type. Smalltalk is dynamically typed, which means that nothing about the type of an object needs to be known before a Smalltalk program is run. Users can not declare an object's type before using it, and the type is determined at run time. Static typing has its advantages and disadvantages. The main advantage is that the compiler can perform type checking to catch certain types of errors and bind calls to definitions. This can marginally speed up method dispatching and prevent "DoesNotUnderstand:" messages. Its main disadvantage is that it limits reuse by freezing the range of objects that can receive or use a message. Although casting can make static typing somewhat dynamic, it violates static typing and negates its advantages. Another disadvantage of static typing is that users have to provide types for all variables, parameters, and return values of methods, requiring extra keystrokes, and increasing the programming time and making code less readable.

Extensibility

Smalltalk's extension mechanism is unlimited and all its source code visible to the programmer on-line. This allows the programmer to view, extend and modify every aspect of the system, including its class library, which can make the programmer's life much easier. In Java, source code is not open and the *final* modifier means that a final class can not be subclassed, a final method can not be overridden, and a final variable can not change its value. That means that unless classes are designed perfectly, either they must be redesigned as their use evolves or new classes duplicating their behavior must be created. However, the fact that the programmer can view and modify every aspect of the system presents the danger of introducing errors into the compiler or debugger and reduces reliability and security.

Access control

Java has four modifiers, *public*, *default*, *protected* and *private*, to let programmers control the visibility of fields, methods, constructors and classes. Using these modifiers, programmers can control the access to make code safe. Smalltalk has no access control. All classes and methods are public, and all variables are private. This makes Smalltalk syntax simple, but programmers lose the control of access to their code.

Productivity

Smalltalk has an integrated and powerful development environment, and includes rich and mature class library, which makes it very productive. Since the source code is available and the language is reflective, developers can extend or customize it as they

wish. Based on studies of on large projects at multiple companies, Smalltalk implements function points with 50% of the code of Java [Capers Jones]. Development using Smalltalk is faster than using Java for the following reasons:

- First, Smalltalk usually needs less code to perform the same operation.

As an example, the following piece of code in Smalltalk gets and displays the contents of a file:

```
'C:\autoexec.bat' asFilename exists
        ifTrue: [Transcript show: ('C:\autoexec.bat' asFilename readStream contents)]
        ifFalse: [Transcript show: 'File does not exist.'].
```

The same task is more complicated with Java:

```
import java.io.*;
public class ReadFile {
public static void main(String args[]) {
 try {
  int c;
  FileInputStream stream = new FileInputStream("C:/autoexec.bat");
  while(( c = stream.read()) != -1) System.out.print((char) c);
  stream.close();
  } catch (FileNotFoundException e){
        System.out.println ("File does not exist.");
        }
   catch (IOException e) {
        System.out.println(e);
```

```
        }
    }
}
```

- Second, Smalltalk compiles code into bytecodes as soon as users finish writing it, taking no noticeable time to do so.

In Java, users usually compile their code after one or more classes are finished and sometimes the compiling process takes long time. Whenever the compiler finds errors, users have to go back to their code, correct the errors and recompile. This consumes much of Java developers' time.

- Third, it is easier to test a piece of code in Smalltalk.

For example, testing the outcome of a method in Smalltalk, users only need to write the code in a Workspace and *inspect* it. In Java, users have to create a file with a "main" method, and then compile and run it to get the result, which is time consuming.

Reflectivity

Smalltalk has a meta-programming mechanism allowing it to reflect on itself. It is flexible enough to let its users access internal information and to modify not only the development and runtime environment but even the language. As an example, developers can change the garbage collection mechanism, manipulate bytecodes, redefine or extend the language, add multiple inheritance, add a privacy mechanism and remove it, change language syntax, etc. Java also has a reflection API, *Object* class and *Class* class provide reflection capability. It resembles to Smalltalk meta-programming, but it is more

complicated and not so powerful. As an example, Java developers cannot make the changes listed above. Smalltalk also provides easier access to the runtime environment. For example, in Smalltalk, users can find all the immediate subclasses of a class by the *subclasses* method, the whole hierarchy of subclasses using *allSubclasses* method, and the class itself with all it subclass hierarchy using *withAllSubclasses* method. Java can not do this directly unless similar methods can be created.

As an example, to access all instance variables of class EMO in Smalltalk, users could use:

EMO instVarNames, then click "inspect" menu item.

In Java, users must write something like this:

```
class GetInstances{
        public static void main(String[] args) {
          EMO emo = new EMO();
          printInstances(emo);
        }
          static void printInstances(EMO o) {
          Class c = o.getClass();
          Field[] Instances = c.getFields();
          for (int i = 0; i < Instances.length; i++) {
            String instanceName = Instances[i].getName();
             System.out.println("Name: " + instanceName);
            }
          }
```

```
}
```

Debugging

Smalltalk's debugger gives users a stack trace, which allows users to look at any of the methods in the stack, inspect or change any of the variable values, redefine the executing message, change objects, and proceed without breaking execution. It also has a browsing capability for looking at senders and implementers. To invoke a debugger, users simply need to add "self halt" to their methods or insert invisible breakpoints. Java's debugger only tells users which line has an error in it and what error it is. Some Java IDEs, such as JBuilder, has more sophisticated debuggers that give users a stack trace and allow them to inspect variable values, but not change values during execution. Similarly, methods cannot be redefined without breaking execution.

Internet Awareness

From the beginning, Java addressed Internet programming and has close ties with the Internet and web browsers. Its class library is more oriented towards building Internet applications than the Smalltalk class library.

Thread-safety

Java class library was constructed to be thread-safe and automatically settle potential conflicts between different running threads, so users only need to concern about their own code to be thread-safe. Most current Smalltalk dialects do not use threads of the underlying operating system and use their own thread mechanism. This means that concurrent programming is easier in Java than in Smalltalk.

Synchronization

Java implements synchronization through the *synchronized* keyword, which eliminates most of the complexity associated with synchronization. In Smalltalk, we must use the *Semaphore* class and its methods to implement synchronization and this may be more complicated.

Runtime object creation

Smalltalk has a global dictionary named *Smalltalk* (VisualWorks5i and higher use namespace objects instead) that contains all the classes in the system, and global variables. Using it users can create an object whose type is unknown until runtime. With the reflection API, Java programmers can also deal with this, but it is more complicated. For example, EMOO needs to create an instance of a class at runtime by its name. In Smalltalk:

```
EMOClass := Smalltalk at: className.
newObject := EMOClass name: objectName.
```

In Java the same effect requires the following code:

```
Class emoDefinition;
Class[] argsClass = new Class[] {String.class};
Object[] args = new Object[] {objectName};
Constructor argsConstructor;
try {
    emoDefinition = Class.forName(className);
```

```
argsConstructor =  emoDefinition.getConstructor(argsClass);
argsConstructor.newInstance(args);
} catch (Exception e) {
   e.printStackTrace();
  }
```

Runtime method invocation

Smalltalk has the *perform:* family of messages that allows the programmer to invoke a method on an object, even if the method is not known until runtime. Java's *java.lang.reflection.Method* class can implement the same function, but again, it is more complicated. Fox example, EMOO needs to invoke a method at runtime according to its name when handling an event. In Smalltalk:

```
anEMO perform: methodName withArguments: parametersArray
```

In Java:

```
try{
   Class cls = anEMO.getClass();
   Class partypes[] = new Class[i];
   for ( int j = 0; j<i; j++) partypes[j] = event.getParameters().get(j).getClass();
   Method meth = cls.getMethod(methodName, partypes);
   Object arglist[] = new Object[i];
   for ( int j = 0; j<i; j++) arglist[j] = event.getParameters().get(j);
   meth.invoke(anEMO, arglist);
   }
```

```
catch (Exception e) {
  e.printStackTrace();
}
```

It is obvious that the code written in Smalltalk is easier to use and understand than Java.

Ease to learn

Java has a more conventional syntax than Smalltalk, so that established programmers quickly feel comfortable with Java, while it takes longer to become comfortable with Smalltalk. However, for new programmers who do not have any program experience, Smalltalk may be easier to learn: It has only five reserved words (Java has 50 reserved words) and very simple syntax with few rules and special cases.

Readability

Each Java source file contains all information about a class including its definition, variables, constructors and methods. In Smalltalk, classes and methods are classified as categories, classes, protocols and methods. Different people have different preferences concerning these arrangements.

Standardization

Smalltalk has an ANSI standard [ANSI, 1998]. Java does not have one yet. However, Smalltalk has several dialects that differ somewhat from one implementation to another. The most popular dialects of Smalltalk are VisualWorks [VisualWorks], VisualAge Smalltalk [IBMST], Dolpin [Dolpin], GNU Smalltalk [GNU], and Squeak [Squeak]. Java

has no dialect. Sun Microsystems, Inc. is the only vendor of the Java development environment and others are vendors of Java IDEs.

Maturity

Smalltalk was unveiled about thirty years ago after ten years of research, while Java was created only ten years ago. Java is still evolving and relatively immature compared with Smalltalk. We can compare Java to a teenager and Smalltalk to an adult. The version of JDK - Java development environment - is often updated and the different JDK versions are somewhat inconsistent, so that Java developers have to work around various immature aspects of the JDK during the development. Smalltalk's core libraries are more mature, but the great interest in Java will hasten its maturation.

7.2.4 Conclusion

Java and Smalltalk are both excellent object-oriented programming languages. They are both suitable for developing virtual environment systems such as MOOs. Based on my experience, I consider Java to be better in the following categories:

- Java is a secure language with many features that facilitate the creation of secure applications.
- Using RMI API to implement a distributed object model is more convenient, more efficient and simpler than using sockets in Smalltalk. (VisualWorks5i.1, the latest version VisualWorks, provides the OpenTalk [Cincom] framework that can implement similar functionality as RMI.)
- To transfer objects, the RMI API uses the Serialization API to marshal and unmarshal objects. This is easier than converting objects to BOSS (BinaryObjectStorageStream)

and vice versa for transferring them in Smalltalk. All we need to do is let object classes implement Serializable interface, we need not define by ourselves. (Recent VisualWorks OpenTalk performs the same function.)

- The Java class library was constructed to be thread-safe and has built-in threading for synchronizing blocks of code or methods. This is easier to use than equivalent in Smalltalk.
- Java is easy to learn and use for a programmer familiar with C/C++.

Smalltalk is better in the following categories:

- Smalltalk has a more mature virtual machine that is faster than current Java virtual machines.
- Smalltalk has a better programming environment and a more mature class library.
- Smalltalk programs are easy to change and reuse because they use dynamic typing with run-time type-checking instead of Java's static typing with compile-time type-checking.
- Smalltalk code is more concise, which makes it suitable for rapid application development.
- Implementing dynamic object creation and method invocation is easier in Smalltalk than in Java.

In conclusion, at present, because Smalltalk is more mature, productive and dynamic, I concluded that it is better using Smalltalk than Java to implement a text-based MOO.

Java is certainly more popular and has a larger user community. Consequently, there is much effort behind Java that pushes it to develop rapidly and make it better and better, but this does not mean that Smalltalk will disappear. Smalltalk is also evolving. Java's onslaught contributed to the emergence of Smalltalk's free versions and created a pressure on further Smalltalk development.

The differences between Java and Smalltalk listed above are diminishing, because Java and Smalltalk are both learning from each other's strong points to offset their weakness. The competition between Java and Smalltalk makes the information technology world more exciting.

Chapter 8 Using EMOO

To put the earlier chapters into context and to help those who want to experiment with EMOO, this chapter describes the use of EMOO and shows how it is installed. Several screenshots show the graphical user interfaces that users will typically encounter while using this system.

8.1 Installing EMOO

All EMOO code is available at http://ace.acadiau.ca/user/ivan/research/CVE/index.html. EMOO class files are contained in a compressed zip file called "emoo.zip". File "java.policy" is used for changing Java security policy and giving permissions to run RMI client and server programs. To install EMOO proceed as follows:

1. Install Java 2 SDK (JDK1.2) or above. This is available at www.javasoft.com.
2. Create a working directory, for example "c:\myclasses".
3. Unzip emoo.zip to the directory created in Step 2.
4. Copy the "java.policy" file to the "jre\lib\security" directory of your JDK installation.

8.2 Starting a Metaserver

To enable EMOO, start a metaserver as follows:

1. Start RMI's registry: Open a console (a DOS window in Windows), and execute the following command in the working directory:

   c:\myclasses> rmiregistry

2. Run EMOO metaserver: Open another console and enter

   (jdk install directory)java –classpath c:\myclasses emoo.EMOOMetaServerManager

   This will open the EMOO metaserver window (Fig. 8-1).

3. Start the metaserver. Click the "Start" button in the EMOO metaserver window.

Figure 8-1. EMOO metaserver user interface

8.3 Starting and saving a Universe

Once a metaserver is running, a server running a universe must be started. Any number of universes may run on a single metaserver. Once a universe is started, any number of users can then connect to it. To start a universe, execute the following steps:

1. Start RMI's registry: Open a console, go to your working directory, and execute the following command:

   c:\myclasses> rmiregistry

   Do not execute this step if you have already started RMI registry.

2. Run EMOO universe: Open another console and execute

   (jdk install directory)java –classpath c:\myclasses emoo.EMOOUniverseManager

   The EMOO universe window will open (Fig. 8-2).

Figure 8-2. EMOO universe user interface

Create a new universe or load one from a file: To create a new universe, click New button to enter the universe name (Fig. 8-3, 8-4). To load a universe from a file, click the Load button (Fig. 8-3, 8-5).

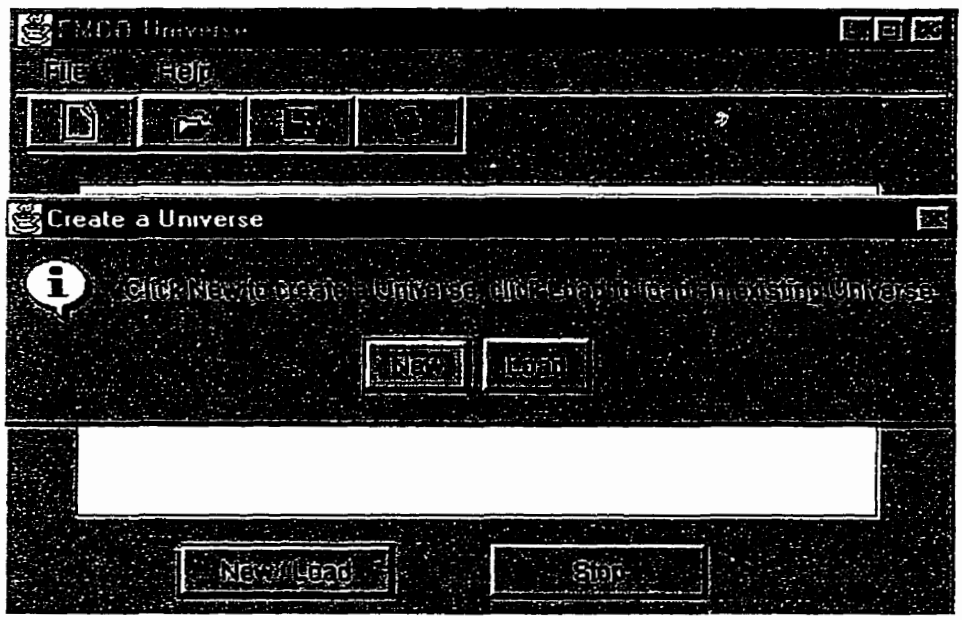

Figure 8-3. Creating a new or loading a saved EMOO universe

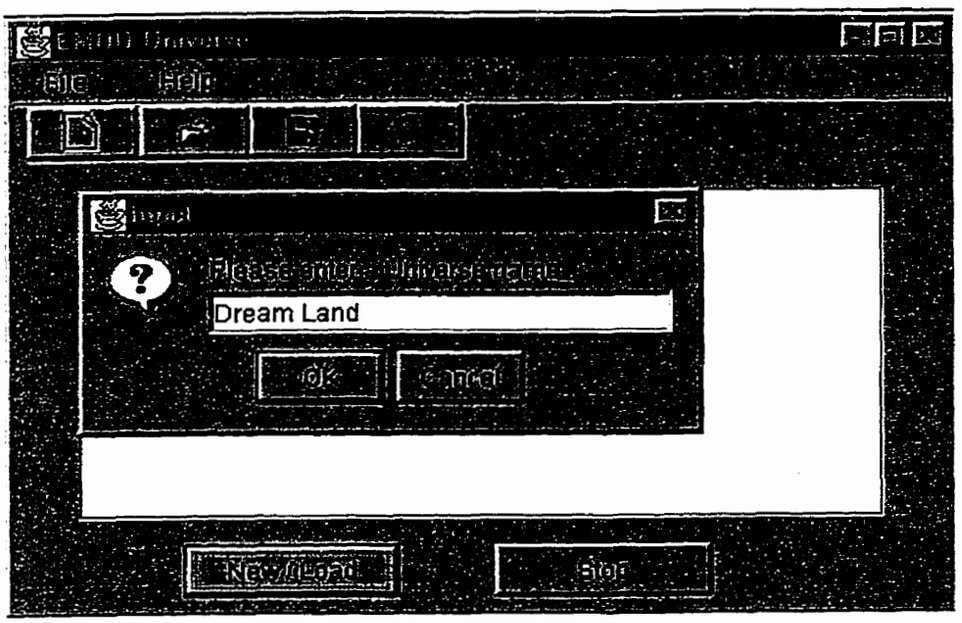

Figure 8-4. Creating a new universe

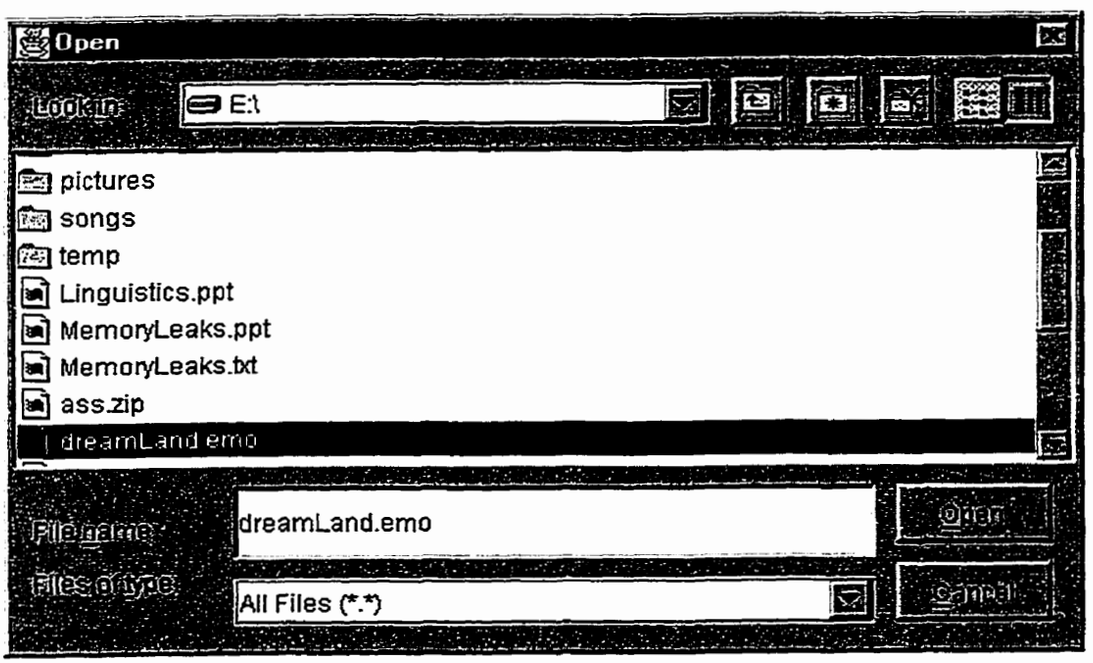

Figure 8-5. Loading a saved universe

When a universe is started, it registers its status on the metaserver (Fig. 8-6).

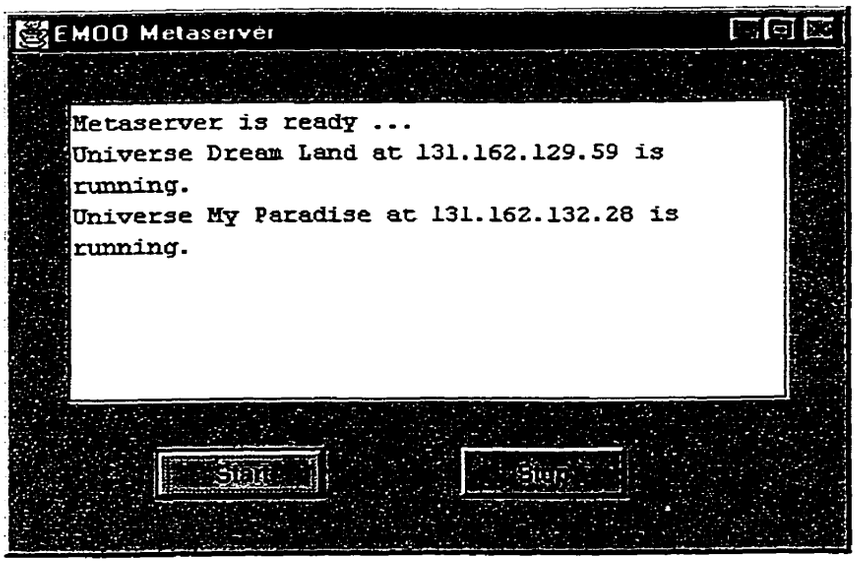

Figure 8-6. Universes registered on the metaserver

To save the currently running universe, click Save button to save it in a file (Fig. 8-7).

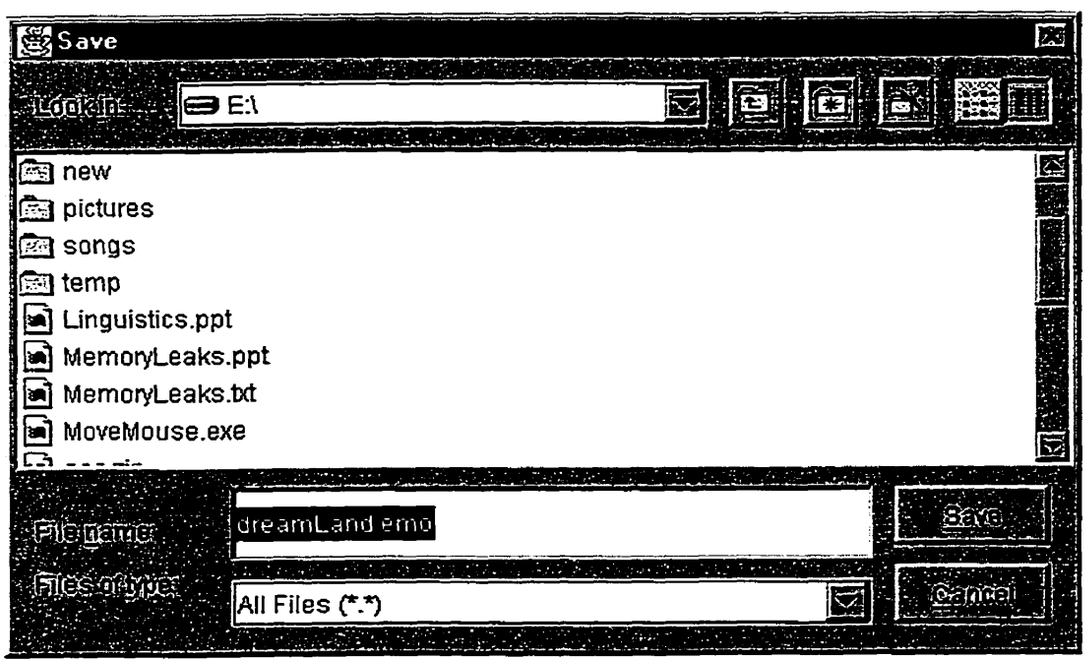

Figure 8-7. Saving a universe

8.4 Running a Client

Opening and closing a client

Use EMOO launcher to contact EMOO universes and connect to them, both locally and remotely, and disconnect. The launcher also provides access to other EMOO tools.

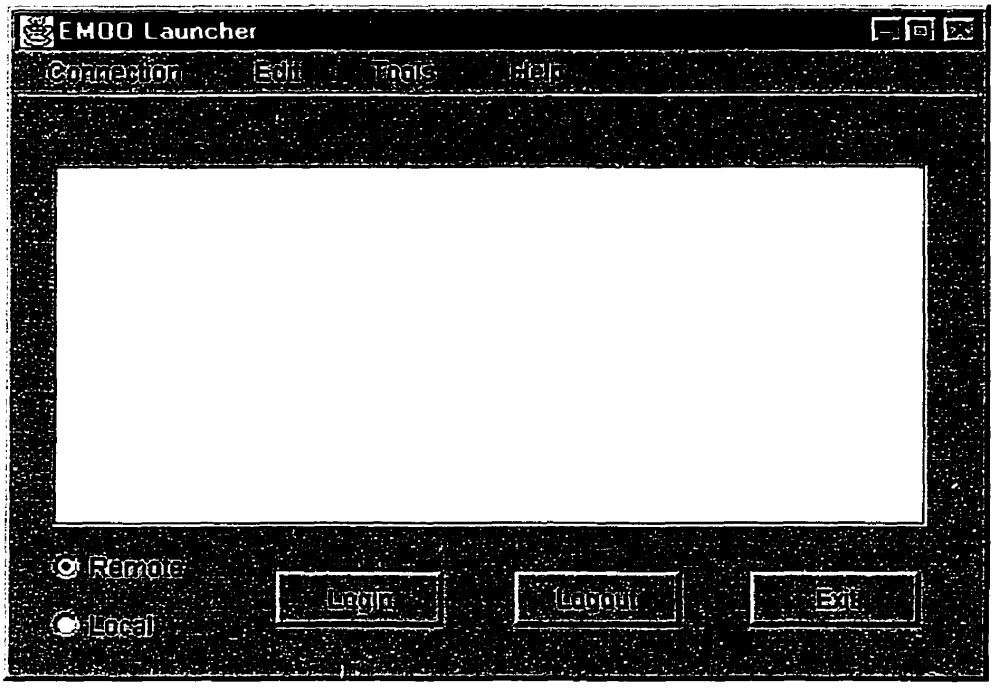

Figure 8-8. EMOO launcher user interface

1. Open EMOO launcher: Open a console and execute

   (jdk install directory)java –classpath c:\myclasses emoo.EMOOLauncher

   The EMOO launcher window will open (Fig. 8-8).

2. Select a remote universe or create a local one: Select Remote radio button and click Login button to open a list of all currently available universes (Fig. 8-9). Select Local to create a local universe (Fig. 8-4).

3. Log in a universe: After selecting a universe, click Select button to open a login dialog (Fig 8-10).

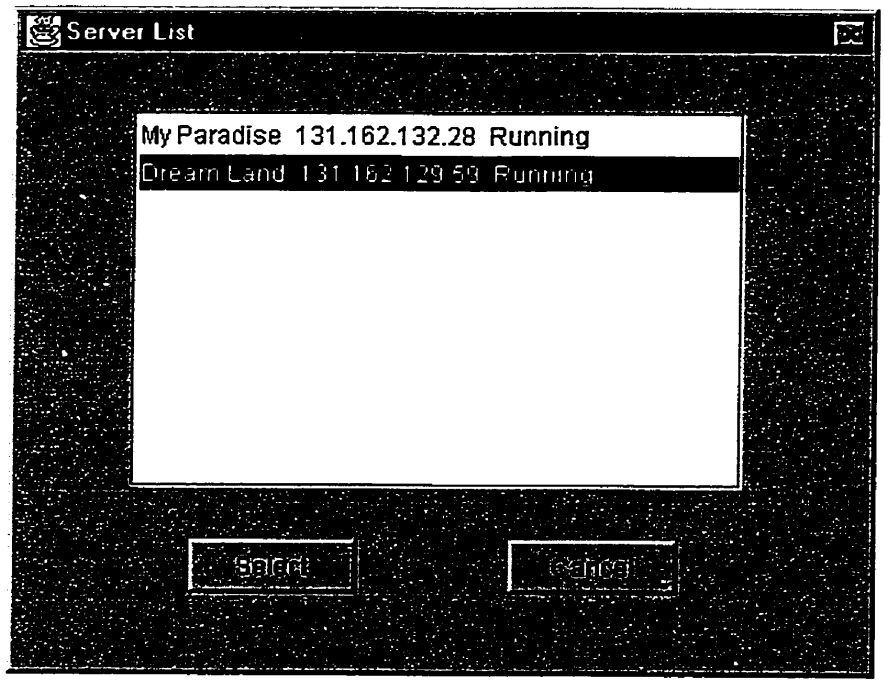

Figure 8-9. Available universes list

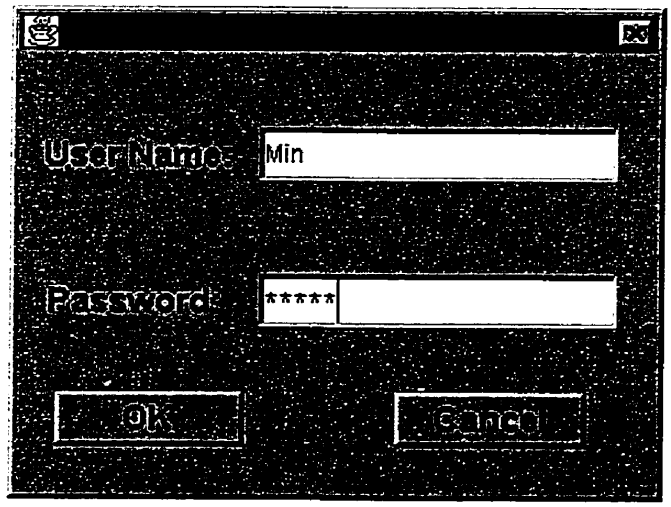

Figure 8-10. Login dialog

After logging in, you can open a tool from the Tools menu (Fig. 8-11). The following section describes the available tools.

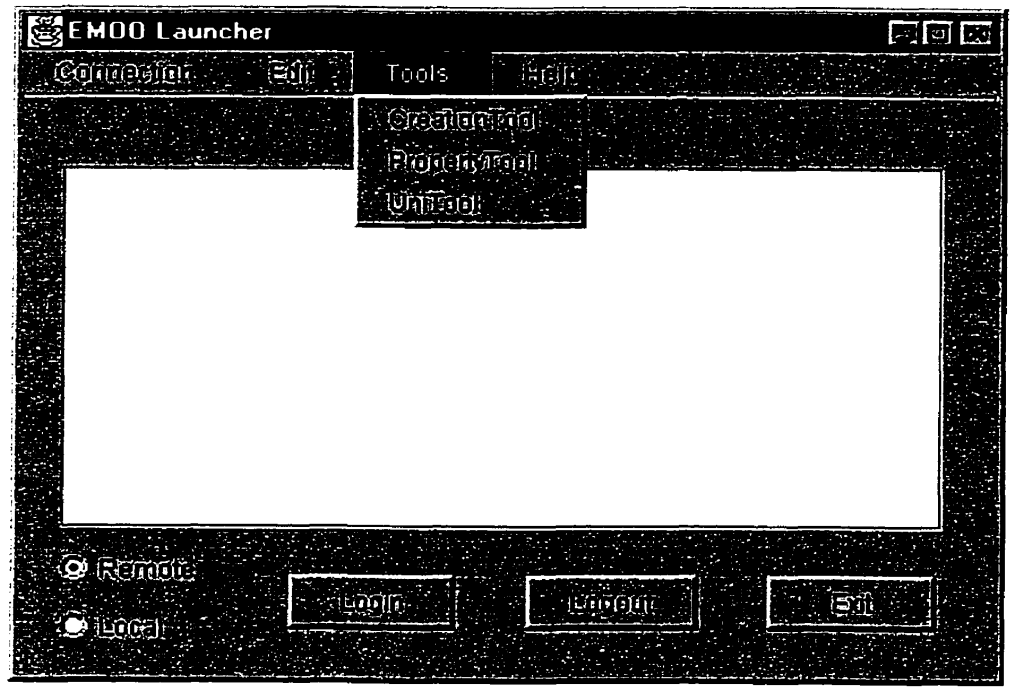

Figure 8-11. Selecting a tool to open

CreationTool

This tool allows the user to create a new instance of an existing object template and to define its properties. The user chooses the object type (class), gives the object a name and description, and clicks "Create" to create the object (Fig. 8-12).

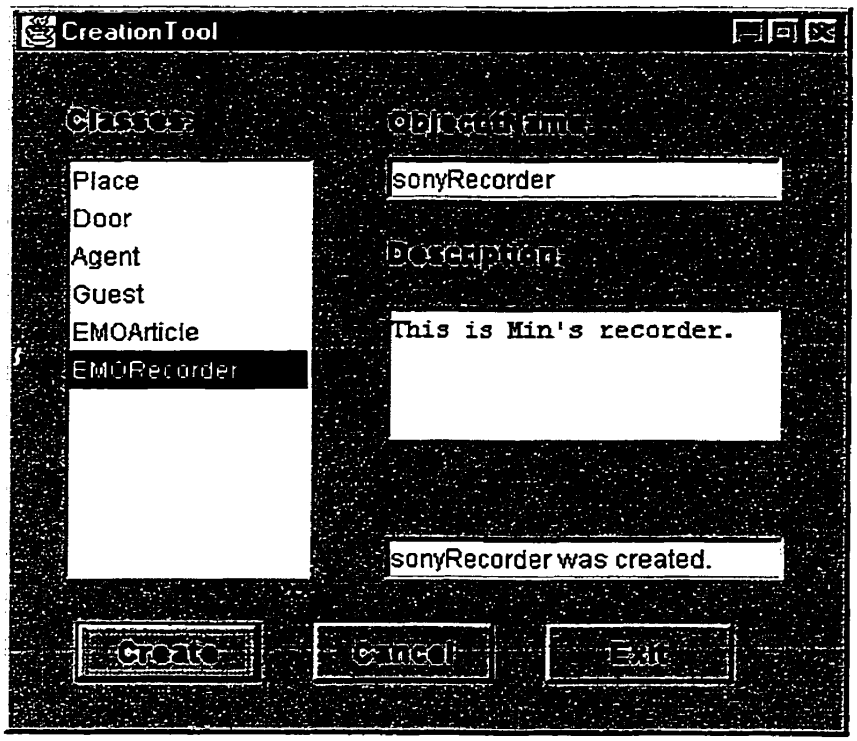

Figure 8-12. "CreationTool" user interface

PropertyTool

This tool displays an object's properties and allows the user to edit them if he or she is authorized to do so. To edit an object's properties, the user first selects the object, then chooses and edits the properties, and clicks Accept button (Fig. 8-13).

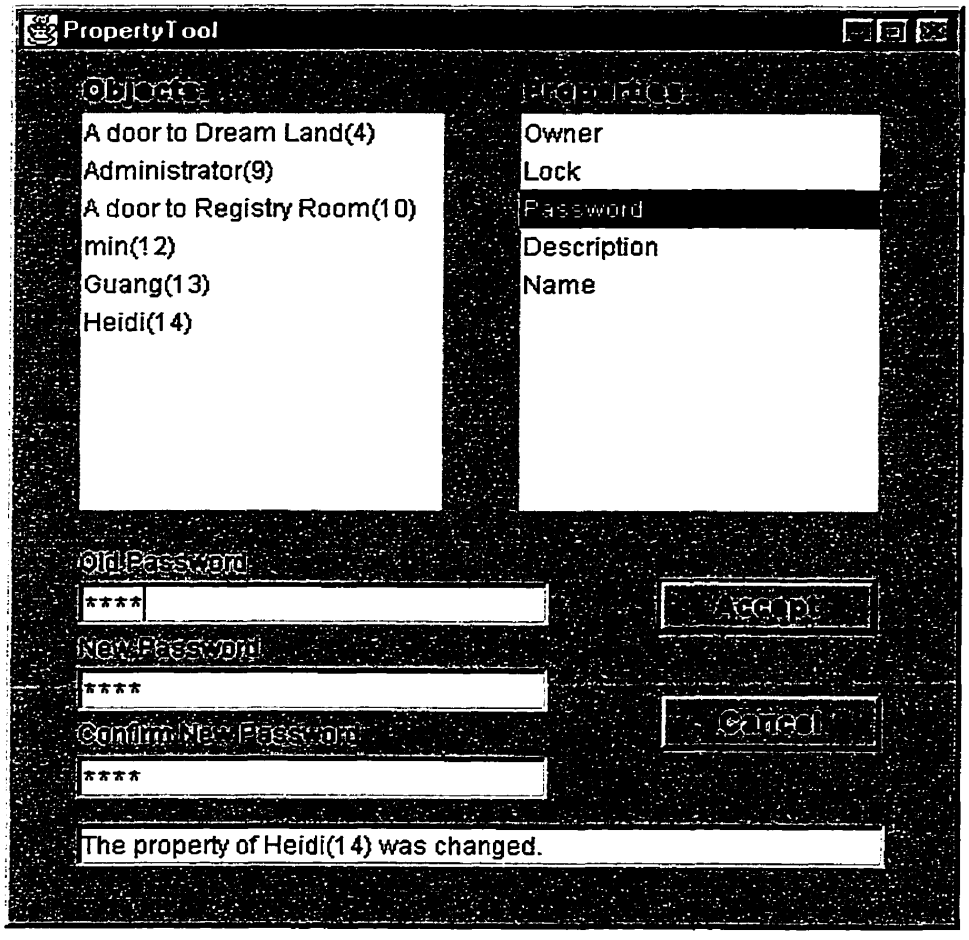

Figure 8-13. "PropertyTool" user interface

UniTool

This tool provides basic communication and navigation functions. It has a communication part with input and output areas, an entity list area showing objects, doors, occupants and personal holdings, and display the commands understood by the selected object, and its subscribable events.

The user can select "Say" or "Emote" to say or emote to everyone in the same room, or select "Whisper" and a person, Heidi in this example to whisper to. Entering the text in the input field executes the command (Fig. 8-14).

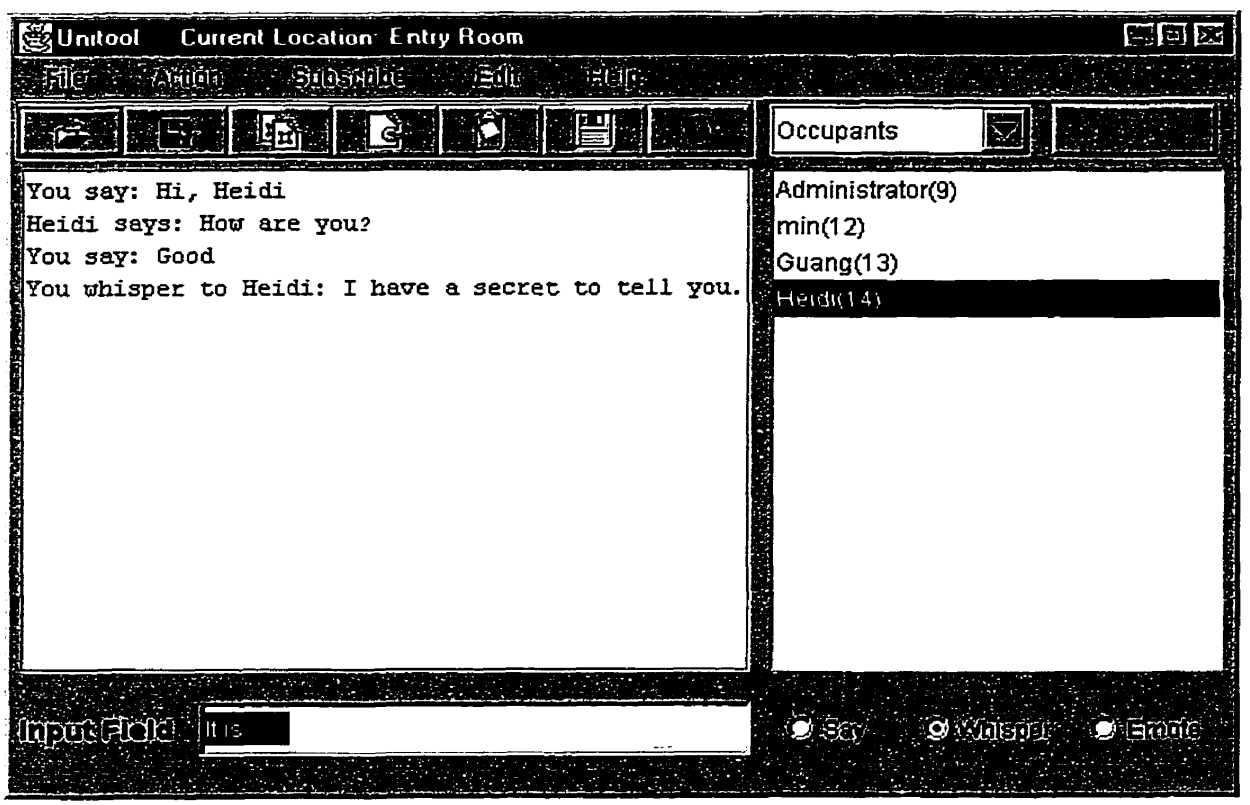

Figure 8-14. Min's UniTool interface when whispering to Heidi

To move from the current room to another one, the user chooses a door from the door list, and clicks Go button (Fig. 8-15).

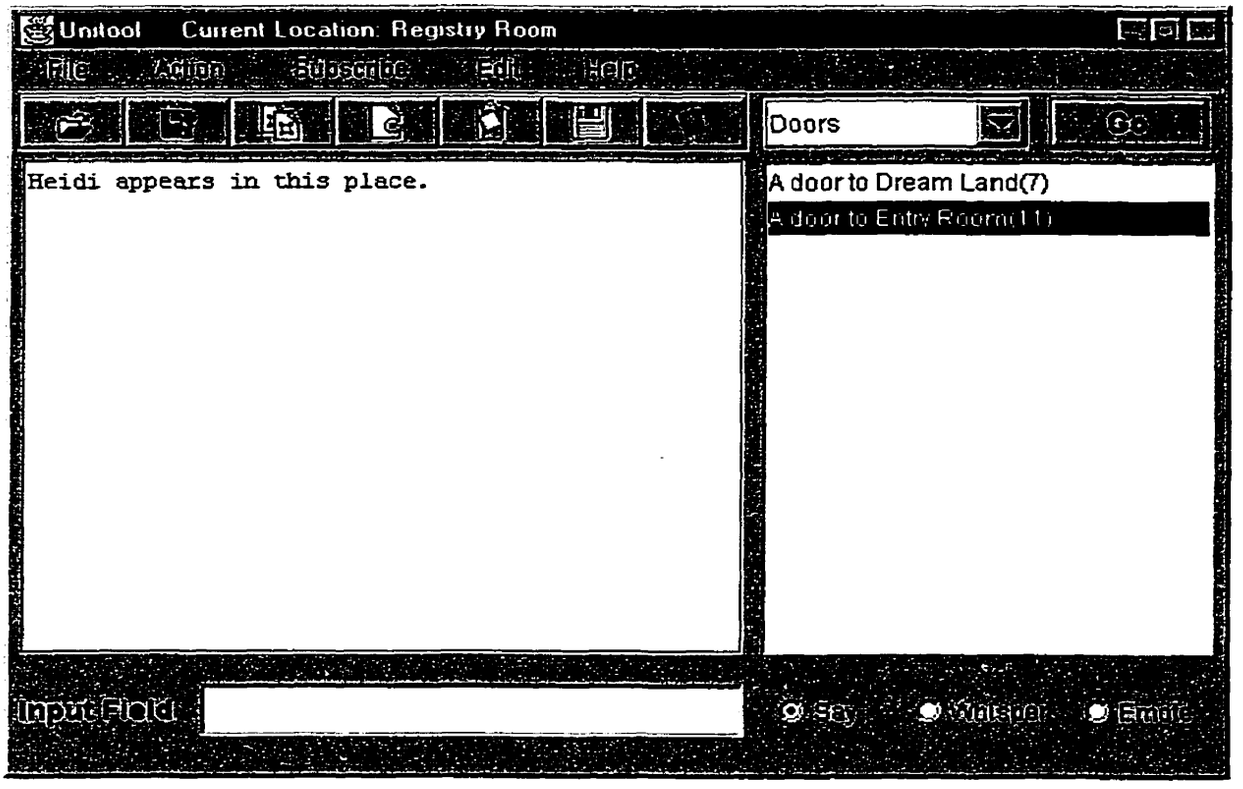

Figure 8-15. Agent Heidi is in "Registry Room" and wants go to the "Entry Room"

To use an object, such as a recorder, the user chooses the object in the UniTool and clicks Open button (Fig. 8-16). The object's tool interface opens (Fig. 8-17).

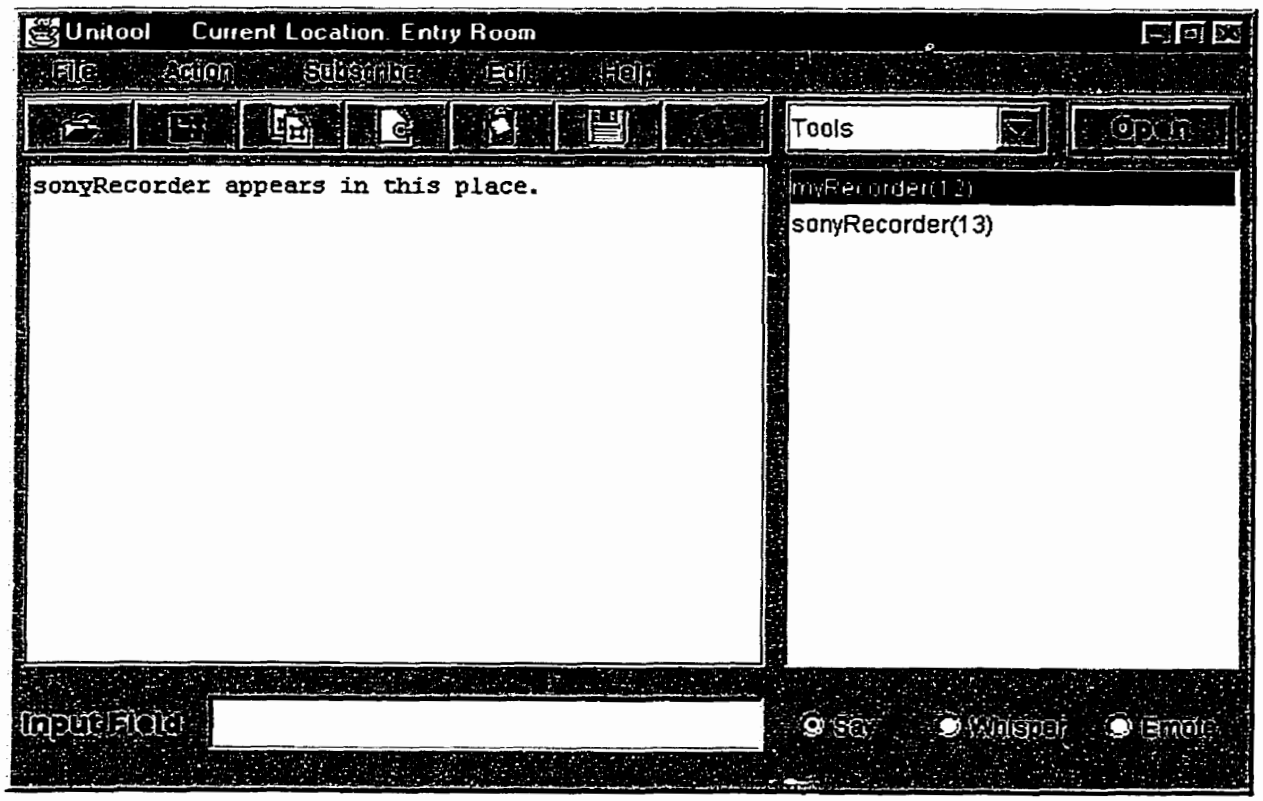

Figure 8-16. Opening a recorder named "myRecorder"

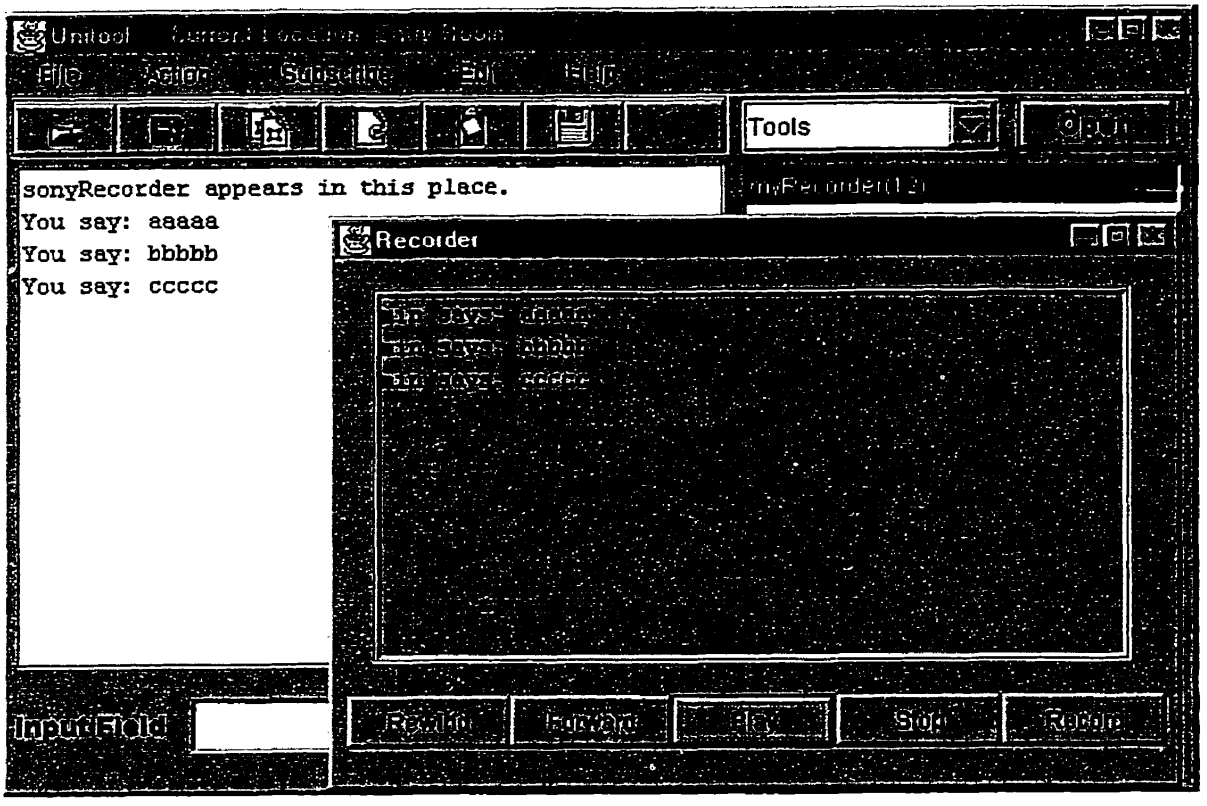

Figure 8-17. Recorder user interface

Manipulating objects

The Action menu in the UniTool allows the user to pick up, drop and destroy an object (Fig. 8-18). The use of the interface is self-explanatory.

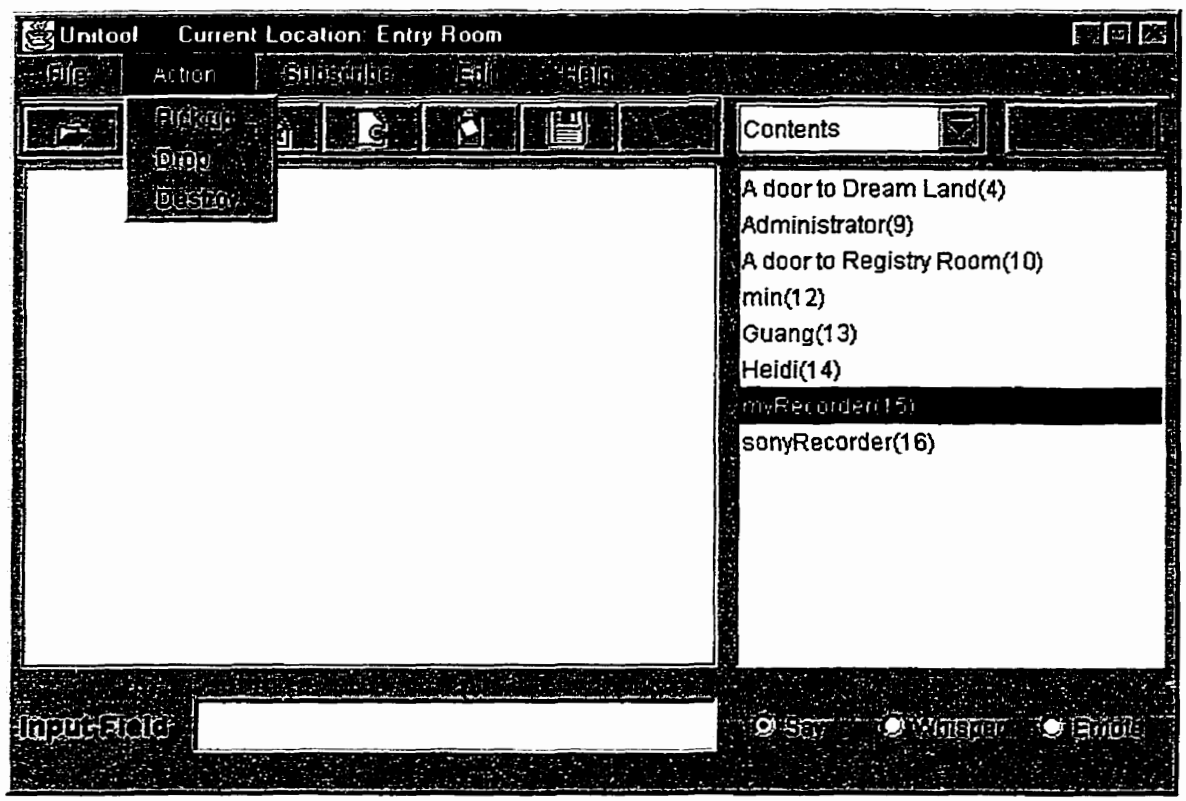

Figure 8-18. User picks up "myRecorder"

Subscribing to events

The user can subscribe to an object's events to be notified when they occur. To subscribe to an event, the user selects the object, opens its events list from the Subscribe menu in the UniTool, chooses an event from the event list, and clicks OK button (Fig. 8-19 and 8-20).

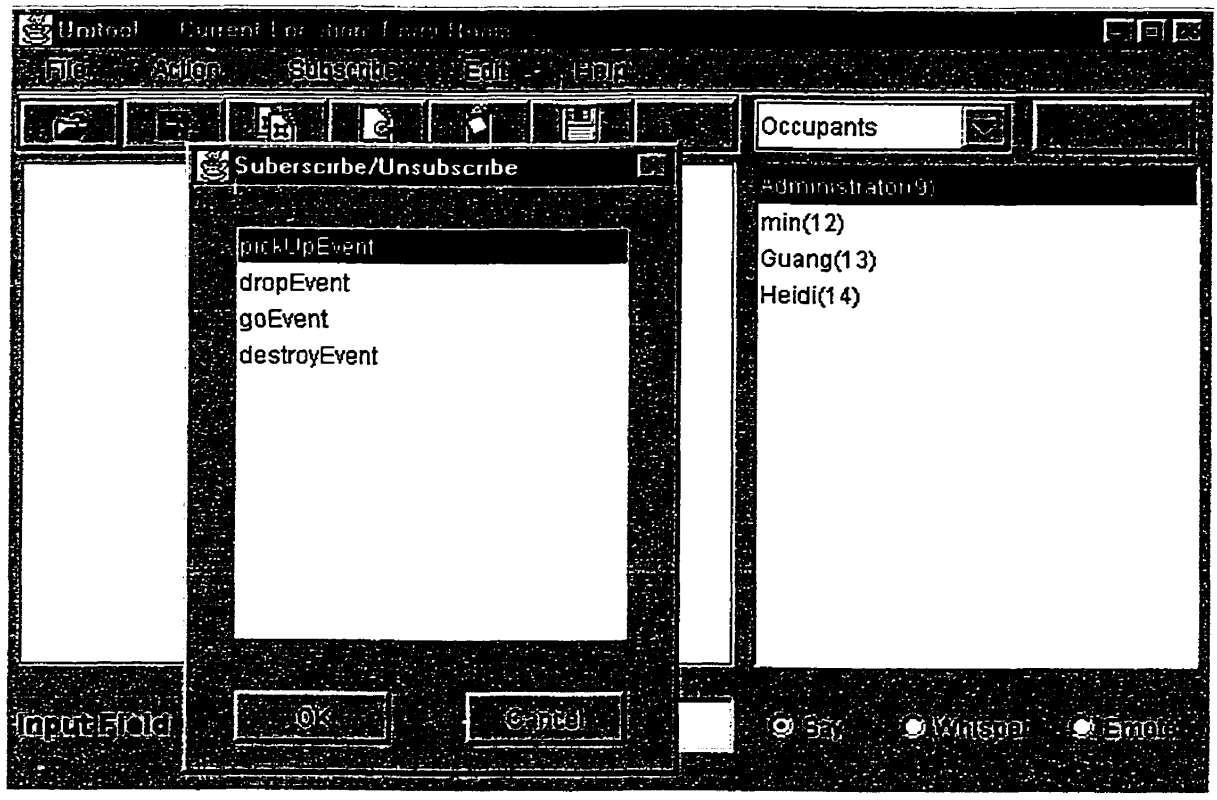

Figure 8-19. A user subscribes to Administrator's pickUpEvent

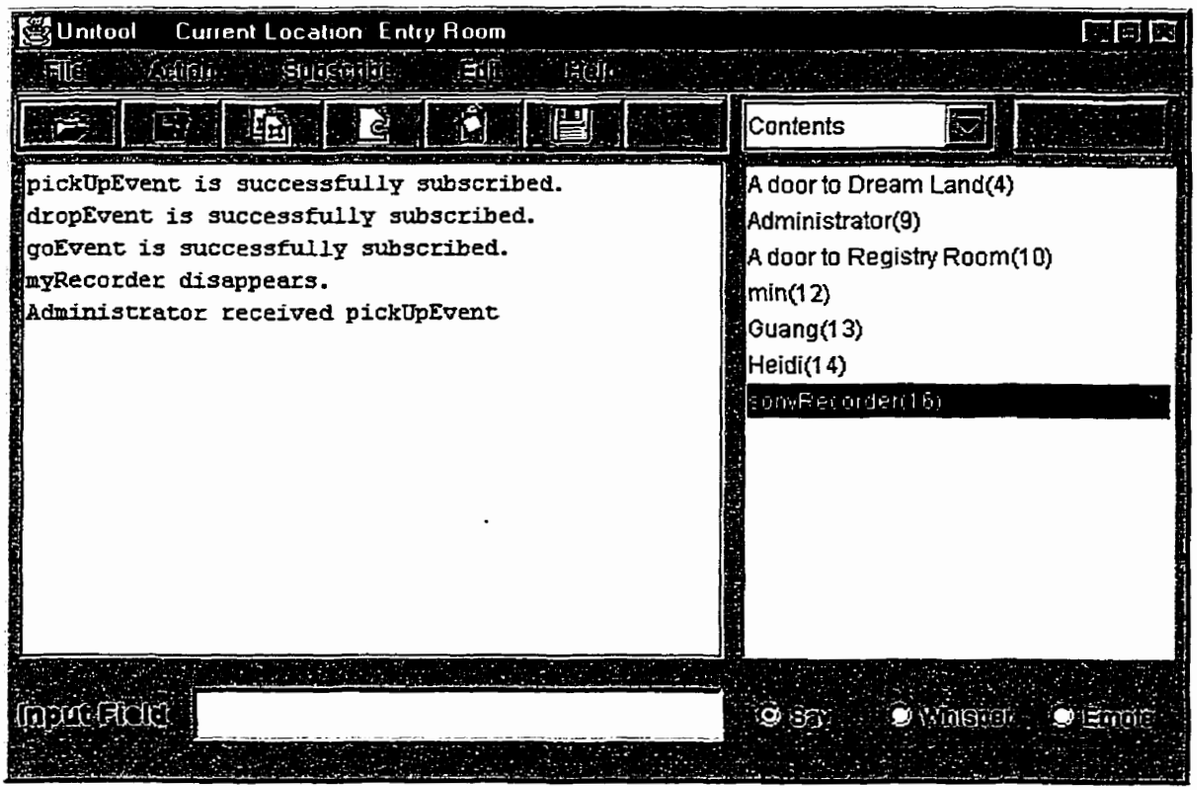

Figure 8-20. The user is notified of event subscription and occurrence

To unsubscribe, the user opens subscribed events from the Subscribe menu in the UniTool, chooses a subscribed event, and clicks OK button (Fig. 8-21 and 8-22).

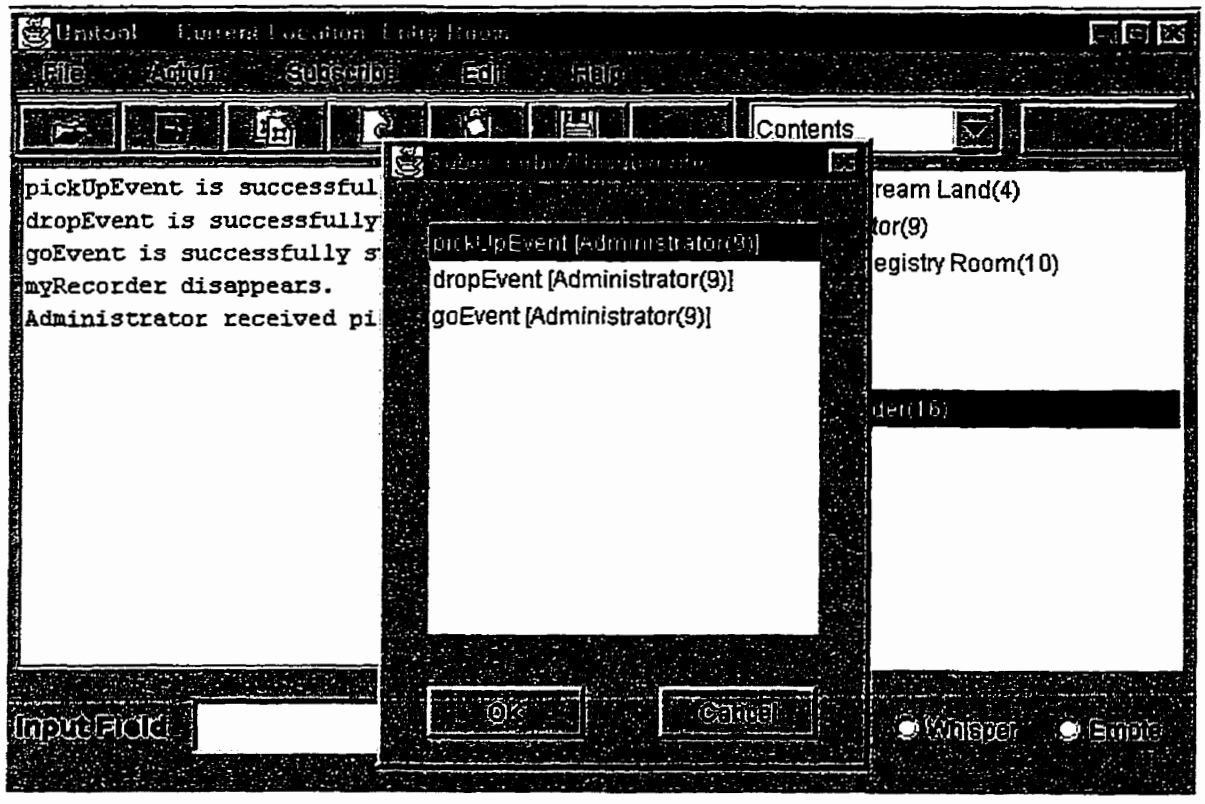

Figure 8-21. User unsubscribes Administrator's "pickUpEvent"

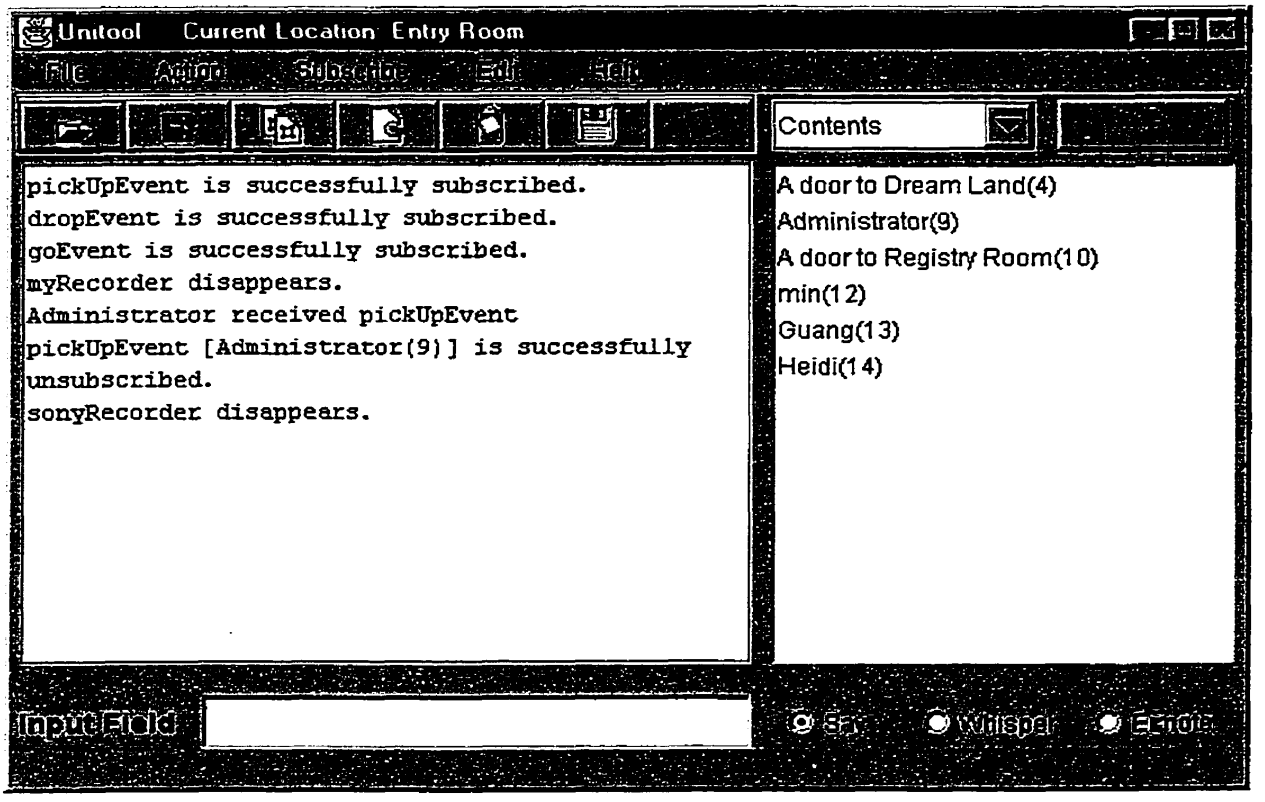

Figure 8-22. Administrator's "pickUpEvent" has been unsubscribed

Chapter 9 Conclusion

9.1 Current state of EMOO

At present, EMOO is fully working and has been tested on a small scale. It has all the basic functions that a traditional MUD/MOO has. Users can communicate with others using the UniTool, create, pick up, drop and destroy objects, walk around, and subscribe to events. It has basic objects including agents, guests, places and doors. Four tools have been implemented including UniTool, Creation Tool, Property Tool and Recorder.

9.2 Conclusion

As we enter the new millenium, distributed software development is becoming more commonplace and there is a growing need for software environments supporting it. Collaborative Virtual Environments (CVE) are one of the promising approaches.

In designing a multi-user virtual environment for Computer Supported Collaborative Work (CSCW) [CSCW], we must consider many aspects including concurrency, efficiency, extendibility, security, complexity, flexibility and usability. Distributed teams require the capability to work together, save and version their work, and merge work done by individuals into a whole. We must consider the management of concurrent access by a potentially large number of users, provide tools to support easy versioning of a design and merging of multiple artefacts. We must also balance speed and functionality, flexibility and security, efficiency and complexity according to the different requirements.

EMOO is an extension of a new approach to the implementation of a virtual environment based on MOOs. It re-implements most of the features of the pilot project MUM in Java, including a simplified version of event-based operation, the possibility to create any number of interconnected universes, a substantial client relieving the server and the network of much activity, and powerful and extendible client-side user interfaces. EMOO is not a simple translation of MUM from Smalltalk to Java, however, and incorporates many changes and improvements in design and implementation to address the problems of balancing speed and functionality, efficiency and complexity (see Chapter 7 for details).

Using Java to implement EMOO also gives us an opportunity to compare Java with Smalltalk – the implementation language of MUM - and learn more about Java and Smalltalk.

9.3 Future work

The first desirable extension of EMOO is to add more EMO objects and tools, for example whiteboard and binder, to support communication and multiple formal software design notations and template-based drawing, such as UML and versions management.

Other possible enhancements include converting the client to an applet, so that users can use a web browser to connect to EMOO without having to invoke the Java interpreter. At present, EMOO is an application because the focus is on conceptual and design issues without the additional complications of applet programming.

Implementing automatic code update, so that when a user connects to a universe the code is automatically updated, and a locking mechanism, for example to manage multiple users working on a single design, are also desirable. Such mechanisms can transform a free-for-all design into an organized and coordinated design effort. Support for multiple languages so that servers can be implemented using different programming languages, such as Smalltalk, Java, C++, should also be considered. All these functions are useful to address CSCW needs.

Glossary

API Application Programming Interface.

Avatar The representation (proxy) of a user in a virtual environment.

CASE Computer-Aided Software Engineering.

CORBA Common Object Request Broker Architecture.

CVE Collaborative Virtual Environment. A virtual environment designed for collaboration.

CSCW Computer-Supported Cooperative Work.

EMOO Experimental MOO.

Function Point A software metric used to measure the complexity of an application from the functional perspective. Also used to predict the amount of source code that must be written to implement an application. Different languages require a different average number of statements to implement one Function Point [Capers Jones].

MOO MUD Object Oriented. MUD implemented in an Object-Oriented language.

MUD Acronym for Multiple User Dimension, Multiple User Dungeon, or Multiple User Dialogue. A virtual software world where a user can extend the environment, communicate with others, navigate, and use objects.

MUM Multi-Universe MOO.

RMI Remote Method Invocation.

UML Unified Modeling Language. A graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system.

VE Virtual Environments.

Virtual Environment A software environment that emulates physical world with a multitude of navigable places, portable objects, and communicating software entities including human users and software agents. Can be implemented according to various paradigms including MUD.

Bibliography

[ANSI, 1998] NCITS J20, ANSI Smalltalk Standard document, March 1998

[Badros] http://www.cs.washington.edu/homes/gjb/doc/java-lang/ppframe.htm

[Booch, 1999] Grady Booch, James Rumbauch, Ivar Jacobson: *The Unified Modeling Language User Guide*, Addison Wesley Longman, Inc., 1999

[Borland] http://www.borland.com/jbuilder/

[Capers Jones] http://www.spr.com/library/0langtbl.htm

[Chimu] http://www.chimu.com/publications/

[CORBA] http://www.corba.org/

[DCOM]

http://www.microsoft.com/ntserver/appservice/techdetails/prodarch/dcom_architecture.asp

[Dolpin] http://www.object-arts.com/Home.htm

[Farley, 1998] Jim Farley: *Java Distributed Computi*ng, O'Reilly, 1998

[Fowler, 1997] Martin Fowler with Kendall Scott: *UML Distilled*, Addison Wesley Longman, Inc., 1997

[GNU] http://www.gnu.org/

[Haynes, 1998] Haynes C, Holmevik JR: *High wired: on the design, use, and theory of education MOOs*, University of Michigan Press, Ann Arbor, 1998

[Horton, 1999] Ivor Horton: *Beginning Java 2*, Wrox, 1999

[IBMST] http://www-4.ibm.com/software/ad/smalltalk/

[IBMVJ] http://www-4.ibm.com/software/ad/vajava/

[JavaSoft] http://www.javasoft.com

[Lambda] http://www.moo.mud.org

[Lingua MOO] http://lingua.utdallas.edu

[Meier, 1999] Carol Meier and George Watson: *Condensed Java*, tutorial notes of OOPSLA'99, 1999

[MicrosoftVJ] http://msdn.microsoft.com/visualj/

[MicrosoftVS] http://www.microsoft.com/office/visio/

[Morrison, 1997] Michael Morrison and Jerry Ablan: *Teach Yourself More Java in 21 Days*, Sams.net, 1997

[MUDline] http://www.apocalypse.org/pub/u/lpb/muddex/mudline.html

[Naughton, 1999] Patrick Naughton and Herbert Schildt: *Java™2: The Complete Reference, Third Edition*, McGraw Hill, 1999

[Okstate] http://www.cs.okstate.edu/~jds/mudfaq-p1.html#q1

[Popkin] http://www.popkin.com/

[Rational] http://www.rational.com/products/rose/index.html

[RMI] http://www.javasoft.com/products/jdk/rmi/index.html

[Sharp, 1997] Alec Sharp: *Smalltalk by Example*, McGraw Hill, 1997

[Smalltalk] http://www.smalltalk.org

[Squeak] http://www.squeak.org/

[Stevens, 1990] W Richard Stevens: *UNIX Newtwork Programming*, Prentice Hall Inc., 1990

[Symantec] http://www.symantec.com/domain/cafe/vcafe30.html

[Together] http://www.togethersoft.com/

[Tomek, 1999] Ivan Tomek, Rick Giles: *Virtual Environment for Work, Study, and Leisure*, Journal of the Virtual Reality Society, 1999

[Tomek, 2000] Ivan Tomek: *The Design of a MOO*, Journal of Network and Computer Applications, to be published

[TWUMOO] http://moo.twu.edu:7000/

[Visual Object] http://www.visualobjectmodelers.com/

[VisualWorks] http://www.cincom.com/

[Whatis] http://www.whatis.com

[Winston, 1997] Patrick Henry Winston: *On to Smalltalk*, Addison Wesley Longman, Inc., 1997

Jersey MOO, MUM & EMOO: http://ace.acadiau.ca/user/ivan/research/CVE/index.html