

The MaDViWorld Software Framework for Massively Distributed Virtual Worlds: Concepts, Examples and Implementation Solutions

P. Fuhrer, G. Kouadri Mostéfaoui, J. Pasquier-Rocha

`Patrik.Fuhrer@unifr.ch`

Abstract

The MaDViWorld project represents an original attempt to define appropriate software architecture for supporting massively distributed virtual worlds systems. A non-massively distributed virtual world system is typically engineered as a client-server application for which a single server or more rarely a small cluster of servers contain all the world pertinent data and assume the world accessibility, consistency and persistence. On the client side, many of them enable interaction with the other users and the various objects of the world.

The main originality of our approach resides in the fact that the server part of the proposed system is no more limited to a few centralized servers, but can be distributed on arbitrarily many of them. Indeed, MaDViWorld, the prototypal software framework already implemented using Java and RMI by our group, allows to create the rooms of a given world on several machines, each running the server application. It is then possible to connect the rooms by way of simple doors and to populate them with active objects. Finally, avatars managed by the client application visit the rooms and interact with the active objects either directly on the remote host or locally by cloning or transporting them first on the client machine.

This paper draws from the experience gained with the development of our first prototype in order to discuss, both at the user's and the vi-world developer's level, the main issues related to this new software engineering approach.

Keywords: Virtual Worlds, MUD, MOO, distributed software architecture, Java, RMI.

Table of contents

1. Introduction	3
1.1 Document versus Virtual World Paradigms	3
1.2 Goal and Outline of this Paper	3
1.3 Conventions	4
2. Virtual Worlds: a Brief Presentation	4
2.1 Terminology	4
2.2 Historical Overview	5
3. Technological Objectives	6
3.1 The User Level	6
3.2 The Vi-World Developer Level	7
4. A Sample User Session	7
4.1 Launching a Server and Setting up a Room	7
4.2 Launching an Avatar	8
4.3 Using Objects	8
4.4 Objects Mobility	9
5. The MaDViWorld Software Architecture	9
5.1 Specification Layer	10
5.2 System Implementation Layer	12
5.2.1 Client classes	12
5.2.2 Server Classes	12
5.2.3 Wizard Classes	13
5.3 Object Implementation Layer	13
6. Concrete Implementation	14
6.1 Software Architecture Revisited	14
6.1.1 Specification Interfaces	14
6.1.2 Server Classes	15
6.1.3 Client Classes	16
6.1.4 Setup Classes	17
6.1.5 Object Classes	17
6.1.6 Utility Classes	18
6.2 Main Dynamic Aspects	19
6.2.1 The Container Concept	19
6.2.2 Objects and Code Mobility	20
6.2.3. Objects – GUI Interaction?	24
6.2.4 Interface Transport	25
6.2.5 Persistence	28
7. Conclusion	30
7.1 Achievements	30
7.2 Future Work	31
7.2.1 Specification Aspects	31
7.2.2 Implementation Aspects	32
References	32
The Official Website	35
Appendices	35
A. The State Files	35
B. Compiling the Applications	36
C. Running the Applications	37
D. Utility Applications	39

1. Introduction

1.1 Document versus Virtual World Paradigms

In today's Internet technology, the distinction must be made between applications based on a document paradigm versus those based on a virtual world paradigm:

- Within the *document paradigm*, documents, often active ones being able to react to various user actions, are made available on one or several servers, and client applications (e.g. web browsers) can be used in order to interact with them. Typically, each user copies the documents onto her local machine and her interactions with them have no direct repercussions on their other connected users. In particular, a user never directly¹ modifies the original document. The underlying metaphor is the one of a huge cross-referenced book where each user browses through the pages totally unaware of other users performing the same task at the same moment. All actions are asynchronous and, thus, there is no need for a central server to coordinate user interactions with the pages of the book or to take care of an event redistribution mechanism.

The main advantage of this approach is that it allows a really massively distributed architecture with thousands of http servers interconnected all over the world. If a crash occurs, only the pages hosted by the failed or the no longer reachable servers become momentarily unavailable. The whole system is extremely robust and its scalability is truly optimal!

- Within the *virtual world paradigm*, multiple users and active objects interact in the *same space* and therefore have a direct impact on each other. Within such systems, if a user interacts with an object, the other connected users can see her and start a dialog with her. Moreover, it is possible for a user to modify some properties of the world and all the other users present in the same subspace (e.g. the same room) must immediately be made aware of it. Examples of the virtual world paradigm range from simple graphical chat to sophisticated 3D virtual worlds used for military simulations, refer to Section 2.2.

At the software architecture level, systems based on the virtual world metaphor are clearly the most complex ones. Indeed, the users directly interact with the original objects of the system and the resulting events must be correctly synchronized and forwarded in order to maintain the consistency of the world. Therefore, most of them are based on a software architecture with one central server containing all the world pertinent data and assuming the world accessibility, consistency and persistence and many clients allowing interaction with the other users and the various objects of the world. This approach has two main weaknesses. First, the whole system depends completely on the central server robustness. Secondly, it does not scale well. Just imagine what the World Wide Web would be today if all its contents would need to be regrouped on a single server having to coordinate all of its users.

1.2 Goal and Outline of this Paper

The goal of our research group is to define and test promising software solutions in order to support the virtual world paradigm presented above, without making concessions to the single server architecture. Actually, *MaDViWorld*, the acronym of the prototypal software framework presented in this paper, stands for Massively Distributed Virtual World, since its subspaces (e.g. rooms) are distributed on an arbitrarily large amount of machines. The only requirement is that each machine containing a part of the world runs a small server application and is connected to other machines through the Internet. This obligation is exactly the same as the one required by the World Wide Web with all its advantages in terms of robustness and scalability. Note that the definition of prototypal architectures for really distributed virtual worlds is a new active field of research tackled both by the vi-world community and the distributed software community. An interesting attempt to build a highly distributed system has been made recently by the V-Worlds Group at Microsoft research as an enhancement to their V-Worlds platform (refer to [VWorlds, 2001] and [Eames, 1999]). Each server manages a room and its content. An SQL server hosted on one of the world server machines handles the synchronization mechanism.

¹ An indirect outcome is, however, possible by means of techniques such as forms, CGI and servlets. For example, a document which is generated on the fly by a program running on an http server depends often on data fetched from a data-base through HTML forms by its users.

Other proposals for completely distributed environments are presented by [Fabre et al., 2000a] in URBI et ORBI, by [Frécon et al., 1998] in DIVE and by [Greenhalgh and Benford, 1995] in MASSIVE.

As pointed out in the previous paragraph, this paper investigates possible solutions for the creation of a highly distributed virtual world software framework. Furthermore, it concentrates on our MaDViWorld prototypical implementation for illustrating the proposed solutions. In order to best achieve this goal, the paper is structured as follows: **Section 2** presents a brief definition and a historical overview of the virtual world metaphor and introduces the reader to the accompanying terminology, i.e. avatars, rooms and objects; **Section 3** summarizes, both at the user level and at the vi-world developer level, the technological objectives, which was the motivation behind the project of creating the MaDViWorld framework; **Section 4** describes a sample user session with a very primitive virtual world implemented on top of the MaDViWorld framework. Its goal is to provide the reader with a feeling of what a possible final product would look like before entering into complex architectural considerations; **Section 5** presents an overview of the software architecture of the current² version of the MaDViWorld framework by sketching the three layers of its class hierarchy. This section is the most technical of the paper and represents its true contribution. For the interested reader, **Section 6** presents the same material at a more detailed level; finally, **Section 7** enumerates some of the conclusions we have drawn from our work, and contains some suggestions for improvement at various levels.

1.3 Conventions

The following formatting conventions are used throughout the present article:

- The ***Bold italic*** font is used for emphasis.
- The Letter Gothic font is used in all Java code and generally for anything which would be literally typed in a program, including keywords, data types, constants, method names, variables, class names, interface names and commands when they are part of an example. This font is also used for commands in the text, email addresses, websites, options, file and directory names.
- The highlighted Letter Gothic font is used in Java code examples in order to outline important pieces of code within the example context.

As far as gender is concerned, we systematically select the feminine when possible.

2. Virtual Worlds: a Brief Presentation

2.1 Terminology

In the book of [Diehl, 2001], ***virtual worlds*** are defined as "computer-based models of three-dimensional spaces of objects with restricted interaction". It is further asserted that ***multi-user worlds*** are distinguished by the fact that several users (working on different machines) can move through the world and interact with one another or with the objects at the same time. Finally, a virtual world is considered as ***distributed*** "if active parts of it are spread throughout different computers in a network".

The present article respects the above definitions on the condition of giving a rather permissive interpretation to the expression "models of 3D spaces". Indeed, we consider that a computer-based model of a given world should not necessarily respect a precise 3D, or even 2D topology. In a text-based model, for example, it is not necessary to precise the coordinates of avatars and objects present in a given room. A simple list which is automatically updated each time a change occurs in the room and which allows to interact with the user or the object of one's choice, might represent a suitable model for many purposes. The same is true for doors. It is certainly more realistic to place them in a precise location within a 3D or 2D wall and to force the user to virtually open them in order to move into another subspace. It is, however, also possible to present a list of available doors in a given room and to automatically transfer the user in a new context (i.e. new room, with new users present and new objects available), when she selects a given door. To summarize, we believe that the feeling of being in a space with other users and objects can be achieved within a very simple topological model (by only defining

² The 1.3 beta version.

rooms, for example). Naturally, the underlying architecture of the world should be conceived in such a way that the addition of a more realistic 2D or 3D topology remains a possible option.

For a good comprehension of the present paper, the three following terms need to be explained in more detail:

1. **Avatars.** Diehl defines them as the "virtual representations of the users". He then explains various perspectives from which the user can look at the scenery (i.e. through the eyes of the avatar, with a camera on top of it or with a camera behind it) and he finally states that in multi-user worlds an avatar should also somewhat be the visual representation of its user.

Within this paper, we adopted the following close but nevertheless easier to apply definition: an avatar is a tool which allows a given user to move through the world, to interact with its inhabitants and objects, and which lets the other users know where she is and what she is doing. This definition does not contradict the one of Diehl, but it also enables an avatar to be a simple client text-based application broadcasting the actions of its user through simple messages to the users of avatars sharing the same subspace.

2. **Rooms.** When a virtual world does not have a precise topology, it is very difficult to distinguish between near and distant elements. Thus, it is essential to divide the world into subspaces where the users might or might not enter and in which all interactions take place. Otherwise, the world would not scale. In the present paper, we call such subspaces rooms, and an avatar can move from one room to another either by using the **doors** available in a given room or by direct transfer if she knows the address of the destination room.
3. **Objects.** In [Kirsh, 1998], the author distinguishes between three categories of objects: (1) **passive objects**, which can only change if a human agent (through an avatar application) interacts with them. These objects do not react to change in other objects. A simple whiteboard on which each user can write short messages is an example of such an object; (2) **reactive objects** which can change their states in response to changes in other objects. These objects typically obey "physical" laws of interaction. When hitting a wall, for example, a ball will rebound at a given angle and velocity; and (3) **active objects**, which can transform themselves. A whiteboard, which would adapt its size to the number of participants in a room, is such an object. An active object has a minimal intelligence built into it.

We strongly think that a virtual world architecture should allow all three types of objects mentioned above and that this can be achieved by modeling them as instances of software objects (i.e. the combination of a state and of methods which can be executed by the host machine). Furthermore, in a distributed world, it should be possible to physically transport a given object from a room on machine X to a room on machine Y (refer to Section 3).

2.2 Historical Overview

The history of virtual worlds began with the first **MUDs** (Multiple-User Dimensions or Dungeons) designed as role-playing adventure games in the mid-seventies. A MUD is a shared textual virtual environment, which can be explored by its users through a series of simple commands such as "look", "go" and "take". As she moves within the world, a user runs across other users with whom she either chats, collaborates for a given task or fights for resources. A detailed description of MUDs is given in [Reid, 1994] and a huge amount of well structured information and resources on actual MUDs (some having evolved from a textual based representation to becoming sophisticated 2D or 3D models) can be found on the [MUD, 2001] website.

In 1990, object oriented features (i.e. the possibility to consider the entities of the world as instances of programmable software objects) were added to the MUDs, which were renamed **MOOs** (MUDs Object Oriented). **LambdaMOO** was the first one and was officially opened at Xerox PARC by Pavel Curtis in 1991 (refer to [Pavel, 1992]). It was designed for social gatherings instead of role-playing. Note that each MOO (or MUD) has its own creator (or God) who sets up all its rules and topology. Thus, it is always entirely contained and managed by a single central server where the users can connect through more or less sophisticated client applications. In that sense, MOOs are interesting for us from an end-user point of view, but their software architectures present very little appeal in terms of robustness and scalability. They are the opposite of being distributed. This limitation holds true for the first 2D and 3D chat worlds, which appeared on the Internet around 1995, as well as for the first multi-user games and

VRML-based virtual worlds. An interesting listing of such applications, as well as an introduction to their terminology and principal techniques (e.g. VRML³), can be found in [Diehl, 2001].

More recently, Microsoft developed its V-Worlds platform (refer to [Vellon et al., 1998]) on top of COM, which is still based on the concept of a central server, but plans exist to make it really distributed (refer to [Eames, 1999]). The direction of research taken by the Microsoft V-Worlds group is typical of the most recent trend in virtual worlds research. Researchers from the software engineering and distributed architecture communities try to apply technologies such as DCOM, CORBA and Java to tackle the difficult problem of defining standard protocols and of creating extensible software frameworks for supporting distributed virtual worlds. We have already mentioned some of these projects in Section 1.2.

3. Technological Objectives

Section 2 has clearly demonstrated that the creation of an appropriate software environment for supporting the development of multi-user distributed virtual worlds is both a new field of research and a very ambitious task. Thus, if one does not want to be overwhelmed by the complexity of the work, one must absolutely follow a feasible strategy based on a few well-defined technological objectives, which will be considered as having first priority. In the next two subsections, we will state these objectives more precisely in the context of the MaDViWorld framework. The first subsection considers the point of view of a simple user (i.e. a person who explores and interacts with a world developed on top of MaDViWorld), while the second one concentrates on the requirements of a system developer, (i.e. one who is ready to do some real programming in order to extend the framework or to create new types of objects).

3.1 The User Level

At the user level, the main idea is to keep the operations in the virtual world as intuitive and transparent as possible. In MaDViWorld, this objective is reached by satisfying the following non-exhaustive list of requirements:

- **World topology.** Rooms (as HTML pages for the world wide web) can be hosted on many machines world wide and they are topologically connected by doors. Avatars access a given room either by specifying its name and the IP address of its host machine or by transparently crossing one of its doors.
- **Interactions.** As soon as she has entered a room, a user is able to interact with other users and/or objects present in it. In principle, the users are informed of her actions.
- **Active/mobile objects.** Initially, each room is populated by a set of objects. These objects can be executed either on the remote machine where they are located or on the local machine of the user. They can also be copied (cloned), removed or simply "physically" transported by an avatar and moved to other rooms. It is further possible to provide these objects with sophisticated graphical user interfaces, which are always transparently executed on the local machine of the user.
- **World creation and extension.** A set⁴ of simple applications (wizards) must allow a user to effortlessly introduce newly programmed objects in the rooms of her choice, as well as to easily extend the global virtual world by creating and managing new active parts (rooms) on any machine running the appropriate server application. Actually this task is very similar to the one of creating new HTML documents and then making them available to other Internet users by running an http server.
- **Persistence and recovery.** The concept of persistence must be implemented. Either a part of the active world (i.e. the one existing on a given machine) or an avatar can be deactivated, its state stored on a local file and then reactivated. This must work both in the case of a voluntary interruption or in the case of a software or hardware failure.

³ Virtual Reality Markup or Modeling Language. At the end of 1997 a revised version VRML97 became an official ISO-standard.

⁴ Possibly a single one.

- **Security.** An appropriate security policy must be enforced both at the machine and at the vi-world level. In the first case, it means that the operating and file systems of the host machines for the client (avatar) and the server (rooms) applications must be protected from potentially dangerous operations (e.g. through a sandbox model). In the second case, rights within the vi-world itself must be defined and enforced (e.g. not every avatar should be allowed to enter any room, nor to interact with, delete, move or copy any objects).

Note that the possibility of creating 3D worlds and/or using audio or video interfaces, although not excluded, is not on our first priority list.

3.2 The Vi-World Developer Level

At the vi-world developer level, the key factor for success is to propose an open architecture, allowing a knowledgeable programmer to easily extend or ameliorate the current version of the framework. Section 5 will describe the technical choices (e.g. adoption of a layered class architecture, separation in well defined packages, etc.) implied by this objective. The non-exhaustive list below provides the reader with a preview of these requirements:

- The developer who wishes to add *new types of objects* to the world must be able to do so (1) by using a standard technology and (2) by respecting a few set of rules. An acceptable solution would be to require her to use the Java programming language and to offer her a structured set of abstract interfaces and concrete classes which have to be implemented and inherited, respectively, in order for an object to be "vi-world compatible".
- Although not trivial, the programming of *a new avatar* should be accessible for an average programmer. This implies that the concrete avatar implementation must be well separated from the other parts of the system (e.g. encapsulated in a Java package) and that the set of minimal conditions it has to respect must be regrouped in a somewhat abstract level (e.g. a Java interface class).
- Finally, the programming of *new types of rooms*, with more features or special behaviors than the standard ones, must also be possible for a developer who is willing to put some effort in understanding the philosophy behind the code of the existing framework. This can be achieved by offering an abstract definition of room properties at the highest possible level. These properties must always be respected (implemented), but they can also be extended or redefined through an object-oriented language inheritance mechanism.

4. A Sample User Session

This section presents a sample user session with a very simple⁵ MaDViWorld-based virtual world. Its purpose is to illustrate some of the main functionalities of our actual prototype, without entering into its more complex technicalities. In order to best achieve this goal, we consider a typical scenario decomposed into seven major steps.

4.1 Launching a Server and Setting up a Room

Step 1. Sylvia launches the MaDViWorld server application on her machine. Then she uses the wizard in order to create a new room "R1" and for adding a new TicTacToe active object in it. She furthermore decides that every other user can enter R1 and play the TicTacToe game.

⁵ It is composed of only two rooms R1 and R2 on two different machines, a simple active TicTacToe object and three users: Sylvia, James and Hans.

4.2 Launching an Avatar

Step 2. By launching the MaDViWorld avatar application, Sylvia enters the virtual world. She chooses an avatar name (actually Sylvia) and specifies the access path of the room she wants to be connected⁶ with (i.e. //134.21.9.252/R1).

Step 3. Another user, James, launches the avatar application from his machine and enters R1. The avatar application displays all the rooms connected with doors to the current one as well as the lists of all the users and objects in the room.

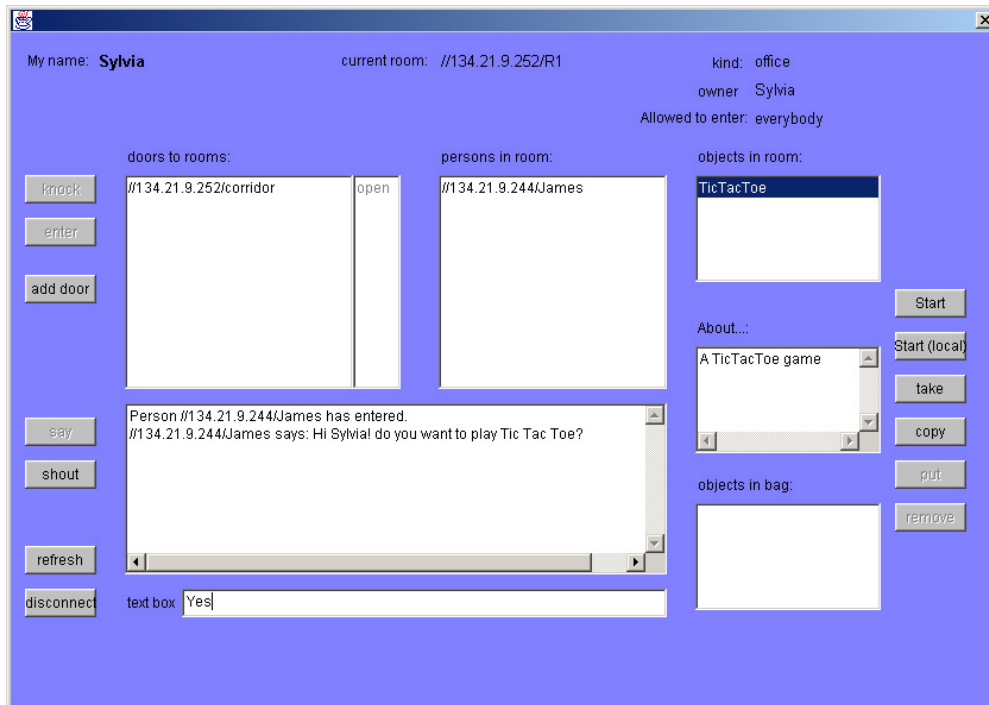


Figure 4.1 Avatar client application

4.3 Using Objects

Step 4. James sees that Sylvia is in R1. He invites her to play the TicTacToe game by sending her a message via the chat window (refer to Figure 4.1). She accepts and both users launch the TicTacToe user interface by selecting "TicTacToe" from the objects list and clicking on the "Start" button.

The game object recognizes the first and the second connected player (refer to Figure 4.2) and after each move waits for the right player to take her or his turn. Each game update is automatically reported on both clients.

⁶ It is also possible to access the corridor (i.e. //134.21.9.252/corridor) which is the default room on any machine running the server application. This is useful when a client does not know yet the existing rooms.

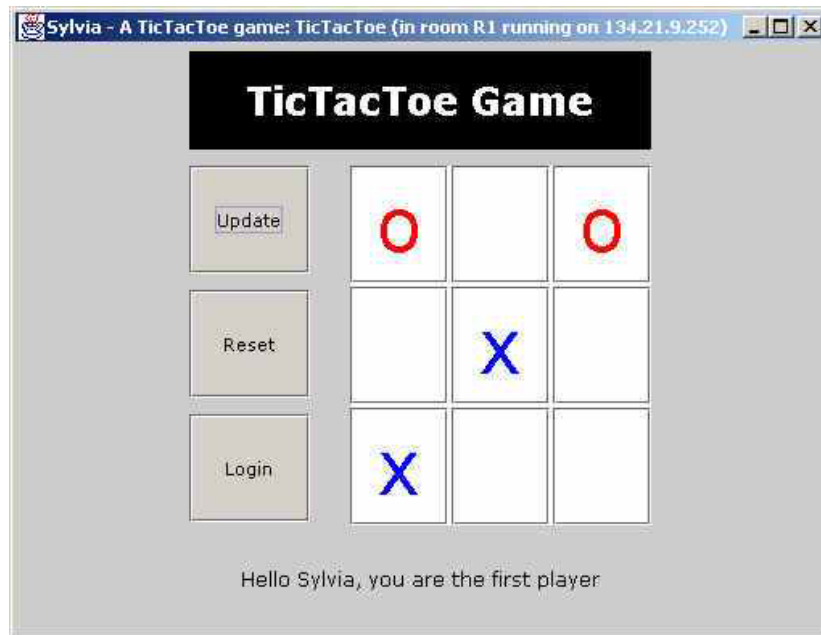


Figure 4.2 TicTacToe game interface

Step 5. A third user, Hans, also enters R1 and launches the TicTacToe object. Since this game does not accept more than two players, Hans is prevented from interacting with it. He can, however, observe the game evolution.

4.4 Objects Mobility

Step 6. Sylvia and James decide to transport the game to another room and play the revenge there. James launches the server application and creates a new room "R2" on his machine. Then he connects it with R1 by creating a door between the two rooms.

Accessing a room via a door is easy, since each avatar automatically displays each new connected room with the current one.

Step 7. In order to move the TicTacToe game to room R2, Hans selects the object and clicks on "take". The object is automatically put into his avatar's bag (refer to Figure 4.1). Then he enters R2 and clicks on "put". Sylvia joins R2 and both users replay the game. Note that the TicTacToe object has been "physically" moved from room R1 to room R2 and that it is no more available in R1. If James had copied it instead of taking it, there would be now two distinct TicTacToe instances, one still in R1 and the other in R2.

It is interesting to notice that even the oversimplified world presented in this section is still rich enough to inspire us with at least two potentially interesting vi-world applications:

- The first consists of a set of rooms full of active game objects, ranging from simple single user arcade games to sophisticated multi-user ones. After having paid a fee, users are allowed to visit the rooms, to watch other users play, to join a game and to exchange their impressions. If she is interested, a user can even copy a given game object onto her own machine by getting the right to clone it.
- A slightly modified version would be to replace the cloneable game object by active pieces of art, which would be unique in the sense that one could only move them around, not copy them.

5. The MaDViWorld Software Architecture

MaDViWorld is a collection of 31 classes organized in 8 packages representing more than 5,000 lines of Java source code. The MaDViWorld platform was developed with the Java 2 platform Standard Edition (J2SE™ version 1.3). The eight packages regroup all the classes needed by the three deployment parts of MaDViWorld in order to set up, manage and use a virtual world. In Figure 5.1,

which illustrates the whole⁷ static diagram of the MaDViWorld framework⁸, these three parts are separated by vertical dotted lines. From left to right, they correspond to the following independent applications:

- a *client application*, called avatar, for exploring the world, as well as interacting with its objects and other avatars;
- a *room server*, for creating, hosting and managing the rooms of the world on a given machine;
- a *room setup utility*, called room wizard, in order to remotely set up rooms and put objects into them.

The packages are further distributed into three hierarchical layers, separated by horizontal dotted lines in Figure 5.1:

- the MaDViWorld *Specification* layer,
- the MaDViWorld *System Implementation* layer, and
- the MaDViWorld *Object Implementation* layer

This section provides a high level description of the MaDViWorld framework software architecture. It is divided into three subsections, each describing one of these layers. Interested readers will refer to Section 6, as well as to the javadoc documentation of the MaDViWorld official website for more information on the precise role played by the many classes and methods of the framework.

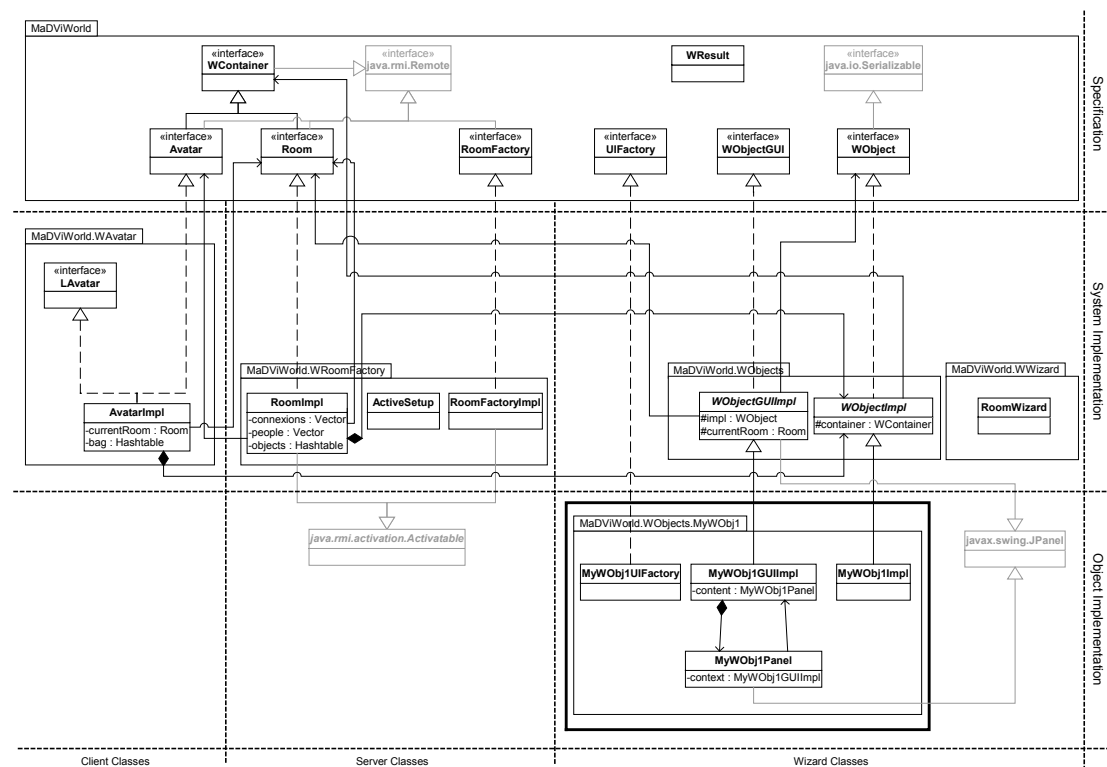


Figure 5.1 Static UML Diagram Class of MaDViWorld

5.1 Specification Layer

The MaDViWorld specification layer classes shown in the upper part of Figure 5.1 are illustrated to a larger extent with all their methods signatures in Figure 5.2. The classes of this layer encapsulate the conventions that each avatar, the room server and the room setup applications must respect in order to function and communicate with one another properly. For example, a room object should offer the

⁷ On this figure only six packages can be counted. Indeed, two utility packages used by every deployment part are omitted here for the sake of simplification. The presentation of these packages is out of the scope of this section but can be found in Section 6.

⁸ The classes in grey are not part of the MaDViWorld framework, but are important Java classes or interfaces which are extended by the framework.

5.2 System Implementation Layer

This section is divided into three parts corresponding to the vertical separations of the system implementation layer shown in the mid section of Figure 5.1. Note that the client, server and wizard classes presented below represent only *one possible implementation among many* of the ideas that were exposed in Section 3 and then clarified by the specification layer protocol of Subsection 5.1. Our goal is that any serious vi-world developer should be able to rewrite (or rather expand) these classes, by simply respecting a few set of well-established rules.

5.2.1 Client classes

The `MaDViWorld.WAvatar` package contains the `LAvatar` interface and an implementation class that, together with some graphical helper classes not shown in Figure 5.1, form the MaDViWorld client avatar application. The latter could be seen as a remote console showing the virtual world through the eyes of a virtual person (avatar) controlled by a human user. Note that the `AvatarImpl` class concretely implements both the `Avatar` and the `LAvatar` interfaces. The former is part of the MaDViWorld specification layer and has to be implemented imperatively in order to allow the avatar to exchange vital information with other avatars and rooms. The latter contains methods that we strongly recommend in order to implement a workable avatar. These methods are self-explanatory and their signatures are summarized in Figure 5.3. In opposition to the methods of the `Avatar` interface, they cannot be invoked remotely. Naturally, the look and feel of the user interface, as well as the way these methods are concretely implemented, are the result of a lot of developers' choices. Therefore the `AvatarImpl` class could be greatly customized or extended by a knowledgeable MaDViWorld developer. For example, within our actual prototype, we have made `AvatarImpl` instances serializable. Thus, a user can save the state of her avatar and reactivate it whenever she wants.

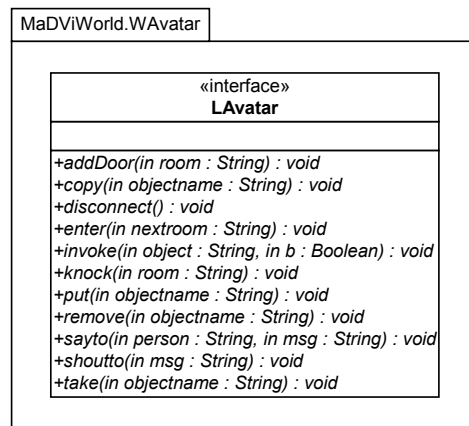


Figure 5.3 The `MaDViWorld.WAvatar.LAvatar` interface

5.2.2 Server Classes

The classes of the server layer are bundled in the `WRoomFactory` package and are used in order to support the creation and the management of a virtual world active part (i.e. the rooms and indirectly their contents). It is important to notice that the rooms are distributed on many machines each running the room server application.

A room creation occurs when a wizard application asks a `RoomFactoryImpl` object to create a proper `RoomImpl` instance. Although these two objects must fully respect the protocols defined for them within the specification layer, they are both highly adaptable. Indeed, (1) the `setRoom()` method of the `RoomFactoryImpl` class takes an array of strings as argument, which basically puts no restriction on the parameters that can be applied to set up a room; and (2) every system developer is free to make her own choices in specifying the way the protection and permission mechanisms of a room work by implementing her own version of critical methods such as `hasAccess()`, `knock()`, `removeObject()` or `setObject()`.

Within our actual prototype, we have made `RoomFactoryImpl` and `RoomImpl` instances activatable and we have used the Java activation model in order to provide a robust server persistence mechanism. This explains the presence of the `ActiveSetup` class in the `MaDViWorld.WRoomFactory` package. For general

considerations on the Java remote object activation framework, the interested reader will find [Flanagan et al., 1999], [Li, 2000], [Oaks, 2000] and [Edwards, 2001] worthwhile studying.

5.2.3 Wizard Classes

The role of the wizard layer classes is to allow users to install their own rooms with their own custom objects on a previously known remote server machine. Within our actual prototype an ad hoc RoomWizard graphic application collects the required information from a human user and then invokes the `setRoom()` method of a remote RoomFactory object, as well as the `setObject()` method on the newly created room with an instance of the custom object class to be added as argument. Note that the server needs no prior knowledge of the objects that will be installed in its room, except that they all implement the `WObject` interface. Each of these custom objects will also have to provide a graphical user interface, for which the only restriction consists in implementing the `WObjectGUI` interface. The concrete implementation of the objects and of their graphical user interfaces represents the topic of the next subsection.

As this wizard application has no other constraints to respect, it offers a lot of extension possibilities in the way the user interface is presented and in the way the pre-programmed objects are made available to the users.

5.3 Object Implementation Layer

It is our main intent that any knowledgeable Java programmer might create new classes of objects and then populate our virtual world with their instances. In order to gain a large community of such users, we made this process both very simple and as little constraining as possible. Suppose a programmer wants to add instances of a new class of objects named `MyWObj1`. She must merely respect the following strategy: (1) she creates a `MaDViWorld.WObjects.MyWObj1` package as shown in the bottom right part of Figure 5.1; (2) once the newly created package has been successfully compiled on her own machine, she uses the wizard application in order to create a local instance of the new object type; and (3) she uses once again the wizard to install the newly created instance in any local or remote room she may have access to. Other users can then copy and/or move this original instance, thus populating the world with many copies of it, each having its own identity and state. Note that the `MaDViWorld.WObjects.MyWObj1` package is always composed of the four following classes, each having a clearly defined role:

- The `MyWObj1Impl` class implements the concrete methods that the object will be able to execute.
- The `MyWObj1Panel` class contains the *graphical user interface*, which is totally separated from the object implementation. We strongly suggest that this class extends the standard `javax.swing.JPanel` class. This way, it will be possible to directly edit its graphic components with any standard Java IDE⁹, like Forte™.
- The `MyWObj1UIFactory` and `MyWObj1GUIImpl` classes are lightweight classes which could be automatically generated. While the first one simply delegates the job of concretely creating the graphic user interface to the `MyWObj1Panel` class, the second just implements the `getUI()` method of the `UIFactory` interface. The reasons behind this apparently complex architectural option are clearly exposed and justified in Section 6. Nevertheless, the general idea is simple. Each object (local or remote) is controlled through a graphical user interface that always has to run locally. Therefore, the latter has to be dynamically downloaded from the room server machine to the client avatar machine. To transparently achieve this goal, we use the factory pattern (refer to [Gamma et al., 1995]) twice. The first factory, `WObject`, generates `UIFactory` objects which are themselves `WObjectGUI` objects factories. This approach was inspired by one of the first successes of the Jini.org Jini Community process, the ServiceUI project, led by Bill Venners of Artima Software [JiniUI, 2000].

⁹ Integrated Development Environment

6. Concrete Implementation

This section is divided into two subsections. The first one comes back with much more details than in Section 5 on the software architecture of our framework, while the second one concentrates on the main dynamic behaviours of the system.

Note that the source code snapshots in this section are simplified (error handling, import statements, etc. are omitted for the sake of conciseness and simplicity). The whole source code, however, can be downloaded from our website (refer to The Official Website).

We shall not explain in detail each of the methods of the different interfaces and classes in this paper. The comments of all the methods can be read in the javadoc documentation (refer to The Official Website).

6.1 Software Architecture Revisited

As already seen in Section 5 the classes and packages constituting MaDViWorld are separated into three deployment parts. Figure 6.1 explicitly shows how the deployment of the six main MaDViWorld packages plus the two utility packages `MaDViWorld.util.WServer` and `MaDViWorld.util.WFetcher` is distributed over our client, server and wizard applications, while Figure 6.2 provides an expanded view of all the classes contained in these packages.

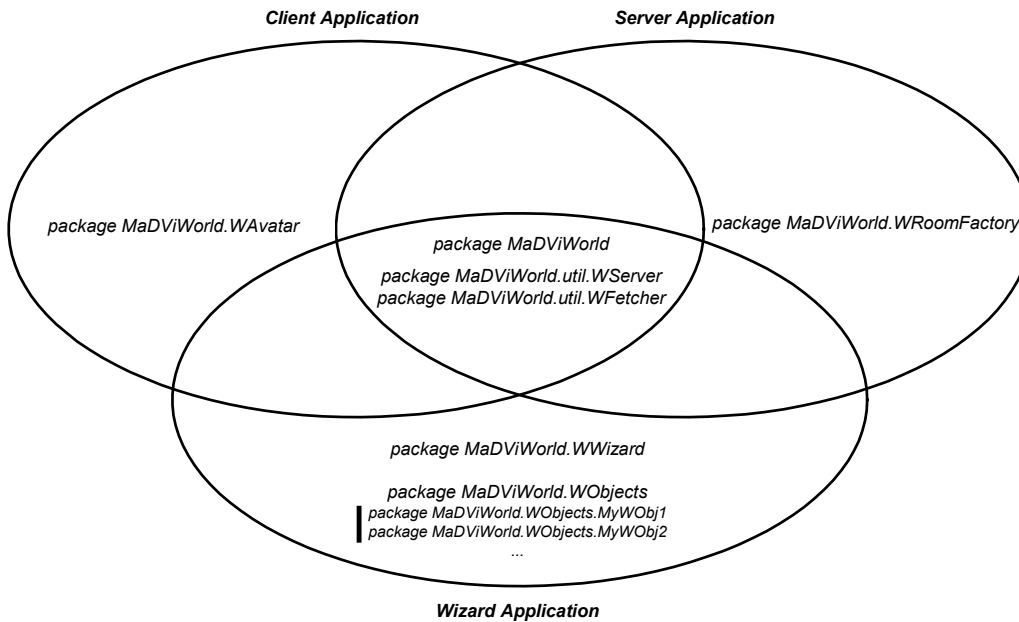


Figure 6.1 Packages layout for client, server and wizard

6.1.1 Specification Interfaces

The `MaDViWorld` package contains the core interfaces specifying all the main MaDViWorld protagonists i.e. avatars, rooms and the roomfactory. Figure 5.2 shows the detailed UML static diagram describing all these specification interfaces. Note that this package contains seven interfaces, plus the `WResult` class, a simple wrapper class for objects returned by object method invocations.

There are four RMI remote interfaces, namely `WContainer`, `Avatar`, `Room` and `RoomFactory`, which therefore all extend the `java.rmi.Remote` interface. Note that the `WObject` interface extends the `java.io.Serializable` interface.

These interfaces should define the minimal set of methods the different elements constituting a distributed world have to offer. At a conceptual level, these interfaces should be totally independent of their future implementation. Unfortunately, within the actual version of our prototype, they have been partially modified and adapted during the development of the implementation level. Therefore some

methods, which would seem redundant on such a specification level, may be removed in further versions and some, which would seem missing, may be added. Other methods will certainly have more generic signatures, like the `getInformation()` method of the `WContainer` interface whose return type is a string instead of a file of a given format (XML, VRML, etc.), which would seem more reasonable.

Package	Interface Abstract Class / Class	Client	Server	Wizard		
MaDViWorld	<i>Avatar</i>	x	x	x		
	<i>Room</i>	x	x	x		
	<i>RoomFactory</i>	x	x	x		
	<i>UIFactory</i>	x	x	x		
	<i>WContainer</i>	x	x	x		
	<i>WObject</i>	x	x	x		
	<i>WObjectGUI</i>	x	x	x		
	<i>WResult</i>	x	x	x		
	util	WServer	WClassServer	x	x	x
			WClassFileServer	x	x	x
		WFetcher	WClassFetcher	x	x	x
			WClassFetcherImpl	x	x	
	WAvatar	LAvatar	x			
		AvatarImpl	x			
		ConnectDlg	x			
		DoorDlg	x			
		LoggedDlg	x			
		QuestionDlg	x			
		SleepingAvatar	x			
		TestFilename	x			
	WRoomFactory	ActiveSetup		x		
		RoomFactoryImpl		x		
		RoomImpl		x		
	WWizard	RoomWizard			x	
		AttentionDialog			x	
	WObjects	WObjectGUIImpl			x	
		WObjectImpl			x	
MyWObj1		MyWObject1UIFactory			x	
		MyWObject1GUIImpl			x	
MyWObj2		MyWObject1Impl			x	
		MyWObject1Panel			x	
		MyWObject2UIFactory			x	
		MyWObject2GUIImpl			x	
		MyWObject2Impl			x	
		MyWObject2Panel			x	
...				x		

Figure 6.2 The 7+n packages and the 27+4n classes of MaDViWorld (n represents the number of objects)

6.1.2 Server Classes

This section takes a closer look at the `MaDViWorld.WRoomFactory` package. First of all, let us consider the `RoomFactoryImpl` class.

The latter is based on the well known *factory pattern* (refer to [Gamma et al., 1995]). Note, however, that our handles to the newly created rooms are not simple pointers, but RMI stubs. Subsection 6.4.1, which concentrated on explaining our vi-worlds persistence model, provides full details on the setting-up of the roomfactory and on the inner working of the `RoomImpl` and `RoomFactoryImpl` classes.

The roomfactory, when it starts, automatically creates a special standard room called "corridor". Thus, if a user knows that a server is running on a given machine, she also knows that there is a room "corridor" she can enter. This special room gives her access to all the rooms having doors to it. It is a kind of centralized registry of public accessible rooms. The creator of a new room has always the choice to set or not to set a door between her room and the corridor.

It is important to note that the parameters, which one passed to the `setRoom()` method, are strings contained in an array and that they are not defined in the specification. In our prototype, these parameters are the name of the room, the type of the room (office, conference room, corridor, etc.), the access rights (owner, professor, student, everybody), the owner, a parameter telling if the room will have a door to the corridor or not and a last parameter describing the room's topology, which is actually left unused. All these parameters are strings and this is too restrictive. In a further version, they will be general java objects, allowing the system developer to adopt a more sophisticated strategy for the description of the room or for the security model.

Concerning the `RoomImpl` class, note that it is still yet have a simplified version of what a true specification for a room would be. Almost all the methods (`setConnexion()`, `getConnexions()`, `getDescriptionOf()`, `getKind()`, `getObject()`, `getOwner()`, `getPeople()` and even `invoke()`, `say()`, `shout()`, `updateLists()`, `setObject()`,...) should take the name and the host of the avatar and/or a password as arguments in order to offer the system a way to check if the method can be invoked or not by this avatar. A reasonable model would be to allow these method invocations if and only if the avatar is currently in the room, i.e. has been granted access to the room by the `hasAccess()` method. For the `connect()` method, the name of the avatar is passed as argument and the room lets the avatar connect if and only if he has access. An analogous check has been done for the `removeObject()` method, which only allows the owner of the room to remove objects from it. The security model of our rooms is very simple and incomplete in this version of our prototype and a clearly defined security mechanism should be integrated in a further stage. The abstract model will be defined in the Specification layer and the Implementation layer will provide a working solution. The way each vi-world developer implements these methods is free, which allows for a great deal of customization!

Below we shall explain, as an example, how the `knock()` method was implemented in our prototype. The goal of the `knock()` method is to give an avatar a chance to enter a room where he is normally not allowed to enter, i.e. in a room whose `hasAccess()` method returns `false`. In our solution, the room explicitly asks the current room's owner, via its `getPermission()` method, if the new avatar is allowed to enter. If the owner of the room is not in the room at this moment, all the avatars that are currently in the room are granted the right to supply for permission. If one of them gives the permission, then the new avatar can enter the room.

6.1.3 Client Classes

This section discusses the `MaDViWorld.WAvatar` package. First of all, note that, this package contains the `MaDViWorld.LAvatar` interface (refer to Figure 5.3), which defines methods that we strongly recommend an avatar to implement. These methods describe a set of functionalities an avatar should be able to perform in order to interact with the world in which he lives, i.e. the rooms and the objects and the avatars in them. The actual avatar client application is composed of the `MaDViWorld.WAvatar.AvatarImpl` class, along with some graphical helper classes (refer to Figure 6.2).

In order to *move from Room A to Room B*, an avatar currently in Room A has to know the name of Room B and the IP address of the machine hosting this room in order to lookup the room in the `rmiregistry` and to get a handle to it. The avatar then invokes the `disconnect()` method of Room A and then the `connect()` method of Room B and, if he is allowed to enter, he is done and the other avatars currently in Room B can see him and interact with him. These functions are supposed to be implemented in the avatar's `disconnect()` and `enter()` methods.

Also note that an avatar can *copy or remove objects from a room and put objects into it* by simply invoking the `getObject()` and `setObject()` methods of his current room. Methods `put()`, `take()`, `copy()` and `remove()` are supposed to implement these functions.

The mechanisms used to implement the other tasks of an avatar (`addDoor()`, `knock()`, `sayto()`, `shoutto()`) are very analogous to those mentioned above, allowing for many improvements in the next version.

To conclude this section, it is necessary to explain how an avatar *invokes methods on objects* that are in the room where it currently stands. This concerns the concrete implementation of the `invoke()` method of the `LAAvatar` interface (refer to Section 6.2.3 for the coding details). For the moment being, just take good notice that the only method an avatar has to invoke itself on an object is the `getUIFactory()` one. All the other interactions have to be made through the object's GUI, which is generated at runtime by the avatar.

6.1.4 Setup Classes

The purpose of the `MaDViWorld.WWizard` package is to provide a simple application to install rooms on a given server. Its role is just to gather all the informations the `RoomFactoryImpl` needs in order to create a new room and to populate it with new objects. In order to allow the creation of such custom objects, the `RoomWizard` absolutely needs to be deployed with the `MaDViWorld.WObjects` package described in the next section.

First let us see concretely how the `RoomWizard` does its job:

```
package MaDViWorld.WWizard;

import MaDViWorld.*;
import MaDViWorld.util.WServer.*;
import MaDViWorld.WObjects.*;
import MaDViWorld.WObjects.MyWObj1.*;
...
public class RoomWizard {
    ...
    void doFinish() {
        ...
        Room r = null;
        RoomFactory factory = null;
        ...
        // Looking up the RoomFactory
        factory = (RoomFactory) Naming.lookup("rmi://" + HostName + "/RoomFactory");
        // Creating the room with the gathered args
        // args contains the name, the access rights, the type and the owner of the
        // room and tells if the room should have a door to the corridor or not.
        r = factory.setRoom(args);
        // Creating a new object instance
        MyWObj1Impl myobj = new MyWObj1Impl("mywobj1","",room);
        // Putting the just created object in the room
        room.setObject(myobj);
        ...
    }
    ...
}
```

Note that the way the input data are gathered is not important and, the choice being left to each `RoomWizard` developer.

6.1.5 Object Classes

This section concentrates at the `MaDViWorld.WObjects.*` packages and the role played by each of its classes.

There are two abstract classes: `WObjectGUIImpl` and `WObjectImpl`, which are used by each concrete custom object. These classes implement all the methods of their interfaces (refer to Figure 5.2), `WObjectGUI` and `WObject` respectively, that are common to the interfaces and implementations of all objects. Concretely, `WObjectGUIImpl` implements all the methods defined in `WObjectGUI` except `getSize()` and `WObjectImpl` all the methods defined in `WObject` except `execute()` and `getUIFactory()`. The implemented methods have self-explicit names and need no more explanations here.

The remaining of the section concentrates on the *four classes* any Java programmer has to provide in order to create a new *custom object*, say `MyWObj1`.

- First she must create a single package `MaDViWorld.WObjects.MyWObj1`.
- This package contains two lightweight classes which could be generated automatically: `MyWObj1GUIImpl` and `MyWObj1UIFactory`. While the first one simply delegates the job of concretely creating the graphic user interface for the object to `MyWObj1Panel` and implements the inherited `getSize()` method, the second just implements the `getUI()` method of the `UIFactory` interface to allow polymorphism with these graphic user interfaces factories.
- The `MyWObj1Panel`, as just seen, implements the concrete graphical user interface and the interactions with the implementation of the object. It extends the standard `javax.swing.JPanel` class and therefore presents the great advantage that it can be developed and edited by every Java IDE. The detailed programming model is described in Subseciton 6.2.2.
- The remaining class, `MyWObj1Impl`, implements the inherited `getUIFactory()` method, which allows polymorphism with the objects implementations and the `execute()` method, which is a dispatcher for the class own methods. Concretely this is done like this:

```
package MaDViWorld.WObjects.MyWObj1;

import MaDViWorld.*;
import MaDViWorld.Objects.*;
...
public class MyWObj1Impl extends WObjectImpl {
    ...
    public WResult execute(Vector args) {
        String method = args.elementAt(0).toString();
        boolean changed_state=false;
        Object resp=null;
        // custom methods of this object
        if (method.equals("myMethod1")) {
            resp = mymethod1(args.elementAt(1));
            broadcast("The state of object "+getName()+" changed!");
            changed_state=true;
        }
        if (method.equals("myMethod2")) {
            resp = mymethod2(args.elementAt(1),arg.elementAt(2));
        }
        ...
        return new WResult(resp,changed_state);
    }
    ...
    private int mymethod1(Object a) {
        //do the job of the method...
        //and return the result
    }
    ...
}
```

The actual version of the `MaDViWorld Wizard`, which can be downloaded from the official website, includes four sample objects:

- a Fibonacci number calculator;
- a simple whiteboard;
- a clock, giving the actual time of the machine it is running on;
- a graphical TicTacToe game.

6.1.6 Utility Classes

If order to run a rmi Java application, an HTTP server for the transfer of stubs is required. Therefore, we provided our custom one in the `MaDViWorld.util.WServer` package. In addition, since we wanted object mobility, a second utility application had to be developed, a kind of dual application of the HTTP server, which we called a "class fetcher". This application is included in the `MaDViWorld.util.WFetcher` package. How this two utility applications collaborate and have concretely been developed is shown in the next paragraphs.

First we look at the class fetcher. It implements a simple remote interface with one method `getClass()`. Thus it is a remote invocable object whose job is to make HTTP requests for files it is told

to get. The arguments of the `getClass()` method are the IP-address and the port of an HTTP server, as well as the file name it has to get.

Secondly, let us consider the HTTP server. It is a simplified standard HTTP server with a slight modification in order to collaborate with a class fetcher. It handles the GET method of the HTTP/1.1 protocol (refer to [Fielding et al., 1999]) and a new method GE2 with the same syntax than GET. The mechanism is simple and is decomposed in the five following steps (refer to Figures 6.4 steps 2 to 6):

1. The HTTP server receives a GET request (from the classloader of a remote JVM¹⁰). For instance:

```
GET /MaDViWorld/Wobjects/WObjectImpl.class HTTP/1.1
User-Agent: Java1.3.0_01
Host: 127.19.8.248:8083
Accept: text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2
Connection: keep-alive
```

2. The server does its expected job serving the requested file to the classloader of the remote JVM.
3. The server looks for a class fetcher on the machine he just delivered the class to and, if it finds one, it invokes the `getClass()` method remotely on this class fetcher.
4. The class fetcher sends a GE2 request to the HTTP server. For instance:

```
GE2 /MaDViWorld/Wobjects/WObjectImpl.class HTTP/1.1
```

5. The HTTP server serves the file a second time to the class fetcher, which saves the file on the local harddisk.

In addition to this strategy, we defined *some conventions*: (1) the HTTP server, if it runs on a wizard machine, listens to port 8083; (2) if it runs on a server machine, listens to port 8084; (3) and if it runs on a client machine, on port 8085. We also know that the HTTP server on a wizard machine and on a client machine will only get requests from a machine hosting a room server and that the HTTP server running on a room server machine only get requests from a machine hosting an avatar application.

This simple mechanism allows the client of the object to download all the classes it will need to further serve this object to another computer. This is exactly what happens in steps 4, 5 and 6 of Figures 6.4 that will be explained in Subsection 6.2.2.

6.2 Main Dynamic Aspects

This section explains the main dynamic behaviours of MaDViWorld, i.e. the *event distribution* mechanism between the objects' GUI's and the objects themselves in Subsections 6.2.1, 6.2.3 and 6.2.4; *object and code mobility* in Subsection 6.2.2; and finally our vi-world *persistence* strategy in Subsection 6.2.5. Our implementation choices will be discussed and, at places where improvements seem possible, an insight on how a more elaborated version of our prototype could work like will be given.

6.2.1 The Container Concept

Considering our virtual world model it is clear that containers are important in the MaDViWorld architecture: rooms contain avatars and objects, avatars contain objects. Between avatars and rooms these relations concern objects on different machines. Therefore, these containing relationships represent an adaptation of the classical container model. Concretely, on the one hand `AvatarImpl` and `RoomImpl` instances have a hashtable with all the object instances they contain, while, on the other hand, rooms just have a vector of strings describing the avatars, which are in the room. These strings allow the room to get a handle to the true avatar via a lookup in a remote `rmiregistry`. Figure 6.3 is extracted from the global static diagram shown in Figure 5.1 and focuses on these containing relationships. An avatar contains one or more objects and is contained in exactly one room. Rooms contain zero, one or more avatars and objects and can have several connexions, i.e. doors, to other rooms. Finally, each object can be contained into exactly one container.

This model induces a simple event model:

- If the state of a room changes (e.g. if an avatar leaves the room or if a door is created) the room immediately informs all the avatars it contains by invoking the proper methods of these latter (e.g.

¹⁰ Java Virtual Machine

PersonLeft(), RoomConnected()). The room could also, even if it is not the case in this version of our prototype, inform the objects it contains from the change and some reactive objects would change their state automatically.

- If the state of an object changes and the object is in a room, it can inform its containing room of this by invoking¹¹ the broadcast() method of the latter. The room can implement its broadcast() method as it wants. In the actual version of the MaDViWorld prototype it informs all the avatars in the room of the change by invoking their StateChanged() method. How this is done concretely is shown in the source code extract of the execute() method of the class MyWObjImpl in Subsection 6.1.5.

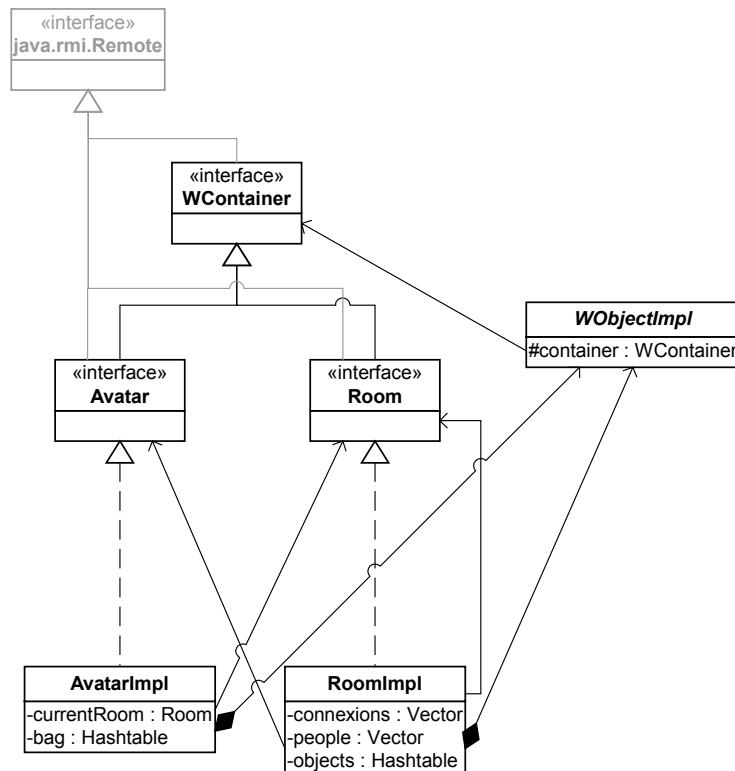


Figure 6.3 Static class diagram of objects and containers

This way, the information is naturally propagated to the right clients associated with avatars, and this without the need to have a kind of "bystander algorithm" as in the Microsoft's V-Worlds platform (refer to [Vellon et al., 1998]), since there is no kind of client cache in our system.

6.2.2 Objects and Code Mobility

The mobility of objects in MaDViWorld should follow the same rules as in the real world. This means that it has to be possible to physically move a given object from one room to another by carrying it in the avatar's bag through the world.

As an example, consider the following concrete scenario:

- A user puts a simple object (myObj) in a remote room with the help of her RoomWizard (refer to Figure 6.4a). Already at this stage, the myObj instance has to "fly" from the machine that created it to the receiving room.

¹¹ within the abstract class WObjectImpl, the method performing this invocation is also named broadcast().

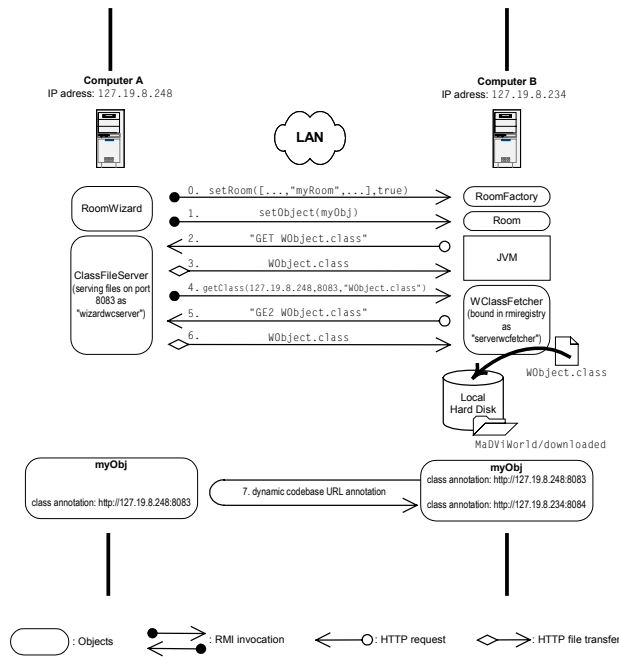


Figure 6.4a Object mobility: a wizard on machine A sets up a room on machine B and puts an object into it

- Then, an avatar running on a third machine comes in the room, likes myObj and wants to take it its bag (refer to Figure 6.4b)

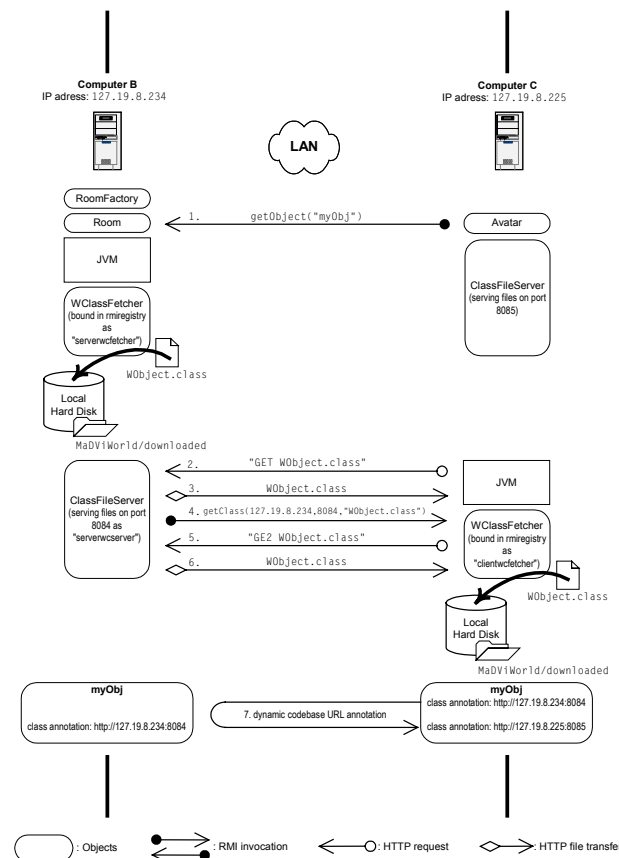


Figure 6.4b Object mobility: an avatar on machine C copies an object from the room on machine B

- In order to put myObj in another room running on a fourth machine (see Figure 6.4c) he walks back. Finally, the avatar decides to deposit myObj into another room, running on a fourth machine (refer to Figure 6.4c)

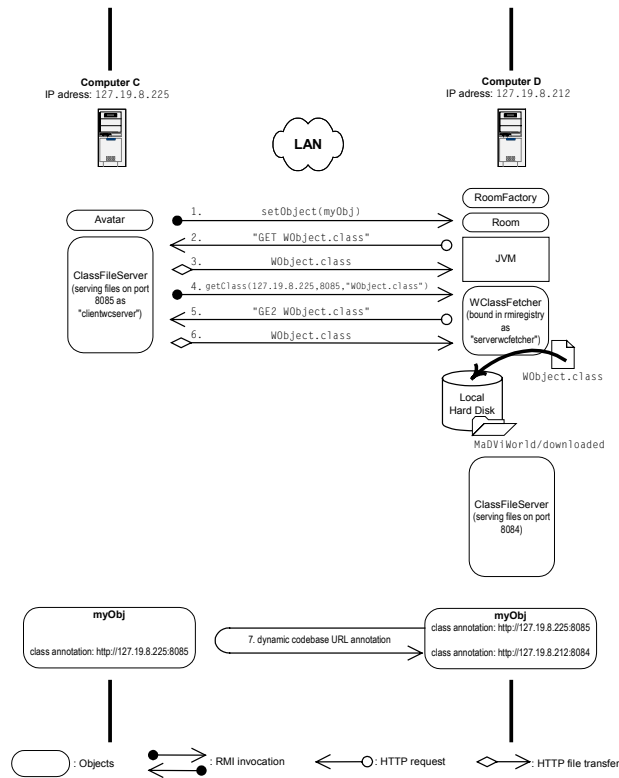


Figure 6.4c Object mobility: an avatar on machine C puts an object into a room on machine D

To sum up this scenario: the object moved from a wizard to a remote server (Figure 6.4a); then from this server to a remote client (Figure 6.4b); and finally from this client to a second remote server (Figure 6.4c). Such a scenario is very simple to achieve in MaDViWorld. Let us consider Figures 6.4a through c with greater care:

- In step 1, a serialized object is either passed as argument to a remote invocable method of a remote object (Room) or is received as a result of such a method invocation. In either case, the object has to be deserialized at the receiving end. As the receiving object does not have corresponding class information, the latter has to be downloaded from the class server, repeating steps 2 and 3 for each needed class, in order to reconstitute a copy of the original object. This is exactly the way RMI provides object mobility.
- The original approach of MaDViWorld are steps 4, 5 and 6, which are appended to the classical RMI mechanism and allow the actual code (.class files) transport for each class involved in steps 2 and 3. The clever part of our work is that we did not change the Java virtual machine classloader. Indeed, by simply listening the requests received by the HTTP server with the help of two utility applications, we transfer all the needed code. These applications ensure that the class files needed to create the myObj, as well as those implementing its graphical user interface, always move along with the object. Note that this strategy does not require that the first machine the myObj originally came from has to run during all the time. The mechanism provided by the MaDViWorld platform differs from the classical RMI object mobility model. Figure 6.5 illustrates this issue, as well as how RMI and MaDViWorld code transfer mechanism differ.
- Step 7 guarantees that the code location is updated by changing the codebase¹² of the object instance. This operation ensures the desired behavior showed in Figure 6.5. This **dynamic codebase annotation** is not possible in the actual Java 1.3 version thus we had to overcome this problem with a little trick. This manipulation is absolutely necessary, but has the drawback that the state of the object is lost. This should be resolved in Java 1.4, code-named Merlin, where better control over codebase annotation of classes is promised. If this feature is not provided, then we will add an ad hoc state recovery mechanism to the objects in a further MaDViWorld version.

¹² A specific HTTP URL, where the needed class files having to be downloaded can be found (e.g. <http://127.19.8.225:8084>).

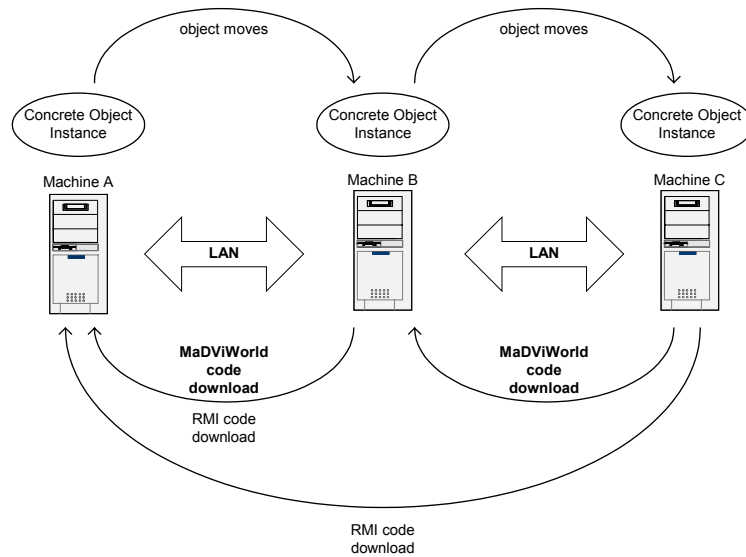


Figure 6.5 Object migration over several machines and the code download mechanism

To conclude, let us see concretely how the dynamic codebase annotation is implemented in the RoomImpl class:

```

package MaDViWorld.WRoomFactory;

import MaDViWorld.*;
...
public class RoomImpl extends Activatable implements Room {
    ...
    public void setObject(WObject o) throws RemoteException {
        ...
        Class c = o.getClass();
        try {
            // The class of c (c.getName()) is dynamically loaded
            // in order to change its class annotation to the current machine,
            // to be specific, to change its codebase dynamically
            // This is only possible if the .class files are already
            // in the classpath of this virtual machine.
            c = Class.forName(c.getName()); // Dynamically load the class
        } catch (Exception e) {...}

        // Create a new instance of the class c, which
        // will have the desired codebase annotation but will have an initial
        // state. Unfortunately, the state is lost.
        try {
            o=(WObject)c.newInstance(); // Dynamically instantiate the class
        } catch (Exception e) {...}
        ...
        //Now we can safely put the correctly annotated object in the room!
        objects.put(name,o);
    }
    ...
}

```

The same work has to be done in the AvatarImpl class in the take() method.

If dynamic object annotation were supported we would probably write this method as follows:

```

public void setObject(WObject o) throws RemoteException {
    String newCodebase;
    ...
    //The following command is not provided in Java!
    RMIClassLoader.setClassAnnotation(o.getClass(),newCodebase);
    //Now we could safely put the correctly annotated object in the room!
    objects.put(name,o);
}

```

6.2.3. Objects – GUI Interaction?

Figure 6.6 is extracted from the global static diagram of Figure 5.1 and focuses on the relationships between the object implementation, object interface and avatar classes.

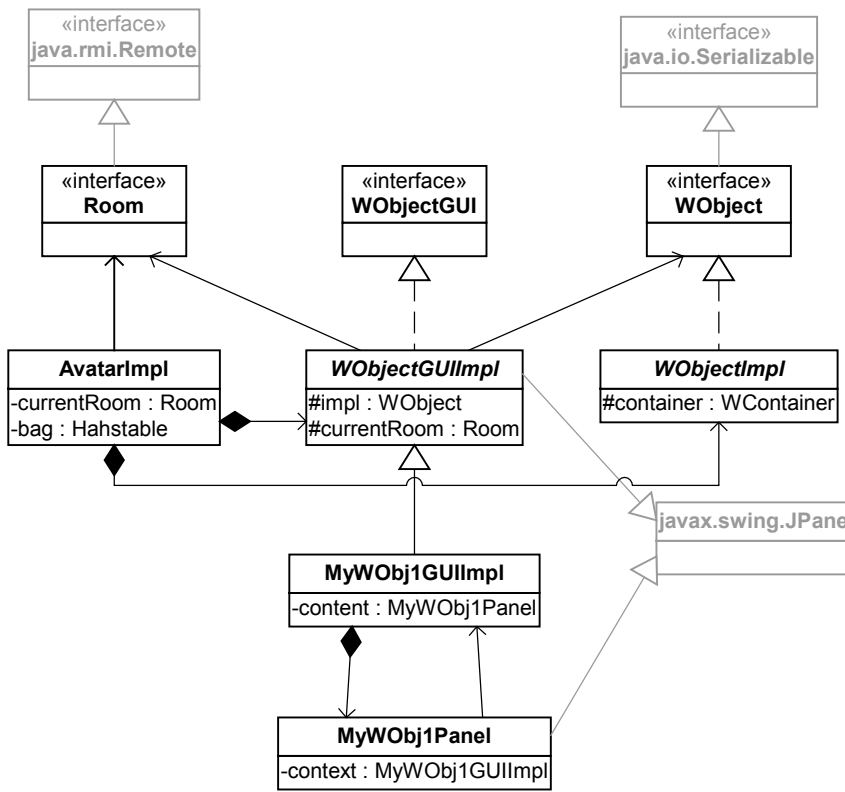


Figure 6.6 Static class diagram of objects and their graphical user interfaces

First of all, note that in the actual version of the MaDViWorld prototype, the `WObjectImpl` has no knowledge of the user interface decorating it. So the title of this paragraph should better be "How Are Methods of an Object Invoked by its GUI?".

As the interactions with an object exclusively happen through its GUI, the specific methods of an object only have to be known by its creator, the MaDViWorld object developer. This allows both a simple and flexible deployment of these objects. Neither the avatar programmer, nor the room a roomserver programmer needs to have any prior knowledge of them! This "magic" is possible because the `WObject` interface defines an `execute()` method, which acts as a dispatcher giving access to the intern custom methods of the object. So avatars, via the object's graphic user interfaces they dynamically instantiate, can interact with the local or remote implementation of the object. While a **local** execution is started by directly calling the `execute()` method of the object implementation, a **remote** invocation is performed via the remote invocable method `invoke()` of the containing room. Note that this latter action is possible although the objects themselves simply implement the `java.io.Serializable` interface and provide no direct remote access to their methods.

Let us see concretely how this works in the different classes that are involved in the mechanism. First of all, in the `MyWObj1Panel`¹³ class we have:

```

package MaDViWorld.WObjects.MyWObj1;

import MaDViWorld.*;
import MaDViWorld.Objects.*;
...
public class MyWObj1Panel extends javax.swing.JPanel {
    private MyWObj1GUIImpl context;
    ...
    public MyWObj1Panel(MyWObj1GUIImpl c) {

```

¹³ Note that this is the only class that an object developer must program himself in order to invoke a method on the object.

```

        this.context=c;
        ...
    }
    ...
    private void exampleButtonActionPerformed (java.awt.event.ActionEvent evt) {
        Vector tmpvec= new Vector();
        //In the following three lines the method invocation is simply done!
        tmpvec.add("myMethod");
        tmpvec.add(myArg1);
        Object o=context.invoke(tmpvec);
        //do whatever you want with the Object o returned by your method invocation
        ...
    }
    ...
}

```

The `invoke()` method of `context` distinguishes between local and remote invocation. If the call is remote, the `invoke()` method of the containing room is executed:

```

package MaDViWorld.WObjects;
...
import MaDViWorld.*;

public abstract class WObjectGUIImpl extends javax.swing.JPanel
    implements WObjectGUI {
    ...
    public Object invoke(Vector v) {
        Object o=null;
        try {
            if (isRemoteInvocation())
                {o=((Room)getWObjectContainer()).invoke(getObjectName(),v);}
            else {o=(getWObjectImpl()).execute(v).getResult();}
        } catch (RemoteException e) {}
        return o;
    }
    ...
}

```

The room is a relay for the remote method invocation and does its job like shown below:

```

package MaDViWorld.WRoomFactory;
...
import MaDViWorld.*;

public class RoomImpl extends Activatable implements Room {
    ...
    private Hashtable objects = new Hashtable();
    ...
    public Object invoke(String object,Vector args) throws RemoteException {
        ...
        WObject wo=(WObject)objects.get(object);
        WResult r = wo.execute(args);
        ...
        return r.getResult();
    }
}

```

What the `execute()` method of the object implementation has been shown in Subsection 6.1.5.

6.2.4 Interface Transport

In this section, we look at the way a user, through her avatar, *invokes methods on objects* that are in the remote room occupied by her avatar. Technically, this concerns the concrete implementation of the `invoke()` method of the `LAvatar` interface (refer to Figure 5.3). It is important to note that the mechanism developed in our prototype was inspired by one of the first successes of the Jini.org Jini Community process, the ServiceUI project, led by Bill Venners of Artima Software [JiniUI, 2000]. The core idea is that the graphical user interface and the implementation parts of an object should be clearly separated (see [Venners, 1999]).

Before we begin let us repeat that our framework offers us two ways to work with objects:

- **locally**: having both the graphical user interface and the implementation of the object running on the local machine hosting the avatar application.
- **remotely**: having only the graphical user interface running locally, and the whole implementation part of the object doing its work on the remote machine.

The dynamic generation of the object interface is done in five steps, which are commented below.

```
package MaDViWorld.WAvatar;
...
import MaDViWorld.*;
...
public class AvatarImpl extends UnicastRemoteObject implements Avatar, LAvatar {
    ...
    public void invoke(String object, boolean remote) {
        ...
        WObjectGUI oGUI=null;
        UIFactory f = null;
        JFrame p=null;
        ...
    }
}
```

Step 1. A main frame, which will "embed" the object GUI, is created.

```
p=new JFrame();
...
```

Step 2. If the methods of the object will be invocated locally, the avatar application has to get a copy of the object named.

```
if (!remote) { //get a serialized copy of the object, which is in the room
    objet=(WObject)currentRoom.getObject(object);
    objet.setContainer(this);
    objet.setHost(hostname);
}
...
```

Step 3. The avatar or the client application has to get a UIFactory by invoking the `getUIFactory()` method of the object it wants to use. It can do this either locally or remotely as shown below.

```
if (remote) { // executed remotely, using the room as relay
    f = (UIFactory)(currentRoom.getUIFactory(object));
} else { // executed locally (on a copy of the object that is in the room)
    f = (UIFactory)(objet.getUIFactory());
}
...
```

Step 4. At this step, the avatar application, thanks to polymorphism, has obtained the appropriate UIFactory, which generates the corresponding `MyWObj1GUIImpl`. Below, what we call **interface transport** really occurs. The codebase of the UIFactory instance `f` points to the machine hosting the room, as well as the object implementation. Thus all the classes needed to build the interface are loaded across the network by the java virtual machine classloader.

```
oGUI = f.getUI(name, objet, cRoom, avatarname, remote);
```

Step 5. The newly obtained panel `oGUI` is added to the frame created in Step 1, which is appropriately adapted. This is possible because the `WObjectGUIImpl` abstract class extends the `javax.swing.JPanel` class.

```
p.getContentPane().add((JPanel)oGUI);
p.setTitle(oGUI.getTitleDescription());
p.setSize(oGUI.getSize());
...
}
...
}
```

Remember that we told in Subsection 6.1.5 that the `MyWObj1GUIImpl` class simply delegates the concrete job of building the graphic user interface for the object to `MyWObj1Panel`. Concretely, the only thing it does is to add a `MyWObj1Panel` instance to itself. The whole source code of this class is shown here:

```
package MaDViWorld.WObjects.MyWObj1;

import java.awt.Dimension;
import MaDViWorld.*;
import MaDViWorld.WObjects.*;

public class MyWObj1GUIImpl extends WObjectGUIImpl {
    private MyWObj1Panel content;

    public MyWObj1GUIImpl(String name, WObject i, Room c, String av, boolean r) {
        super(name,i,c,av,r);
        initComponents (); }

    private void initComponents () {
        content = new MyWObj1Panel(this);
        add (content);
    }

    public Dimension getSize() {return content.getSize();}
}
```

Figure 6.7 provides a pictural summary of this whole process. The reader is invited to compare it with the concrete example of Figure 4.2.

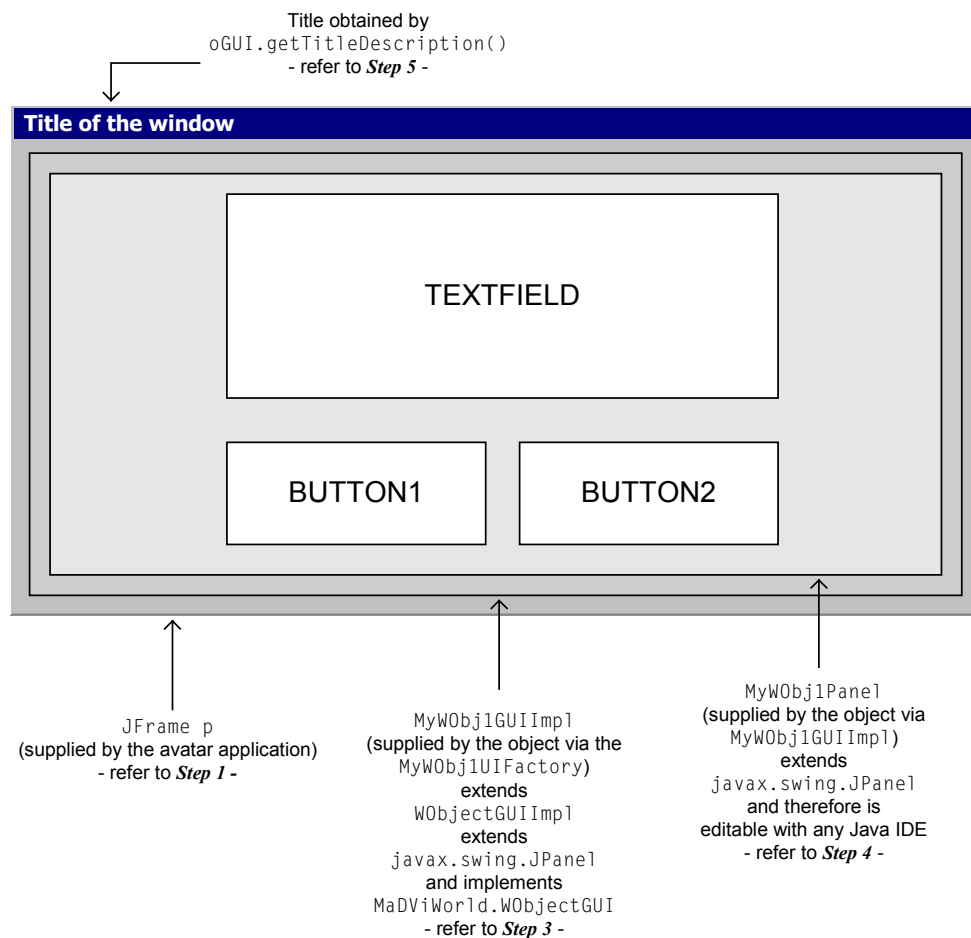


Figure 6.7 Composition of an object user interface

At this stage it is important to note that, when an object moves from one machine to another (for instance when avatars do a `put()` or a `take()`), the system implementator should always program the generation of a user interface, even an artificial one, in order to enforce steps 5, 6 and 7 of the code transport mechanism seen in Subsection 6.2.2. Otherwise the object GUI classes might not be transported. This is done in the `RoomImpl` class:

```

package MaDViWorld.WRoomFactory;
...
import MaDViWorld.*;

public class RoomImpl extends Activatable implements Room {
    ...
    public void setObject(WObject o) throws RemoteException {
        //artificially enforcing GUI generation in order to force the
        //transfer of the code.
        ((UIFactory)(o.getUIFactory())).getUI(null,o,null,null,false);
    }
    ...
}

```

Of course, the same holds for the avatar's `take()` method.

For the people, to whom this architecture seems overly complicated, let us see why this class structure was chosen. What we need is that `MyWObj1GUIImpl` directly extends `JPanel` in order to be directly edited with any Java IDE, like Forte™ for instance. However, `MyWObj1GUIImpl` must also extend the abstract class `WObjectGUIImpl`. In this case, the diagram class would be the one of Figure 6.8a. As multiple inheritance is not provided in Java, we had to simulate it as shown in Figure 6.8b, which is a snapshot of the whole static diagram already seen in Figure 5.1. This technique was inspired by [Malak, 2001].

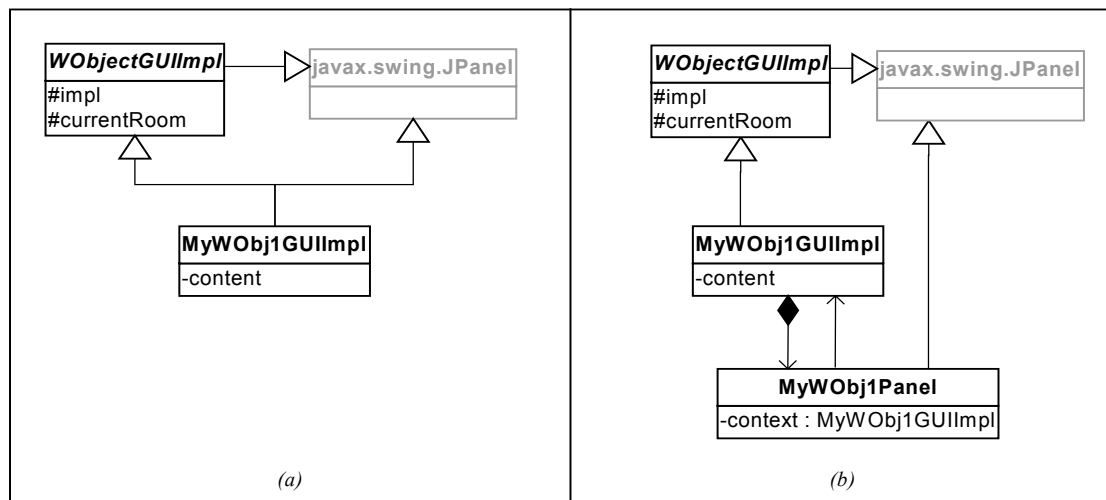


Figure 6.8 (a) "Ideal" class diagram that cannot be done in Java (b) MaDViWorld's solution

6.2.5 Persistence

In MaDViWorld, persistence plays an important role. Indeed, in every distributed application, robustness is a difficult task to achieve. A lot of runtime errors can happen. Therefore a persistence mechanism is needed to recover safely from them. Persistence, however, is not needed by all parts of the framework. The room wizard, for example, needs no persistence. It is just a setup utility that does its job and shuts down. It has no state and can be restarted as many times as necessary without any loss of information. On the other hand, the avatars, the rooms and the roomfactory do have a state and clearly need a persistence mechanism. This section presents our set of solutions to this problem.

The scenario illustrated by Figure 6.9 shows the worst failure that can happen, i.e. a hardware failure of a server machine.

- In the upper part of the figure, a small portion of a virtual world is presented. The world is composed of two servers ("server A" and "server B"), each hosting two rooms ("corridor" and

"Room1", respectively) and running on two different machines. There is also a client application, an avatar application, running on its own machine, and a wizard, which will be setting new rooms on server A. **All machines are running correctly.**

- In the lower part of the figure, a **hardware failure** of the machine running server A has occurred, implying that the two rooms it was hosting become unavailable to the other parts of the world. Therefore, if one tries to invoke a method on this server, a `java.rmi.ConnectException` is returned. The critical machine is rebooted and now "server A" has to recover. This is done by the following **simple recovery strategy**: (1) The registry daemon is restarted; (2) The HTTP class server (class `MaDViWorld.util.WServer.ClassFileServer`) is restarted; (3) The rmi daemon is launched; (4) And finally the `MaDViWorld.WRoomFactory.ActiveSetup` class is launched. Thus all the rooms with the objects they contain are restored and we return to a stable state.

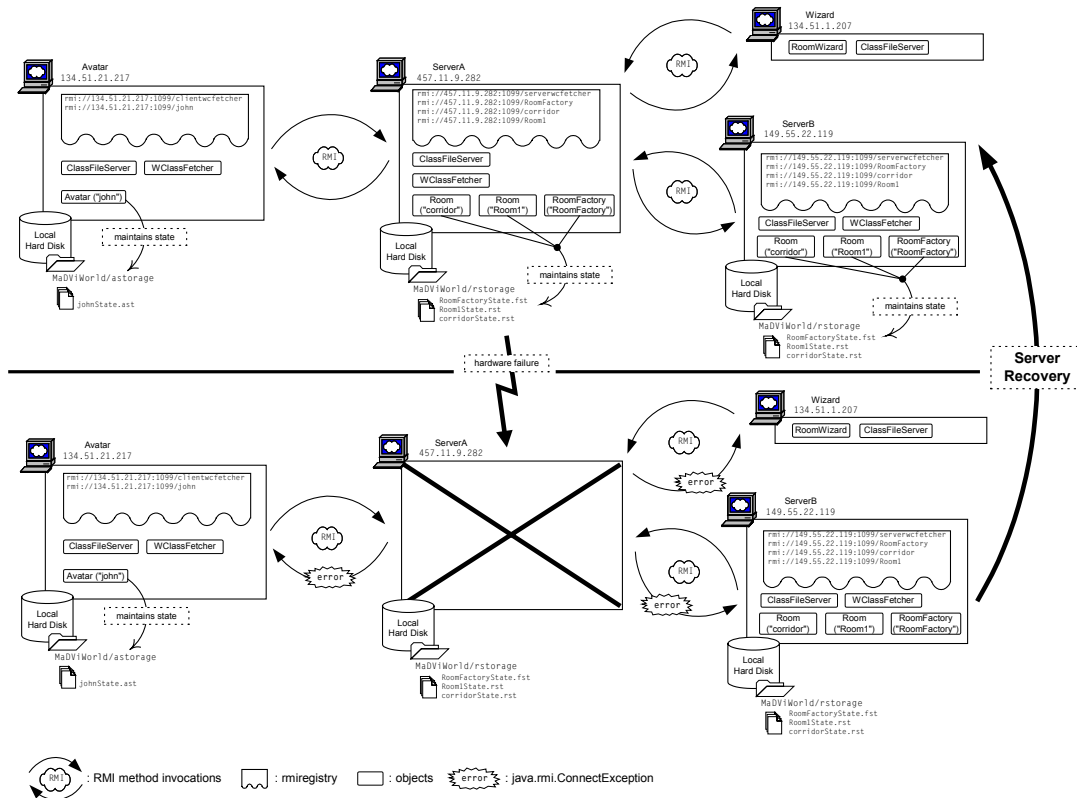


Figure 6.9 Persistence of MaDViWorld: crash and state recovery

The recovery of such a worst-case failure works perfectly well in our prototype. Note that other failures could occur. Only the `ClassFileServer` could crash. In this case, restarting it would be sufficient. Or we could have a software failure, i.e. the `rmdaemon` would crash. Then only the `rmid` should be restarted and all would go well again. We shall explain below our strategy for making our prototype robust. We shall consider first the server application and then the client application:

- **Persistence of the server (Rooms and RoomFactory):** The persistence of the MaDViWorld server parts, needs that each room factory has its own recovery mechanism and manages the state of the room it hosts. In MaDViWorld, the java activation system is used. The room factory and the rooms are `Activatable` objects, i.e. they extend the `java.rmi.activation.Activatable` class. This way, these RMI objects can live forever and attain virtual immortality. Moreover this means that these classes do not need a `main()` method. Instead, the activation system daemon (RMID) will be starting them on our behalf. For each activatable object or RMI service, we need a setup program that does most of the work for an `Activatable` object system. The setup program for the `RoomFactory` is the `ActiveSetup` class and the `RoomFactory` class acts as setup program for the rooms it will manage. These setup classes are common setup classes for activatable objects, but the properties of the activation group have to be set carefully and the mechanism to preserve state information between activations has to be done properly. The state of the roomfactory is written to a `.fst` file and each room has a `.rst` file containing its state. The interested reader is invited to

consult the source code on the official website for more details on how this is done in the MaDViWorld prototype as well as the appendices for the exact content of the state files.

- **Persistence of avatars:** An avatar walks through the world, collects some objects that it puts in its bag and chats with other avatars. Thus it has a history and a state. The user should be able to store this information and retrieve it afterwards. In our avatar application prototype, the state of the avatar can be serialized to a file stored on the local hard disk on the user's demand. Figure 6.9 shows how avatars maintain their state. The state is written to a `.ast` file (refer to Appendices). It would be possible to move this file to another machine and reactivate the avatar. We could even imagine an internet site serving avatar files for some custom avatars one could download. In a further version this mechanism will probably be improved. More important, method signatures supporting the persistence of avatars should become part of the `MaDViWorld.LAvatar` interface.

7. Conclusion

Our conclusion is divided into two subsections. While the first briefly summarizes the MaDViWorld framework main benefits, the second subsection concentrates on the work and research yet to be done.

7.1 Achievements

Considering the actual version of the MaDViWorld prototype, its main benefits are commented in a few words below:

- MaDViWorld is written in *pure Java* and benefits from all the related advantages. Particularly, it is platform independent and every computer with the standard JRE installed can use it.
- An *original and powerful use of Java RMI* and of its basic facilities, essentially object serialization and dynamic class loading, provides the cornerstone of MaDViWorld.
- MaDViWorld is for a *large audience*. There are no other material requirements besides a standard personal computer with a connection to a local network or the Internet.
- Virtual worlds built over the MaDViWorld framework are *really distributed*. Their rooms are spread over several computers connected via a network, and there is no need for a single host to have full knowledge of the world.
- MaDViWorld offers a very effective *persistence* mechanism. Each server guarantees the persistence of the part of the world it manages.
- Our framework is *extensible*. By merely respecting some standard interfaces, any Java programmer can customize and extend MaDViWorld or create her own classes of objects. Then, the end-user can add these completely new objects to the world while the system is running.
- MaDViWorld offers *class and object mobility* facilities. Objects are installed and moved around transparently, without the user or even the vi-world developer having to worry about the peculiarities of these tasks. Within MaDViWorld the avatar client applications automatically download all the code needed to interact with the objects selected by its user. This is a vital requirement for a flexible and extensible distributed virtual world, since the client application does not know a priori the particular implementation of the objects it will handle.
- All avatars in a common room *share the same objects* hosted on the server, which is the sole responsible for managing their states and persistence. So the clients do not need to locally cache a part of the world, and it is the task of the server to manage events distribution.
- Under the virtual world metaphor, MaDViWorld tackles many interesting and very actual technological issues. MaDViWorld is *at the crossroad of many Java products*: applets, Java Web Start, RMI, Jini and Javaspaces. It has characteristics of each of these and thus represents an original alternative. Figure 7.1 shows a comparison of MaDViWorld and these other technologies.

Finally, note that the goal of MaDViWorld was to implement a really distributed virtual world only using Java remote method invocation distributed technology and some basic Java features, and our prototype shows that this is feasible in many interesting ways. So we clearly decided that *MaDViWorld would not be a new Java middleware system*, like FlexiNet (refer to [Herbert et al., 1999]) and we chose not to use a legacy middleware platform that supports mobile services like Voyager (refer to

[Voyager, 2001]). We also ruled out to work with one of the many "agent" systems available because the majority of these systems basically consists of mobile script platforms that allow scripted agents to move around in a distributed network. Examples of these systems include Aglets (refer to [Aglets, 2001]). Nevertheless in order to have all the pieces of this "puzzle" fit together, we needed some "glue". Object providers and their users (rooms and avatars) have to agree upon a general collection of interfaces defining a standard protocol. The clearly defined and well-separated MaDViWorld Specification Level with its interface files shared between avatars, wizards and servers provides us this "glue".

	MaDViWorld	Applets	Java Web Start	RMI	JINI	JavaSpaces
Methods ¹⁴ can be executed locally	x	x	x			x
Methods ¹⁴ can be executed remotely	x			x	x	
Persistence	x				x	x
GUI of service/objects	x	x	x			
Class mobility ¹⁵	x	x	x			
Object mobility (from host to host)	x			x ¹⁶	x ¹⁶	x ¹⁶

Figure 7.1 Comparison between MaDViWorld and some of the main Java technologies

7.2 Future Work

Future improvement efforts will clearly be divided into two directions: (1) a total revision of the specification protocol; and (2) improving the concrete implementation.

7.2.1 Specification Aspects

The specification protocol of MaDViWorld has to be reviewed in more detail. The methods that have to be offered in the interfaces of the specification layer need to be refined. Some of the aspects that should be considered but are still not present in the current version of MaDViWorld are listed below:

- The `RoomFactory` should be much closer to a **room manager**. A `removeRoom()` method, for instance, is required.
- The **security model of the world**, i.e. of the rooms, is not well defined. The actual prototype just provides a minimal ad hoc security model. In a more elaborated version, rooms should have a password associated with them, and thus the new methods `setPassword()` and `changePassword()` should be available. So the room would be able to give access to methods like `changePermission()`, `changeKind()`, `changeOwner()`, `changeTopology()` only to the owner.
- The object concept is still too poor and should be extended. **Objects should interact** with each other: changes in a given object should be allowed to induce changes in other objects of the room. For instance, if we have a box and we put objects into this box, then moving the box implies that we also move all the objects in the box. This concept implies that some **objects are containers** of other objects (e.g. boxes, tables, bookshelves,...) and naturally, these new objects induce an **object visibility concept**. There would be closed (a box) and open (a box, a table) objects, and their visualization would be treated differently. [Vellon et al., 1998] discusses such a model and even proposes noticeable objects which are visible even if their containers are closed.
- Avatars, objects, rooms and their doors should have concrete **topological and geometrical attributes** like size, color, coordinates,... Constraints induced by such information should be managed by the world. So the definition of "moving" will change. In the actual version avatars and objects move only when they change container.

¹⁴ of objects/services provided.

¹⁵ not just dynamic class loading but physical downloading of class files.

¹⁶ only under the condition that the HTTP server initially referenced by the object's codebase is up and running. This is not necessary in MaDViWorld.

7.2.2 Implementation Aspects

On the implementation side there are still many improvements to be made. Some of them are the direct consequence with respect to the additional specifications we have just mentioned. Let us enumerate in a non-exhaustive list what these challenges may be:

- MaDViWorld uses the *java security model* and its security policy files. In the future, we will have to provide specific policy files adapted for room servers, avatars, and so on.
- Some implementation details should be optimized. For instance, we use the serialization classes of `java.io` to store the states of the rooms, of the avatars and of the room factory into files. It would be conceivable to use the *compression classes* of `java.util.zip` to keep the size of these files as small as possible. Furthermore, the MaDViWorld packages as well as the object packages could be stored into *JAR files* to improve class downloading.
- The `getInformation()` method of the `WContainer` interface should not just return a short description string. An *XML* file containing all the *topological, geometrical* and other relevant *information* would perhaps be the best alternative. The client applications should then show other avatars, objects and the rooms in a more *sophisticated graphical model* (2D or 3D) and the rooms should provide an adapted events broadcasting mechanism.
- Using Java ORB over RMI could give the potential to *access objects written in other languages* (e.g. C++) and to exploit CORBA services.

References

The books, articles and websites mentioned in this bibliography are in alphabetical order according to their first author and then in numerical order in accordance with their publication date. They are divided into the following two groups: (1) these, which are explicitly referred to in the text of the article. They are recognizable by their asterisk [*]; (2) those, which concern one of the topics discussed in this article but have not been directly referred to.

[Agllets, 2001]*

IBM, *IBM Aglets Software Development Kit*, [on line], <http://www.tr1.ibm.com/aglets/>, (accessed on 12th July 2001).

[Chung et al., 1998]

R. Chung, W. Ng, H. Siu, K. Young, *Introduction to DIMUD Virtual World*, <http://www.cs.cornell.edu/home/raph/c5514/index.html>, 1998.

[Cicognani, 1998]

A. Cicognani, *A linguistic characterisation of design in text-based virtual worlds*, PhD thesis, Department of Architectural and Design Science, Faculty of Architecture, University of Sydney, 1998.

[Diehl, 1998]

S. Diehl, "Towards Lean and Open Multi-User Technologies", in *Proceedings of the International Symposium on Internet Technology ISIT'98*, Taipei, Taiwan, 1998.

[Diehl, 2001]*

S. Diehl, *Distributed Virtual Worlds: Foundations and implementation techniques using VRML, Java and CORBA*, New York, Springer, 2001.

[Eames, 1999]*

R. Eames, *The Future of the Virtual Worlds Platform*, <http://vworlds.research.microsoft.com/Docs/Future/VWFuture.htm>, 1999.

[Eames, 2000]

R. Eames, *Architecture Implementation Diagrams*, <http://www.vworlds.org/Docs/ArchImpl/VWFlow.htm>, 2000.

[Edwards, 2001]*

W. K. Edwards, *Core Jini*, 2nd Edition, Prentice Hall PTR, 2001.

- [Fabre et al., 2000a]***
Y. Fabre, G. Pitel, L. Soubrevilla, E. Marchand, T. Géraud, A. Demaille, "A Framework to Dynamically Manage Distributed Virtual Environments", in *Virtual Worlds, Proceedings of the Second International Conference, VW 2000*, Paris, France, July 2000, p. 54-64.
- [Fabre et al., 2000b]**
Y. Fabre, G. Pitel, L. Soubrevilla, E. Marchand, T. Géraud, A. Demaille, *An Asynchronous Architecture to Manage Communication, display, and User Interaction in Distributed Virtual Environments*, submitted to the 6th Eurographics Workshop on virtual Environments, Amsterdam, The Netherlands, 2000.
- [Fielding et al., 1999]***
R. Fielding, J. Gettys, J.C. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee, *Hypertext Transfer Protocol – HTTP/1.1*, Network Working Group, Request for Comments: 2616, <ftp://ftp.isi.edu/in-notes/rfc2616.pdf>.
- [Flanagan et al., 1999]***
D. Flanagan, J. Farley, W. Crawford, K. Magnusson, *Java Enterprise in a Nutshell*, O'Reilly & Associates, 1999.
- [Flanagan, 2000]***
D. Flanagan, *Java Examples in a Nutshell*, 2nd edition, O'Reilly & Associates, 2000.
- [Fowler, 2000]**
M. Fowler, K. Scott, *UML distilled: a brief guide to the standard object modeling language*, 2nd Edition, Addison Wesley, 2001.
- [Frécon et al., 1998]***
E. Frécon, M. Stenius, *DIVE : A Scaleable Network Architecture for Distributed Virtual Environments*, Distributed Systems Engineering Journal (special issue on Distributed Virtual Environments), 5 (3) 1998, p. 91-100.
- [Freeman et al., 1999]**
E. Freeman, S. Hupfer, K. Arnoold, *JavaSpaces: Principles, Patterns and Practice*, Addison-Wesley, 1999.
- [Gamma et al., 1995]***
E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional Computing Series, 1995.
- [Greenhalgh and Benford, 1995]***
C. Greenhalgh, S. Benford, "MASSIVE: a Distributed Virtual Reality System Incorporating Spatial Trading", in *Proceedings of the 15th International Conference on Distributed Computing Systems*, IEEE Computer Society Press, Vancouver, Canada, 1995, p. 27-34.
- [Harold, 1999]**
E. R. Harold, *XML Bible*, IDG Books Worldwide, 1999.
- [Herbert et al., 1999]***
A.J. Herbert, R.J. Hayton, M. Bursell, "Mobile Java Objects" in *BT Technology Journal*, Vol. 17, No. 2, April 1999, *Networked distributed systems*, <http://www.bt.com/bttj/>.
- [JiniUI, 2000]***
Jini serviceui Project, [on line], <http://developer.jini.org/exchange/projects/serviceui/>, (accessed on 12th July 2001).
- [Kirsh, 1998]***
D. Kirsh, "Adaptive Rooms, Virtual Collaboration, and Cognitive Workflow" in N. Streitz et al., *Cooperative Buildings – Integration Information, Organization, and Architecture*, Lecture Notes in Computer Science, Springer, Heidelberg, 1998, p. 94-106.
- [Kulczycki, 1997]**
P. Kulczycki, *An Application Framework for the Distributed Simulation of Virtual Worlds by Spatial Decomposition*, Polytechnic University, Department of Software Engineering, Hagenberg, Austria, supervised by Research Institute for Symbolic Computation, Johannes Kepler University, Linz, Austria, 1997.
- [L'Heureux and Vandelac, 1999]**
A. L'Heureux, J.-N. Vandelac, *Overview of Distributed Virtual Worlds*, <http://jota.sm.luth.se/~antlhe-9/01.html>, 1999.

- [Li, 2000]***
S. Li, *Professional Jini*, Wrox Press Ltd., 2000.
- [McClain, 2001]***
J. McClain, *A Short Presentation on Codebase*, [on line],
<http://developer.jini.org/exchange/users/jmccain/codebase/codebase.htm>, (accessed on 12th July 2001).
- [Malak, 2001]***
M. Malak, "Simulating Multiple Inheritance", in *Journal of Object-Oriented Programming*, April 2001, Vol. 13, No. 12, <http://www.joopmag.com>.
- [Mitchell, 1995]**
D. Mitchell, *From MUDs To Virtual Worlds*,
<http://research.microsoft.com/scg/papers/3DVW.htm>, 1995.
- [MUD, 2001]***
The Mud Connector, [on line], <http://www.mudconnector.com>, (accessed on 12th July 2001).
- [Oaks, 2000]***
S. Oaks, H. Wong, *Jini in a Nutshell*, O'Reilly & Associates, 2000.
- [Parris et al., 1993]**
M. Parris, C. Mueller, J. Prins, A. Duggan, Q. Zhou, E. Erikson, "A Distributed Implementation of an N-body Virtual World Simulation" in *Proceedings of IEEE Workshop on Parallel and Distributed Real-Time Systems*, IEEE, 1993.
- [Pavel, 1992]***
C. Pavel, "Mudding: Social Phenomena in Text-Based Virtual Realities", in *Proceedings of the 1992 Conference on Directions and Implications of Advanced Computing*, Berkeley, May 1992.
- [Reid, 1994]***
E. Reid, *Cultural Formations in Text-Based Virtual Realities*, Masters Thesis, English Department, University of Melbourne, 1994.
- [Sommers, 2001]**
F. Sommers, Object mobility in the Jini environment, JavaWorld, January 2001,
<http://www.javaworld.com/javaworld/jw-01-2001/jw-0105-jiniology.html>.
- [SUN, 2001]***
Sun Microsystems Inc., Java 2 Runtime Environment, Standard Edition, [on line],
<http://java.sun.com/j2se/1.3/jre/>, (accessed on 12th July 2001).
- [Thorpe, 1987]**
J. Thorpe, "The New Technology of Large Scale Simulator Networking: Implications for Mastering the Art of Warfighting", in *Proceedings of the Ninth Interservice Industry Training Systems Conference*, 1987.
- [Vellon et al., 1998]***
M. Vellon, K. Marple, D. Mitchell, S. Drucker, "The Architecture of a Distributed Virtual Worlds System", in *Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, Santa Fe, New Mexico, April 27-30, 1998.
- [Venners, 1999]***
B. Venners, How to Attach a User Interface to a Jini Service, JavaWorld, 1999,
<http://www.javaworld.com/javaworld/jw-10-1999/jw-10-jiniology.html>.
- [Voyager, 2001]***
ObjectSpace, *ObjectsSpace: Products – Voyager 4.0*, [on line],
<http://www.objectspace.com/products/voyager/>, (accessed on 12th July 2001).
- [VWorlds, 2001]***
Microsoft Corporation, *Virtual Worlds Group*, [on line], <http://www.vworlds.org>, (accessed on 12th July 2001).
- [Waldo, 2001]**
J. Waldo, Mobile Code, Distribute Computing, and Agents, IEEE Intelligent Systems, 2001,
<http://www.computer.org/intelligent/ex2001/pdf/x2010.pdf>.

[WebStart, 2001]

Sun Microsystems, *Java™ Web Start*, [on line],
<http://java.sun.com/products/javawebstart/>, (accessed on 12th July 2001).

[Zyda et al., 1992]

M. Zyda, J. D. R. Pratt, J.G. Monahan, K.P. Wilson, "NPSNET: Constructing a 3D Virtual World",
in *Proceedings of 1992 Symposium on Interactive 3D Graphics*, Computer Graphics, 1992.

The Official Website

The official website of the MaDViWorld project is:

<http://www-iiuf.unifr.ch/sde/projects/madviworld/>

This site provides the following information:

- the whole source code of the latest MaDViWorld version, which can be freely downloaded;
- an installation guide;
- the up-to-date javadoc documentation of MaDViWorld;
- actual publications related to MaDViWorld in electronic format.

Appendices

A. The State Files

Section 6 explained the persistence mechanism of the avatars, rooms and room factory. The state of these objects are stored in local files, with the extension `.ast`, `.rst` and `.fst`, respectively. The files are stored on the local hard disk and their contents are described below.

A.1 Avatar State File

Let us consider an Avatar State File (`.ast`) for an avatar named `myAvatar`:

- Default location of the file: `astorage` subfolder of the MaDViWorld avatar application folder.
- Name of the file: `myAvatarState.ast`.
- Contents of the file: a `SleepingAvatar` composed of:
 1. A `String` with the name of the avatar.
 2. A `String` with the name of the room in which the avatar currently is.
 3. A `String` with the type of the avatar.
 4. A `String` with the avatar's "history" (contents of the message field of Figure 4.1).
 5. A `Hashtable` containing all the objects it has in its bag.

A.2 Room State File

Let us consider a Room State File (`.rst`) for a room named `myRoom`:

- Default location of the file: `rstorage` subfolder of the MaDViWorld server part folder.
- Name of the file: `myRoomState.rst`.
- Contents of the file:
 1. The `Hashtable` with the objects it contains.
 2. The `Vector` with the name of the other rooms this room is linked to by a door.
 3. The `Vector` with the name of the avatars it contains.

A.3 Factory State File

Let us consider a Factory State File (.fst) for a roomfactory:

- Default location of the file: `rstorage` subfolder of the MaDViWorld server part folder.
- Name of the file: `factoryState.fst`
- Contents of the file:
 1. The `String` with the host name.
 2. The `Vector` with the name of the rooms it manages.
 3. The `String` containing the backup path where it saves the room state files.
 4. The `String` containing the path of the download folder, where all the files needed at runtime are saved.

B. Compiling the Applications

This appendix explains step by step how to compile each of the three MaDViWorld applications. All the batch files mentioned in this section can be downloaded (refer to The Official Website):

- The root directory for the server application is `d:\MaDViWorlds` and has a subfolder `MaDViWorld` containing the `MaDViWorld` package and all the subpackages relative to the server application.
- The root directory for the avatar application is `d:\MaDViWorldc` and has a subfolder `MaDViWorld` containing the `MaDViWorld` package and all the subpackages relative to the avatar application.
- The root directory for the wizard application is `d:\MaDViWorldw` and has a subfolder `MaDViWorld` containing the `MaDViWorld` package and all the subpackages relative to the wizard application.

This structure also holds for Appendix C and you eventually have to adapt it to your corresponding configuration. We recall that JRE 1.3 must be installed in order to run the system. It can be downloaded from [SUN, 2001].

B.1 Compiling the Room Server

Step 1. Compile all the files. In your server application root directory type:

```
javac MaDViWorld/WRoomFactory/ActiveSetup.java
javac MaDViWorld/util/WServer/ClassFileServer.java
javac MaDViWorld/util/WFetcher/WClassFetcherImpl.java
```

Step 2. Create stubs and skeletons for remote objects by typing:

```
rmic -v1.217 MaDViWorld.util.WFetcher.WClassFetcherImpl
rmic -v1.2 MaDViWorld.WRoomFactory.RoomFactoryImpl
rmic -v1.2 MaDViWorld.WRoomFactory.RoomImpl
```

All this work can be automated by running the `cpws` batch file.

B.2 Compiling the Avatar

Step 1. Compile all the files. In your server application root directory type:

```
javac MaDViWorld/WAvatar/AvatarImpl.java
javac MaDViWorld/util/WServer/ClassFileServer.java
javac MaDViWorld/util/WFetcher/WClassFetcherImpl.java
```

Step 2. Create stubs and skeletons for remote objects by typing:

```
rmic -v1.2 MaDViWorld.WAvatar.AvatarImpl
rmic -v1.2 MaDViWorld.util.WFetcher.WClassFetcherImpl
```

All this work can be automated by running the `ccclient` batch file.

¹⁷ Create stubs for the 1.2 JRMP (Java Remote Message Protocol) stub protocol version only

B.3 Compiling the Wizard

Step 1. Compile all the files. In your server application root directory type:

```
javac MaDViWorld/util/WServer/ClassFileServer.java
javac -classpath %CLASSPATH%;MaDViWorld\WObjects\TicTacToe
MaDViWorld/WWizard/RoomWizard.java
```

Step 2. Compile all the custom objects you have developed:

```
javac MaDViWorld/WObjects/MyWObj1/MyWObj1GUIImpl.java
```

All this work can be automated by running the `cpw` batch file.

C. Running the Applications

This appendix explains step by step how to launch each of the three MaDViWorld applications. All the batch files mentioned in this section can be downloaded (refer to The Official Website). A short presentation about codebase can be found in [McClain, 2001] and for further explanations refer to [Flanagan et al., 1999].

C.1 Running the Room Server

The very first time you launch a server, do the following steps:

Step 1. Start the registry daemon by running the `rmiregistry` command as follows:

```
start rmiregistry (in a Windows system or rmiregistry & on a Unix system)
```

It creates and starts a remote object registry¹⁸ on the default port 1099. The registry daemon produces no output and is typically run in background. Note that in order for the RMI registry to recognize and pass along the codebase property you specify, it has to be started in such a way that it cannot find any of the remotely loaded classes on its `CLASSPATH`. Therefore, it is crucial to start your RMI registry with a `CLASSPATH` that does not include stub/skeleton classes, etc. Later you will run your RMI server with a `CLASSPATH` that includes all the required classes.

Step 2. Start the http server. The command line that launches the http server is shown in Figure A.1:

```
java MaDViWorld.util.WServer.ClassFileServer 8084 serverwcserver d:\MaDViWorlds
d:\MaDViWorlds\MaDViWorld\downloaded
```

Figure A.1 Command line launching the http server

Let us take a closer look at this command line:

- ① is the main class for the http server.
- ② is the first command line argument. It defines the port number the http server listens to.
- ③ is the second command line argument. It defines the type of the http server.
- ④ is the third command line argument. It precises the first path to the files served by the http server.
- ⑤ is the fourth command line argument. It precises the second path to the files served by the http server.

All this work can be automated by running the `sclassserver` batch file.

Step 3. Start the `sdaemon`. The command line to launch the `rmidaemon` is shown in Figure A.2:

¹⁸ A remote object registry is a bootstrap naming service that is used by RMI servers on the same host to bind remote object to names. Clients on local and remote hosts can then look up remote objects and make remote method invocations.

```

①
↓
rmid -J-Djava.security.policy=policy.all
-J-Djava.rmi.server.codebase=http://134.21.9.244:8084/
-C-Djava.rmi.server.codebase=http://134.21.9.244:8084/ -log d:\MaDViWorlds\log
④
↑
⑤
↑

```

Figure A.2 Command line launching the `rmidaemon`

Let us take a closer look at this command line:

- ① is the `rmid` tool. It starts the activation system daemon.
- ② is a command line option passed to the java interpreter running `rmid`. It specifies a java security policy file allowing to resolve classes remotely.
- ③ is a `c` ommand line option passed to the java interpreter running `rmid`. It sets the codebase property for the activation daemon.
- ④ is a `c` ommand line option specifying the codebase property to pass to each JVM spawned by the activation system daemon.
- ⑤ is a `c` ommand line option specifying the name of the directory the activation system daemon uses to write its database and associated information.

All this work can be automated by running the `sdaemon` batch file.

Step 4. Start the server active setup application with the command line shown in Figure A.3 (from the root directory of the server application).

```

②
↓
java -classpath %CLASSPATH%;d:\MaDViWorlds\MaDViWorld\downloaded
-Djava.security.policy=policy.all -Djava.rmi.server.codebase=http://134.21.9.244:8084/
MaDViWorld.WRoomFactory.ActiveSetup /MaDViWorlds/MaDViWorld/rstorage/
d:\MaDViWorlds;d:\MaDViWorlds\MaDViWorld\downloaded
④
↑
⑥
↑
①
↓
⑤
↑
③
↓
⑦
↑

```

Figure A.3 Command line setting up the server

Let us take at closer look to this command line:

- ① is a command line option allowing to include the downloaded files in the `CLASSPATH`.
- ② is a command line option¹⁹ specifying a java security policy file allowing to resolve classes remotely.
- ③ is a command line option, which sets the codebase property for the class bytecodes.
- ④ is the main class for the server setup application.
- ⑤ is the first command line argument. It specifies the path where `.fst` and `.rst` files are stored.
- ⑥ is the second command line argument. It specifies the classpath for the JVM spawned by the activation system daemon.
- ⑦ is the third command line argument. It specifies the path were the downloaded files (i.e. those fetched by the `WClassFetcher`) are stored.

All this work can be automated by running the `sserver` batch file.

To restart the server after a shutdown or a failure, apply the following simple procedure:

- If the http server crashed, simple restart it by repeating **Step 2** above.
- After a software failure, i.e. after a crash of the `rmidaemon`, simply repeat **Step 3** above.
- After a hardware failure (machine reboot), implying that the `rmiregistry` also crashed, just repeat all **four steps** above.

C.2 Running the Avatar

Step 1. Start the `rmiregistry` as follows:

¹⁹ `-Dproperty=value` Sets a system property value.

```
start rmiregistry
```

Step 2. Start the client application. To do this, use the command line shown in Figure A.4 (from the root directory for the server application).

```
java -classpath %CLASSPATH%;d:\MaDViWorldc\MaDViWorld\downloaded -Djava.security.policy=policy.all -Djava.rmi.server.codebase=http://134.21.9.244:8085/MaDViWorld.WAvatar.AvatarImpl /MaDViWorldc/MaDViWorld/astorage d:\MaDViWorldc\MaDViWorld\downloaded d:\MaDViWorldc d:\MaDViWorldc\MaDViWorld\downloaded
```

Figure A.4 Command line launching the avatar application

Let us take a closer look at this command line:

- ① is a command line option allowing to include the downloaded files in the `CLASSPATH`.
- ② is a command line option that specifies a java security policy file allowing to resolve classes remotely.
- ③ is a command line option, which sets the codebase property for the class bytecodes.
- ④ is the main class for the avatar application.
- ⑤ is the first command line argument. It specifies the path where `.ast` files are stored.
- ⑥ is the second command line argument. It specifies the path where the downloaded files (i.e. those fetched by the `WClassFetcher`) are stored.
- ⑦ is the third command line argument. It precises the first path to the files served by the http server.
- ⑧ is the fourth command line argument. It precises the second path to the files served by the http server.

All this work can be automated by running the `sclient` batch file.

C.3 Running the Wizard

Step 1. Start the wizard. To do this, use the command line shown in Figure A.5 (from the root directory for the server application).

```
java -Djava.security.policy=policy.all -Djava.rmi.server.codebase=http://134.21.9.244:8083/MaDViWorld.WWizard.RoomWizard d:\MaDViWorldw
```

Figure A.5 Command line launching the wizard application

Let us take a closer look at this command line:

- ① is a command line option specifying a java security policy file allowing to resolve classes remotely.
- ② is a command line option, which sets the codebase property for the class bytecodes.
- ③ is the main class for the wizard.
- ④ is the first command line argument. It precises the path to the files served by the http server.

All this work can be automated by running the `swizard` batch file.

D. Utility Applications

Both our custom HTTP server and our class fetcher produce output to the console window giving information of the tasks they are fulfilling. As an example, let us consider the transfer of a class file named `MyFile.class` of package `MaDViWorld.MyPackage`:

D.1 The HTTP Server

For each requested file it has to serve it displays the following information.

- If the request was a standard GET request from a remote java class loader:

```
reading: MaDViWorld.MyPackage.MyFile (requested by iiufpc31)
```

- If the request was a GE2 request from a remote ClassFetcher.

```
reading: MaDViWorld.MyPackage.MyFile (fetched by iiufpc31)
```

D.2 The Class Fetcher

For each file it is getting, the following information is displayed:

```
getting: MaDViWorld/MyPackage/MyFile (obtained from iiufpc32)
```