Department of Informatics
University of Fribourg (Switzerland)

Distributed Virtual Worlds

Abstract Model and Design of the MaDViWorld Software Framework

Accepted by the Faculty of Science of the University of Fribourg (Switzerland) on the recommendation of:

Prof. Dr. Heinz Gröflin, University of Fribourg (jury president)
Prof. Dr. Jacques Pasquier-Rocha, University of Fribourg (Ph.D. Supervisor)
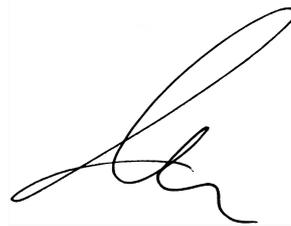Prof. Dr. Jacques Savoy, University of Neuchâtel (Second reporter)

Fribourg, September 10th, 2004

Ph.D. Supervisor: Faculty Dean:

Prof. Dr. Jacques Pasquier-Rocha   Prof. Dr. Marco Celio

To my love, Carole and to my son, Théo.

Acknowledgements

There are many people who helped me finishing this thesis. The following list is therefore by no means exhaustive.

First of all, I am deeply indebted to my supervisor, Prof. Jacques Pasquier-Rocha who gave me the opportunity to write this thesis and whose continuous support, stimulating suggestions, enthusiasm and encouragements helped me in all the time of research.

I would also like to thank Prof. Jacques Savoy of the University of Neuchâtel who kindly accepted being referee of my dissertation and provided the feedback I needed.

Debts are owed to my friend Jessen Page and to my brother Tobias who patiently read the draft manuscript improving the language of my thesis and correcting many spelling mistakes.

On a more personal level, there is of course my love Carole, who supported me at all times and was always at my side. Thanks for dealing with me being absent even when I was home. As much time as put into this thesis, I never stopped being distracted by thinking of you.

One of the best experiences that we lived through in this period was the birth of our first son Théo, who provided an additional and joyful dimension to our life.

Lastly, and most importantly, I whish to thank my parents for helping me start off with a good education, from which all else springs.

Abstract

Distributed virtual worlds are multi-user applications running on several computers connected by a network. They go beyond the traditional document based World Wide Web since multiple users interact within a shared space and are aware of each other. Distance learning, telemedicine, adventure games, virtual shopping malls, virtual conferencing, virtual museums, virtual e-banking are just some examples in the wide spectrum of applications for distributed virtual worlds.

The first contribution of this thesis to the field of virtual worlds is the elaboration of an original and general abstract model defining the different components of a virtual world. This formal approach is intended to provide a common basis for many possible software implementations.

The second major contribution is the creation of MaDViWorld, a concrete highly distributed implementation of one of the possible model instantiations. MaDViWorld is an extensible Java and Jini-based software framework supporting distributed virtual worlds. Its main originality is a highly modular structure integrating a lot of carefully documented software engineering concepts. MaDViWorld allows for creating the rooms of a given world on several machines, each running a 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. This approach opens promising perspectives for future applications.

The MaDViWorld project further served as a test-bed for many student projects thus presenting a real academic and pedagogical interest. Furthermore, involving external people as programmers using and testing the framework was very useful for its improvement and validation.

Résumé

Les mondes virtuels distribués sont des applications multi-utilisateurs fonctionnant sur plusieurs machines reliées par un réseau. Ils vont au-delà du "World Wide Web" traditionnel basé sur les documents car plusieurs utilisateurs, évoluant au sein d'un espace commun et conscients de leur présence mutuelle, peuvent y interagir. L'enseignement à distance, la télemédecine, les jeux d'aventures, les téléconférences, ainsi que les supermarchés, musées ou e-banking virtuels ne sont que quelques exemples du large éventail d'applications possibles de ces mondes.

La première contribution de cette thèse dans le domaine des mondes virtuels est la conception d'un modèle abstrait original et géneral définissant les différentes composantes d'un monde virtuel. Cette approche formelle vise à fournir une base commune à une multitude d'implémentations logicielles possibles.

La deuxième contribution principale est la création de MaDViWorld, une implèmentation concrète hautement distribuée d'une des instantiations possibles du modèle abstrait. MaDViWorld est un cadre logiciel réutilisable pour le développement de mondes virtuels distribués. Ce cadre logiciel est programmé en Java et repose sur la technologie Jini. Sa principale originalité réside dans sa structure hautement modulaire intégrant une multitude de concepts du génie logiciel documentés avec minutie. MaDViWorld permet la création de pièces d'un monde virtuel sur plusieurs machines, chacune hébergeant une application serveur. Il est alors possible de connecter les pièces entre elles par de simples portes et de les peupler avec des objets actifs. Finalement, des avatars gérés par des applications clientes visitent les pièces et interagissent avec les objets actifs. Cette approche ouvre de prometteuses perspectives pour des applications futures.

De plus, le projet MaDViWorld a servi de base pour plusieurs projets d'étudiants présentant ainsi un réel intérêt académique et pédagogique. En outre, le fait d'inclure des personnes externes en tant que programmeurs utilisant et testant le cadre logiciel a été très utile pour l'améliorer et a fortement contribué à sa validation.

Table of Contents

List of Figures

List of Tables

1
Introduction

The greatest challenge to any thinker is stating the problem in a way that will allow a solution.
—Bertrand Russell

1.1 Motivation

The present thesis has been undertaken in the Software Engineering Group of the Department of Informatics of the University of Fribourg in Switzerland. This research group has been interested since the late 80's in the development of well designed object oriented applications. At the beginning, the Object Pascal language was used. Later, the eclosion of Macintosh like graphical user interfaces and the application framework MacApp [170] boosted the development of convivial and powerful products. This is the case of Intermedia, developed at the IRIS Institute at the University of Brown [211], of NoteCards developed at Xerox Parc [87] or of the Apple's famous Hypercard [80]. Hypertext and electronic books became a hot topic, and based on some early experiments documented in [151], this application domain served as context for the software engineering efforts of the group [150]. These resulted in the development of an electronic book framework called WEBS[1] (Woven Electronic Book System), which was later enhanced with an object-oriented scripting environment and was renamed into WEBSs [138, 139]. The project

---

[1]The ancestor of WEBS within the University of Fribourg was EBOOK3 [168, 169]. This prototype, however, was built on a totally non object-oriented software architecture.

focused on the integration of electronic textbooks in the education process and carried out some pedagogical experiments with the developed prototypes [36]. Since 1995 proprietary products, such as those developed at the University of Fribourg, were outplayed by the generalization and standardization of hypertext offered by the World Wide Web [21] and HTML. We have now been accustomed to seeing the Cyberspace as a great sea of World Wide Web documents. This approach corresponds to the *document paradigm*.

More fantastic views of Cyberspace portray it as a labyrinth of interconnected virtual worlds inhabited in real time by millions of people represented as 'avatars' and being aware of each other's presence and actions. This vision corresponds to the *virtual world paradigm* and moves the Cyberspace beyond a simple pile of linked web documents, a dead library, to a social and communicative space: a web of human relationships–a community. With this evolution, users transformed themselves from "surfers to settlers" [25]. The use of cyberspace to overcome the expense of travel and overcome the physical limitations of reality has sparked the imagination of many visionaries and networked virtual environments have been the product of science fiction [179, 78] for many years. The Internet represents the ideal medium for synthesizing huge virtual worlds accessible from a large community of users. The technological innovations which accompanied this new trend–beside the fact that network plays now a central role–were the appearance of the Java™ object-oriented programming language, the development of generic environments and the use of extensible and flexible application frameworks. Therefore, based on the solid experience acquired with electronic books, the Software Engineering research group investigations focus since 1998 [65, 66] on the development of a flexible and extensible distributed software application framework supporting virtual worlds. Figure 1.1 summarizes the evolution of the researches lead at Fribourg.
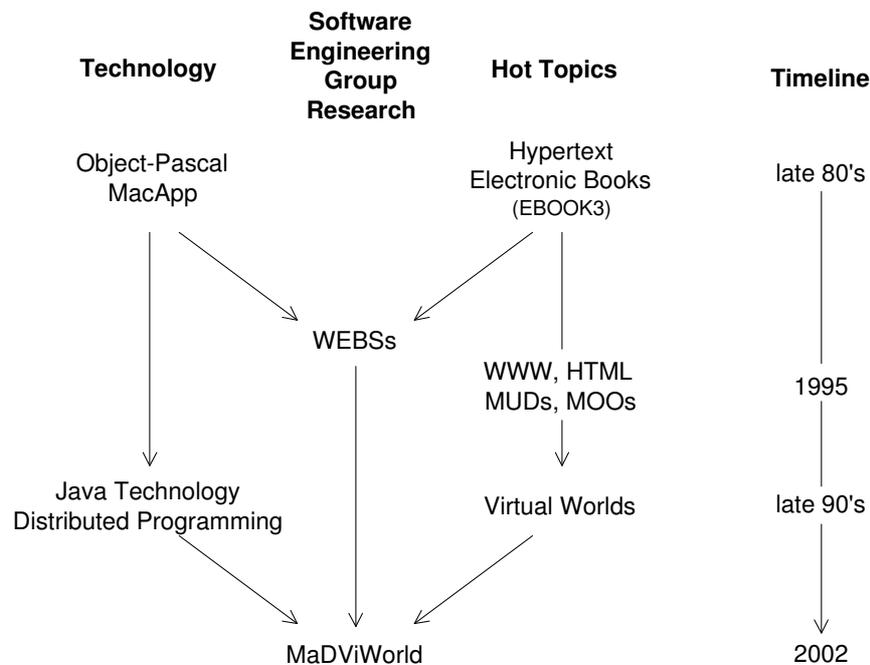
Figure 1.1: Historical background

1.2 Document versus Virtual World Paradigm

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 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 to interact with them. Typically, each user copies the documents onto her local machine and her interactions with them have no direct repercussions on the other connected users. In particular, a user never directly 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 truly 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, since the connection of new decentralized servers is always possible, there is no limit to its growth.

- 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 (see Section 2.3).

At the software architecture level, systems based on the virtual world metaphor are clearly the most complex ones. Indeed, the users interact directly 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. To face these issues and support scalability, virtual world developers have to choose an appropriate software architecture. Classical architectures are often restricted to a single central server containing the whole virtual world and guaranteeing its consistency with many clients connected to it.

1.3 Basic Virtual Worlds Concepts

For a good comprehension of this dissertation, the following four terms need to be briefly explained:

1. *Avatars* are the virtual representation of the users. Concretely, an avatar is a tool that allows a given user to move through the world, to interact with its inhabitants

and objects and that lets the other users know where she is and what she is doing. The word Avatar comes from the Sanskrit: 'Earthly incarnation of a Hindu god or goddess'. The reference is in particular to a God called Visnu that is able to reincarnate himself into several and different faces. Among people working on virtual reality and cyberspace interfaces, the word Avatar is used to describe the "object" (icon, two or three-dimensional photo, design, picture, animation or representation) representing the user in a shared virtual reality. In other words, an avatar is an instantiation of the user's body in the computerized medium. In text-based virtual realities, such as MUDs, avatars consist of a short description which is displayed to the users whose avatars 'look' them.

2. In order to distinguish between near and distant elements it is essential to divide the world into subspaces where the users might or might not enter and in which all interactions take place. We call such subspaces *rooms*.

3. Rooms are connected by *doors*, which an avatar can use for moving from one room to another.

4. *Objects* populate the rooms. In [108], the author distinguishes between three categories:

   a) passive objects, which can only change if a human agent (through an avatar application) interacts with them. These objects do not react to changes in other objects. A simple whiteboard on which each user can write short messages is an example of such an object;

   b) 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;

   c) 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.

Furthermore, in a distributed world, it should be possible to "physically" transport a given object from one room on a given server to another room running on a different machine.

The conceptual model, that emerges from these considerations, is shown in Figure 1.2. It represents a very simple world with four rooms, three avatars (James, Sylvia and Hans) and a single game object (TicTacToe). One can also see how the rooms are interconnected by three doors.

The terminology and the conceptual model will be defined and extensively discussed in Section 4.2.

1.4 Contributions

Virtual Worlds represent an active research field, with many related topics. Various projects are concerned with gaming and simulation aspects, which are not considered in the present thesis. Indeed, the research described here is much closer to the networked
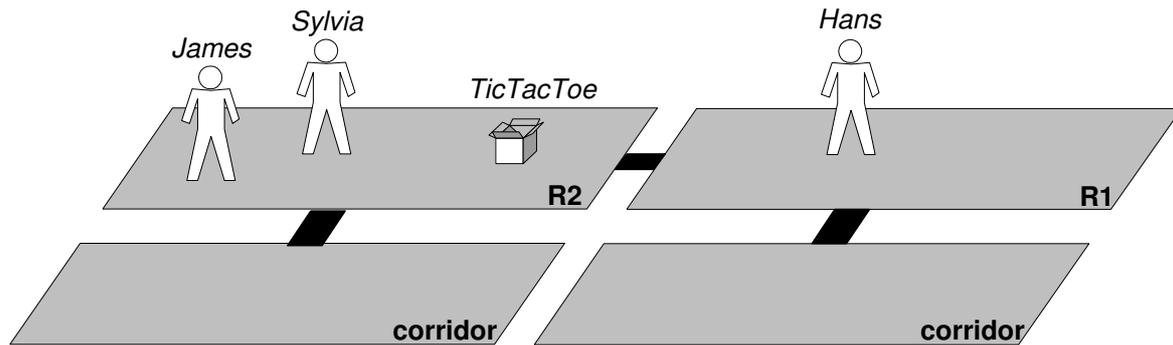
Figure 1.2: Conceptual model of a simple world

virtual environments and to Bruce Damer's Virtual Community [47] concept. This means that neither the simulation of living systems or the study of evolution of artificial like systems, nor the issues of user immersion in computer generated 3D spaces are addressed in this dissertation.

The goal of this thesis is to define extensible and flexible software solutions in order to support the virtual world paradigm, without making concessions to the single central server architecture. The two main features that should be supported by the proposed software architecture are:

- massive distribution: elimination of any central backend, which would be a communication and processing bottleneck. The full distribution of the system should allow optimal scalability.

- user customization of the world: the user has to be able to easily extend the virtual world by creating new parts of it or by populating it with new objects while she is visiting it.

Actually, MaDViWorld, the acronym of the implemented software framework, stands for Massively Distributed Virtual Worlds, since its subspaces are distributed on an arbitrarily large amount of machines. As in the case of the world wide web, no single host knows the current state of the whole world. 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 network. Such an architecture combines the great advantages (scalability, robustness,...) found in the document paradigm with those of the shared virtual world metaphor.

As already mentioned, issues related to social interactions and graphic aspects are not the first priorities, and are thus only supported to provide the minimal required feeling of immersion to the user.

The expected contributions of this thesis are:

- A purely theoretical definition and description of the concept of virtual world paradigm.

- The adaption of a new and original bottom-up composition of the virtual world, which contrasts with the top-down decomposition adopted by other analog projects.

- The validation of the abstract model, by the implementation of a working framework, named MaDViWorld. This framework achieves both the world customization and the distributed deployment goal.

- The creation of a small but active and open community of users and developers around the MaDViWorld project.

1.5 Organization

This thesis is divided into eight chapters. After this introductory chapter, Chapter 2 summarizes the beginning of the Internet and the rising of the first virtual environments. Then it presents some more recent projects tackling the challenge of developing distributed virtual worlds. It also gives an overview of some important concepts related to framework development and network computing, thus providing the global background of the present work. Chapter 3 identifies seven major problems that virtual world implementations have to solve.

The fourth chapter introduces an original total abstract model for virtual worlds, containing formal definitions and some basic properties. It further explains how to derive a concrete and programmable model from the general view. Chapter 5 shows how to smoothly pass from the conceptual model to a computer-based solution. To reach this goal, some preliminary considerations are followed by the presentation of the basic object-oriented software architecture, which breaks down into several abstraction layers. The first layer, based on the theoretical contributions of Chapter 4, is devoted to an abstract specification of the different participants of the distributed system. The lower layers provide a concrete implementation of the top layer and some helper classes. An user-oriented point of view completes the general presentation of the software framework, which is named MaDViWorld. Chapter 6 describes the implementation layers from a technical perspective, explaining the principal design choices and describing the core classes of the framework with UML class and sequence diagrams. Chapter 7 is advocated to a brief presentation of some existing virtual world objects and to a comparison with the world of software agents. Finally, Chapter 8 summarizes this research and proposes possible future work.

1.6 Notations and Conventions

The following conventions are adopted in this dissertation:

- Formatting conventions:

  - **Bold** and *italic* are used for emphasis and to signify the first use of a term.
  - SansSerif is used for web addresses.
  - Code is used in all Java code and generally for anything that would be typed literally when programming, including keywords, constants, method names, and variables, class names, and interface names.

- The present dissertation is divided in Chapters. Chapters are broken down into Sections. Where necessary, sections are further broken down into Subsections, and Subsections may contain some Paragraphs.

- Figures are numbered inside a chapter. For example, a reference to Figure j of Chapter i will be noted *Figure i.j*.

- UML (specifically class and sequence diagrams) is extensively used to illustrate software concepts–to document the patterns and describe the static and dynamic interactions.

- Software examples are illustrated with screenshots of the prototype distributed virtual world programmed in the scope of this dissertation. The size of screenshots has been adapted to the page layout of this document.

- A little icon in the margin emphasizes a key idea that will be used further in the dissertation.

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

2

Background

It would appear that we have reached the limits of what it is possible to achieve with computer technology, although one should be careful with such statements, as they tend to sound silly in five years.
—John von Neumann

The background of this thesis is twofold: one is domain specific, i.e. virtual communities and virtual worlds, and one is rather technical, i.e. object-oriented programming, object-oriented frameworks, distributed applications. The intersection between these two domains is the scope of this thesis. More precisely, this dissertation comments the design of an extensible, distributed object-oriented software architecture, namely a framework, and documents it, showing how to extend and use it in order to support innovative virtual worlds.

The first three sections of this chapter focus on the domain specific related work and follow the historical evolution over the last four decades: from the very beginnings of the Internet to the recently developed virtual world platforms. The goal of the last section is to present some important software engineering concepts that were used in the present work.

2.1 Internet

MUDs and MOOs were the first virtual environments. Their history, presented in the next section, is closely related to the history of Internet [213]. Thus, we begin this chapter by recalling the main steps the Internet technology went through since its beginnings.

2.1.1 History of Internet

- 1958. In response to the first artificial earth satellite, Sputnik, launched by USSR in 1957, the United States forms the Advanced Research Projects Agency (ARPA) with the Department of Defense (DoD) to establish US lead in science and technology applicable to the military.

- October, 1963. Thomas Marill, Daniel Edwards, and Wallace Feurzeig publish "DATA-DIAL: Two-Way Communication with Computers from Ordinary Dial Telephones" [130]. The host computer is a DEC PDP-1. This is the first application that allows users to communicate with computers remotely without using special expensive equipment. Bolt, Beranek and Newmann Inc. (BBN) also registers the patent on the modem on June 17, 1963, and subsequently develops the foundations of modern computer networks.

- 1969. ARPANET is created commissioned by the DoD for research into networking. AT&T provides 50kbps lines. As BBN builds IMP (Information Message Processors), nodes are set up.[1] The Internet is born and the first four nodes composing ARPANET are (see Figure 2.1):

    - the University of California Los Angeles (UCLA);

    - the Stanford Research Institute (SRI);

    - the University of California Santa Barbara (UCSB);

    - the University of Utah.

---

[1]The IMPs were Honeywell DDP 516 mini computers with 12k of memory.
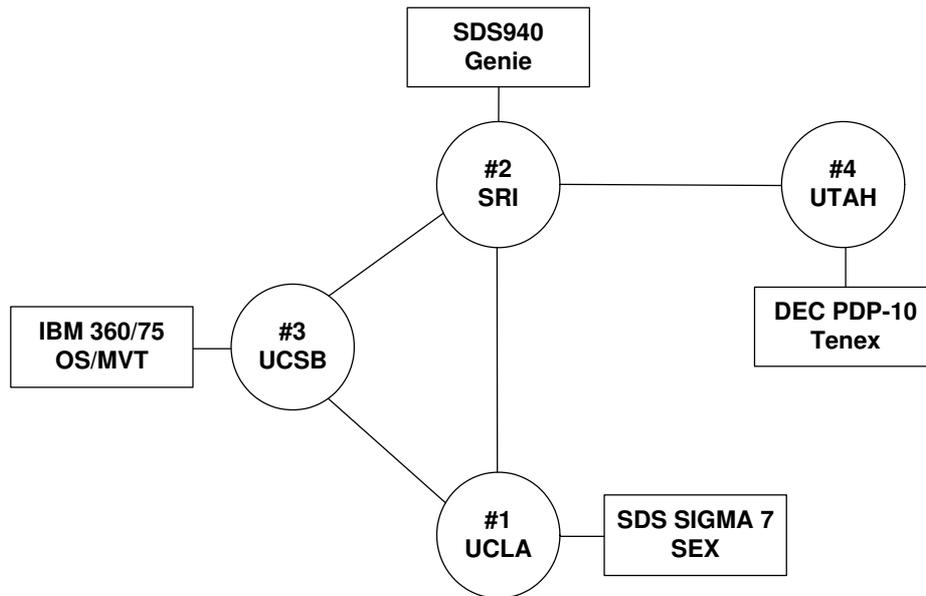
Figure 2.1: 4-Node ARPANET topology (December 1969)

- 1969. The Request For Comments (RFC) document series begins. It is a set of technical and organizational notes about the ARPANET (later the Internet). Memos in the RFC series discuss many aspects of computer networking, including protocols, procedures, programs, and concepts, as well as meeting notes, opinions, and sometimes humor. The RFC Editor [158] is the publisher of the RFCs and is responsible for the final editorial review of the documents.

- 1971. The network grows up to 15 nodes (23 hosts): UCLA, SRI, UCSB, University of Utah, BBN, MIT, RAND, SDC, Harvard, Lincoln Lab, Stanford, UIU(C), CWRU, CMU, NASA/Ames.

- 1971. Ray Tomlinson of BBN invents email programs to send messages across a distributed network. The original program was derived from two others: an intra-machine email program (SENDMSG) and an experimental file transfer program (CPYNET).

- 1972. The Advanced Research Projects Agency (ARPA) is renamed to Defense Advanced Research Projects Agency (or DARPA).

- 1972. The telnet specification (RFC 318) is published.

- 1972. In October the International Network Working Group (INWG) is created to establish technical standards to enable any computer to connect to the ARPANET. The Chairman of the INWG is Vinton Cerf.

- 1973. Development begins on the protocol later to be called TCP/IP. It is developed by a group headed by Vinton Cerf from Stanford and Robert Kahn from DARPA. This new protocol allows diverse computer networks to interconnect and communicate with each other.

- 1973. First international connections to the ARPANET: University College of London (England) via NORSAR (Norway).

- 1973. Robert M. Metcalfe's Harvard PhD Thesis outlines the idea for Ethernet. The concept is tested on Xerox PARC's Alto computers, and the first Ethernet network is called the Alto Aloha System (May).

- 1973. The FTP (File Transfer Protocol) is published. It allows information to be sent from one computer to another in bulk.

- 1974. Vinton Cerf and Robert Kahn publish "A Protocol for Packet Network Internetworking" [33] which specifies, in detail, the design of a Transmission Control Protocol (TCP). They use the term Internet for the first time.

- 1976. Dr. Robert M. Metcalfe develops Ethernet, which allows coaxial cable to move data extremely fast. This was a crucial component to the development of Local Area Networks (LANs).

- 1976. The Department of Defense (DoD) begins to experiment with the TCP/IP protocol and soon decides to require it for use on ARPANET.

- 1978. The first MUD–an adventure game supporting multiple players–is developed by Roy Trubshaw and Richard Bartle at Essex University in England (see Subsection 2.2.1 for more details).

- 1982. The International Network Working Group (INWG) establishes the Transmission Control Protocol (TCP) and Internet Protocol (IP), as the protocol suite, commonly known as TCP/IP, for ARPANET. This leads to one of the first definitions of an "internet" as a connected set of networks, specifically those using TCP/IP, and "Internet" as connected TCP/IP internets. The DoD declares TCP/IP suite to be standard for DoD.

- 1984. The Domain Name System (DNS) is introduced. The DNS is a distributed Internet directory service. DNS is used mostly to translate between domain names and IP addresses and to control Internet email delivery.

- 1985. The Internet Protocol software is being placed on every type of computer and universities research groups also begin using in-house networks known as Local Area Networks or LAN's. These in-house networks start using Internet Protocol software so one LAN can connect with other LAN's.

- 1986. One LAN branches out to form a new competing network, called NSFnet (National Science Foundation Network). NSFnet first links together the five national supercomputer centers, then every major university, and soon starts to replace the slower ARPAnet (which is finally shutdown in 1990). NSFnet forms the backbone of what we call the Internet today.

- 1987. On October 22nd 1987, "SWITCH - Swiss Academic and Research Network" was established as a foundation by the Swiss Confederation and eight university cantons (Basle City, Berne, Fribourg, Geneva, Neuchâtel, St. Gall, Vaud and Zurich). The Berne-based foundation has as its objective "to create, promote and offer the necessary basis for the effective use of modern methods of telecomputing in teaching and research in Switzerland, to be involved in and to support such methods". It is a non-profit foundation that does not pursue commercial aims.

Date	Hosts	Date	Hosts
12/1969	4	07/1989	130'000
04/1971	23	10/1990	313'000
06/1974	62	01/1991	376'000
03/1977	111	10/1991	617'000
08/1981	213	04/1992	890'000
05/1982	235	10/1992	1'136'000
08/1983	562	04/1993	1'486'000
10/1984	1'024	10/1993	2'056'000
10/1985	1'961	07/1994	3'212'000
02/1986	2'308	01/1995	5'846'000
11/1986	5'089	07/1996	16'729'000
12/1987	28'174	01/1998	29'670'000
07/1988	33'000	07/1999	56'218'000
10/1988	56'000	01/2001	109'574'429
01/1989	80'000	07/2002	162'128'493

Table 2.1: Internet Growth (hosts = computer systems with registered IP address)

- 1990. 21 years after its creation, ARPANET ceases to exist.

- 1990. The following countries connect to NSFNET: Argentina (AR), Austria (AT), Belgium (BE), Brazil (BR), Chile (CL), Greece (GR), India (IN), Ireland (IE), Korea (KR), Spain (ES), Switzerland (CH).

- 1991. The World Wide Web (WWW) is released by CERN (European Organisation for Nuclear Research, in French Conseil Européen pour la Recherche Nucléaire) in Geneva, Switzerland. The web has been conceived and developed by Tim Berners-Lee with help from Robert Cailliau.

  Tim Berners-Lee defined HTTP (Hypertext Transfer Protocol), HTML (Hypertext Markup Language) and URL (Universal Resource Locator), which are the cornerstones of the WWW[2]. He is currently the Director of the World Wide Web Consortium[3], the group that sets technical standards for the Web.

  It is Vannevar Bush who first proposed the basics of hypertext in 1945, and who is therefore the inventor credited with the principles underlying modern hypertext research.

- 1994. No major changes were made to the physical network. The most significant thing that happened was the growth. Many new networks were added to the NSF backbone. Hundreds of thousands of new hosts were added to the INTERNET during this time period (see Table 2.1).

---

[2]On April 15th 2004, Tim Berners-Lee, the inventor of the World Wide Web, was named recipient of the first-ever Millennium Technology Prize from the Finnish Technology Award Foundation (see http://www.technologyawards.org/ (accessed December 28, 2004)). This honor is bestowed as an international acknowledgement of outstanding technological innovation that directly promotes people's quality of life.

[3]See http://www.w3.org/ (accessed December 28, 2004).

- 1995. New WWW technologies emerge: Java, JavaScript, ActiveX, VRML. The main idea is to allow for the creation of reactive web pages.

- 1996. Microsoft enters the Internet market and the Internet Explorer versus Netscape browser war begins.

- 1997. Sun Microsystems develops a promising standard for server-side components: Enterprise JavaBeans (EJB) and releases its first draft specification in December.

- 1998. XML is among the emerging technologies.

- 1999. Technologies of the year comprise e-commerce, e-auctions and portals.

- 2001. Web services and their send-an-object-get-an-object model enable computer-to-computer interactions on the Internet.

- 2003. The first official Swiss online election takes place in Anières (7 January).

2.1.2 Main Concepts Summarized

The Internet has developed tremendously since the creation of its ancestors ARPANET and later NSFnet. Today it counts hundreds of millions of computers connected all around the world and is still growing as the private connections are getting cheaper and the connection speeds are increasing. The main application are electronic mail, file transfer, remote access to other computers and browsing the huge amount of web pages offered on the WWW. All the computers are connected by a complex network of cables and wireless transmission means, which one could compare to roads, and information is transmitted according to the TCP/IP protocol, which one could compare to the circulation rules. The resulting underlying architecture is quite simple and is roughly sketched in Figure 2.2.

2.2 MUDs and MOOs

This section describes MUDs and MOOs and their history[4]. A huge amount of well structured information and resources on actual MUDs and MOOs can be found on the [143] website.

2.2.1 History of MUDs and MOOs

The early MUD games were partly influenced by the fantasy board game Dungeons and Dragons, or D&D for short, developed by Dave Arneson and Gary Gygax in the early 1970's. For example, the game developers Will Crowther, Dave Lebling, and Richard Bartle mentioned below were D&D aficionados. D&D games were very intricate, complex games where players could take on the aspects of various characters from a fantasy world, such as a warrior, wizard, prince, and so on, acquire and lose magical powers, and progress through fantastic adventures involving travel through wild and wonderful worlds. These

---

[4]Sources for these histories mainly include Reid's thesis [157], providing a well-researched history and analysis of MUDs, as well as the MUDdex site [29].
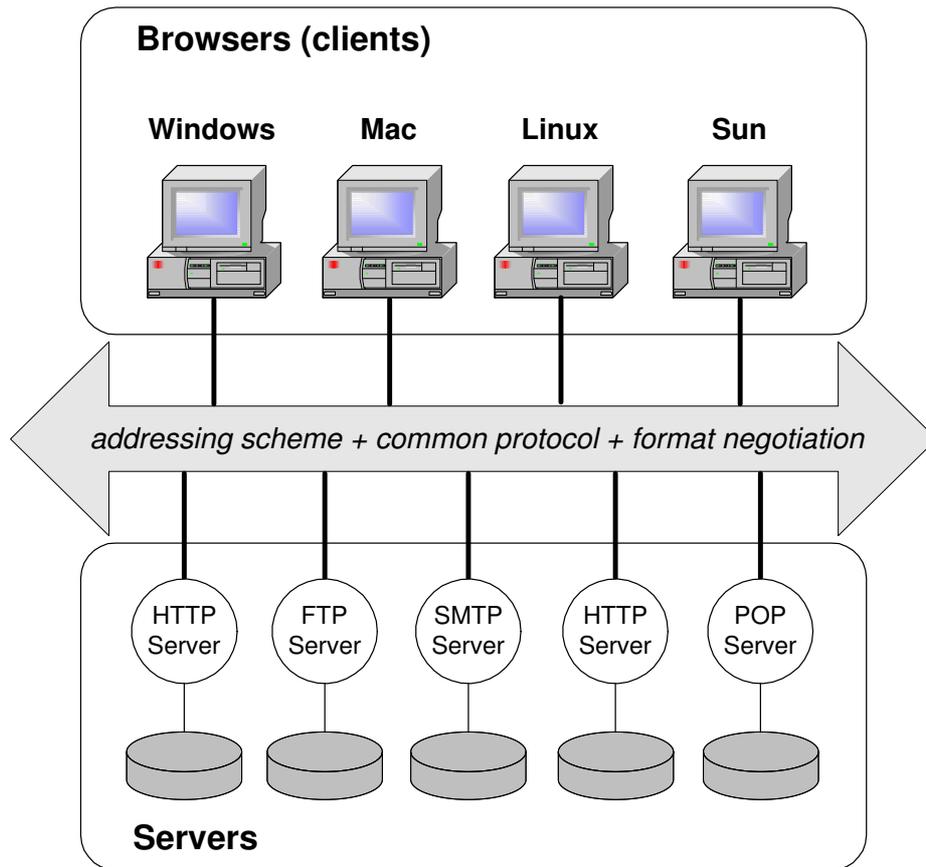
Figure 2.2: Global architecture of the Internet (inspired from a presentation of Robert Cailliau)

games were strongly inspired by J.R.R. Tolkien's *Lord of the Rings* (1937) and its complex fantasy world.

The major milestones in the development of MUDs are described below:

- The first widely used computer adventure game was created in 1973 by Will Crowther on a DEC PDP-10 computer, who coincidentally had earlier also worked on the ARPANET IMP. The program depicted an explorer seeking treasure in a network of caverns. It was an entirely text-based game. There were no graphics at all, just descriptions of localities and prompts asking players where they wished to go or what they wanted to do next.

- The game was then significantly extended in 1976 by Don Woods at Stanford University. It was called "Adventure", but was often referred to as "ADVENT" since the length of a file name on the TOPS-10 operating system was limited to six-letters. Containing many of the features of a D&D game, it added an interesting twist: the dungeon master, the "person" who sets-up and runs a D&D world, was played by the Adventure computer program itself. In "Adventure" the player assumed the role of a traveller in a Tolkienesque setting, fighting off enemies, overcoming obstacles through clever tricks, and eventually discovering treasures.

  Crowther and Woods are the inventors of the very first computerized virtual reality game. Crowther's caves, and Wood's more complex fantasy world, were figured by

players as places which they could enter through the computer. Adventure quickly became extremely popular and a host of similar games began to appear.

- Adventure and its cousins did not run on computer networks. They were single player games. However, at the same time as they were being written, most US universities were, as mentioned earlier, joining ARPANET. By the late 1970s most research institutions in the United States had joined ARPANET.

- In 1977 the interest of networking, interactivity, and virtual reality games met to produce the first networked, multi-user game. Mazewar, written by Jim Guyton, involved the extremely simple scenario of multiple participants wandering around a maze, trying to shoot one another. It was a kind of multi-participant Spacewar, which was the first computer game consisting of a spaceship appearing on the screen, to be shot by the player.

- Inspired by the game Adventure, Dave Lebling, Marc Blank, Tim Anderson, and Bruce Daniels, a group of students at M.I.T., wrote a game called Zork [118] in the summer of 1977 for the PDP-10 minicomputer which became quite popular on the ARPANET.

- A version called DUNGEN was later developed in the FORTRAN language by a programmer at DEC and ported to many different machines. In 1980, Blank and Joel Berez, with some help from Daniels, Lebling, and Scott Cutler, produced a version for the company Infocom that ran on the TRS-80 and Apple II microcomputers and was later ported to several other microcomputers. Although Zork did not borrow any code from Adventure, it built on the same concepts and added several more features. Like Adventure, Zork was a single player game.

- The first Multi-User Dungeon was usually just called MUD, and its first version was written and completed at the end of 1978 by Roy Trubshaw, a student at the University of Essex in England, originally in the MACRO-10 language for a PDP-10. MUD was the first adventure game to support multiple users. The name was chosen partly as a tribute to the DUNGEN variant of Zork, which Trubshaw had greatly enjoyed playing. Trubshaw converted MUD to BCPL (the now obsolete predecessor to the C language), and then handed over development to Richard Bartle, also a student at Essex University in England who joined him in working on MUD. It was a networked multi-user game which allowed users to communicate with one another, to cooperate on adventures together, or to fight against each other. The original version merely allowed a user (player) to move about in a virtual location. Later versions provided for more variation including objects and commands which could be modified on or offline. Although this original MUD game did not ever gain a high level of popularity, it nevertheless had great influence on those who were to develop later games. The number of people who played Bartle and Trubshaw's MUD was small, but many of them went on to design the systems that were popular later.

Trubshaw began his work with two purposes in mind: to make a multi-player computer adventure game and to write an interpreter for a database definition language. The goal for developing the first MUD was to test a newly developed shared memory system, the gaming aspect came later.

- The original MUD was available on the UK CompuNet network for two years until the DECsystem-10 computers were decommissioned. You can still play the original version (see Figure 2.3) at British-Legends.com[5], a web version converted by Viktor Toth from BCPL into C/C++ in a thirteen day marathon. Foreseeing the future popularity of the game, fortunately Bartle put the word "MUD" and the concept into the public domain. In his words: "MUD development had been funded by public money, therefore I felt the fruits of this should be returned to the public". Bartle and Trubshaw have continued to be involved in MUDs and gaming, and are currently Directors of MUSE (Multi-User Simulated Environment).

Figure 2.3: Screenshot of a connection to Bartle's original MUD

- Alan Cox[6] was one of those who spent a lot of time playing the original MUD game, and in 1987 he decided to design his own. AberMUD, named after the university where it was written, University of Wales at Aberystwyth, has evolved through numerous versions. In 1988, AberMUD spread on the Usenet and started being used in North America, after which its use spread rapidly among research and academic organizations.

- Jim Aspnes of Carnegie-Mellon University was another fan of Bartle and Trub-

---

[5]Or by directly telnetting to host.british-legends.com 27750

[6]Alan Cox is now working for Red Hat (http://www.redhat.com (accessed December 28, 2004)) and is one of the key players in the Linux kernel development area.

shaw's MUD. In August of 1989 he began to work on TinyMUD, which was to
introduce a whole new flavour of game to the genre. TinyMUD was designed to run
on computers running the UNIX operating system, and the growing popularity of
UNIX made possible the popularity of Aspnes' creation. TinyMUD was the first of
what were to be called 'social' MUDs. Indeed, TinyMUD players were encouraged
to center their play around cooperation and interaction rather than competition
and mastery. TinyMUD gave players extensive abilities to add items and rooms to
the game database, by giving them access to a library of commands that allowed
them to create and describe objects and areas, and made them behave in certain
ways in response to input from other players.

- Lars Pensjl, a swede, created LPMUD, the first MUD to have a built-in C-like lan-
  guage, which became one of the most popular MUDs by the early 1990's. LPMUDs
  are extensible adventure-style combat MUDs.

- From 1990 onward the number of MUD programs in circulation increased rapidly
  and the environments portrayed on them have become far more varied. The Tolkie-
  nesque fantasy worlds are still the most common, closely followed by science fiction
  worlds, but MUD environments based on actual or historical places have appeared.
  The meaning of the term 'MUD' has changed to reflect this. The original acronym
  'Multi-User Dungeon' has been joined by 'Multi-User Dimension' and 'Multi-User
  Domain', and the term has come to refer not to the original program written by
  Richard Bartle and Roy Trubshaw but to the entire program genre.[7]

- Stephen White of the University of Waterloo developed TinyMUCK and the MOO
  systems, the first version of which he released on May 2nd, 1990. These were
  extensions to TinyMUD that allowed users to write scripts controlling objects.
  MOO stands for MUD, Object-Oriented. White admits that the name is some-
  what pretentious–class and object creation is object-oriented, but the programming
  language is not.

- LambdaMOO, based on Stephen White's MOO, was brought up for the very first
  time on October 30th, 1990, with a tiny core database. Pavel Curtis is responsible
  for extensive modifications to the original MOO code (almost none of White's is
  still existent), as well as securing a permanent home for LambdaMOO at Xerox
  PARC (Palo Alto Research Center). MOOs extend the concept of a configurable
  MUD with a built-in object-oriented or object-based language.

- Since the public domain release of the MOO server code and its subsequent port-
  ing to various operating systems, MOOs have become a vastly popular form of
  communication and learning. They serve a variety of functions, both social and
  educational, and are not simply games (although certainly people do play games of
  one sort or another on MOOs, often involving skill or intellectual challenge). Some
  people use them to work on their programming skills: the MOO server code has a
  built in programming language for developing objects and verbs (commands) in the
  virtual reality interface and making them interactive and interesting. The language,
  called MOOCode, is a combination of C++ and LISP, but it is its own language

---

[7]Some would insist that MUD has come to stand for Multi Undergraduate Destroyer, in recognition of
the number of students who may have failed their classes due to too much time spent MUDding.

altogether in some respects. The MOO Programmer's Manual [41], written by Pavel Curtis, is the official guide to this language.

Figure 2.4: MUD and MOO by Liz Manicatide

- There are many MUDs and MOOs being used for academic purposes. For instance, the TecfaMOO[8][184] project which was started in fall of 1994 for education technology, research and life at TECFA, School of Psychology and Education, University of Geneva, Switzerland. A screenshot of a connection to TecfaMOO is shown on Figure 2.5.

2.2.2 Main Concepts Summarized

MUD usually stands for multi-user dimension or multi-user dungeon, and MOO stands for MUD Object Oriented. MUDs and MOOs are places, not physical places, but virtual spaces created on the network. They are housed on a computer, called a server, and are accessed through a text relaying client program, such as telnet, through the address of the server plus a port number. These are games where one can explore imaginary places, kill monsters, collect treasures, build new realms, or interact with other players[9]. Their development and success was supported by the rise of the wide area network. MUDs and MOOs have the following main characteristics:

- Several people can play at once.

- The game is partitioned into virtual spaces ("rooms") such that people and objects in one room cannot directly interact with people and objects in another room.

- All interactions take place in text, not pictures or sounds.

- Communications are handled with TCP sockets.

---

[8]TecfaMOO is located at: tecfamoo.unige.ch 7777.

[9]A lively description, drawn from experience, of these first virtual worlds can be found in J.C. Herz's book [89].

Figure 2.5: Screenshot of a connection to TecfaMOO

- Combinations of objects, rooms and exits allow simple puzzles, while some MUDs and MOOs with their own programming languages allow for much more complicated puzzles and toys.

- Even MUDs/MOOs created for serious purposes retain some of the original Adventure or role-playing game atmosphere;.

- As people virtually share a common place, social interactions take place.

- The software architecture is essentially based on many clients connecting to a single central server (see Figure 2.6).

While the players of MUDs only interact with the proposed virtual reality, MOOs give them the possibility to bring their own contribution to the virtual space by allowing them to become programmers. Thanks to an object-oriented[10] or at least object-based[11] language, the MOO player can create its own objects and rooms, and so enhance the virtual world, which becomes dynamic and unbound.

A second difference between MUDs and MOOs is that initially a MUD was considered a game oriented community while a MOO has been labelled as a social community.
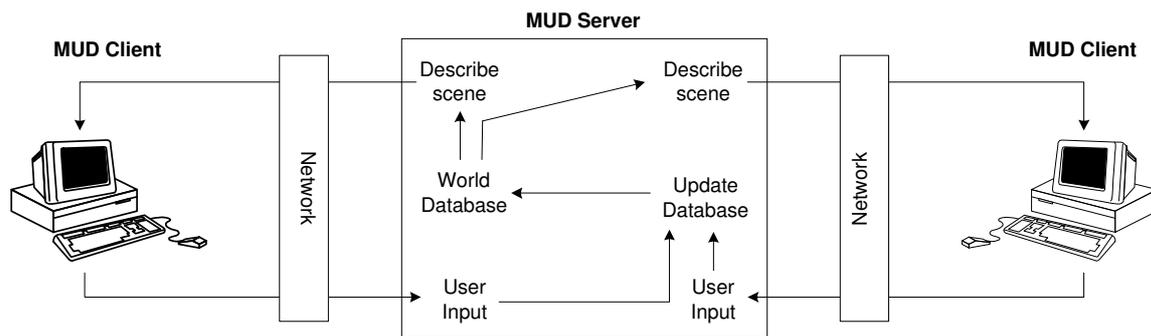
Figure 2.6: MUD architecture [153]

We will retain the basic concepts of MOOs but will enhance them by:

1. Distributing the server architecture.

2. Proposing "real" objects, that can be moved and executed.

3. Respecting standards, such as a common object-oriented programming language (e.g., Java) well suited for distributed programming.

2.3 Virtual Worlds

The various MUD's and MOO's running on the Internet have a lack of realism because they have no graphics at all and only a limited number of people with a lot of computer

---

[10]The programmer creates her own classes as templates for new objects. Inheritance and polymorphism are supported.

[11]The programmer creates her own objects essentially by cloning and parametrization.

knowledge could enter these. Thus, further developments to the experience of shared virtual environment took place.

First of all, it is worth noting that the terminology encountered in literature is diverse. One might call these environments distributed virtual environments, networked virtual environments or multi-user networked virtual environments. Other sources use collaborative virtual environments [19], distributed virtual reality, shared spaces [25], inhabited virtual worlds, distributed virtual worlds [45] or simply virtual worlds [47].

Heudin [47] introduces Virtual Worlds as a new field of research that studies complexity by attempting to synthesize digital universes on computers. This includes models of simple abstract worlds such as cellular automata to more sophisticated virtual environments using Virtual Reality and Artificial Life techniques. Moreover, Bruce Damer, in the fifth chapter of [47], introduces the concept of inhabited virtual worlds or Virtual Community.

- *Virtual Reality* is concerned with the design of graphical spaces using advanced three-dimensional image synthesis. Besides displaying 3D spaces, two other important aspects are involved: immersion and interaction. The operator typically evolves in the generated world thanks to data suit, head mounted display and data gloves. The idea is to feel "physically" present in the virtual environment and to interact with it.

- *Artificial Life* is focused on the simulation of living systems, and addresses the study of complex phenomena such as self-organization, reproduction, development, evolution of artificial life-like systems and emergent behavior. This field of research was initiated by Christopher Langton [113]. There is a large number of possible related approaches: for instance Cellular Automata [196, 74], Simulation of Ecological Systems, Bio-inspired Multi-agent Systems, Evolutionary Computing, Evolving Robots, and many others including related philosophical issues.

- *Virtual Community* or Inhabited Virtual World (IVW) finds its roots in the earliest text-based multi-user games: MUDs and MOOs. Continuing the trend was the development of IRC[12] and conferencing systems like the WELL in the 1970s and 80s [159]. Finally 'inhabited' 2D and 3D virtual spaces in Cyberspace have risen by the merging of text-based chat channels with a visual interface in which users are represented as 'avatars'. Virtual Community focus on 3D worlds, 3D avatars and essentially on social interaction and chat.

Over the past twenty years, three closely related classes of distributed real-time applications have evolved in parallel: *(i)* military simulations [42] *(ii)* networked virtual environments (NVEs) [175], and *(iii)* multi-player computer games (MCGs) [177]. According to this classification, an overview of the related work in these domains is given below. Sources for this section mainly include [175] and the extensive review found in [176].

2.3.1 Military Simulations

The United States Department of Defense (DoD) was one of the first to develop networked virtual environments with its SIMNET system and is the largest developer of networked

---

[12]IRC is the acronym for Internet Relay Chat.

virtual environments for use as simulation systems. SIMNET (Simulator Networking) is a distributed military virtual environment originally developed for DARPA by Bolt, Beranek and Newman (BBN), Perceptronics, and Delta Graphics. SIMNET was begun in April 1983 and delivered to U.S. Army at the end of March 1990. The goal of the SIMNET project was to develop a "low-cost" networked virtual environment for training small units to fight as a team.

Further attempts to formally generalize and extend the SIMNET protocol led to the Distributed Interactive Simulation (DIS) protocol, which was issued as IEEE Software Standard 1278 in 1992. The purpose is to allow any type of player on any type of machine to participate in the simulation. It is consequently used in many specialized systems such as NPSNET (Naval Postgraduate School Net) [123].

Current military research efforts aim at providing a general architecture and services for distributed data exchange. These systems are based on *High Level Architecture* (HLA), which was issued as IEEE Standard 1516 in 2000 [43]. HLA does not prescribe any specific implementation or technology and could be used with different type of simulation (even non-military).

The development of large-scale networked virtual environments by the Department of Defense has been poorly documented and was not publicly available. This fact pushed the academic community to reinvent and extend much of what the DoD did and, subsequently, to document that experience in public literature.

The Performance Architecture for Advanced Distributed Interactive Simulation Environments (PARADISE) project was initiated in 1993 by David Cheriton, Sandeep Singhal, and Hugh Holbrook of the Distributed Systems Group at Stanford University [149]. Unlike most academic research projects of the period, which primarily focused on the graphical aspects of networked virtual environments design, the PARADISE system explicitly addressed network software architecture issues facing environments containing thousands of users and its goal was to reduce bandwidth requirements throughout the system, correcting several mistakes made by SIMNET and DIS.

2.3.2 Networked Virtual Environments

Contrarily to military research, networked virtual environments focus less on large-scale systems and pay closer attention to the virtual representation of the participants (i.e. avatars) and the collaboration between the participants (for example operating at the same time with a shared object).

Habitat

The first true multi-user Virtual World started back in 1985. This Virtual World was Lucasfilm's Habitat (see Figure 2.7) and was created by Lucasfilm Games, a division of Lucas Arts Entertainment Company. It was one of the first attempts to create a very large scale commercial multi-user virtual environment, and was also the first one that used 'avatars'.

Habitat was not something that ran on expensive hard- and software in a laboratory. This Virtual World could be entered with a normal inexpensive home computer (a Commodore

64 computer) over ordinary commercial online services. The system supported thousands of users in a single shared cyberspace.

After 6 years of intensive use the world ended. The lead creators of Habitat, Chip Morningstar and Randy Farmer summarized their experiences in [141], where the following description of Habitat is given:

"Habitat is a 'multi-player online virtual environment'. Each user uses her home computer as a frontend, communicating over a commercial packet-switching data network to a centralized backend system. The frontend provides the user interface, generating a real-time animated display of what is going on and translating input from the user into requests to the backend. The backend maintains the world model, enforcing the rules and keeping each player's frontend informed about the constantly changing state of the universe. The backend enables player to interact not only with the world but with each other."

The following interesting points may also be retained:

- Avatars, in this case represented by animated figures, can move around, pick up, put down and manipulate objects, talk to each other, and gesture, each under the control of an individual user.

- Many objects are portable and may be carried around in an avatar's pocket.

- The Habitat world is made up of a large number of discrete locations called "regions", and doors, which may be open, closed, locked or unlocked, transport avatars from one region to another.

Figure 2.7: A typical Habitat scene (©1986 Lucas Arts Entertainment Company)

MR Toolkit

Minimal Reality (MR) Toolkit [172, 171, 198], from the University of Alberta Department of Computing Science, focuses on providing software that supports real-time interaction with a virtual environment using a head-mounted display (HMD). MR Toolkit divides the virtual environment into four components: presentation, interaction, geometric model, and computation. These components can be distributed among the nodes in a network. The Minimal Reality Toolkit Peer Package (MR-TPP), an extension of MR Toolkit, allows the master processes of different application instances to communicate with each other across the Internet, by using UDP (User Datagram Protocol) packets. Thus, multiple users can share the same virtual environment.

WAVES

WAVES [105, 106] (WAterloo Virtual Environment System) provides increased performance by exploiting distribution and parallelism. A virtual world, in this model, is comprised of a number of *Message Managers* which mediate communication between a number of *Hosts*. Host are processes which simulate a number of objoids (term used rather than *object* which carries too much baggage with it from object-oriented programming) in the virtual world, and provide some services to those objoids. The virtual world is composed of nothing but objoids.

To reduce the message volume in the system, a message manager filters (upon request) the messages which a particular host receives. These filters are either semantic (based on particular attributes, i.e. only send me objoids that make sounds) or positional (only send me objoids which are located in this particular region of space). Moreover, hosts and the message manager typically maintain a cache of objoids which they are interested in, and can update this cache at any time.

BrickNet

BrickNet [173, 174] is the work of Gurminder Singh at the Institute of Systems Science (ISS) at the National University of Singapore. It is a virtual environment toolkit that provides support for graphical, behavioral, and network modeling of virtual worlds. BrickNet system uses a client/server architecture. Each client connects to a server to request objects of interest and to communicate with other clients. A client can deposit its own objects to the server and thus share them with other clients. The object sharing operations are mediated by the BrickNet server. The primary contribution of BrickNet in the early networked virtual environments arena is that it explored the client-server space and did not require each client to have a complete copy of the virtual environment database.

RING

It is a client/server system that supports interaction between large numbers of users in a shared three dimensional virtual environment. RING [68, 69] takes advantage of the fact that state changes must be propagated only to hosts containing entities that can possibly perceive the change. This filtering is carried out by servers and is based on object space

visibility algorithm computations (line-of-sight visibility information). This approach reduces the message traffic required to maintain consistent state during multi-user visual simulations.

DIVE

The Distributed Interactive Virtual Environment (DIVE) [63] (see Figure 2.8), developed at the Swedish Institute of Computer Science, uses a replicated world database and peer-to-peer communication. When an object is updated, it is done in the local replica and a message is sent to all peers to update their own replicas accordingly. Naturally, this subjects the replicated object to inconsistencies due to network delays. DIVE compensates them by employing dead reckoning[13] and by sending periodically synchronization information. Scalability is achieved by making extensive use of multicast techniques and by partitioning the virtual universe into smaller regions. Furthermore, for improved scalability and fault tolerance DIVE does not rely on any central service (except the diveserver, a mostly passive name mapping service).

Figure 2.8: 3D human avatars around a desktop inside of a room in DIVE

---

[13]Technique that consists in computing missing information by approximation calculus. This technique is further explained in Subsection 3.2.

MASSIVE

MASSIVE [17, 84, 83, 82] (Model, Architecture and System for Spatial Interaction in Virtual Environments) supports different computer platforms and allows the users to interact with each other using arbitrary combinations of graphics, audio and text media over local and wide area network. The systems controls communications by a so-called spatial model of interaction so that one user's perception of another user is sensitive to their relative positions and orientations. The *awareness* which one object has of another is controlled by the observing object's *focus* and the observed object's *nimbus*, which describe, for each medium, regions of interest and projection, respectively.

COVEN

The efforts of DIVE and MASSIVE systems are combined and coordinated in the COVEN [51] project. COVEN (COllaborative Virtual ENvironments) is a European project that brings together twelve academic and industrial partners and that seeks to develop comprehensive approach to the issues in the development of Collaborative Virtual Environment technology.

Spline

Spline [200] (for Scalable Platform for Large Interactive Network Environments) shares many common goals and design choices with DIVE, although they have evolved separately. The Spline collaborative virtual environment was developed by the Mitsubishi Electric Research Laboratories (MERL). In the Spline software platform, the whole virtual environment or *world model* is divided into sub-regions called *locales*, which have arbitrary size and shape and are each associated with a separate multicast communication channel. This allows Spline to scale based solely on the maximum number of users that are gathered in any one locale, rather than on the total number of users in the virtual world. Spline uses a distributed architecture, with no centralized process, where each node maintains a partial copy of the virtual environment corresponding to the focus of attention.

Community Place

Community Place [115, 116, 117] is a system developed at Sony, that partially stems from a research collaboration with the Swedish Institute of Computer Science (SICS). Community Place is a shared multi-user VRML [8] system designed to work in the Internet. It consists of a VRML (Virtual Reality Modeling Language) browser, a multi-user server architecture and an application support environment. In Community Place, each entity sends position information to a server. The server uses spatial filtering based on auras to decide which other entities need to be aware of these position changes. The server also distributes updates to local scenes and events. The static elements of a scene are loaded as VRML, while dynamic data is managed through local scripts and message parsing. If the server becomes a bottleneck, it can be replicated, what allows to scale the entire system.

SmallTool

Smalltool [28, 27] is a toolkit to support shared virtual environments on the Internet. It consists of a VRML based parsing and rendering library, a device library, and a network library, DWTP. The Distributed Worlds Transfer and communication Protocol (DWTP), provides an application independent networking architecture to support large multi-user environments. DWTP uses a set of daemons for transmitting virtual world contents, detecting transmission failures and recovering lost packages. In addition to being replicated, each object in the virtual world can specify what is the event's maximum update frequency and whether a particular synchronisation event requires a reliable distribution or not. Synchronisation is performed by the sensor nodes, which catch an external event and make it available to the VRML scene.

DVECOM

Distributed Virtual Environment Collaboration Model (DVECOM) [35] is a centralized system which guarantees both synchronization of the display from the user point of view and consistency of scene rendering. The system provides QoS (quality of service) by degrading rendering of the virtual environment, if the client's resources are lacking. The main aspects of DVECOM are implemented by the proprietary virtual reality platform VirtualArch.

Urbi et Orbi

URBI ET ORBI [192, 53, 54] is a framework to dynamically manage completely distributed virtual environments, which relies on a dedicated scripting language, *Goal*, supporting code migration. The main characteristics of URBI ET ORBI are *(i)* its full and symmetric (peer-to-peer) distribution; *(ii) Goal* allows to describe the objects and their behavior including their distribution policy in addition to the usual geometric descriptions. To keep the pressure upon the network as low as possible, if the motion of an object is fully predictable, deterministic (for instance a windmill), it is replicated on each node in the network, and each node is responsible for updating the local replica. This avoids frequent updates of its position which would waste bandwidth. An unpredictable, indeterministic object (for instance an avatar) is distributed to one node from where it begins to multicast (group communication) update messages to the network.

PaRADE

PaRADE (Predictive Real-time Architecture for Distributed Environments) [161, 162] has been implemented at the University of Reading. In PaRADE system replicated databases are maintained through the communication of non-deterministic events and local calculation of deterministic events, thus greatly reducing bandwidth requirements. Events can be discrete (e.g., a state change) or continuous (e.g., an audio stream). Discrete events require that the before-after relations are preserved, whereas continuous events are stamped with a wall clock time that is kept synchronized in each node.

CIAO

CIAO (Collaborative Immersive Architectural layOut) [183] is a multi-user, large-scale 3D layout system. It uses a multicast-based, optimistic method for concurrency control for replicated objects. An update is carried out immediately in the local replica and transmitted to the remote nodes. If a conflicting operation occurs, a token associated with the manipulated object is used to maintain consistency. Initially, the object's creator has the token. When the owner of the token receives the update message, it validates the operation by giving the token to the node which originated the update.

Real-Time Transport Protocol (RTP/I)

(RTP/I) [195, 131] is a real time application level protocol for distributed interactive media (e.g., shared whiteboards, distributed virtual environments, networked computer games). RTP/I stands for Real-Time Application Level Protocol for Distributed Interactive Media, and reuses many aspects of RTP. It includes three methods for ensuring that all application instances look as if all operations have been executed in the same order. Inconsistencies caused by operation delays (e.g., network latency) are handled by voluntarily delaying the local updates to match the transmission delays; this approach is called *local lag*. Each node keeps a history, and if an inconsistency occurs, the situation is rolled back, the conflicting operation is carried out, and situation is rolled forward back to the current time. This is an improved *timewarp* algorithm, which can repair inconsistencies using exclusively local information without burdening the network. As a last resort, the protocol includes also a method for explicit state request for handling exceptional situations.

Virtual Object System

The Virtual Object System (VOS) [9] is an open infrastructure for object-oriented network communication, and for building flexible, distributed object networks for a variety of purposes. Its primary application is multiuser collaborative 3D virtual environments. The core VOS library is made up of C++ classes.

ActiveWorlds

Active Worlds [3] is a streaming 3D world for the Windows platform and unix servers with in-world building metaphor. Components are downloaded, cached and replicated. It allows users over the Internet with nothing more than a Windows PC and a modem connection to navigate and build in a large virtual space while interacting with others.

V-Worlds

The Virtual Worlds Group, part of Microsoft Research, has implemented the V-Worlds platform [187], which facilitates the development of shared virtual environments. V-Worlds is based on standard COM functionality and uses a client-server distributed architecture. Each client keeps a locally cached copy of the objects that it needs to render the virtual environment and changes made to the "master copy" at the server have to be

reflected to all clients that have local copies of the object. An event mechanism allows this information propagation, and the *bystander* algorithm relying on a hierarchy of *containment* of V-Worlds objects provides for limiting communication needs. The platform also provides persistent state management (using a logging mechanism), security and ease of development (through object inheritance).

2.3.3 Multi-player Computer Games

Flight

Gary Tarolli of Silicon Graphics, Inc. (SGI) is the original programmer of the Silicon Graphics demo program, *Flight*, in the summer of 1983. In that program, the user selects any one of a number of airplanes and uses the keyboard of the workstation to accelerate and to steer. The entire purpose of the demonstration was to see if you were coordinated enough to land your plane.

Beginning in 1984, networking was added to *Flight*. In the networked *Flight* game, users could see each other's planes, although there was no other interaction between players.

Dogfight

Sometime after the release of the networked version of *Flight*–in early 1985 it is believed– SGI engineers modified the code of *Flight* to produce the demonstration program *Dogfight*. This modification dramatically upgraded the visibility of networked virtual environments, as players could now interact by shooting at each other.

Flight/Dogfight inspired the development of more networked virtual environments and games.

Amaze

Amaze is thought of as the first modern networked game/virtual environment, dating from 1984 [20]. In Amaze the players are in a 2D maze and their goal is to shoot missiles to other players. Using point-to-point communication, each node transmits once a second position and velocity updates (thus allowing dead reckoning).

Doom

While workstation-based virtual environments are some of the earliest inspiration for networked virtual environments, the PC has taken the desire and interest for such connected worlds to the next level. On December 10th, 1993, id Software released its shareware game *Doom* [91]. This networked ability to blast people in a believable 3D environment created an enormous demand for further 3D networked games. An estimated 15 million shareware copies of *Doom* have been downloaded around the world.

Doom's play is the archetypical shooter. You are in a 3D space, a space oozing with toxic waste and full of monsters. In networked form, you play against the would-be online monsters represented by the other players.

XPilot

XPilot [178] is a game which allows you to guide a triangle shaped space craft in a two-dimensional cave-like environment you share with other players. By configuring some parameters, several playing modes are possible, as races through a course between the cave walls or dogfights with machine guns. XPilot uses a simple client/server architecture and because it does not employ dead reckoning, the responsiveness is effectively determined by the network latency.

Artery

Artery [34] is a distributed system, which is specifically designed to support network game applications by providing a high level application program interface and by taking advantage of application-specific semantics to optimize the network performance. Networking is based on DIS protocol. The system features such network bandwidth reduction techniques as dead reckoning, message aggregation and interest management.

MiMaze

MiMaze[14] [75, 76] is based on iMaze [107]. iMaze is a 2D "Pacman" game with a server based architecture. Avatars are "Pacmen" that evolve in a maze where they try to kill each other. MiMaze uses iMaze rules but includes 3D graphics, sounds and video and its architecture is totally distributed, i.e. the state of the game is computed by each participant (there is no server to compute the game state). Because of the distributed (i.e. serverless) architecture of this application, a synchronization mechanism (called bucket synchronization) has been designed to cope with different transmissions delays among the participants. Delays between participating hosts are evaluated by using a global clock time. A message issued at absolute time is delayed according the measured transmission delay so that all participants can evaluate it at the same time. If the message is missed or arrives too late, dead reckoning is used to extrapolate the information.

Distributed Entertainment Environment (DEE)

DEE [153] is an architecture for supporting a game genre known as graphical multi-user dungeons. It proposes that a graphical MUD environment may be treated and implemented as three distinct models: a conceptual model (i.e. high level game related concepts, rules and object attributes), a dynamic model (i.e. interaction at the spatial level), and a visual model (i.e. rendering information). Thus, where the conceptual model knows that a door can be opened, and the dynamic model knows how it opens, the visual model knows what it looks like as it opens. The three types of models are coupled together by message channels and to reduce network traffic the conceptual and the dynamic models are stored in a server, while each participating client has its own instance of the visual model.

---

[14]MiMaze is available from http://www-sop.inria.fr/rodeo/MiMaze/ (accessed December 28, 2004).

2.3.4 Main Concepts Summarized

Definition

The preceding subsections provide a good feeling of what Inhabited Virtual Worlds or shortly Virtual Worlds are. But let us recall the five features which distinguish them according to [175]:

- A shared sense of space: All participants have the illusion of being located in the same place, such as in the same room. That shared place represents the common location within which other interactions can take place. The place may be real of fictional. The shared space must present the same characteristics to all participants. For example, all participants should get the same sense of temperature and weather, as well as the acoustics. Though it needs not be presented graphically, the most effective networked virtual environments provide an immersive three-dimensional representation of the shared place.

- A shared sense of presence: When entering the shared place, each participant takes on a virtual persona, called an avatar, which includes a graphical representation. Upon entering the networked virtual environment, each participant can see the other avatars that are located in the shared place, and those users can see the new participant's avatar. Similarly, when a participant leaves the networked virtual environment, other participants should see the avatar depart.

- A shared sense of time: Participants should be able to see each other's behavior as it occurs. In other words, the networked virtual environment should enable real-time interaction to occur.

- A way to communicate: Though visualization forms the basis for an effective networked virtual environment, most networked virtual environments also enable some communication among the participants–by gesture, by typed text, or by voice. This communication adds a necessary sense of realism to any simulated environment. It is a fundamental component of engineering or training systems.

- A way to share: The aforementioned elements effectively provide a high-quality video conferencing system. However, the true power of networked virtual environments derives from users' ability to interact realistically not only with each other but also with the virtual environment itself. Users should be able to pick up, move and manipulate items that exist in the environment, and they should be able to give items to other participants. Users might even be empowered to manipulate the environment by destroying existing parts of it or by building new ones.

The ability to share objects differentiates networked virtual environments from traditional chat rooms, and the real-time interactivity differentiates networked virtual environments form traditional Web browsing or electronic mail.

The following definition is given by Diehl in [45]:

> *Virtual worlds* are computer-based models of three-dimensional spaces and objects with restricted interaction. A user can move through a virtual world and interact with those objects in various ways.

A *multi-user* virtual world is a virtual world where several users interact at the same time. These users work at different computers which are interconnected. In multi-user worlds the avatar plays a central role.

A virtual world is *distributed* if active parts of it are spread throughout different computers in a network. There is no need for a single host to have full knowledge of the world.

Application Domains

Even if Virtual Worlds are still young and evolving, applications have already been found and may be found in the future. Some examples are given below:

- In majority, people use Virtual Worlds as an extension of their real world social lives.

- There are companies who set up a Virtual World for better communication between their world-wide employees and give them a feeling of togetherness.

- Some universities provide support to their students in a virtual world, commonly called virtual campus.

- Other domains where Virtual Worlds could find applications in the future are virtual conferencing, virtual diplomacy, virtual museums, telemedecine or the development of virtual shopping malls.

- Groupware and WYSIWIS (What You See Is What I See) applications could use virtual worlds as supporting platform.

2.4 Technological Background

This section presents the background of the present thesis at the technological level. The implementation work consists in the design, implementation and documentation of an application framework, and therefore has been influenced by object-oriented programming concepts and software reusability techniques used in other frameworks. Furthermore, the developed framework is intended for distributed applications. Hence, the issues related to this kind of software and approaches to solve them have also been studied.

2.4.1 Frameworks and Application Frameworks

We have experienced a significant increase in software reusability and an overall improvement in software quality due to the application of object-oriented programming concepts in the development and (re)use of *semifinished software architectures* rather than just single components, small building blocks or procedures and subroutines libraries. A semifinished architecture is a collection of cooperating classes that make up a reusable design solution for a given problem domain. It typically includes objects that provide default behavior and programmers use it by inheriting some of that default behavior and overriding other behavior so that the system calls application code at the appropriate times. The term *framework* is used for these architectures.

There are three main differences between frameworks and class libraries [40]:

- *Behavior versus protocol.* Class libraries are essentially collections of behaviors that you can call when you want those individual behaviors in your program. A framework, on the other hand, provides not only behavior but also the protocol or set of rules that govern the ways in which behaviors can be combined, including rules for what a programmer is supposed to provide versus what the framework provides.

- *Inversion of control.* With a class library, the code the programmer writes instantiates objects and calls their member functions. It is possible to instantiate and call objects in the same way with a framework–that is, to treat the framework as a class library–but to take full advantage of a framework's reusable design, a programmer writes code that is called by the framework (see Figure 2.9). This is usually handled by overriding some methods or by implementing superclass abstract ones. The framework manages the flow of control among its objects. Writing a program involves dividing up responsibilities among the various pieces of software that get called by the framework rather than specifying how the different pieces should work together. This relationship is expressed by the Hollywood Principle [114]: "Don't call us, we'll call you."
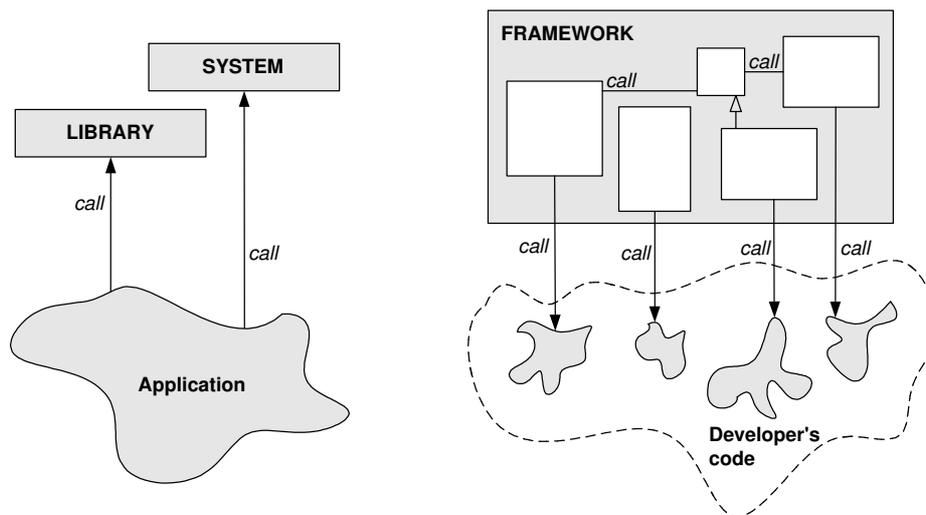
Figure 2.9: Class libraries versus frameworks

- *Implementation versus design.* With class libraries programmers reuse only implementations, whereas with frameworks they reuse design. A framework embodies the way a family of related programs or pieces of software work. It represents a generic design solution that can be adapted to a variety of specific problems in a given domain. For example, a single framework can embody the way a user interface works, even though two different user interfaces created with the same framework might solve quite different interface problems.

A framework often embodies specific domain expertise. Application code customizes the framework in a way that solves a particular application problem within the general domain of the problem the framework was designed to solve. In the case of an *application*

framework like MacApp (see Subsection 2.4.4), the expertise is in the domain of writing programs for the Mac OS that display menus and windows and perform other basic tasks consistently and reliably. In the case of a framework developed by a bank, the domain expertise embodied in the framework might be how customer accounts or certain kinds of financial transactions work. In this case, a programmer might customize the framework to create specific kinds of accounts or financial instruments

2.4.2 Hot Spots and Frozen Spots

The signature quality of a framework is that it provides an implementation for the core and unvarying functions, as well as includes a mechanism to allow a developer to plug in the varying functions, or to extend the functions. Therefore, it is considered that an application framework consists of frozen spots and hot spots [154, 155].

Frozen spots define the overall architecture of a software system–its basic components and the relationships between them. These remain unchanged in any instantiation of the application framework.

Hot spots represent those parts of the application framework that are specific to individual software systems. Hot spots are designed to be generic–they can be adapted to the needs of the application under development.

When creating a concrete software system with an application framework, its hot spots are specialized according to the specific needs and requirements of the system.

2.4.3 Black-box and White-box Frameworks

There are essentially two types of frameworks, each representing a different degree of maturity of the design. The types may be called white-box and black-box frameworks. They are discussed in a Smalltalk context by Johnson and Foote in [101].

In a *white-box* framework, tailoring is done by replacing chosen operations by any new behavior seen fit: this is reuse by *inheritance*. This gives fine grain control but often makes it difficult to modify the default behavior without detailed knowledge of the internal design of the framework classes. The major problem with such a framework is that every application requires the creation of many subclasses. A second problem is that a white-box framework can be difficult to learn to use, since learning to use it is the same as learning how it is constructed.

In a *black-box* framework, the user may only supply new behavior using visible features defined in the framework. The user can reuse components by plugging them together and not worrying about how they accomplish their individual tasks: this is reuse by *composition*. Only the visible interface of its components must be understood. Black-box frameworks are easier to learn to use than white-box frameworks, but are less flexible and harder to design.

However, most real-world frameworks are between these two categories and are called *gray-box* frameworks. Gray-box frameworks are matured white-box framework with some pre-built components which can be instantiated and 'plugged' as in black-box frameworks. Thus, a gray box framework allows tailoring by extension and/or by composition. Gray-box frameworks are designed to avoid the disadvantages presented by white-box and

black-box frameworks. In other words, a good gray-box framework has enough flexibility and extendibility, and also has the ability to hide unnecessary information from the application developers [212].

2.4.4 Framework Examples

There exist a lot of frameworks in various application domains. Examples are application frameworks for visual programming like FOIBLE [97], for controlling real-time psychophysiology experiments [58], for the development of operating systems [124], for visual language systems [67], for building graphical editors for different domains like artistic drawing, music composition, and mechanical CAD [187, 99], for helping building compilers for different programming languages and target machines [102], for helping building financial modeling applications [22], for creating graphical editing applications (Unidraw [193, 194] based on the InterViews toolkit), or for constructing Smalltalk-80 user interfaces (Smalltalk Model/View/Controller (MVC) [110, 79]). All these frameworks can be customized to a particular application by creating application-specific subclasses of abstract classes from the framework. Some other frameworks are now briefly commented:

- MacApp [206, 38, 170], from Apple Computer, was one of the first commercially successful application frameworks. It provides working code for a generic Mac OS application, which is not easy to write from scratch. The generic application is built from objects, so MacApp programming involves using object-oriented techniques even though the objects are implemented on top of a procedural operating system. An abstract MacApp application consists of one or more windows, one or more documents, and an application object. A window contains a set of views, each of which displays part of the state of a document. MacApp also contains commands, which automate the undo/redo mechanism, and printer handlers, which provide device independent printing.

  The generic application includes a working menu system that takes care of pulling down menus, highlighting, dimming items that aren't active, implementing command-key equivalents, and so on. It also includes standard File and Edit menus.

  The File menu includes working New, Open, Close, Save, and Save As commands and the dialog boxes for locating files and disks that go with them. The File menu also includes the Print and Page Setup commands, with the corresponding code for printing multiple copies on various sizes of paper, portrait or landscape orientation, and so on. The last default command in the File menu is the Quit command, for exiting the program.

  The Edit menu includes Cut, Copy, Paste, and Undo, all supported by generic code, which sets up and runs the commands, with stubs for the code to be provided by a specific application.

  A programmer does not have to write any of these menu commands from scratch. They are working and ready to use, although in some cases they cannot really do much without additional application-specific code. To make an application do anything useful, it is necessary to override at least some of the application framework's member functions.

Most application classes do little besides defining the class of their documents. They inherit a command interpreter and menu options. Most document classes do little besides defining their windows and how to read and write documents to disk. They inherit menu options for saving the documents and tools for selecting which document to open next. An average programmer rarely makes new window classes, but usually has to define a view class that renders an image of a document.

MacApp not only ensures that programs meet the Macintosh user-interface standard, but makes it much easier to write interactive programs. The first version of the framework was written in Object Pascal, then the global architecture was modified and implemented in C++.

- ET++ [203, 72, 201, 48, 202, 204] was developed by André Weinand and Erich Gamma in Prof. Marty's group at the University of Zurich. The same team was responsible for the evolution of ET++ at the Union Bank of Switzerland's Informatics Laboratory (UBILAB) in Zurich. ET++ is a portable and homogenous object-oriented class library integrating user interface building blocks (for example, buttons and menus), basic data structures, and high level application framework components which predefine as far as possible the look and feel of ET++ applications. The main goals in designing ET++ were the desire to substantially ease the building of highly interactive applications with consistent user interfaces following the well known desktop metaphor, and to combine all ET++ classes into a seamless system structure. ET++ is implemented in C++ and can be used on several operating systems and window system platforms.

- The Microsoft Foundation Class Library (MFC) [156] is an "application framework" for programming in Microsoft Windows. Written in C++, MFC provides much of the code necessary for managing windows, menus, and dialog boxes; performing basic input/output; storing collections of data objects; program features like property sheets ("tab dialogs"), print preview, and floating, customizable toolbars; and so on. All you need to do is add your application-specific code into this framework. And, given the nature of C++ class programming, it is easy to extend or override the basic functionality the MFC framework supplies. Furthermore, MFC gives easy access to "hard to program" user-interface elements and technologies, like Active technology, OLE, and Internet programming and simplifies database programming through Data Access Objects (DAO) and Open Database Connectivity (ODBC), and network programming through Windows Sockets.

- JUnit [103] is an open source Java testing framework written by Erich Gamma and Kent Beck. It is used by the developer who implements unit tests in Java. It is an instance of the xUnit architecture for unit testing frameworks. JUnit features include:

  - assertions for testing expected results,
  - test fixtures for sharing common test data,
  - test suites for easily organizing and running tests,
  - graphical and textual test runners.

- Java's Swing is the typical example of a GUI framework. It provides many classes and interfaces for core GUI functions. Developers can add specialized widgets by

subclassing from the Swing classes and overriding certain methods. Developers can also plug in varying event response behavior to predefined widget classes (such as JButton) by registering listeners or subscribers based on the Observer pattern [73].

- Gachet [71] presents the design of a distributed software framework for developing distributed decision support systems (DSS) characterized by highly decentralized, up-to-date data sets coming from various sources. The proposed Java-based framework relies mostly on the Jini technology and its JavaSpaces service. It has been developed at the Decision Support Laboratory at the University of Fribourg, Switzerland, for crisis management in the food supply sector. A complete description of the framework can be found in [70].

The six preceding framework examples would all be categorized as gray-box frameworks.

2.4.5 Framework: a Summary Definition

There are a lot of definitions and discussions for what a framework is. A complete and precise one is given in [114] but let us take the definition found in [30] which summarizes well the discussion contained in this section:

> "A framework is a partially complete software (sub-) system that is intended to be *instantiated*. It defines the architecture for a family of (sub-) systems and provides the basic building blocks to create them. It also defines the places where adaptations for specific functionality should be made. In an object-oriented environment, a framework consists of *abstract* and *concrete classes*. The instantiation of a framework involves composing and subclassing the existing classes. A framework for applications in a specific domain is called an *application framework*."

2.4.6 Design Patterns

The architect Christopher Alexander, known for his work on urban planning and building architecture [6, 5], first introduced the concept of design pattern and inspired the object-oriented community. Even though his patterns encode knowledge of the design and construction of communities and buildings, his definition is true about object-oriented design patterns:

> "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice".

Software patterns first became popular with the object-oriented Design Patterns book [73], also known as the Gang of Four book (Gamma, Helm, Johnson and Vlissides), which offers a catalog of such patterns and where the following description can be found:

"A design pattern names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design. The design pattern identifies the participating classes and instances, their roles and collaborations, and the distribution of responsibilities. Each design pattern focuses on a particular object-oriented design problem or issue."

Thus design patterns are concerned with the design of reusable object-oriented software and have some similarities with frameworks, as both are attempts to reuse design. But they are different in three major ways:

- Patterns are more abstract than frameworks.

- Frameworks are at a different level of abstraction than patterns: design patterns are less specialized than frameworks.

- Design Patterns are smaller and are the architectural elements of frameworks.

The main purpose of design patterns is to capture the design of a framework and its individual classes without revealing implementation details. Such abstract design descriptions should allow communication of mature designs in an efficient way, and are therefore a valuable means of documenting existing frameworks. An added benefit comes when the framework is documented with the design patterns it uses.

2.4.7 Framework Documentation

Since frameworks are reusable designs, not just code, they are more abstract than most software, which makes documenting them difficult. According to [99], documentation for a framework must describe in the following order:

1. the purpose of the framework, i.e. the problem domain for which it is designed. It is usually hard to specify a problem domain precisely, but a small set of examples usually makes the general area clear. These examples are intended to show what the framework is good for;

2. how to use the framework. Indeed, most users of a framework are not interested in a description of the design of the framework, but want a kind of cookbook, giving detailed instructions for using it efficiently;

3. the detailed design of the framework, since the major weakness of cookbooks is that they only describe the typical way the framework will be used. However, a good framework will be used in ways that its designers never conceived. Design patterns are a good way to describe how the framework works.

One can notice that in a framework documentation the theory follows practice. This is not a new idea and is part of the folklore of frameworks. The documentation for MacApp [37] has long contained a cookbook and the first documentation for Model-View-Controller (MVC) was called a "Cookbook for Model-View-Controller" [110].

Examples also play a key role in the documentation of frameworks, since they make them more concrete, make it easier to understand the flow of control and help the reader to determine whether she or he understands the rest of the documentation.

2.4.8 Unified Modeling Language (UML)

To discuss the design of an application, to express an object-oriented design, or to document a design pattern, a visual modeling language is required. In the design patterns book [73] a notation based on the Object Modeling Technique (OMT) [165, 164] is adopted for graphically representing the classes. To illustrate sequences of requests and collaboration between objects, interaction diagrams taken from the Object-Oriented Software Engineering (OOSE) method [94] and from the Booch [23] method are also used.

In the present work, the more modern UML is employed. UML stands for Unified Modeling Language. It is the successor of the wave of object-oriented analysis and design methods that appeared in the late 80's and early 90's. UML most directly unifies the methods of Booch, Rumbaugh (OMT) and Jacobson, and became the OMG (Object Management Group) standard (see Figure 2.10). The three authors at the origin of this unified notation are often referred to as the three amigos. Recently UML is being widely accepted in both industry and academia as a language for architecture description.

Most methods consist of both a modeling language and a process. The modeling language is the (graphical) notation that methods use to express design. The process is their advice on what steps to take in doing a design. The process developed besides UML is the RUP, for Rational Unified Process [93, 111]. Note that you do not have to use the process (RUP) in order to use the modeling language (UML), as they are clearly separate.

The RUP is not used in this work and is not further considered in the present dissertation. UML, however, is used in order to communicate certain concepts and document the design of the developed framework. More precisely, static class diagrams and dynamic sequence diagrams are employed. A good and concise description of UML is given by [61] and more detailed and longer study of UML can be found in [24, 166].

2.4.9 Distributed Computing

With the emergence of the Internet, the vast majority of all computing today is distributed, i.e. programs that make calls to other address spaces, possibly on another machine. In [197], the authors outline core distinctions between local and distributed design and show the weaknesses of models that ignore or deny these differences, like the Object Management Group's Common Object Request Broker Architecture (CORBA) [86]. Indeed, in CORBA an object, whether local or remote, is defined in terms of a set of interfaces declared in an interface definition language and the underlying mechanisms used to make a method call are hidden from the programmer.

Thus, the programmer has to face the realities of network programming and deal with problems like partial failure, concurrency issues and latency. Peter Deutsch [44] states the following eight assumptions, that must **not** be made when building a distributed application:

1. The network is reliable.

2. Latency is zero.

3. Bandwidth is infinite.

4. The network is secure.

Grady Booch' s methodology
(Booch method)

Object Modeling Technique (OMT)
(James Rumbaugh)

Unified Method 0.8

1994: Grady Booch and James Rumbaugh started working
together at Rational to unify their separate methodologies

Object-Oriented Software
Engineering method (OOSE)
(Ivar Jacobson)

1995: Ivar Jacobson joinded Rational and the three began working unifying
the three methods

UML 0.9

1996: During 1996 the "three amigos" worked on a new method, renaming
it the Unified Modeling Language (UML)

UML 1.0

September 1997: Submission to the Object Management Group (OMG)

UML 1.1

November 1997: OMG accepts the UML as the official industrial standard
for software engineering notation

UML 1.2
1998 (minor revision)

UML 1.3
1999 (minor revision)

UML 1.4
May 2001 (minor revision)

UML 1.5
2002 (minor revision)

UML 2.0 (1st revision)
June 2002: OCL, infrastructures

UML 2.0 (2nd revision)
January 2003: diagram interchange, superstructures

UML 3.0
200?: Planned major version

Figure 2.10: A brief history of UML

5. Topology does not change.

6. There is one administrator.

7. Transport cost is zero.

8. The network is homogeneous.

Java Remote Method Invocation (RMI) [85, 146], the framework for Java technology-based distributed applications, allows for addressing the issues mentioned above and supports true object-oriented polymorphism. That is the reason why, the application framework documented in this dissertation is built on top of Java RMI and takes advantage of features such as dynamic classloading, which allows for the development of mobile objects. Furthermore, in order to have a system which is robust, flexible and highly adaptive to change, the Jini™ [11, 112] technology is also exploited. Jini is a Java-based solution that enables all types of services to work together in a federation or community-organized without extensive planning, installation, or human intervention.

2.4.10 Design by Contract and Unit Testing

This subsection briefly presents the concepts of *Design by Contract* and Unit Testing and then shows how they are related.

Design by Contract

One main goals of every program is reliability, that is correctness and robustness. A program is correct if it performs according to its specification. It is robust if it handles situations that were not covered in the specification in a graceful manner. One way to prove the correctness of a program with respect to a (formal) specification is the Hoare [10, 90] calculus. Based on this formal method, Bertrand Meyer developed a method of software engineering called *Design by Contract* [135]. It prescribes that software designers should define precise checkable interface specifications for software components based upon the theory of abstract data types and the conceptual metaphor of a legal contract.

The central idea of *Design by Contract* is that a class and its clients have obligations to each other: the client must guarantee certain conditions before calling a given method on the class (the preconditions), the class guarantees certain properties after the call (the postconditions)). If the pre- and postconditions are included in a form that the compiler can check, then any violation of the contract between caller and class can be detected immediately. Furthermore, programming by contract is a way to provide rigorous specifications in a way that is accessible to a good technical programmer.

The programming language that offers the best support for *Design by Contract* is Eiffel[15] [134], designed by Bertrand Meyer. Specification languages for expressing *Design by Contract* constructs such as invariants, pre- and postconditions are also available, as for example, the Object Constraint Language (OCL) [199]. OCL is part of the UML standard and allows construction of logical expressions thus improving the precision of a UML specification.

---

[15]The first version of Eiffel was released at the end of 1986.

In the programming part of this thesis, the Java object-oriented language has been preferred. Java does not directly support *Design by Contract* but there are many external tools attempting to incorporate "contract" into a Java program. Some examples are jContractor [104, 2], iContract [109], Jass [12], JMSAssert [98], JContract [95] or the DBCProxy framework [49]. Yet these tools are not used in this work, the spirit of *Design by Contract* has been respected while specifying and documenting the public interfaces of the various components of the framework (see Subsection 6.1.1).

Unit Testing

Unit tests are at the heart of Extreme Programming (XP) [14], a software development method, which was first introduced by Kent Beck in 1996. Unit tests are a set of executable pieces of code that exercise "a unit" of production code in isolation from the full system and check that the results are as expected. So unit tests provide a way to check code for correctness. Furthermore, unit tests are automated and run without human interaction.

Developers who implement unit tests in Java take advantage of JUnit [103], a concrete testing framework written by Kent Beck and Erich Gamma. But unit testing is possible with any object-oriented programming language, and the XUnit family of testing frameworks contains a lot of members (e.g., SUnit for Smalltalk, cppUnit for C++, DUnit for Delphi, PyUnit for Python, csUnit for C# and other .NET languages).

Some key software components have been verified with the help of unit tests, but this technique is not extensively used in the present work.

The difficult part with unit testing is to select appropriate test cases, i.e. to elaborate "good" scenarios which cover all critical and special situations one wants the code to handle properly. Generating concrete test cases actually consists of two tasks [125]:

- Select the path through the program, i.e. the sequence of methods to be called.

- Select the input data for all method calls.

Both are hard tasks and are difficult to automate. Algorithms like the one presented in [13] help selecting a judicious path through the program. The second task is more difficult but a possible approach is sketched below.

Design by Contract versus Unit Testing

Design by Contract and Unit Testing are both techniques for writing correct and reliable software and are very similar. Nevertheless, there are still some fundamental differences between *Design by Contract* assertions checking and unit testing and each has different strong points.

Unit tests generally focus on a single class and do not exercise classes in concert. In particular, unit tests are a poor vehicle for checking preconditions. Unit testing and *Design by Contract* postconditions validate individual module behavior, with unit testing focusing on specific situations and *Design by Contract* postconditions focusing on general behavior.

There are several advantages to running unit tests, even in a *Design by Contract* environment:

- Better code coverage: unit tests provide more thorough code coverage than what a typical application would do. *Design by Contract* assertions are useless if the code is never called.

- Repeatable: unit tests are automatic and repeatable. You do not have to rely on a user manually running a program.

- Reportable: the results of the unit test are easily summarized for reporting.

In order to automate the testing process, one has to know how to select input test data for all method calls. To address this issue, the notion of *equivalence partitioning* [16] is useful, and *Design by Contract* can help the programmer to describe the different equivalence partitions. By combining these two concepts, a unit testing tool, which generates and evaluates test cases automatically, has been developed and is briefly presented in [126].

3

Virtual Worlds Main Problems

A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.
—Leslie Lamport

Despite the many attempts (see the long yet not exhaustive list of Section 2.3) there is no widespread adopted virtual world technology available. The reason behind this fact is that developing a distributed virtual environment is hard. As also stated in [9], the major hurdles include, but are not limited to: performance of 3D rendering, network performance, resource discovery and organization, single points of failure, security, privacy, trust, ease of use (by users, content creators and programmers), extensibility, persistence, and finally the insufficiency of existing protocols and software in addressing most of these issues. This chapter briefly presents and discusses these problems, that an ideal distributed virtual environment should tackle.

3.1 Presentation

The most important point when defining a virtual environment is to isolate the presentation level and the conceptual level from each other as much as possible. Indeed, defining a virtual environment in terms of the configuration and behavior of objects, rather than

their presentation, enables us to span a vast range of computational and display capabilities among the participants in a system. So it would be possible to have two users looking at the same tree in a same place and talking to each other as they do so, but while the first user's interface just displays a simple "There is a tree here", the second user could see a high resolution 3D graphic representation of the tree. Rendering performance for such 3D representations is certainly not a problem with recent almost standard hardware. Furthermore this approach allows the system to evolve as tomorrow's technology develops.

3.2 Network Performance

While developing multiuser distributed virtual worlds, one has to deal with problems of network performance, and in particular latency[1], bandwidth[2] and reliability[3]. Good communication and data distribution architectures and techniques that help to optimize the network traffic allow to reduce bandwidth usage and improve latency.

Network Traffic

A big issue virtual worlds platforms have to face, is the reduction of the network bandwidth requirements. The most common techniques used to achieve this goal are (for more details see [176, 175]):

- packet compression and aggregation: Packet compression reduces the number of bits needed to represent particular information. Thanks to *lossless* compression techniques the size of data can shrink down to half, and *lossy* compression techniques can be employed to achieve higher compression ratios, for example, for audio and image data. Packet aggregation consists in merging several packets and transmitting their content in one larger packet. Thus, the overhead caused by packet headers is smaller. Message compression and aggregation saves bandwidth but requires extra computation.

- interest management: To save bandwidth, update packets should only be sent to those nodes that are interested in them. Interest management techniques allow the nodes to express interest in only the subset of information that is relevant to them [142]. For example, message filtering can be based on semantic or positional criteria [105, 106] or on space visibility algorithm computations [68, 69]. Another successful approach is the introduction of the notion of *Aura* or *area of interest*. *Aura* is a subspace where interaction occurs, and it usually depends of the sensing capabilities of the system being modelled. Thus, when two avatars' aura intersect, they can be aware of each others actions. This leads to a symmetric interest management. However, aura can be further divided into *focus* (observer's perception)

---

[1]In a network, latency, a synonym for delay, is an expression of how much time it takes for a packet of data to get from one designated point to another.

[2]Bandwidth has a general meaning of how much information can be carried in a given time period (usually a second) over a wired or wireless communications link.

[3]Reliability means that systems can logically assume that data sent is always received correctly. Reliability often forces a compromise between bandwidth and latency [122].

and *nimbus* (observed object's perceptivity) [18, 82]. Thus, awareness requires that the observer's focus intersects with the observed nimbus.

- dead reckoning: Especially in the case of position information, one can reduce bandwidth use by sending update packets less frequently. To maintain consistency, the lack of information between the packet updates is compensated by approximation calculus. Based on previously received information (commonly a known starting point and velocity), the node predicts movement of a particular object. The predicted state of the object is used in the application until new information is received from the source node. Then a convergence algorithm is used to smooth the transfer from the approximated to the actual location. Therefore, dead reckoning consists of two parts: a *prediction technique* and a *convergence technique.*

Related techniques allow to control the quality and response rate of the simulation [32]:

- level of detail: for representing the geometrical data in different resolutions.

- motion level of detail: for sending the state update messages to remote hosts in different rates, depending on their distance.

- synchronization: so that each host machine has similar states of the virtual world.

Network topology

Virtual Worlds support varying numbers of geographically distributed users and keep them up to date with changes in the world and support communication and interaction between them. A communication architecture can be chosen among different models, which can be arranged as communication graphs according to their degree of deployment (see Figure 3.1). In a communication graph, the nodes represent the processes running on remote computers and the links denote that the nodes can exchange messages.
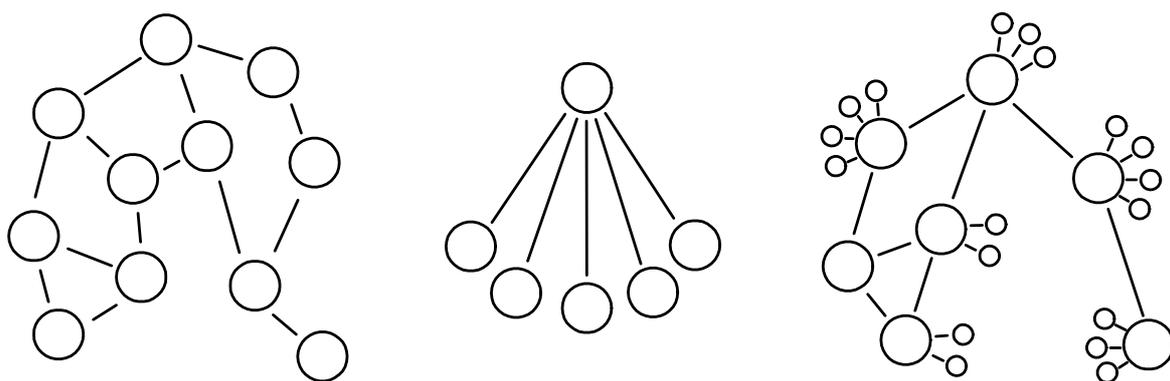
Figure 3.1: Three common network topologies: decentralized (peer-to-peer), centralized (client/server) and centralized+decentralized (server-network)

- In decentralized (or peer-to-peer) architecture we have a set of equal nodes connected by a network. There are three subcategories of this architecture according to the related transmission technique (see Figure 3.2):

– unicast: Each individual client program sends information directly to other client programs, as appropriate.

– multicast: Similar to unicast peer-to-peer, except the same information is sent simultaneously and directly to many other client programs.

– broadcast: Each node broadcasts its message to every node in the network.
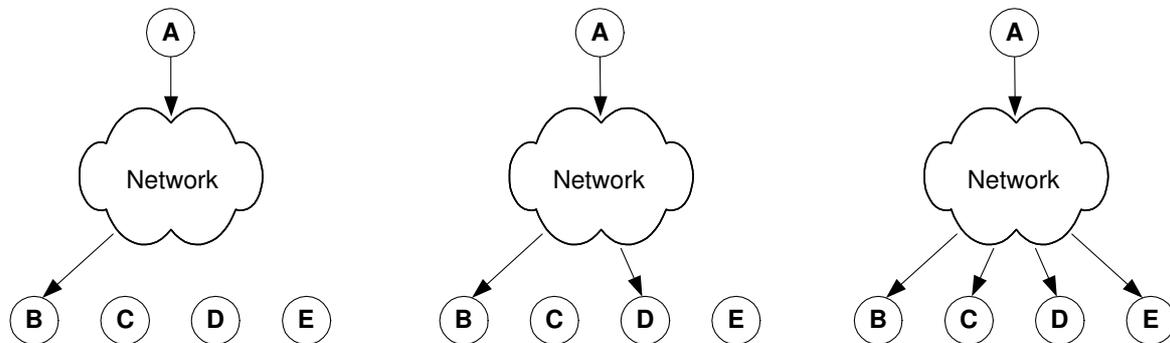
Figure 3.2: Examples of unicast (message is sent to a single receiver), multicast (message is sent to one or more receivers that have joined a multicast group) and broadcast (message is sent to all nodes in the network)

The decentralized approach does not scale up easily if the communication graph is dense or complete, i.e. if each node exchange messages with all other nodes.

• In centralized (or client/server) architecture, one node is promoted to the role of server. All communication is handled through this server node, that is responsible for passing messages on to other clients as appropriate. In this approach, the server becomes the critical part, the whole system depends completely on the central server robustness and scalability is not optimal.

• In server-network (or server pool) architecture, there are several interconnected servers. This is a hybrid combination of the two other network topologies. Here, the communication graph can be thought of as a peer-to-peer network of servers over a set of client/server subnetworks. A client is connected to a local server, which is connected to the remote servers and, through them, to the remote clients. By reducing the capacity requirements imposed on a server this approach provides better scalability but increases the complexity of handling the network traffic.

There are other possible topologies for distributed systems [136], like ring, hierarchical and the limitless possibilities in combining these various kinds of architectures. The interested reader will find a good evaluation of each of these system designs and a discussion of their relative merits in [137].

Data Distribution

The communication architecture also encompasses all the pertinent data. Determining where to put the data relevant to the state of the virtual world and its objects is a difficult decision. There are several conceivable ways of distributing persistent data. Three common data architectures (see Figure 3.3) are briefly described below.

- In **centralized** data architectures, one (data server) node stores all data. This is the model used by MUDs, where clients connect to a central server that does almost all the computation and maintains the state of the virtual world.

- In **distributed** data architectures, the data are distributed among the nodes. A spatial model can be used for data partitioning among the nodes.

- In **replicated** data architectures, each node manages a replica of all data. Traditionally, every node participating in the distributed environment is initialized with all static information. Communicated among all the users of the environment are relatively small messages describing the state changes.
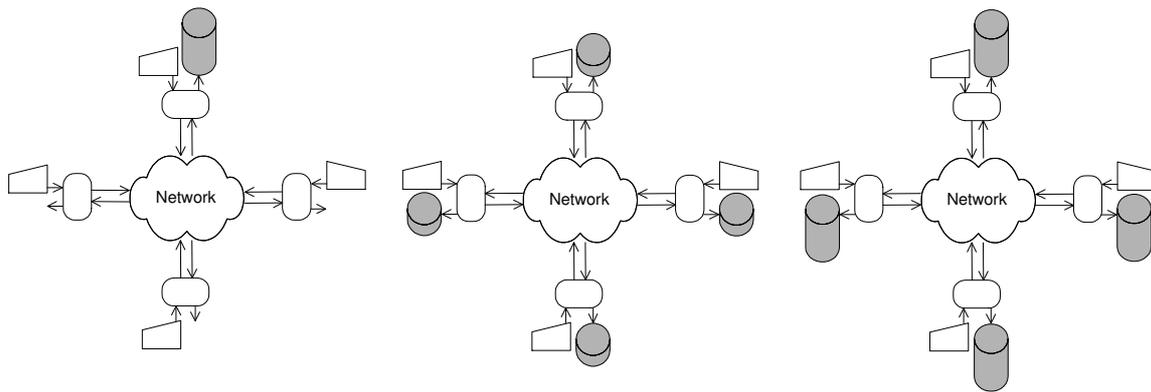
Figure 3.3: *Left to right:* centralized, distributed and replicated data architecture [176]

3.3 Resource Discovery

Resource discovery is the problem of knowing what things (objects, avatars, rooms) are in a world and how they relate to one another. In order to be meaningful, elements of a virtual world must be able to seamlessly link to each other. In the same sense, that the World Wide Web would be thoroughly useless without hyperlinks between the web pages.

In addition to that, awareness is a key aspect within a virtual world. This means that avatars must be informed in "real-time" of changes of its virtual environment. This includes the appearing and disappearing of elements. That is the reason why distributed virtual worlds should be built on top of infrastructures that provide some self-healing properties and automatic discovery services.

3.4 Robustness

For a virtual reality system to be robust, it must not contain a single point of failure. As already mentioned in Section 1.2, the document paradigm based World Wide Web is very instructive. If a single web server goes down, the effects are isolated to the immediate content hosted on that server and the rest of the Internet continues on its way. Thus the

whole system is extremely robust and, since the connection of new decentralized servers is possible with only a bit of connectivity, highly scalable.

The goal for a software architecture supporting virtual worlds, is to be as close as possible to the World Wide Web in terms of robustness and scalability.

3.5 Security

Security, privacy and trust are crucial elements in virtual world systems. One has to distinguish between two levels of security concerns: the system level and the virtual world level.

- At the system level, security concerns of distributed virtual world architectures are the same as in any other distributed computing system. This includes problems like passing through firewalls, encrypted communication protocols, permissions, etc. The security problem becomes even harder in a totally decentralized system with code that moves around and where downloaded proxy code must be trusted in order to protect the operating and file system of the host machines from potentially dangerous operations. The support of these features are all criteria that should strongly influence the choice of the appropriate language and technology for the development of virtual world applications.

- At the virtual world level, all the actors who interact need security facilities. This includes preventing an avatar snooping in on a conversation between two other avatars, protecting privacy and personal information of an avatar, verification and identification of other users, controlling access of parts of the world, managing permissions, etc. These issues can be addressed by using encrypted communications, public key cryptography, digital signatures and password protection.

3.6 Ease of Use

In order to be popular, virtual worlds need to be easy to use.

- To the **end user**, represented by her avatar, this means a seamless experience of moving inside the world, talking to other avatars and interacting with objects she may encounter. These objects may be executed, customized, transported or copied by the avatar.

- To **content creators**, this means the possibility to easily and rapidly create and arrange the elements of a world in an intuitive fashion. The created world part must be straightforwardly published on any machine with a bit of connectivity. Newly programmed objects should be effortlessly introduced into an existing virtual world.

- At the **application programmers** level, the key factor for success is to propose an open architecture, allowing a knowledgeable programmer to effortlessly understand and smoothly extend or improve the virtual world architecture building blocks, which should follow the principles of object-oriented design.

3.7 Persistence

Persistence is a feature that must be provided by a virtual environment. Either parts of the virtual world or avatars can be deactivated, its state stored to secondary storage (e.g., into a local file) and later reactivated. This must work both in the case of voluntary interruption and in the case of a software or hardware failure.

4

Virtual Worlds: A Conceptual View

Every great movement must experience three stages:
ridicule, discussion, adoption.
—John Stuart Mill

The goal of this chapter is to clearly define all the concepts involved in a virtual world in an implementation independent manner. To best achieve this goal, the key concepts are identified with the help of an informal scenario. The second section of the chapter attempts to define a formal abstract mathematical model for statically describing virtual worlds and anchors the ideas sketched in the first part. How to concretely instantiate this model is shown in the third section by means of two examples. Sections four and five complete the abstract model by including interaction and security considerations. However, these aspects are presented in an abstract way. Finally, the present chapter is concluded by a short summary, outlining the main characteristics, benefits and originalities of this abstract approach.

4.1 The Conceptual Components

The first part of this section is dedicated to a typical scenario, which should be possible in a virtual world. Building on this story, we then identify and extract the main concepts that are involved.

4.1.1 A Short Scenario

Suppose we have a virtual world a user wants to "live" in. As the user strolls through the world, she discovers it and its components, objects and other users like her. Suddenly, she finds an interesting object, a fibonacci number calculator. She can use it to compute some numbers of the famous series. Then, she goes on with her discovery of the world. Doing so, she comes to a place where she sees two other users playing a tic-tac-toe game and a little crowd watching them. She joins the observers and, after a while, she says to her neighbor: "Do you want to play this game with me?" As the other agrees, they go to another place where the user has placed a copy of the previous board game and they start to play. When they are finished she puts the tic-tac-toe object back in her bag and leaves the other user.

4.1.2 The Key Concepts

We consider that the main participants of our model emerging from this simple scenario are:

- **Avatar**: The human user needs a representation in the virtual world and is therefore personified by an avatar. Through her avatar the user can walk, "fly", look around the virtual world, manipulate objects and perform virtual actions. In other words, avatars allow users to interact with the virtual world and other users and also allow navigation through the world. We see one's avatar as being her representative in Cyberspace. In text-based virtual realities, such as MUDs, one's avatar consists of a short description which is displayed to other users who have their avatars 'look' at her. In a 3D graphical world, the avatar can take the shape of an animated cartoon or of her favorite fantasy hero.

- **Object**: In the discussed virtual world, there are objects, not just simple passive data objects, but active objects the avatar can execute (e.g., the fibonacci calculator). These objects can even be multi-user (e.g., the two players tic-tac-toe board) and can have many observers like in our little story. Last, but not least, objects can be copied and/or moved by the avatar.

- **Location**: Avatars and objects have a location. This concept is natural and necessary, since it supports navigation.

- **Navigation**: The action of going from a given location to another. Avatars can do that, either through a **link** between these two locations, or directly if they know the **address** of the subspace.

- **Subspaces**: Each location can be seen as a subspace of the whole virtual world. It is natural to consider that the shared virtual space is composed of many different subspaces. Furthermore, the avatars and objects are always **contained** within one given subspace.

  In order to achieve interaction between these participants, the concepts of *event* and *event propagation* as well as the two roles *event producer* and *event consumer* need to be introduced.

- **Event**: The avatar has to be aware of its environment. This awareness is achieved by the concept of events. An avatar moving, a played move in the tic-tac-toe game are simple examples of such events.

- **Event producer**: An event always has a source which produces it. For instance, the game could produce a "game finished event".

- **Event consumer**: An event can be caught and interpreted by an event consumer, which then reacts properly or simply ignores it. For instance, the player of the game understands that the game is finished.

- **Event propagation**: Each event has an event propagation space. This space is a delimited zone around an event producer, within which an event consumer will be aware that the event has occurred.

The following sentence, simply resumes the last four concepts. "An event is consumed by each interested event consumer in the event propagation space of the event producer."

4.2 Formalization

All the concepts identified in Subsection 4.1.2 need to be integrated in our theoretical model. Before formalizing them in the most general way, however, a brief "non-mathematical" summary should help the reader:

1. At the core of the model, there is the whole virtual space $\mathcal{V}$. $\mathcal{V}$ has a global metric space associated to it, which allows subspaces to have a location and a volume.

2. The subspaces are inhabited by avatars, objects and link extremities. The latter have a location and a volume relative to the local metric space associated with their container.

3. Further axioms and rules achieve the description of the relations between all these concepts.

4.2.1 The Global Virtual Space

- Let $\mathcal{V} := (\mathcal{V}_V, \mathcal{V}_A, \mathcal{V}_N)$ be the whole virtual world. It represents the entire shared virtual space. It is seen as a *directed graph*[1] consisting of a set of vertices $\mathcal{V}_V$ and a set of arcs $\mathcal{V}_A$. The vertices are also called *nodes* or *points*; the arcs could be called *directed edges*. An arc is an ordered pair of vertices and is expressed by:

$$v_i \rightarrow v_k \text{ with } v_i, v_k \in \mathcal{V}_V \tag{4.1}$$

- For a given arc $v_i \rightarrow v_k$, v_i is called the *head* and v_k is called the *tail* of the arc. They are respectively defined by the following functions:

$$h(v_i \rightarrow v_k) := v_i \text{ and } t(v_i \rightarrow v_k) := v_k \tag{4.2}$$

- $\mathcal{V}_V$ is the set of all the *subspaces* v_i composing the world $\mathcal{V}$:

$$\mathcal{V}_V := \bigcup_{i \in I} \{v_i\} \text{ where } I \text{ is a set of indices} \tag{4.3}$$

- $\mathcal{V}_A$ is the set of all the *links* between the subspaces v_i of $\mathcal{V}_V$. It is of the form:

$$\mathcal{V}_A := \{v_i \rightarrow v_k : v_i, v_k \in \mathcal{V}_V\} \tag{4.4}$$

- $\mathcal{V}_N$ is the *namespace* used for the addresses of the rooms.

- Each subspace has an address given by the function:

$$\lambda : \mathcal{V}_V \longrightarrow \mathcal{V}_N \tag{4.5}$$

- Each subspace $v_i \in \mathcal{V}_V$ has a *location* in $\mathcal{V}$ given by the function:

$$\overline{\pi} : \mathcal{V}_V \longrightarrow (\overline{\Sigma}, d_{\mathcal{V}}) \tag{4.6}$$

- Each subspace $v_i \in \mathcal{V}_V$ has a *volume* in $\mathcal{V}$ given by the function:

$$\overline{\rho} : \mathcal{V}_V \longrightarrow \mathcal{P}((\overline{\Sigma}, d_{\mathcal{V}})) \tag{4.7}$$

Where $\mathcal{P}(\overline{\Sigma})$ is the powerset (set of all subsets) of $\overline{\Sigma}$.

---

[1]Directed graphs are well introduced in [4] and further discussed in [39].

- $(\overline{\Sigma}, d_{\mathcal{V}})$ is a metric space[2]. So there must be a metric $d_{\mathcal{V}}$ defined on the location space $\overline{\Sigma}$ of $\mathcal{V}$.

$$d_{\mathcal{V}} : \overline{\Sigma} \times \overline{\Sigma} \longrightarrow \mathbb{R} \qquad (4.8)$$

- Two subspaces v_k and v_l are *adjacent* if the following property is respected:

$$\overline{\rho}(v_k) \cap \overline{\rho}(v_l) \neq \emptyset \text{ and } \overline{\rho}(v_k) \neq \overline{\rho}(v_l), \quad v_k, v_l \in \mathcal{V} \text{ and } k \neq l \qquad (4.9)$$

4.2.2 Avatars, Objects and Transport Points

- Each subspace $v_i \in \mathcal{V}_V$ is a set of avatars, objects and transport points.

- Let A be the set of all *avatars* α_i living in the subspaces of $\mathcal{V}_V$:

$$A := \{\alpha_1, \alpha_2, ..., \alpha_n\} \qquad (4.10)$$

- Let Ω be the set of all *objects* ω_i populating the subspaces of $\mathcal{V}_V$:

$$\Omega := \{\omega_1, \omega_2, ..., \omega_m\} \qquad (4.11)$$

- Let us associate to each link $x \in \mathcal{V}_A$ the notion of a *leaving point* and an *entry point*:

$$\begin{aligned} x_{e\rightarrow} \text{ is the leaving point of } x, \text{ with } x_{e\rightarrow} \in h(x) \\ x_{e\leftarrow} \text{ is the entry point of } x, \text{ with } x_{e\leftarrow} \in t(x) \end{aligned} \qquad (4.12)$$

- Let the set of all leaving and entry points be regrouped in the set of *transport points* Δ defined by:

$$\Delta := \bigcup_{x \in \mathcal{V}_A} \{x_{e\rightarrow}, x_{e\leftarrow}\} := \{\delta_1, ..., \delta_k\} \qquad (4.13)$$

The cardinality, i.e. the number of elements, of Δ, noted $\#\Delta$, is given by the following relation: $\#\Delta = 2 \cdot \#\mathcal{V}_A$

- Let Θ be the set of all entities 'living' in the virtual world, including avatars, objects and transport points:

$$\begin{aligned} \Theta := A \cup \Omega \cup \Delta &= \{\alpha_1, ..., \alpha_n, \omega_1, ..., \omega_m, \delta_1, ..., \delta_{\#\Delta}\} \\ &=: \{\theta_1, \theta_2, ..., \theta_{n+m+\#\Delta}\} \end{aligned} \qquad (4.14)$$

- The avatars, objects and transport points must be located in *exactly one subspace* of the world. This means, that each element of Θ has a location, i.e. its container in $\mathcal{V}$ given by the function π:

$$\pi : \Theta \longrightarrow \mathcal{V}_V, \text{ so that } \theta \in \pi(\theta) \in \mathcal{V}_V \qquad (4.15)$$

In other words:

$$\theta \in v_i \Leftrightarrow \pi(\theta) = v_i \text{ with } \theta \in \Theta \text{ and } v_i \in \mathcal{V}_V \qquad (4.16)$$

---

[2]For a general introduction about metric spaces see [59] or [163].

- Naturally the following statement is always verified:

$$\forall \theta_j \in \Theta \ \exists! v_i \in \mathcal{V}_V \text{ with } \pi(\theta_j) = v_i$$
$$1 \leqslant j \leqslant n + m + \#\Delta \text{ and } i \in I \tag{4.17}$$

- Furthermore, the following property, which is the corollary of (4.12), is also respected:

$$\forall x \in \mathcal{V}_A \quad \pi(x_{e\rightarrow}) = h(x)$$
$$\text{and } \pi(x_{e\leftarrow}) = t(x) \tag{4.18}$$

4.2.3 The Local Subspaces

- Let A_{v_i} be the set of all avatars living in a given subspace v_i of $\mathcal{V}_V$:

$$A_{v_i} := \{x \in A : \pi(x) = v_i\} \tag{4.19}$$

- Let Ω_{v_i} be the set of all objects populating a given subspace v_i of $\mathcal{V}_V$:

$$\Omega_{v_i} := \{x \in \Omega : \pi(x) = v_i\} \tag{4.20}$$

- Let Δ_{v_i} be the set of all transport points located in a given subspace v_i of $\mathcal{V}_V$:

$$\Delta_{v_i} := \{x \in \Delta : \pi(x) = v_i\} \tag{4.21}$$

- Let Θ_{v_i} be the set of all entities located in v_i:

$$\Theta_{v_i} := A_{v_i} \cup \Omega_{v_i} \cup \Delta_{v_i} := \{x \in \Theta : \pi(x) = v_i\} \tag{4.22}$$

- Each element of Θ_{v_i} has a *location relative* to its container v_i, given by the function:

$$\pi_{v_i} : \Theta_{v_i} \longrightarrow (\Sigma_{v_i}, d_{v_i}) \tag{4.23}$$

- Each element of Θ_{v_i} has a *volume relative* to its container v_i, given by the function:

$$\rho_{v_i} : \Theta_{v_i} \longrightarrow \mathcal{P}((\Sigma_{v_i}, d_{v_i})) \tag{4.24}$$

- (Σ_i, d_{v_i}) are metric spaces. So there must be a metric d_{v_i} defined on each location space Σ_{v_i} of v_i.

$$d_{v_i} : \Sigma_{v_i} \times \Sigma_{v_i} \longrightarrow \mathbb{R} \tag{4.25}$$

- The elements of $\mathcal{V}_V$, i.e the subspaces, as well as those of $\Omega \cup A$, i.e. the objects and avatars, can fire events. Each event source s has an event propagation space defined by the function p:

  − If $s \in \Omega \cup A$

$$p(s, k) := \{x \in \varsigma := \pi(s) : d_\varsigma(\pi_\varsigma(s), \pi_\varsigma(x)) \leqslant k\}$$
$$\text{with } k > 0 \in \mathbb{R} \text{ constant} \tag{4.26}$$

Note that an event cannot be propagated outside the subspace containing the event source.

– If $s \in \mathcal{V}_V$

$$p(s) := s \tag{4.27}$$

In other words, if the event source is a subspace the event propagation space is the subspace itself.

In Section 4.4 the events are further explained.

4.2.4 Remarks About Time

- The virtual world $\mathcal{V}$ is in constant evolution with time. Further, we consider that time is discrete. So all sets and functions defined above depend actually on a parameter t defining the instant t. This parameter has been omitted for sake of simplicity but it would indeed be more accurate to speak about a virtual world $\mathcal{V}_t$ to designate the state of the world at time t. Then one would also have to write $\mathcal{V}_{Vt}$, $\mathcal{V}_{At}$, I_t, A_t, Ω_t, Δ_t, Θ_t, π_t, A_{v_it}, Ω_{v_it}, Δ_{v_it}, Θ_{v_it}, π_{v_it}, ρ_{v_it} and $p_t(s, k)$.

- Each event corresponds to a transition between two states of the world $\mathcal{V}_t \longrightarrow \mathcal{V}_{t+1}$. An event is considered either at the local level, or at the global level or both.

- Each event occurs at a given instant t, so there are no simultaneous events in the model.

- In order to consider events at the global level, a *global clock* T is needed.

- The local level events can either use the global clock if there is one or define their own *local clock* τ.

- The virtual world clocks T and τ are actually simple counters incrementing themselves when the state of the world changes. For convenience reasons, let us further define two reference clocks (giving the time in the real world for example) $\widetilde{T}$ and $\widetilde{\tau}$ in the usual sense of the world. Of course, two events corresponding to two instants t_1 and t_2 can occur at the same time $\widetilde{t}$.

- A non exhaustive list of possible *global level* transitions (i.e. events) would be:

  – the creation of a new subspace
  – the destruction of a subspace
  – an avatar who moved from one subspace into another
  – a new avatar who joins the world (logs in)
  – an avatar who leaves the world (disconnects)
  – the creation of a link between two subspaces
  – the destruction of a link between two subspaces

A non exhaustive list of possible *local level* transitions would be:

  – an avatar who arrives in the subspace

– an avatar who leaves the subspace

– an object added to the subspace

– an object removed from the subspace

– an internal state change of an object in the subspace

– the creation of a link between the subspace and another

– the destruction of a link between the subspace and another

- An in-depth look into the event model is provided in Section 4.4.

4.2.5 General Considerations

- Within a running model, the virtual world $\mathcal{V}$ may avoid overlapping and collision of its subspaces v_i. In this case the following invariant is always verified:

$$\overline{\rho}(v_k) \cap \overline{\rho}(v_l) = \emptyset, \quad v_k, v_l \in \mathcal{V} \text{ and } \forall k \neq l \tag{4.28}$$

- Each subspace v_i has its own location function π_{v_i}, volume function ρ_{v_i} and system of coordinates.

- Within a running model, a subspace v_i may avoid overlapping or collision of the objects θ_k it contains. In this case the following invariant is always verified:

$$\rho_{v_i}(\theta_k) \cap \rho_{v_i}(\theta_l) = \emptyset, \quad \theta_k, \theta_l \in \Theta_{v_i} \text{ and } \forall k \neq l \tag{4.29}$$

- Within a running model, only symmetric links can be allowed. The following implication should then be always verified:

$$v_i \rightarrow v_k \in \mathcal{V}_A \Longrightarrow v_k \rightarrow v_i \in \mathcal{V}_A \tag{4.30}$$

In this case, let us define the notion of symmetric arc and note $v_i \leftrightarrow v_k \in \mathcal{V}_A$.

- If a running model satisfies (4.30), then it can also always guarantee:

$$\pi_{v_i}((v_i \rightarrow v_k)_{e\rightarrow}) = \pi_{v_i}((v_k \rightarrow v_i)_{e\leftarrow}) \tag{4.31}$$

- One can define a measure[3] m on the algebra $\mathcal{P}(\overline{\Sigma})$ in order to have the measure space $(\overline{\Sigma}, \mathcal{P}(\overline{\Sigma}), m)$. Thus, the definition of two adjacent spaces (4.9) can be defined in a more intuitive way by requiring the following additional property to be verified:

$$m(\overline{\rho}(v_k) \cap \overline{\rho}(v_l)) = 0 \tag{4.32}$$

This means that an intersection exists but it is negligible, i.e. its measure is null.

---

[3]For a good introduction in measure theory see [88].

4.2.6 Final Definition

To summarize, one can define a **virtual world system** VW as a 5-tuple:

$$VW = (\mathcal{V}, \Theta, \overline{\rho}, \overline{\pi}, (\overline{\Sigma}, d_{\mathcal{V}})) \tag{4.33}$$

And each subspace v_i, element of $\mathcal{V}_V$, can be defined as a 4-tuple:

$$v_i = (\Theta_{v_i}, \rho_{v_i}, \pi_{v_i}, (\Sigma_{v_i}, d_{v_i})) \tag{4.34}$$

For sake of simplicity, we consider the whole model at a fixed time t and omit the time parameter.

4.3 Model Instantiation

In this section, we show two concrete instantiations of the abstract model. To achieve this goal, we have to define the functions $\overline{\pi}, \pi_{v_i}, \overline{\rho}, \rho_{v_i}$, the location spaces $\overline{\Sigma}$, Σ_{v_i} and their respective metrics $d_{\mathcal{V}}$, d_{v_i} (see Figure 4.1). The parameter k of formula (4.26) has to be fixed as well.
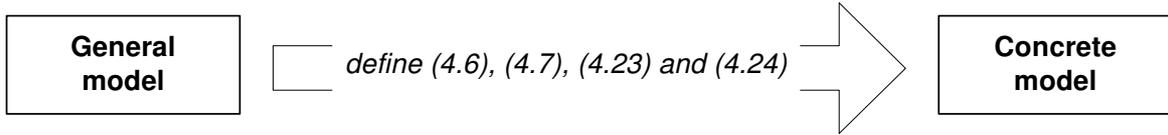
Figure 4.1: The model instantiation mechanism

4.3.1 The "Natural" Instantiation of the Model

The "natural" way to instantiate the model, consists of considering an arbitrarily large number of subspaces located in the three dimensional space.

$$\mathcal{V}_V := \bigcup_{i \in I} \{v_i\} \subseteq \mathbb{R}^3$$

Here we define $\overline{\Sigma} = \mathbb{R}^3$, and the function $\overline{\pi}$ gives the position of the subspaces v_i in the natural coordinates (e.g., the 'lower' corner of the bounding box). The function $\overline{\rho}$ associates each subspace v_i to the 3D volume it occupies in the space. These functions are illustrated on Figure 4.2.

The functions

$$\pi_{v_i} : \Theta_{v_i} \longrightarrow \Sigma_{v_i} := (\mathbb{R}^3, d) \ \forall i \in I$$

associate each element of Θ_{v_i} to its position in natural coordinates (e.g., the 'lower' corner of the bounding box). The functions ρ_{v_i} associate each element of Θ_{v_i} to a 3D volume it occupies in the subspace. We then define the traditional euclidian metric d on all location

spaces $\Sigma_{v_i} = \overline{\Sigma} = \mathbb{R}^3$:

$$d_{\mathcal{V}}(x,y) = d_{v_i}(x,y) = d(x,y) = \sqrt{\sum_{j=1}^{3}(x_j - y_j)^2}$$

$$\forall i \in I, \forall x := (x_1, x_2, x_3), y := (y_1, y_2, y_3) \in \mathbb{R}^3$$

The event propagation space is defined by the function:

$$p(s,k) := \{x \in \pi(s) : d(\pi_{\pi(s)}(x), \pi_{\pi(s)}(s)) \leqslant k\}$$
$$\text{with } k > 0 \in \mathbb{R} \text{ constant}$$
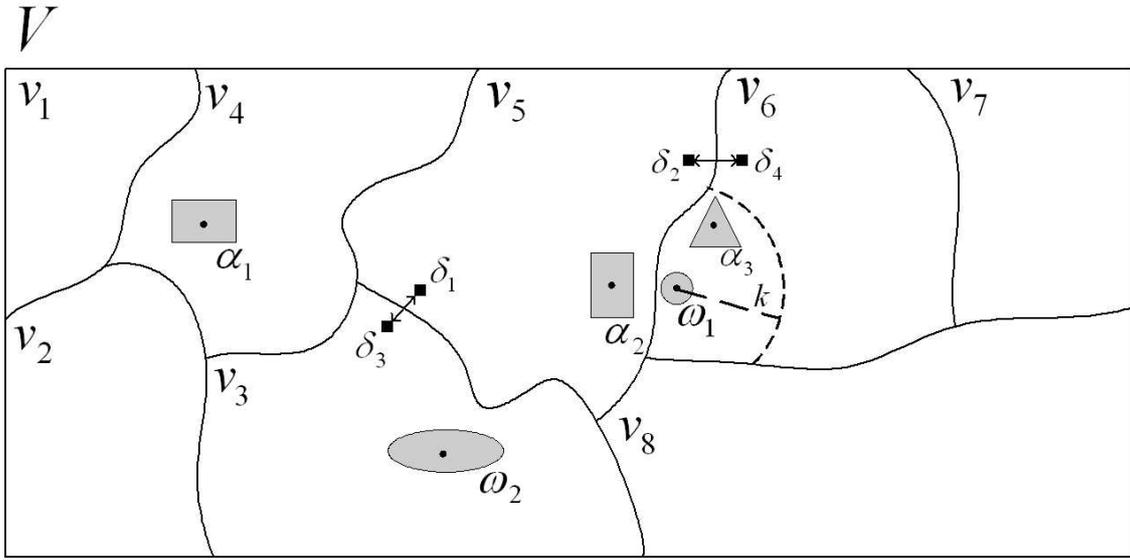
Figure 4.2: The "natural" instantiation of the model

Figure 4.2 illustrates a concrete example. On this figure one can see the following relations:

- Eight subspaces form the virtual world:

$$\mathcal{V}_V := \bigcup_{i=1}^{8}\{v_i\} \subseteq \mathbb{R}^3$$

- There are two links between subspaces. One between v_3 and v_5 and one between v_5 and v_6:

$$\mathcal{V}_A = \{v_3 \leftrightarrow v_5, v_5 \leftrightarrow v_6\}$$

- The object ω_1 and the avatar α_3 are located in the same subspace v_6.

$$\pi(\omega_1) = \pi(\alpha_3) = v_6$$

The observations below can be made:

- $p(\omega_1)$ is delimited by the dotted line.

- $\rho(\alpha_1)$ is represented by the shadowed zone around $\pi_{\pi(\alpha_1)}(\alpha_1) = \pi_{v_4}(\alpha_1)$, which is represented by a black dot.

- The entry points are represented by a black square ($\delta_1 := \pi_{v_5}((v_3 \rightarrow v_5)_{e\leftarrow})$).

- The world shown in this example respects both properties (4.28) and (4.29).

- Furthermore, properties (4.30) and (4.31) are also respected.

- Considering the subspaces v_i as closed subsets of $\mathbb{R}^3$ (see Figure 4.2) the world of this example has only links between two adjacent (4.9) subspaces. Thus these links support the metaphor of doors.

4.3.2 The MaDViWorld Instantiation of the Model

The implemented MaDViWorld framework supports a more pragmatic instantiation of the model. Indeed, one considers only a finite number of subspaces, which have no "real" location.

$$\mathcal{V}_V = \bigcup_{i=1}^{l} \{v_i\}$$

We define $\overline{\Sigma} := \{1\}$, and the function $\overline{\pi}$ gives the unique and same position for all subspaces v_i, i.e. 1. We define the trivial metric $d_\mathcal{V}$ on the location space $\overline{\Sigma} := \{1\}$:

$$d_\mathcal{V} : \{1\} \times \{1\} \longrightarrow \mathbb{R}, d(1,1) = 0$$

The function $\overline{\rho} \equiv \overline{\pi}$. This means that the different subspaces composing the virtual world have no position and occupy no space.

We define the same degenerated location functions π_{v_i} for all subspaces $v_i, 1 \leqslant i \leqslant l$:

$$\pi_{v_i} : \Theta_{v_i} \longrightarrow (\Sigma_{v_i}, d) \text{ where } \Sigma_{v_i} := \{1, 2, 3\}$$

$$\pi_{v_i}(x) := \begin{cases} 1 & \text{if } x \in A_{v_i} \ (x \text{ is an avatar}) \\ 2 & \text{if } x \in \Omega_{v_i} \ (x \text{ is an object}) \\ 3 & \text{if } x \in \Delta_{v_i} \ (x \text{ is a transport point}) \end{cases}$$

We define the same simple metric d on all location spaces $\Sigma_{v_i} = \{1, 2, 3\}, \forall i \quad 1 \leqslant i \leqslant l$:

$$d_{v_i} = d : \{1, 2, 3\} \times \{1, 2, 3\} \longrightarrow \mathbb{R},$$

$$d(x, y) := \begin{cases} 0 & \text{if } x = y \\ 1 & \text{if } x \neq y \end{cases}, \quad \forall i, 1 \leqslant i \leqslant l$$

The function $\rho_{v_i} \equiv \pi_{v_i}$. This means that the elements in the subspace have no "real" position and no volume. We can just say that there are three different lists, one of transport points, one of objects and one of avatars.

The event propagation space is defined by the function:

$$p(s) := \pi(s)$$

This means that an event fired by an object s can be consumed by any other object in the same subspace.
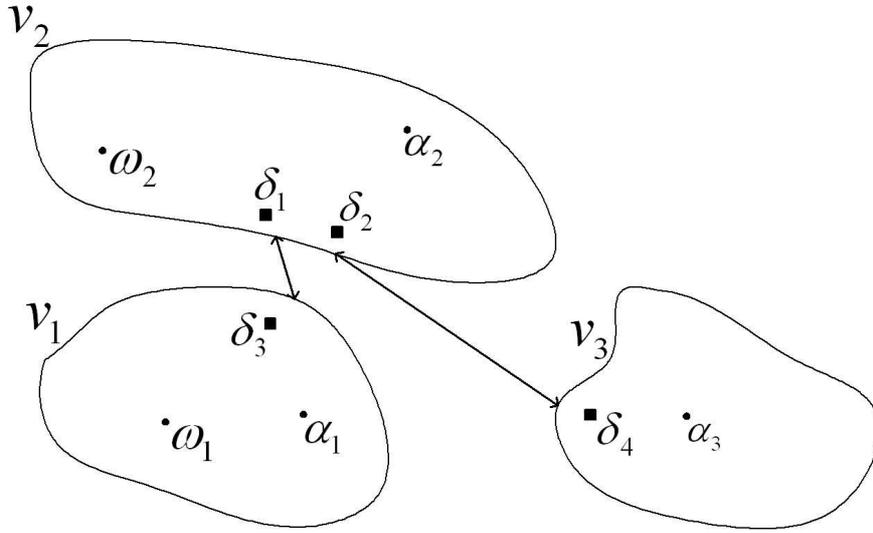
Figure 4.3: The MaDViWorld instantiation of the model

Figure 4.3 sketches an example of a concrete world within this model. On this figure one can see the following relations:

- Three subspaces compose the virtual world:

$$\mathcal{V}_V = \bigcup_{i=1}^{3} \{v_i\}$$

- There are two links between subspaces. One between v_2 and v_1 and one between v_2 and v_3.

$$\mathcal{V}_A = \{v_2 \leftrightarrow v_1, v_2 \leftrightarrow v_3\}$$

- The object ω_1 and the avatar α_1 are located in the same subspace v_1.

$$\pi(\omega_1) = \pi(\alpha_1) = v_1$$

- The event propagation space of the object ω_1 is the whole subspace it is contained in, v_1.

$$p(\omega_1) = \pi(\omega_1) = v_1$$

- The volume of ω_1 in v_1 is the same as its position, i.e. a single point.

$$\rho_{v_1}(\omega_1) = \pi_{v_1}(\omega_1)$$

The two observations below can be made:

- Properties (4.30) and (4.31) are respected and links support the metaphor of "teleportation gates".

- As the location function in this model is partially degenerated (all avatars share the same location, all objects share the same location,...), it is clear that neither property (4.28) nor (4.29) are respected.

4.4 Events and Interaction

This section describes the event concept and shows how the event traffic is achieved inside a virtual world. We already hinted these concepts in [66] but it is worth clarifying it by formalizing and giving some definitions.

4.4.1 Global View

The four main concepts supporting interaction between the components of a virtual world are explained below:

- The *event source* or *event producer* is the object that fires an event. It can be an avatar, an object or a subspace.

- The *event listener* works on behalf of a given event consumer. Its role is to "catch" events that occur in the event consumer's environment.

- The *event consumer* is the object that is interested in events and that will consume them. It reacts to received events. If an object c is interested in events fired by a given source object s, it has to register an event listener l to s. This is only possible if c is in the event propagation space of s^4. Note that an event consumer can have several event listeners working for it.

- The *event* is an object describing an action or a state change.

This is a known technique of enabling entities to register interest in a particular resource. When the resource changes state, all listeners are automatically notified. This obviates the need to periodically poll the resource to see if anything has changed.

4.4.2 Formalization

Let us write l^c for a listener l working on behalf of the event consumer c.

For a given event source s, we define the set of associated registered event listeners s_l as follows:
$$s_l = \{l_1^{c_1}, l_2^{c_1}, ..., l_n^{c_m}\} \text{ where } c_i \in p(s) \quad \forall i, 1 \leqslant i \leqslant m$$

An event $\varepsilon_{s,\mathcal{R}}^{i,t}$ has several attributes:

- A sequence number i (i^{th} event produced by the source s).

- A timestamp t giving the time at which the event occurred (either relative to a global clock T or a local clock τ).

---

[4]The *event propagation space* is analog to the notion of *aura* introduced on page 46.

- A set of recipients $\mathcal{R} \subseteq s_l$.

- A source s.

- A type $\tau(\varepsilon)$ which can be a (avatar), s (subspace) or o (object).

As expected, avatars can fire events of type a, objects can fire events of type o and subspaces can fire events of type s. Furthermore, subspaces can forward received events of type a or o.

The listeners can be registered to an event producer by the event consumers according to one of the following policies γ:

1. γ_{all} : "Inform me of all events."

2. γ_a : "Inform me only of events of type a."

3. γ_o : "Inform me only of events of type o."

4. γ_s : "Inform me only of events of type s."

5. γ_{me} : "Inform me only of events explicitly addressed to me."

Therefore the set s_l can be defined through the following non disjoint union:

$$s_l = s_l^{\gamma_{\text{all}}} \cup s_l^{\gamma_{\text{me}}} \cup s_l^{\gamma_a} \cup s_l^{\gamma_o} \cup s_l^{\gamma_s}$$

where s_l^{γ} is the subset of listeners registered with the policy γ.

4.4.3 Benefits

Figure 4.4 shows an event source firing events to its registered event listeners. This model allows for the optimization of the event traffic between the different objects of the virtual world. Indeed, the event source does not fire events to listeners, which do not care about these events. Reciprocally, the event listeners do not have to check if the events they received are relevant to them or not.

4.5 Security

The goal of this section is to explain how the security issues inside the world are addressed in our abstract model.

4.5.1 Main Concept

When talking about security, one has to determine under what circumstances some interactions can take place or not. Inside our virtual world, the following questions relative to security may arise:

1. Who can access a given subspace?
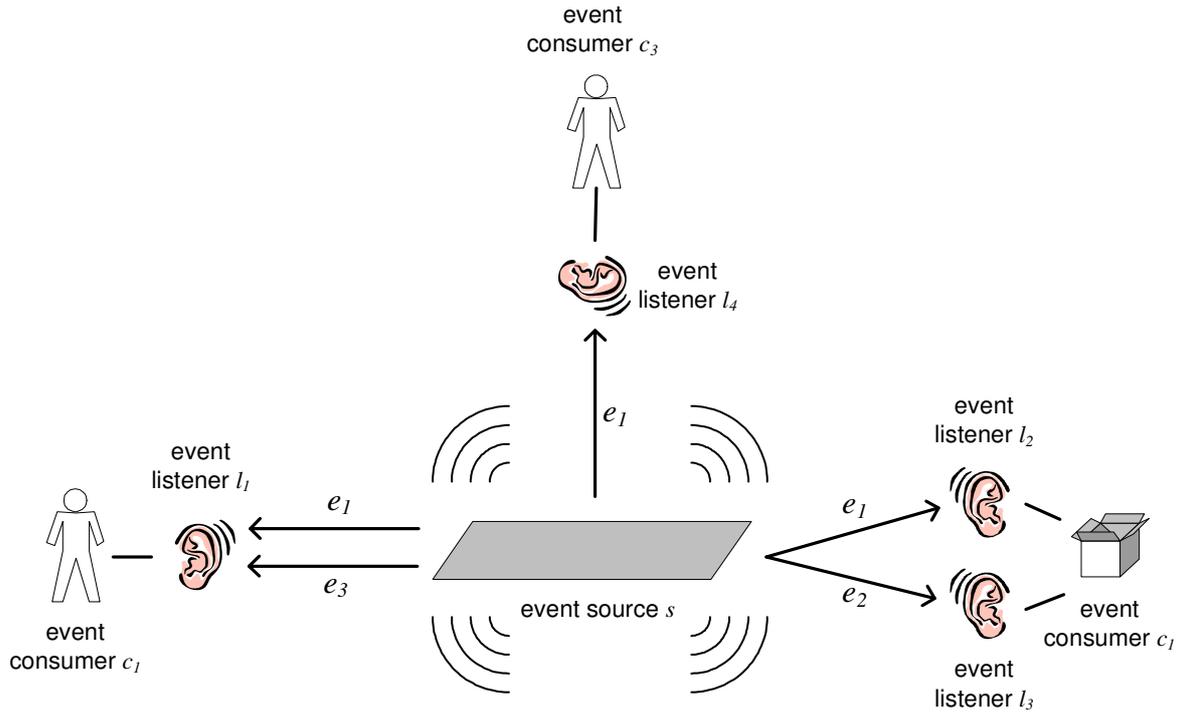
Figure 4.4: An event source with its listeners and consumers

2. Who can use a given object in a subspace?

3. Who can add an object to a subspace?

4. Who can copy a given object from a subspace?

5. Who can add a link from one subspace to another?

6. Who can remove an object from a subspace?

One should notice that the interactions involved in the questions above all concern the subspaces and another entity. This second actor is normally an avatar, but it could also be an active object directly interacting with the subspace it is located in. Transport points however are purely passive and thus are not involved in any security problems.

The basic principle is that *the subspace grants access rights* or privileges *to the avatars and objects*.

4.5.2 Formalization

In order to precise and formalize the concepts of the preceding subsection, let us consider a given subspace v_i.

- The set of all avatars living in this subspace is noted A_{v_i}.

- The set of all objects populating it is noted Ω_{v_i}.

- Let us further define $I_{v_i} := A_{v_i} \cup \Omega_{v_i}$ as the set of all the entities which interact with v_i.

The subspace v_i has four access right levels and grants them to the elements of I_{v_i}, classifying them among the four following groups[5]:

- Guest group G_{v_i}. Guests are prevented from making changes to the subspace.

- User group U_{v_i}. Users have the most usual access privileges, including those of guests.

- Administrator group A_{v_i}. Administrators have complete and unrestricted access to the subspace.

- Agent group O_{v_i}. Members of this group are typically active mobile objects and have some administrative rights.

There is also a fifth possibility: elements of I_{v_i} can belong to the "access denied" group. In this case, the concerned avatar or object has no rights at all to interact with the given subspace.

4.5.3 Benefits

This security model is very flexible, since each subspace manages its own security policy independently.

In conclusion, let us answer the six identified security questions. The answer for the two first questions is: members of the guest, user or administrator group. The answer of the question 3 and 4 is: users and administrators. Finally, the answer of the last two questions is: only administrators.

4.6 Main Concepts Summarized

Before concretely implementing a virtual world and delving into more technical implementation aspects it is worth defining a common terminology and to clarify some main concepts. The theoretical foundation provided by this chapter is general enough to allow for many concrete implementation architectures, ranging from traditionally centralized to completely distributed solutions. The software framework developed in the context of this thesis is one of these possible solutions and its design is extensively presented in the following chapters.

Even if the model presented in this chapter aims to be as general as possible, depending on the implementation choices and special aspects one wants to focus on, some of its parts should however be adapted. For instance, if three dimensional graphic presentation with continuous avatar motion and navigation has to be supported, the event mechanism will certainly have to be refined and some techniques used in the related projects presented in Chapter 2 can then be applied.

The main particularity of the abstract model is to define the virtual world as a federation of subspaces, forming a dynamic set of nodes within a directed graph. Thus subspaces

---

[5]The separation in user groups was inspired by the security mechanism of file systems defined by the NFS (Network File System) protocol designed by Sun Microsystems and which is often associated with UNIX systems.
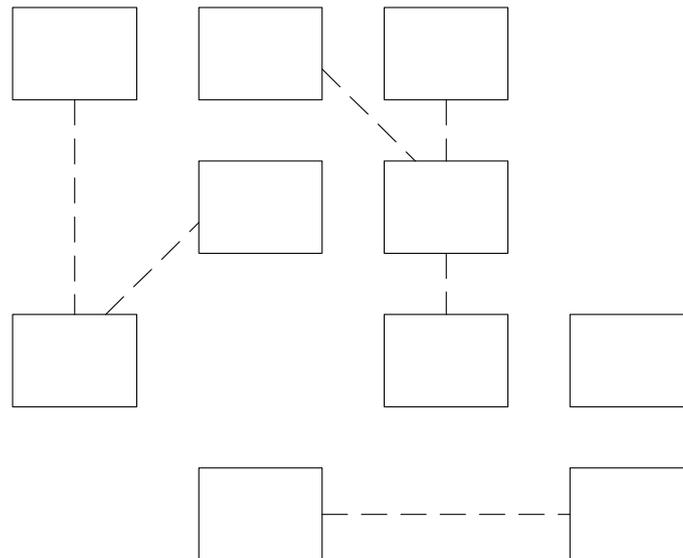
Figure 4.5: Loosely coupled MaDViWorld rooms

are conceptual atomic units of decomposition joined together in a very flexible way in order to form a complete world (see Figure 4.5).

On the other hand, the approach adopted by the big majority of the projects presented in Chapter 2 is to start from a conceptually monolithic global virtual world, and only then to partition it into subspaces (commonly into a spatial subdivision of cells), in order to distribute the world and to address scalability issues (see Figure 4.6). According to [31] four spatial partitioning schemes are possible: *(i) separate servers* (that separate the world into independent worlds), *(ii) uniform geometrical structure* (that divides the world uniformly), *(iii) free geometrical structure* (that divide the world based on users choices), *(iv) user-centered dynamic structure* (that divide the world based on interactions between users).

Another capital point, which cannot be explicitly seen in the mathematical description of the model, but which is clearly stated in Subsections 4.1.2 and 4.1.1, is the mobility of the objects. It is important to note that the objects cannot only move inside a given subspace, but also from one subspace to another; usually carried by an avatar, but autonomously as well.
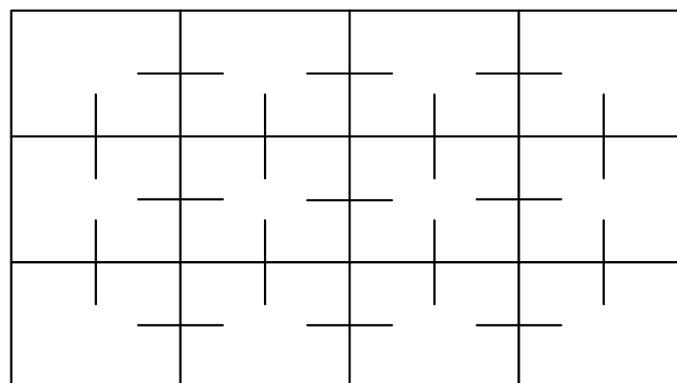
Figure 4.6: Partition of a virtual world in tightly coupled parts

5

The MaDViWorld Framework:
A First Approach

Simplicity does not precede complexity, but follows it.
—Alan J. Perlis

This chapter really marks the beginning of the documentation of the MaDViWorld framework. While the first chapters of the dissertation described the purpose of the framework, presented an example scenario and clarified the concepts, the present one is essentially devoted to explain how to use the framework, but without delving into too many details. The guidelines for documenting a framework suggested in Subsection 2.4.7 are thus clearly respected. In order to best achieve this goal, the first section makes some necessary reflections prior to implementation. Section two then provides a bird's eye view of the framework design and shows how it was deduced from the theoretical model. Finally, the last section contains a cookbook providing detailed instructions for using the framework.

5.1 Preliminary Implementation Considerations

Starting from the theoretical background provided by the previous chapters, we want to develop a distributed software solution supporting the abstract and general model presented in Section 4.2. In order to achieve this goal, several preliminary decisions have to been taken to instantiate the model properly and to adopt an appropriate deployment strategy.

5.1.1 Technology

It is now time to choose the best adapted technologies. These include an appropriate software architecture and a programming language.

First of all, since one of our main concerns is to populate the world with an ever growing set of active objects, the **object-oriented technology** seems to be the natural way to face our implementation problems. In addition to the objects, two other major actors have been identified in the previous chapters: the avatars and the rooms. In order to keep the consistency of the world, two roles related to the events are associated to these three major actors: event producers and event consumers. The services that these five components should provide have to be defined in an abstract communication protocol that is as close as possible to the theoretical model of Section 4.2. Object-oriented technology also fits well here, since one can define a set of interfaces or abstract classes, that can be implemented or specialized in a further stage through an inheritance "is-a" relation. This already hints a layered **software framework** approach[1], leading from abstract to always more concrete classes. Moreover, this results in an open and extensible architecture.

Java[2] is the programming language that has been chosen to implement the MaDViWorld software framework. Some of the reasons for Java's suitability for the development part of this thesis are:

- Java is a complete object-oriented language.

- Java has built-in support for network programming facilities (see Subsection 2.4.9).

- The popularity of Java opens the MaDViWorld project to a large developer's community.

- Java is platform-neutral and its "write once, run anywhere" architecture guarantees the portability of compiled code.

- The Java Virtual Machine has a built-in, fine-grained, and very configurable security control mechanism.

Another reason for choosing Java is the support of the **Jini**[3] [145, 119] technology. Jini is a middleware providing a set of classes, interfaces, helper utilities, services, and related

---

[1] Software frameworks have been defined and extensively discussed in Chapter 2.4.

[2] The Java programming language was invented by Sun Microsystems, and the source for Java technology is http://java.sun.com/ (accessed December 28, 2004).

[3] The central place and resource for the Jini Community is http://www.jini.org/ (accessed December 28, 2004). The latest version (v2.0_002) can be downloaded from this site. Note that the preceding version (v1.2.1_001) is used in this work.

network protocols, for building scalable, robust, distributed systems using Java [70]. A Jini network is a network of many services able to find each other [145]. These services are dynamically combined in groupings called federations, or communities. Particulary interesting for unpredictable virtual worlds where rooms go on- and offline, is the Jini service location mechanism: a client locates a service through an intermediary service called a *lookup service*. Querying the lookup service is known as lookup, and a client lookup is typically performed based on *functionality* [70]. This mechanism allows avatars to find rooms without previous knowledge of their existence or of their address.

5.1.2 Global Position and Volume

Properties (4.6) and (4.7) are the only ones presenting a real problem if there is no central server. Indeed, in order to implement these two functions, there must be a central unit which knows all the subspaces existing at every moment and their respective locations. This central unit would then be able to maintain the "coherence" of the world relative to its chosen metric. That is the reason why, in MaDViWorld, we made the choice that the different subspaces composing the virtual world have no position and occupy no space. Loosing only these two restrictions, it is possible to develop a completely distributed virtual world in a quite natural manner. This corresponds to choosing a model in the right branch of the tree shown in Figure 5.1.
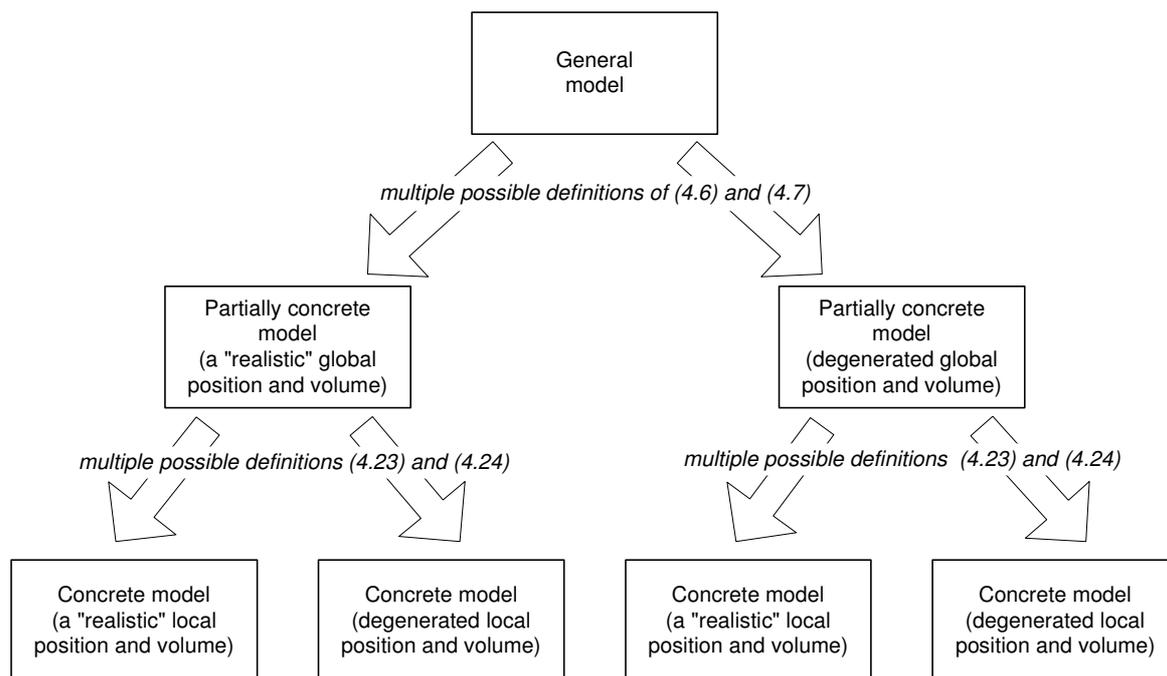
Figure 5.1: Model instantiation tree

5.1.3 Local Behavior

Concerning the subspaces "internal behavior", the model presents no constraints in the way each of them manages their objects and topology. The subspaces are completely free to control the behavior, permissions and motion of the avatars and object they enclose.

In the scope of this thesis, we also chose to have the simplest possible subspaces: the objects in them have no position and do not occupy a "physical" volume in the space. Indeed, we will implement the concrete model presented in Subsection 4.3.2. The fact that 3D features and the social aspects of interactions are not the first priority of the MaDViWorld project, justifies this choice. The model, however, allows for the addition of more realistic or sophisticated topologies.

5.1.4 Time and Events

Pertaining to the time, if the world manages and diffuses events consistently at the global level, there must be a central clock. One, however, can restrict events to the subspace containing the event source. Thus, only independent local clocks are required and no centralized 'time server' is needed.

5.1.5 Terminology

From now on when speaking of the implemented software architecture, the same metaphor as in MUDs will be used. Concretely, instead of speaking of subspaces and transport points we will use the terms *rooms* and *doors*.

5.1.6 Topology

As already mentioned, the subspaces are the conceptual unit of decomposition of a virtual world, but there is no constraint on the granularity at implementation and deployment time. At one end, all or several subspaces could be deployed on a same server, and at the other end a single subspace could be distributed over several machines. In the adopted software implementation described in the forthcoming sections, we consider that the model is sufficiently fine grained and that there is no necessity to split a subspace into several parts.

One could say that the goal is to carry out the analogy with the World Wide Web architecture (see Table 5.1). According to the classification presented in Subsection 3.2 this results in a *centralized+decentralized* network topology. Such architectures eliminate single points of failure and prove very robust (see Subsection 3.4). Figure 5.2 illustrates this topology even more precisely: the black nodes represent rooms; the white nodes represent avatars; the numbers on the room nodes represents a room server hosted on a computer. This figure emphasizes that a room server can manage one or several rooms, and that a room is not spread over multiple room servers.

World Wide Web	Virtual World
HTTP Server	Room Server
Web Browser	Avatar
Web Pages	Rooms
Applets, Servlets, Scripts	Objects
Hyperlinks	Doors

Table 5.1: Analogy between the World Wide Web and Virtual Worlds
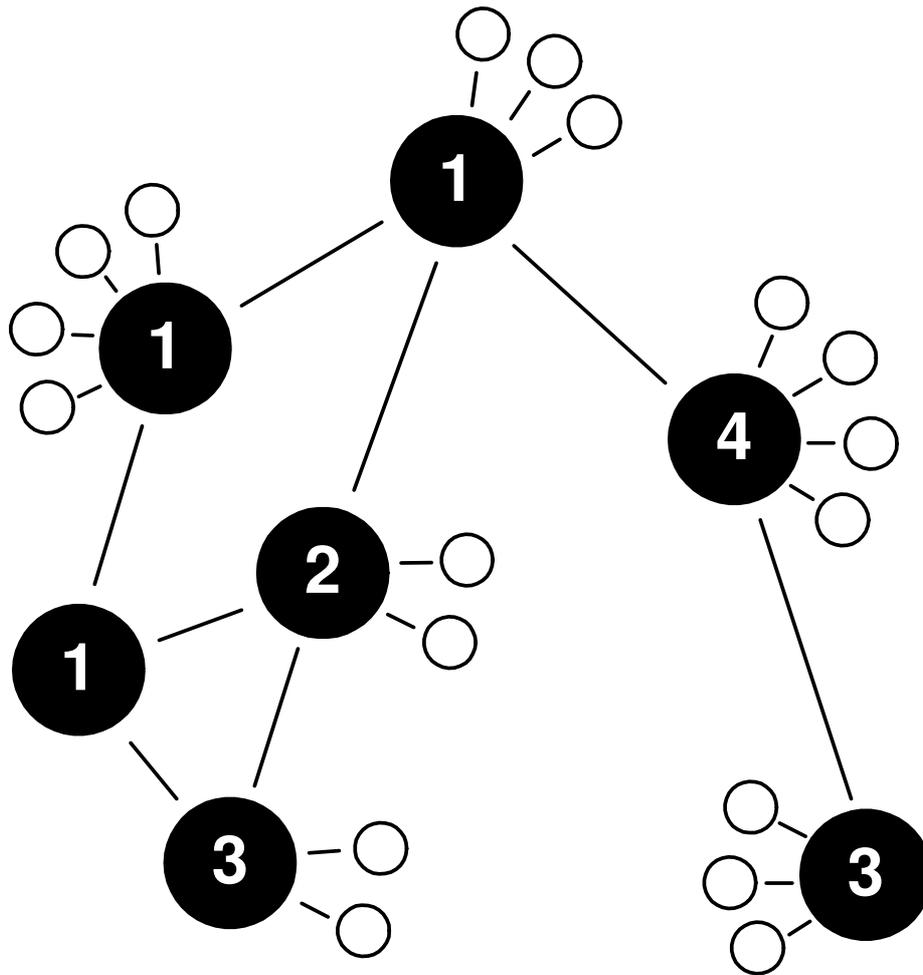
Figure 5.2: The MaDViWorld network topology: *centralized+decentralized*

Web server applications, in order to have failover and load-balancing capabilities, often use a cluster of machines arranged in a ring to act as a distributed server. To handle high clients load, room servers can first manage only one single room, and if this is not sufficient, could adopt the same solution as web servers and distribute a room on a cluster of machines. According to Minar's [136, 137] classification, one would have a *(centralized+ring)+distributed* hybrid network topology (see Figure 5.3). This solution would be a serious option to think about if the rooms should support three dimensional aspects and the high event traffic those would inexorably engender.

5.1.7 Notation

The present chapter and the next one present many figures to illustrate the inner structure of the framework. Most diagrams are designed as UML[4] class diagrams or as UML sequence diagrams.

Furthermore, this thesis uses a customized notation extending the UML class diagrams to clearly separate *(i)* the classes provided by the Java language, *(ii)* the classes of the framework, and *(iii)* the classes that the developer must create to customize the virtual world.

---

[4]The Unified Modeling Language (UML) is briefly introduced in Subsection 2.4.8.
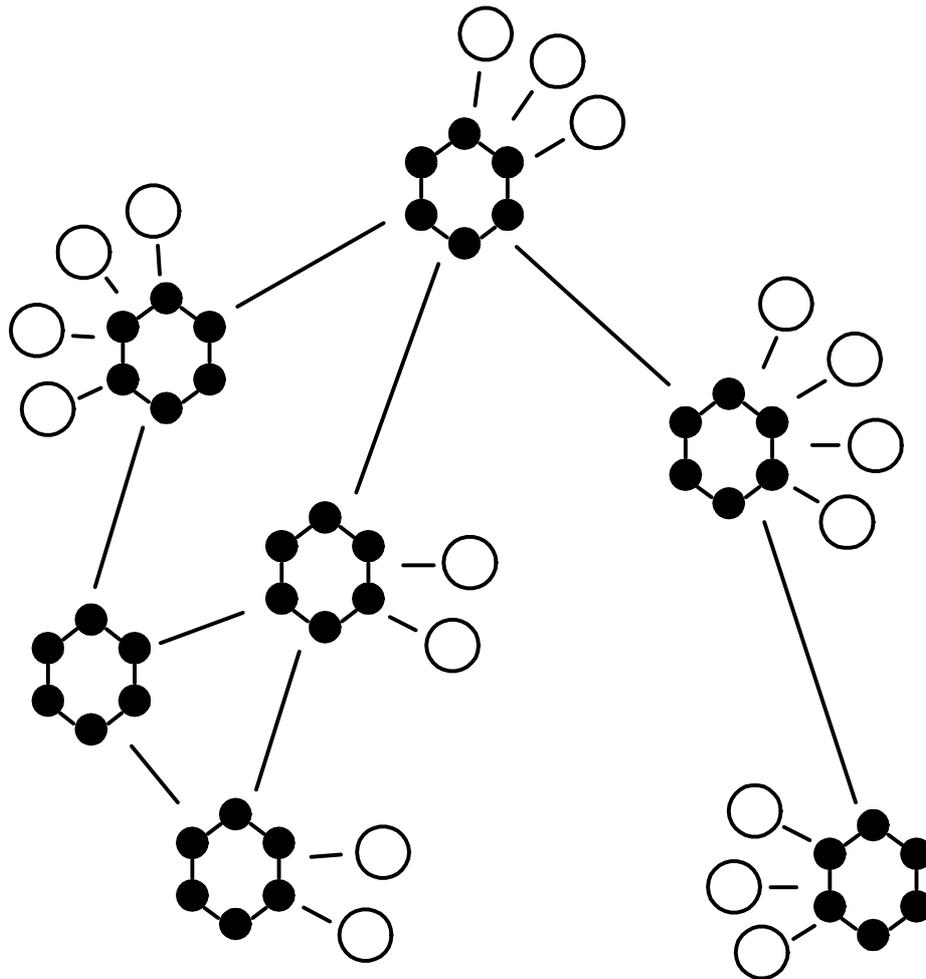
Figure 5.3: An improved network topology: *(centralized+ring)+distributed*

This solution offers a good readability and is also used in [70]. Figure 5.4 exemplifies this notation. It shows a ConcreteClass extending AbstractClass and implementing myRemoteInterface. myRemoteInterface extends the java.rmi.Remote interface and, as with all Java classes, AbstractClass is a subclass of java.lang.Object. The surrounding boxes clearly indicate which classes belong to the Java SDK, which classes belong to the framework, and which classes belong to the implementation of the specific virtual world. These boxes are used throughout the next chapters, when needed. To avoid cluttering the figures and befuddling the reader's mind, the arguments of the methods are never displayed. For the same reason, it was renounced to show the software packages[5].

The UML sequence diagrams are slightly customized as well. The goal of this adaptation is to visibly differentiate local and remote method invocations. Figure 5.5 illustrates this notation. It shows aClient first locally invoking localMethod() of aLocalObject and then remotely calling aRemoteObject's remoteMethod(). The thick black vertical line represents the network, and whenever a method invocation arrow crosses this line it represents a remote method invocation.

---

[5]The detailed class structure is described in Appendix A.
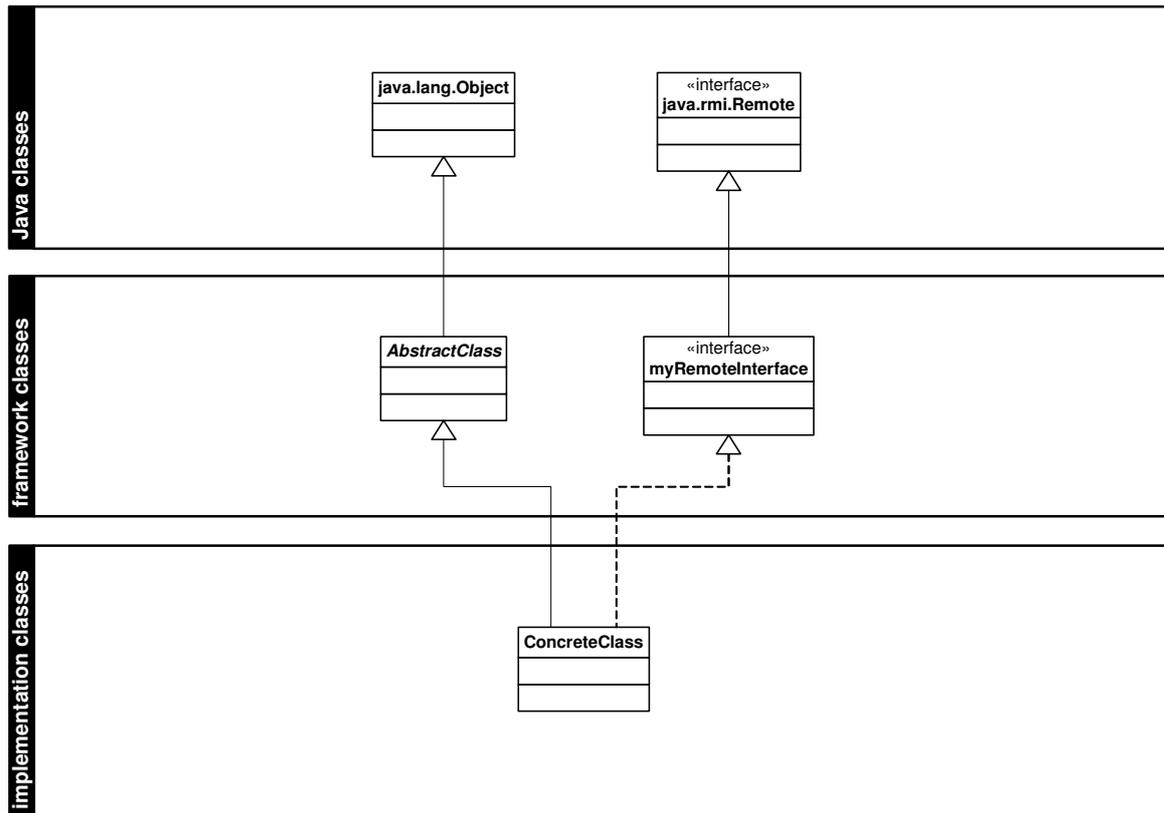
Figure 5.4: An example of an extended class diagram

5.2 The Software Architecture

It is now time to work out a software architecture bearing in mind all the considerations and definitions of the previous chapters. This section does not explain the details of the software design, but rather sketches the central classes and proposes a basic communication protocol between the different building blocks. This is the contact point between the theoretical and the practical parts of this thesis. The second subsection provides a bird's eye view of the whole framework structure.

5.2.1 The Basic Architecture

At this stage one has enough elements in hand to identify the major classes of the framework, as well as to find out their main methods. The structure should not be considered as exhaustive, but as a basic skeleton that will be the starting point for the development of the software architecture that will be detailed in the forthcoming sections.

The main classes of the virtual world framework clearly are:

- **Avatar**: This class implements the client application and represents the end user in the virtual world.

- **Room**: Rooms are server objects representing the atomic subspaces where interactions between avatars and objects can take place.
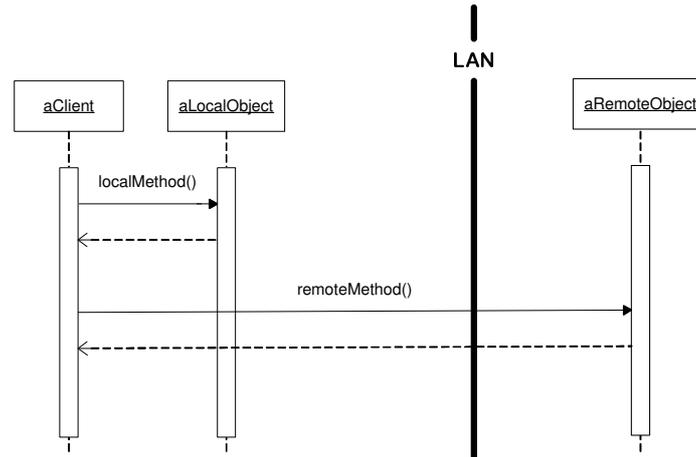
Figure 5.5: An example of an extended sequence diagram

- **WObject**: This class represents all objects that will populate the rooms. In order to avoid collision with the Java root class **Object** and because the term "object" is extremely broad in English language as well as having a variety of technical meanings, we use the term **WObject** to unambiguously refer to the abstraction provided by MaDViWorld.

- **Door**: The door class corresponds to a leaving point in the abstract model.

- **EventProducer**: Event producer is a role that has been defined in the event model. Indeed, in the virtual world there are event sources that produce some events.

- **EventListener**: Event listeners can be notified of events fired by the event producers.

- **Event**: This class represents events. An event has a source, a timestamp, a sequence number, and a type (see Subsection 4.4.2). The event producer sends the events to the registered listeners.

The methods or services theses classes must offer as well as their relationships can be directly deduced from the abstract model. The UML class diagram of Figure 5.6 shows the cornerstone of the framework, and is the consequence of following specific parts of the model:

- Property 4.15 states that avatars, doors and objects are contained in exactly one room. Thus avatars and objects know the room they are contained in and rooms provide setters and getters for avatars, objects and doors.

- Definition 4.26 states that rooms, avatars and objects can fire events. Thus, the classes representing these three classes "are" (inheritance relation) event producers.

- For the event and event listener classes, the methods are a direct consequence of the event model.

The dark grey boxes of the Figure 5.6 represent the implementations of the abstract interfaces contained in the light grey upper box. The methods are shortly described in Table 5.2 where the arguments are not displayed for sake of simplicity.
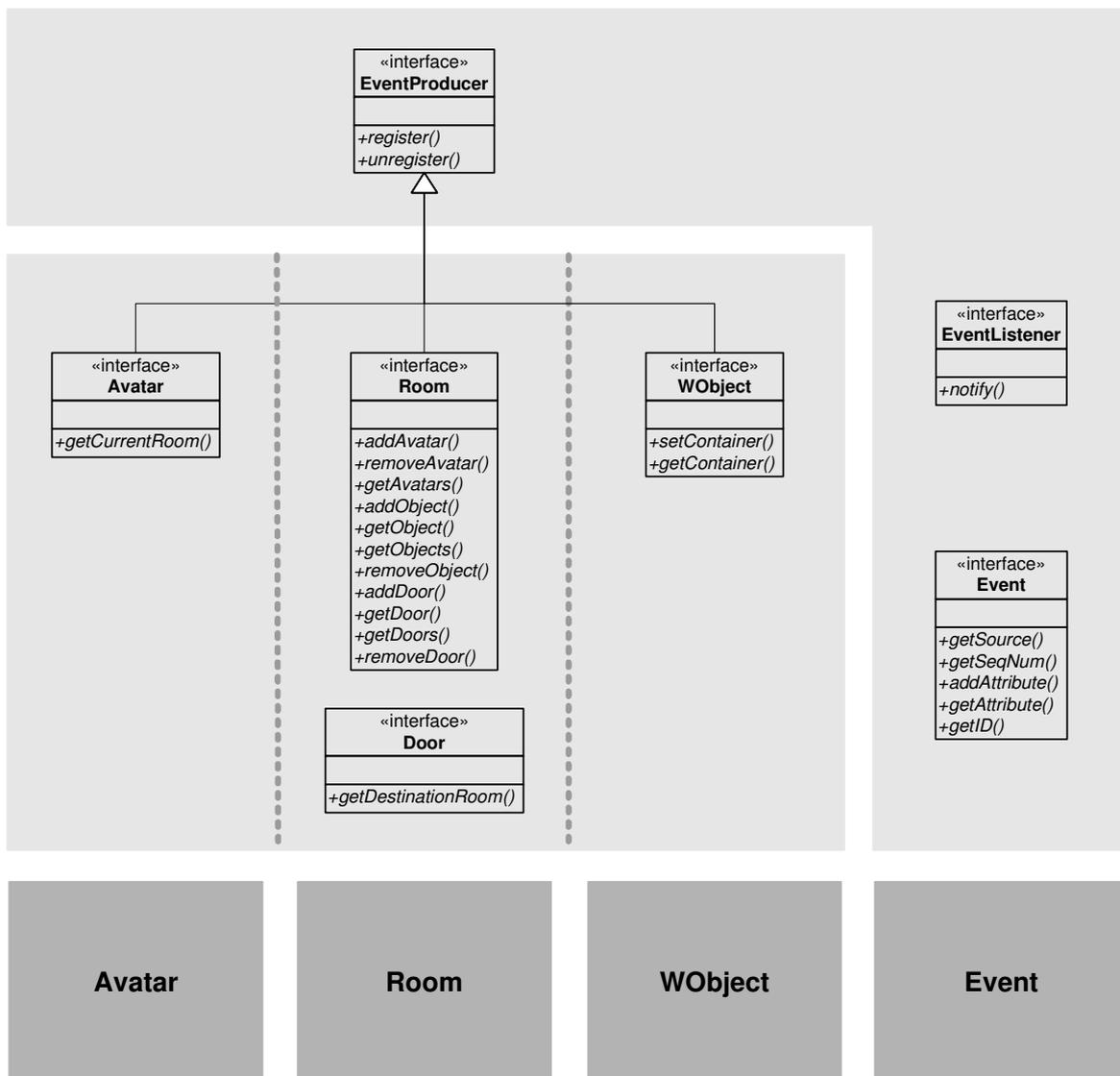
Figure 5.6: The starting point for the distributed framework

Avatar	
getCurrentRoom();	Returns the current room where the avatar is.
WObject	
getContainer();	Returns the container of the object (e.g., a room).
setContainer();	Sets the container of the object (e.g., a room).
Room	
addAvatar();	Sets a given avatar into the room.
removeAvatar();	Removes a given avatar from this room.
getAvatars();	Returns a list of the avatars in the current room.
addObject();	Sets a new object into the room.
getObject();	Returns a given object in the room.
getObjects();	Returns a list of the objects in the current room.
removeObject();	Removes a given object from the room.
addDoor();	Adds a door to a given room.
getDoor();	Returns a given door in the room.
getDoors();	Returns a list of doors to other rooms.
removeDoor();	Adds a door to a given room.
Door	
getDestinationRoom();	Returns a handle to a room.
EventProducer	
register();	Registers an interested event consumer with this producer.
unregister();	Unregisters a registered event consumer with this producer.
EventConsumer	
notify();	Notifies the event consumer of an event.
Event	
getSource();	Returns the source of the event.
getSeqNum();	Return the sequence number of this event, relative to its source.
addAttribute();	Attaches a given attribute to the event.
getAttribute();	Returns a given attribute of the event.
getID();	Returns the ID of the event or the event type.

Table 5.2: Some important method candidates of the main interfaces/classes.

5.2.2 Structure of the Framework

In order to satisfy extensibility and customization requirements a layered software architecture, leading from abstract to always more concrete classes, has been adopted. An overview of the whole framework is shown in Figure 5.7. First, let us consider a *horizontal decomposition* into three level of abstraction:

- The upper layer of the framework defines the communication protocol between the different components. It contains essentially exported interfaces that are accessible to the clients of the components.

- The middle layer consists of the default implementation packages of the framework. It contains the actual code for the functionality provided by the components.

- The lower layer, finally, is for the concrete applications, where all the application specific classes are placed. This layer may provide specializations of the features provided by the middle layer.

Second, let us decompose the blocks of Figure 5.7 *vertically*. From left to right one finds respectively all the packages and classes relative to the client application (i.e. avatars), then those relative to the server application (i.e. rooms) and those relative to the active objects populating the rooms. Finally, there are two utility packages, the event package and the rightmost one containing packages and classes used by the framework (such as http file servers, custom classloaders, etc.). Obviously, the avatar, room and wobject packages contain the *hot spots* of the MaDViWorld framework. Indeed, the framework user can:

- develop new types of objects by providing the appropriate implementations of the abstract classes of the wobject package.

- define and develop room and/or avatar applications by extending the framework classes in order to provide her own features.

- program all the applications from scratch respecting the interfaces defined in the framework's upmost level.

As recalled by Opdyke [147], a framework is a mixture of abstract and concrete classes. The MaDViWorld framework has a large library of concrete subclasses of each abstract class, so that most of the time an application can be plugged together from existing components. Even when new subclasses are needed, they are easy to produce; their abstract superclass provides their design and much of their code, and the already existing concrete subclasses provide examples of how to subclass from the abstract superclass.

5.3 A Utilization Scenario

The goal of this section is to provide end users and content creators with a preview tour explaining the utilization of the different applications of the MaDViWorld framework without entering into its more complex technicalities.
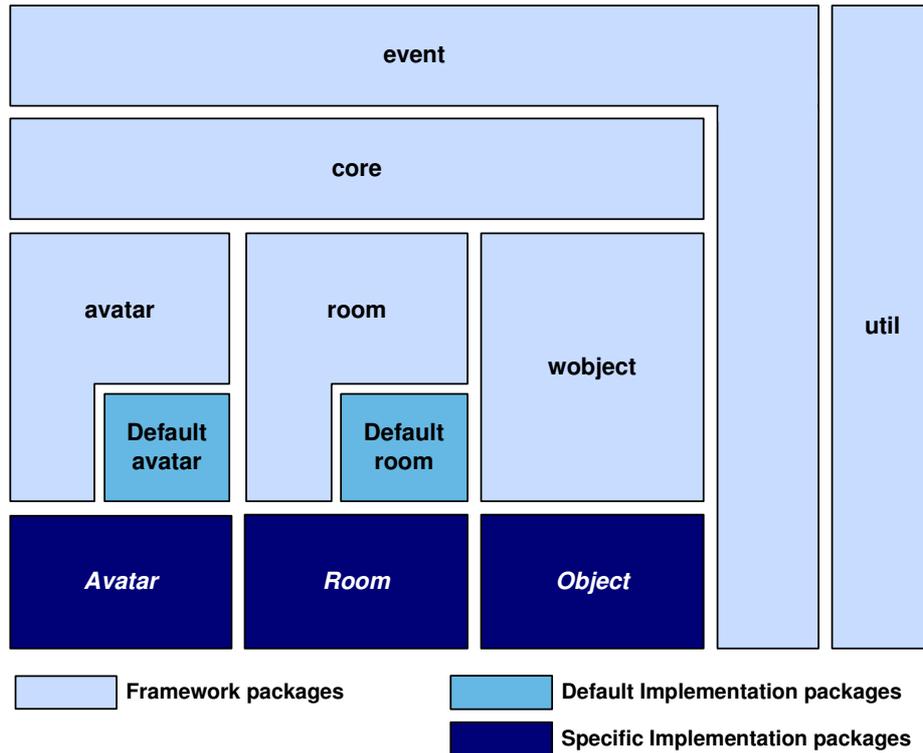
Figure 5.7: Overview of the MaDViWorld framework

5.3.1 End User View

This subsection illustrates a short albeit typical virtual world visit by an end user named George who has a MaDViWorld avatar application allowing for a seamless experience of moving inside an existing virtual environment.

- First George launches the client application and a click on the "Show available rooms on whole network" button shows him the list of the possible entry points of the virtual world (see Figure 5.8).

- George chooses to start his virtual trip by visiting the room R1. The avatar application displays the room (see Figure 5.9) and George notices that there is a tic-tac-toe game and that at this moment he is the only inhabitant of R1.

- He decides to put the tic-tac-toe object in his bag and hopes to meet someone to play with him later.

- As the room contains a door leading to room R2, George opts for this destination.

- In R2, he meets Jack and Mary–two other avatars–and using the chat object (see Figure 7.3) located in this room, he invites them to play with him. Mary accepts, but Jack prefers to watch how they play.

- Thus, George puts his tic-tac-toe into the room, and all the avatars join the game. George and Mary log in as players and Jack as a passive observer (see Figure 5.10).
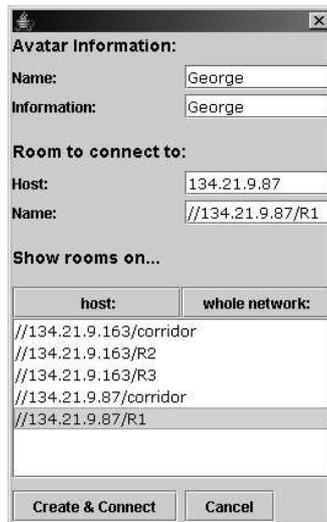
Figure 5.8: Startup screen of the avatar application

Figure 5.9: An avatar visiting a room

Figure 5.10: A MaDViWorld game with two players and one observer

- After several entertaining duels, George decides to stop playing. He can prolong his visit of the virtual world by accessing other rooms, interacting with other users and using further objects. But George can also stop his virtual trip by closing the client application and saving the avatar's state. Thus, it is possible to later reactivate the avatar: the objects in its bag are then recovered and the avatar always tries to return to its last location.

5.3.2 Content Creator View

The use of the MaDViWorld applications is also intuitive for a content creator, but needs a little bit more work. This section explains how a user, named Lucas, in possession of the server and the room setup applications can install a new virtual world with several rooms and objects within them.

- First Lucas must have at least one server machine connected to a network, in order to host the server application and to allow remote clients to connect with it. In our example, one considers that Lucas will launch a room server application on three different computers–H1, H2 and H3–on his local network. This application has no particular user interface and simply turns in the background waiting for client connections.

- Lucas further needs the room setup application to setup some rooms and one or more MaDViWorld object packages to enhance the rooms with them. There are two available room setup applications: the simpler one lets its user to set one room at the same time and the second one, a simple graphic editor, allows for the creation of an arbitrarily large amount of rooms in one single bunch. Both versions require the content creator to configure each room by giving such information as the room's name, the IP address of the room's host, the objects it will contain, security parameters (i.e. the passwords) and the doors to other rooms. Lucas has

three object packages: the tic-tac-toe, the chat and the clock. Figure 5.11 shows how Lucas uses the more sophisticated setup application to install and connect the rooms composing the simple world used in the preceding subsection, where:

– R1 is hosted on H2 and contains a tic-tac-toe game.

– R2 is hosted on H3 and contains a chat object.

– R3, R4 and R5 are hosted on H1 and each contain a simple clock.
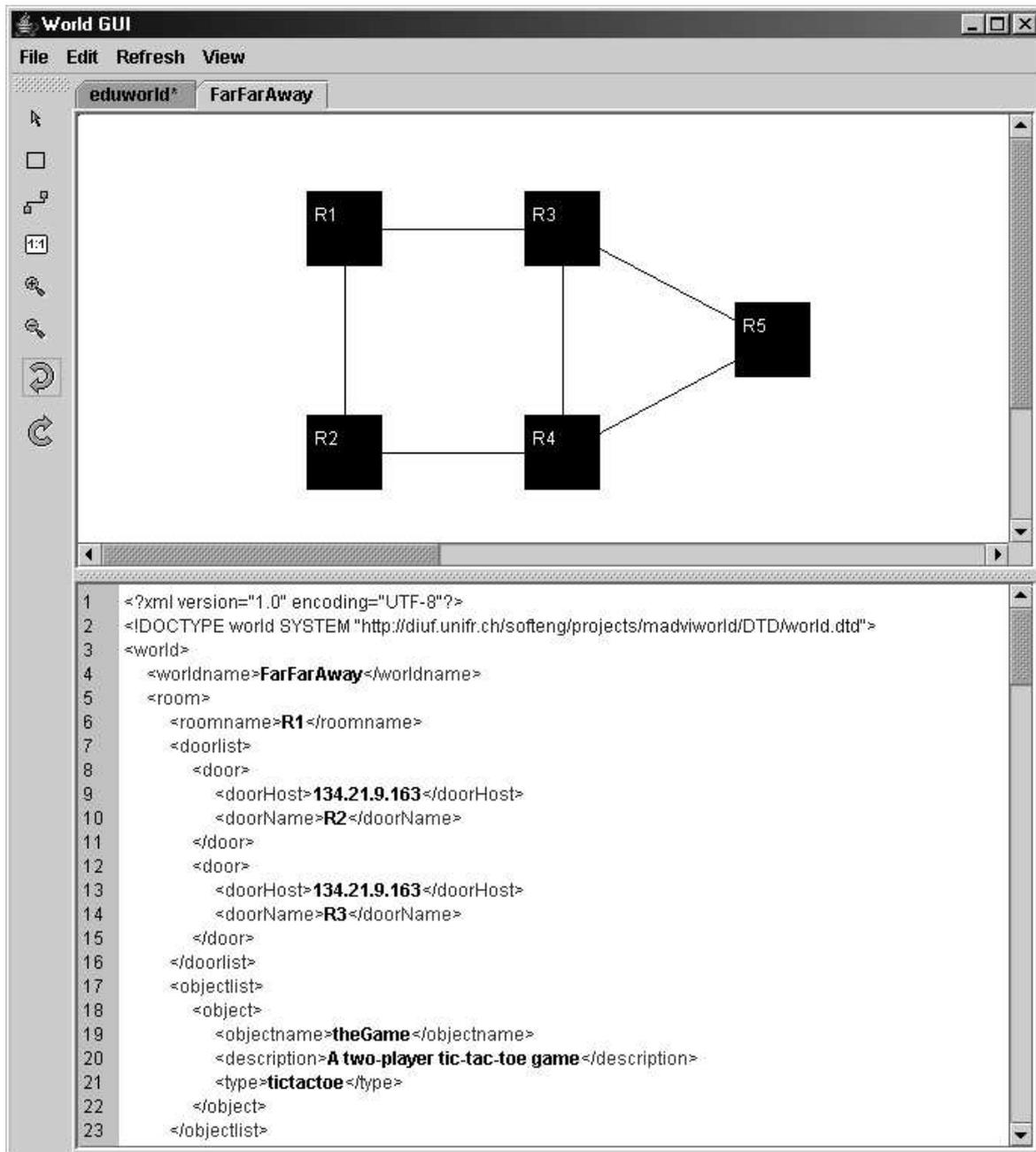
Figure 5.11: Creating a little world with the graphic editor

On Figure 5.11 one can see that the GUI of the setup application [152] is divided into two parts. The lower part displays an XML file corresponding to the virtual world graphically edited in the upper part. Hence a knowledgeable content creator

can also configure a virtual world by directly editing its corresponding XML source. The mechanism adopted by the setup application is summarized in Figure 5.12: *(i)* the graphical interface creates and displays a virtual world object; *(ii)* the virtual world data structure is synchronized with a XML file; *(iii)* an installation utility communicates with the room servers and attempts to set up the rooms with their objects.
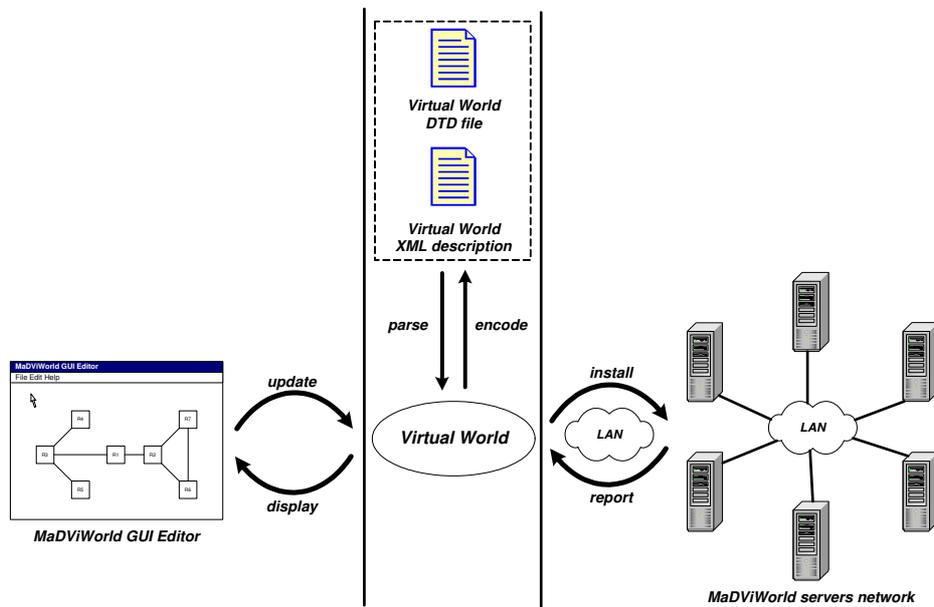
Figure 5.12: MaDViWorld setup application and XML description files

- While this graphic setup application eases the creation of a whole virtual world configuration, the simpler setup application (see Figure 5.13) presents the advantage that its user can easily change one single already existing room. Indeed, the content creator may add a new object to a running room or change its security parameters.
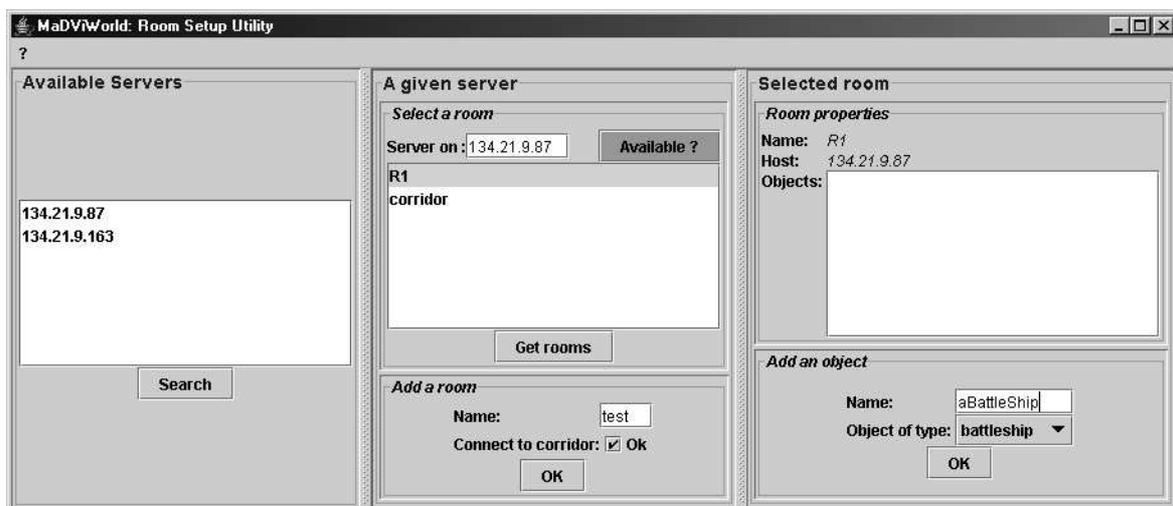
Figure 5.13: Customizing a room with the setup application

Note that the room setup applications can run on any computer on the network as it collaborates remotely with the room server applications. Therefore the first step must

not necessarily be incumbent upon the content creator. If room servers are already setup on the network, anyone in possession of a room setup application can install new rooms on the remote hosts.

<div align="right" style="font-size:4em">6</div>

The MaDViWorld Framework: Software Design and Special Topics

*Any fool can write code that a computer can understand.
Good programmers write code that humans can
understand.*
—Martin Fowler

This more technical chapter fosters the presentation and the explanation of the design choices made during the development of the MaDViWorld framework. It should offer application programmers the general philosophy and the key notions needed to extend or improve the existing architectural building blocks. The concepts and design principles are documented essentially by means of design patterns and UML diagrams (static class diagrams, dynamic sequence diagrams and deployment diagrams). The first section tackles general principles that are true for the whole framework. The second section focuses on some selected topics of particular interest such as the remote event mechanism, the security model and the 'strong' object mobility. It is worth noting that the presented designs are the result of several adjustments. The pain of designing a framework from scratch is already described in [208]: "Good frameworks are usually the result of many design iterations and a lot of hard work."

6.1 Design Choices

This section presents general design choices made for building the overall structure of the framework (multi-layer and multi-tier decomposition, separation of interface and implementation, adaptation by extension and by implementation, logic and presentation separation).

6.1.1 A Layered Software Framework

MaDViWorld is a *distributed* framework and adopts a multi-layered and multi-tiered architecture. More precisely there are abstraction layers and orthogonal deployment tiers. This decomposition allows for an optimal separation of concerns between the different building blocks. Figure 6.1 illustrates the global structure of the framework and is actually a refinement of Figure 5.7.

First, let us recall the roles of each abstraction layer, which alltogether embody the fundamental principle called *separation of interface and implementation* [30]:

- The *upper abstraction layer* (core) contains the interface parts of all the main components of the system. It defines the functionality of each component and provides clients with guidelines for using them. The specification of these interfaces could be strengthened by using *Design by Contract* (see Subsection 2.4.10). As MaDViWorld is implemented in the Java language which does not support *Design by Contract*, rigorous specification must be provided by a good documentation of the interface methods[1]. Thus, this first layer defines clear boundaries between the components and defines a communication protocol between them.

- The *middle layer* consists of the default implementation packages of the framework. It contains the implementation part of the components and the actual code for the functionality they provide.

- The *lower layer*, finally, is for the concrete applications, where all the application specific classes are placed. This layer may provide specializations of the features provided by the middle layer.

The main idea behind this decomposition could be summarized with the following idiom: "Program against interfaces, not classes." Adopting this technique is a way to achieve information hiding and encapsulation and results in a low coupling of components. This approach supports changeability and eases the task of altering a component's behavior or representation (see Subsection 6.1.3). The Bridge [73] pattern, for example, addresses this principle.

Second, let us give some details about the vertical tiers which correspond each to one of the three main applications interacting when using virtual worlds.

- *Avatar* application: This leftmost tier contains the classes and packages implementing the avatar. As already mentioned, it is a client application allowing for the connection to rooms, and for the interaction with objects and other avatars.

---

[1] Javadoc is the JDK tool that helps creating your documentation and keep it up-to-date. The javadoc tool is used for generating API documentation in HTML format from doc comments in source code (see http://java.sun.com/j2se/javadoc/index.jsp (accessed December 28, 2004)).

- *Room Server* and *Rooms*: The second tier is composed of two parts. The implementation of the room interface supports a single room. The second component of this layer is dedicated to a room server application that acts as a room factory. A factory, in this context, is a piece of software that implements one of the "factory" design patterns introduced in [73]. The room server manages the rooms existing on a given host and controls the creation of new ones on behalf of a setup application.

- *Setup Application* and *Objects*: This tier contains the packages concerning the objects. A room setup application allows for the creation and customization of rooms on distant room servers, and for the installation of objects into them.

There remain two building blocks that were not discussed yet: event and util. These are in fact two utility packages. The first one is dedicated to the remote event mechanism and the second one contains packages and classes used by all the components of the framework (such as http file servers, custom classloaders, etc.).
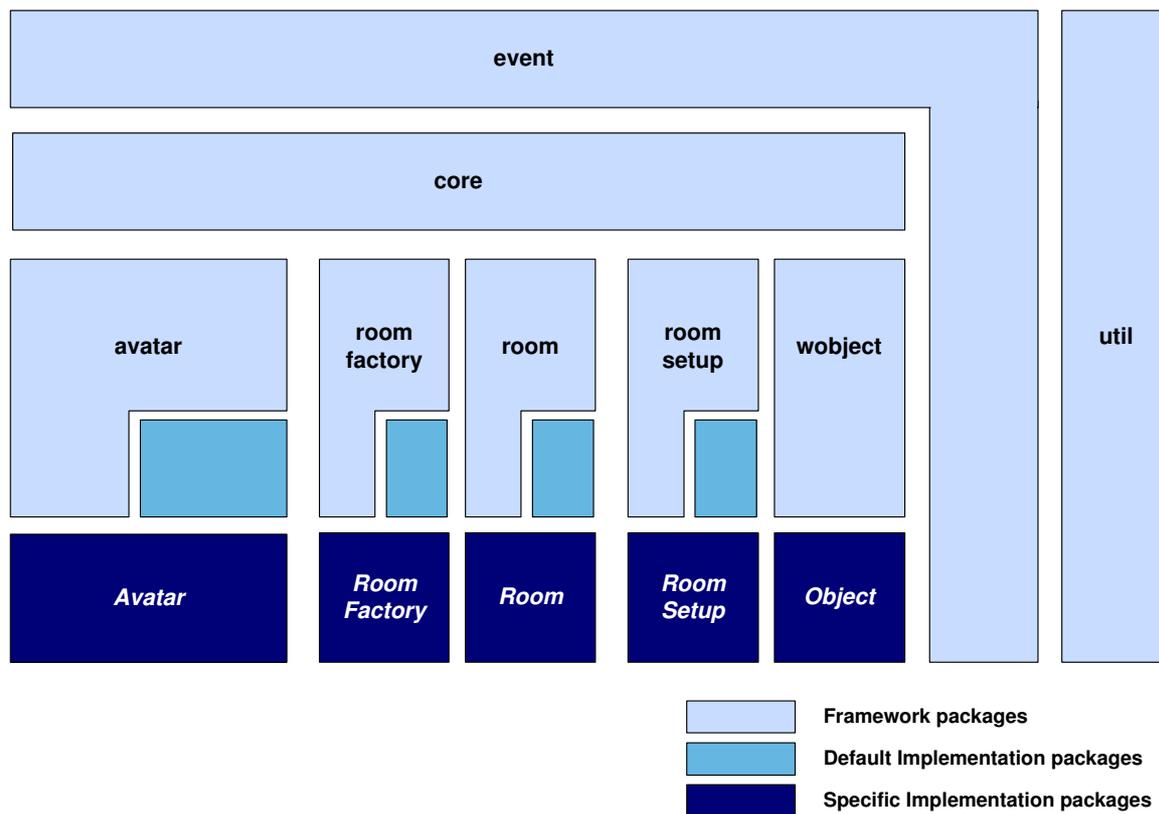
Figure 6.1: Vertical and horizontal layers of the MaDViWorld framework

Each of the three main tiers can be deployed separately. The applications are deployed with the packages directly concerning themselves, as well as those common to all applications, i.e. the core layer, as well as the event and util packages. Figure 6.2 shows how the different packages are bundled for deployment and Figure 6.3 depicts a UML deployment diagram capturing the runtime configuration of the framework's elements. The package of a given object is deployed on the setup computer at the system startup, but as objects are mobile this component is shown on the three deployment nodes.
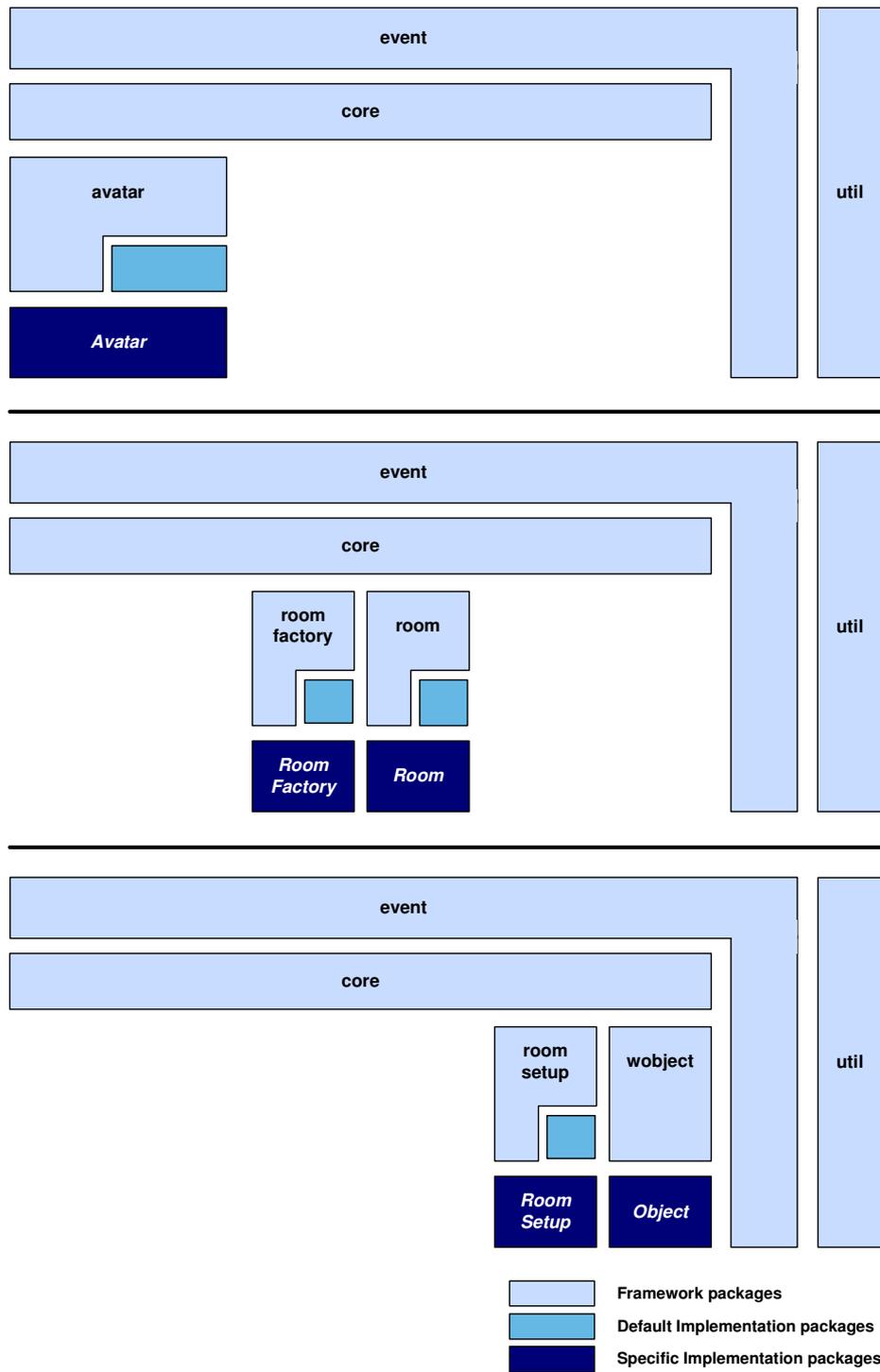
Figure 6.2: *Top to bottom*: Packages required for the deployment of avatar, room server
and room setup applications
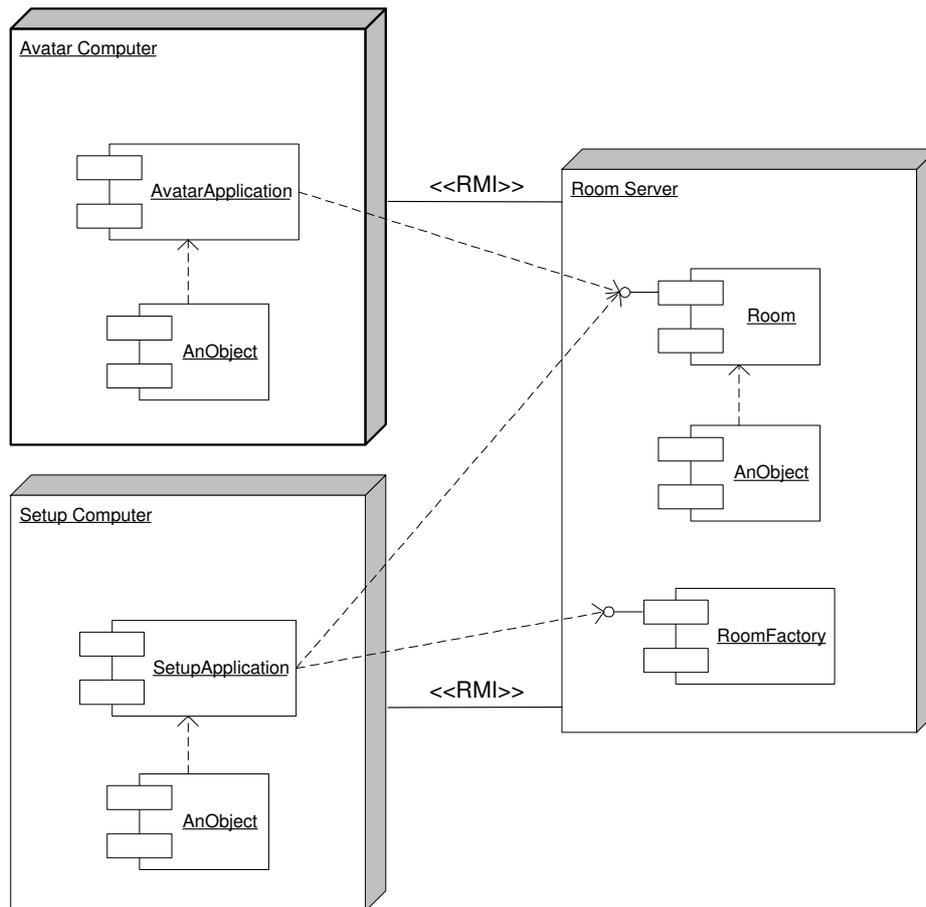
Figure 6.3: UML deployment diagram for the MaDViWorld framework

6.1.2 Extension Mechanism

Frameworks were extensively discussed in Section 2.4 and we have seen that a good framework has to include a mechanism to allow a developer to plug in the varying functions, or to extend the proposed default functions. The places where adaptations for specific functionality should be made, i.e. the hot spots, have to be defined as well. While the latter will be shown in the following sections, the extension mechanisms of the proposed MaDViWorld framework are now discussed. There are three means of adaptation for specific functionality: *(i)* adaptation by *extension*, *(ii)* adaptation by *implementation* and *(iii)* adaptation by *extension and implementation*.   Figure 6.4 helps to explain these concepts.

In many situations, the framework offers one Java interface and one Java class providing a default implementation of this interface. If this default implementation is adapted to the virtual world programmer's needs or if she wants to develop a prototype very quickly, she can either use the DefaultImplementation on its own or she can easily build a trivial subclass of DefaultImplementation and use it in her specific virtual world. However, if the default implementation is not adapted to the virtual world programmer's needs or if she has a different approach about the implementation of the ImplementableInterface, she can directly write her own interface implementation. This takes much more development

time, but eventually produces a class better suited to her needs. The stereotype[2] **optional** shows that the framework does not formally require the class **AdaptationByExtension**. For the avatar and room setup application, as well as for the rooms, these two extension schemes fit well.

In some cases, the framework provides only partial implementation of the **ImplementableInterface** providing some general purpose functionalities common to all subclasses. These **ExtendableClasse**s must further be appropriately implemented by **AdaptationByExtensionAndImplementation** class. This mechanism is the one used for the objects.
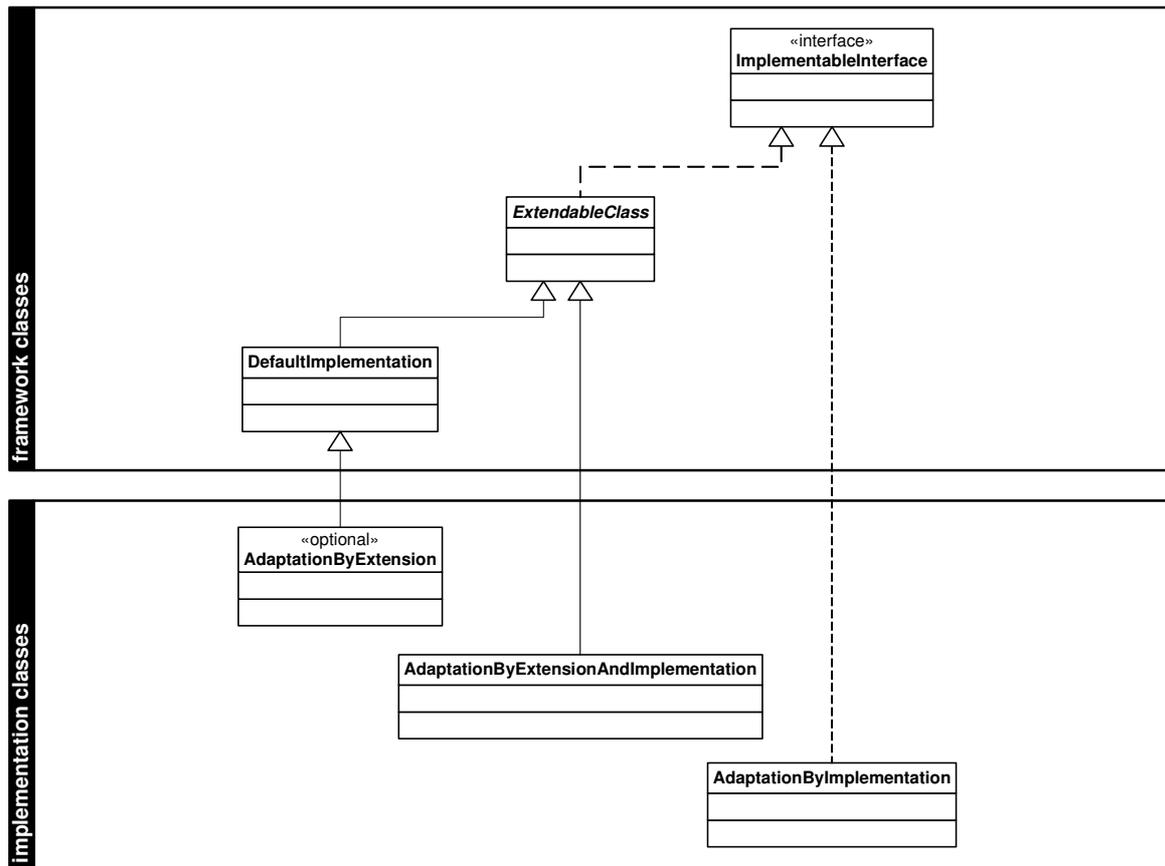
Figure 6.4: The three modes of adaptation offered to the framework user

6.1.3 Separate Logic From Presentation

A well-layered framework–particularly one supporting distributed virtual worlds (see Subsection 3.1)–provides a neat separation between the code that handles the user interface (UI), often called *presentation*, from code that handles the business *logic* or domain logic [189]. This principle presents the advantage that it separates two complicated parts of the framework into pieces that are easier to modify independently. It also allows multiple presentations of the same business logic; for example a simple text-based system, a complex graphical user interface, or more exotic user interfaces (web-based applications, cell phones, etc.).

---

[2]Stereotypes are used to extend the UML notational elements.

This practice is encouraged by [60] where Fowler named it *Separate Domain From Presentation*. In the MaDViWorld framework these guidelines were respected and lead to the basic pattern illustrated in Figure 6.5. The framework provides abstract and concrete classes defining the logic of the different building blocks, as well as concrete classes handling all the operations related to the presentation part ConcretePresentation. The virtual world designer can extend the ConcreteLogic class to add the specific logic of her target system (RefinedLogic class). She also has the possibility to extend the default user interface (RefinedPresentation class) or write her own from scratch.
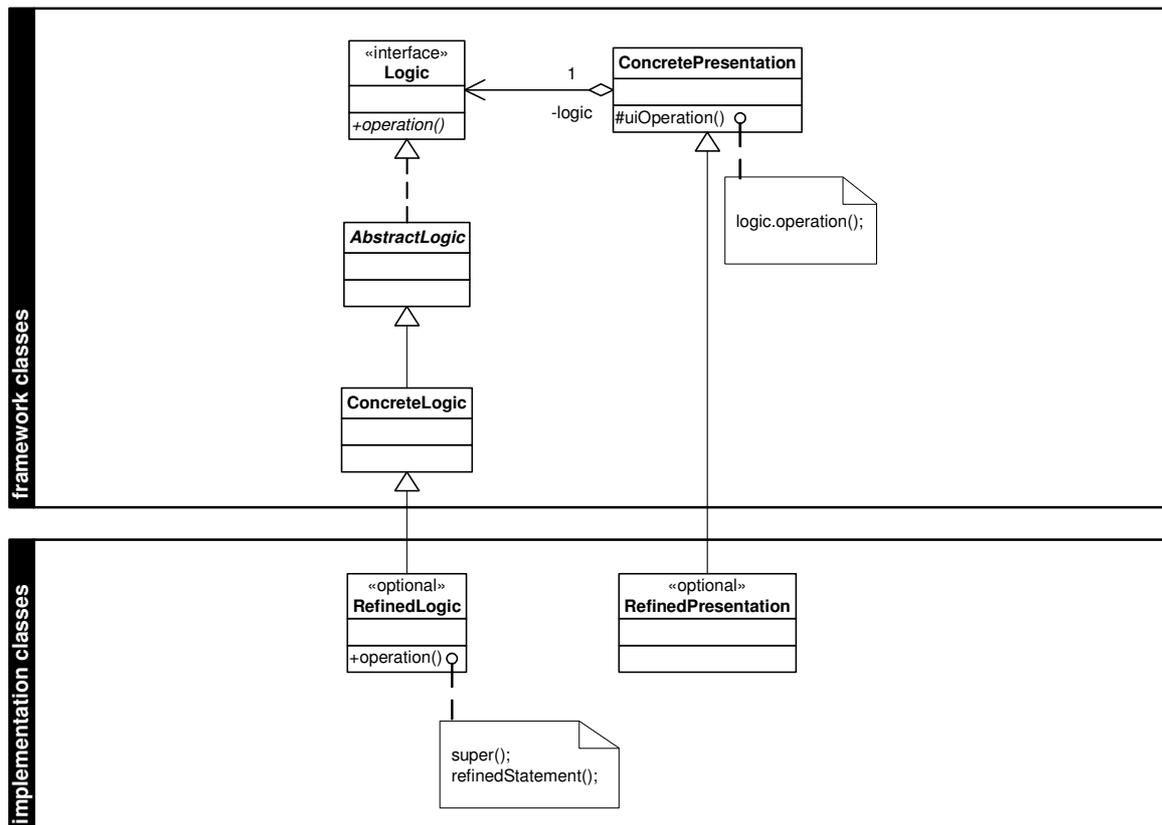
Figure 6.5: Presentation/Domain Separation

When the ConcretePresentation class needs to execute a logical operation, it simply delegates the method call to its own ConcreteLogic class[3] (logic.operation()). The Logic usually does the real work. Sometimes however, Logic is designed as a Facade [73], providing a unified high-level interface to a set of interfaces in a subsystem.

6.2 Special Topics

In addition to the general principles discussed in the preceding section, there are several special aspects worth presenting. Each subsection succinctly exposes the solution of the main problems encountered during the development of the distributed MaDViWorld framework.

---

[3]Note that the ConcreteLogic has no knowledge of the ConcretePresentation. Subsection 6.2.4 shows how the logic can interact with its presentation thanks to events.

6.2.1 Lookup and Registration

Within an environment composed of distributed components, like the virtual worlds we are interested in, there must be a way to locate the existing objects or services (room servers, rooms, avatars, objects). Thus the system must provide a lookup mechanism, that faces the resource discovery (see Section 3.3) issue. This subsection analyzes the software design adopted by MaDViWorld to solve this problem. It offers the possibility to make use of two complementary mechanisms allowing avatars, room servers and rooms: *(i)* to register themselves in local rmiregistries offered by standard Java RMI; and/or *(ii)* to act as well-behaved Jini services by registering themselves to remote Jini lookup services (reggie). Furthermore, the framework's software architecture allows the developer to add new registration and lookup technics[4] in the future using, for example, CORBA[5] or Web Services[6] technologies.

Let us comment the UML diagram of Figure 6.6 which depicts the different classes allowing to manage the RMI and Jini technologies. The main class supporting registration and lookup operations is LookupAndRegistrationSystem which is a Composite [73] in the sense of the design pattern. The LookupAndRegistrationStrategy interface is the 'Component' and the two concrete strategies (the 'Leafs' of the pattern) are RMILookupAndRegistration and JiniLookupAndRegistration. If a developer wants to add a new LookupAndRegistrationStrategy she just has to provide an appropriate implementation (adaptation by *implementation*). Clients do not care if they are dealing with one or several lookup and registration strategies and the general principle of *separation of interface and implementation* evoked in the preceding section is respected.

Moreover, the implementation of the 'Composite' LookupAndRegistrationSystem, which is just responsible for management tasks, follows the Singleton [73, 77] design pattern. This design ensures that only one instance of this class exists throughout the system. The private selectedStrategies field contains one instance of each strategy it manages and is set up at initialization time.

This design was adopted for its flexibility because it allows for the combination of an arbitrarily large amount of concrete strategies. It was preferred to a classic Strategy [73] pattern which enforces the user to choose one single strategy at the same time.

The classes RMIManager and JiniManager, also visible on Figure 6.6, are utility classes used by the respective strategies. They encapsulate the configuration and the management proper to each underlying technology and hide low level details from the strategy. These classes are designed as Singletons as well. Another utility class, not shown on Figure 6.6 and called Administration, frees the programmer from all the tasks associated with the creation of the a Jini federation, such as the launch of an rmiregistry, an activation system,

---

[4]Naturally the different strategies must all implement the RMI object semantics.

[5]CORBA is the acronym for Common Object Request Broker Architecture, OMG's open, vendor-independent architecture and infrastructure that computer applications use to work together over networks. Using the standard protocol IIOP, it is useful to achieve interoperability in heterogenous (different programming languages, operating systems,...) software environments.

[6]Web Services are standards-based software components that can be accessed over the Internet. In a typical Web services scenario, a business application sends a request to a service at a given URL using the SOAP protocol over HTTP. Based on open standards, Web Services can utilize any platform, object model, or programming language. In addition of SOAP (Simple Object Access Protocol), three other XML-based technologies enable this: WSDL (Web Services Description Language), UDDI (Universal Description, Discovery, and Integration) and XSLT (Extensible Stylesheet Language Transformation).

the instantiation of a shared virtual machine, of HTTP servers, of lookup services, etc. All these operations are needed to bootstrap the system and can be tricky to implement for a Java programmer lacking in experience with Jini. All the methods of the Administration utility class are static and most of them are used during the system launch.
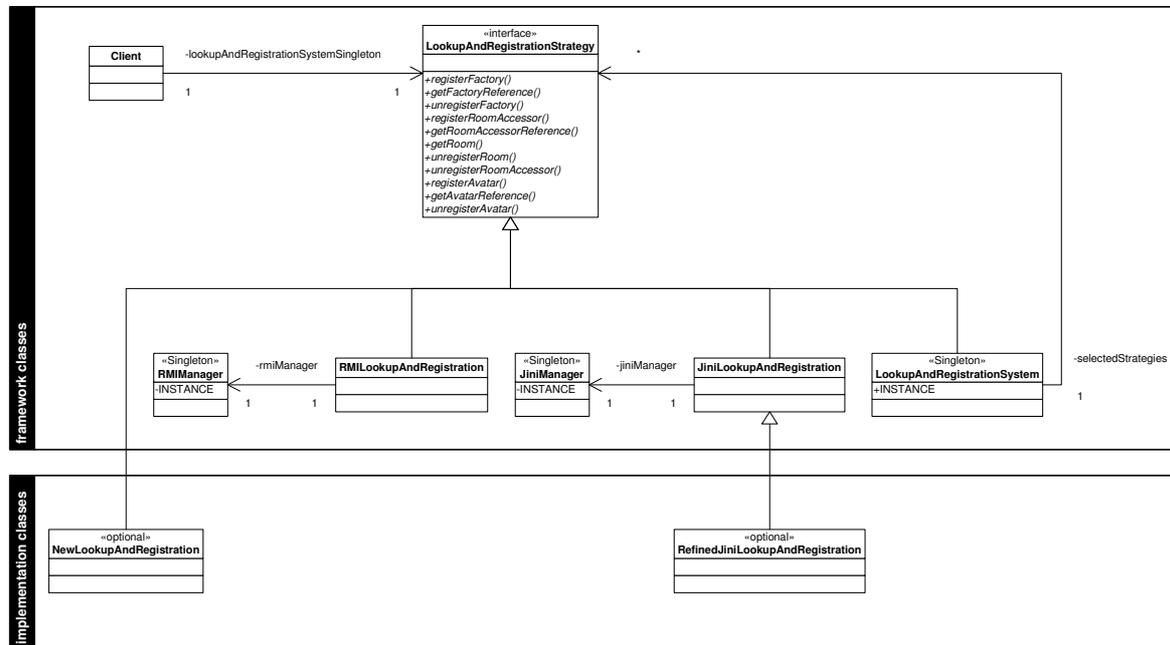
Figure 6.6: Managing the RMI and Jini technologies

The use of Jini offers great benefits for looking up rooms and avatars when one has no *a priori* knowledge of the running world (i.e. on which machine to find a room for example). In addition to that, the template matching mechanism of Jini lookup presents a lot of possibilities (e.g., searching all german speaking avatars or all rooms intended for games). Unfortunately, these lookup services become bottlenecks if the world becomes too wide.

The classical rmiregistry approach offers the shallow and flat "search by name" lookup method. Since each rmiregistry only contains the rooms and avatars running on a given machine (and those are not so many), it offers a good performance and avoids bottleneck.

The combination of these two publishing methods offers great advantages without altering the scalability of the world.

6.2.2 Distributed Event Model

Events play a crucial role in the MaDViWorld framework because they glue its different components together. Indeed, events are the only communication channel between rooms and avatars, rooms and objects and between two objects. Subsection 6.2.4 shows yet another situation where remote events play a central role. Schematically, each time the state of one of the world components changes, a corresponding event is triggered by the altering subject and consumed by the registered listeners, which react appropriately. The management of all these events is a complex task for several reasons: *(i)* they are in reality remote events and several network related problems can occur; *(ii)* some of the events have to be fired to only a subset of all the listeners; *(iii)* some listeners may not

be interested in every type of event. The *distributed event model* of the framework must handle all these situations thus contributing to improve the global network performance (see Subsection 3.2).

The two last points listed above, lead to the elaboration of an abstraction for creating unique identifiers. DUID is the acronym for Distributed Unique ID and is implemented in the DUID class[7]. Each room, room server, object or avatar has an associated DUID that is generated by the framework and that never changes during its life cycle, so that it can be identified without ambiguity. The use of such a DUID was inspired by [70].

It is now time to take a closer look at the framework classes which aim to solve the mentioned problems (see Figure 6.7):

- The RemoteEventListener interface extends the java.util.EventListener interface and defines the single notify() method. Any object that wants to receive a notification of a remote event needs to implement it.

- The RemoteEventProducerImpl class implements two interfaces: *(i)* RemoteEventProducerRemote is an interface defining the methods that interested event consumers can remotely invoke to register their listeners; *(ii)* RemoteEventProducerLocal does not extend java.rmi.Remote since the methods it defines are not offered to remote clients. Therefore RemoteEventProducerImpl provides the methods needed to register, unregister and notify event listeners used to communicate between different parts of the system. The register method takes as parameter the event type the listener is interested in. The five event types defined in Subsection 4.4.2 are: *all* events, *avatar* events, *object* events, *room* events and *"events for me"*. With the latter, the listener is only informed of events addressed explicitly to it (thanks to its DUID), without paying attention by whom.

- The RemoteEventNotifier helper class notifies in its own execution thread a given event listener on behalf of a RemoteEventProducerImpl.

- The RemoteEvent class defines remote events passed from an event producer to the event notifiers, which forward them to the interested remote event listeners. A remote event contains information about the kind of event that occurred, a reference to the object which fired the event and arbitrarily many attributes.

The design pattern illustrated by Figure 6.7 is used through the whole framework for the collaboration between the three different parts of MaDViWorld (i.e. avatars, rooms and objects) and the utility event package. Note that the three of them are both implementing the RemoteEventProducerRemote interface and are client of its default implementation, RemoteEventProducerImpl. The operations defined by the interface are just forwarded to the utility class. With this pattern we have the suited inheritance relation (a WObject 'is a' RemoteEventProducer) without duplicating the common code. A lot of similarities with the Proxy pattern defined in [73] can be found. This composition based design is more flexible and better adapted to our class hierarchy than the straightforward approach consisting of just inheriting of a common RemoteEventProducerRemote implementation.

---

[7]The DUID is the combination of a java.rmi.server.UID (an identifier that is unique with respect to the host on which it is generated) and of a java.net.InetAddress (a representation of the host's IP address where the object was created which makes the UID globally unique).

Anyway, the main inspiration of this structure comes from the Observer [73] pattern and its *publish-subscribe* interaction kind.

To sum up the whole event mechanism, the UML sequence diagram of Figure 6.8 dwells on all the operations, from the registration phase to the firing and notification of an event. First *(a)*, the event consumer registers a **RemoteEventListener** to a room, avatar or object whose events it is interested in. Second *(b)*, due to a state change an event is fired and all interested listeners are notified, each by a **RemoteEventNotifier**. The informed listener can then do the appropriate work with regard to the type of the event. On Figure 6.8, one can also see the different methods invoked remotely across the LAN. This pattern presents some similarities with the *Jini distributed event programming model*, which is specified in [11] and thoroughly explored in [119].
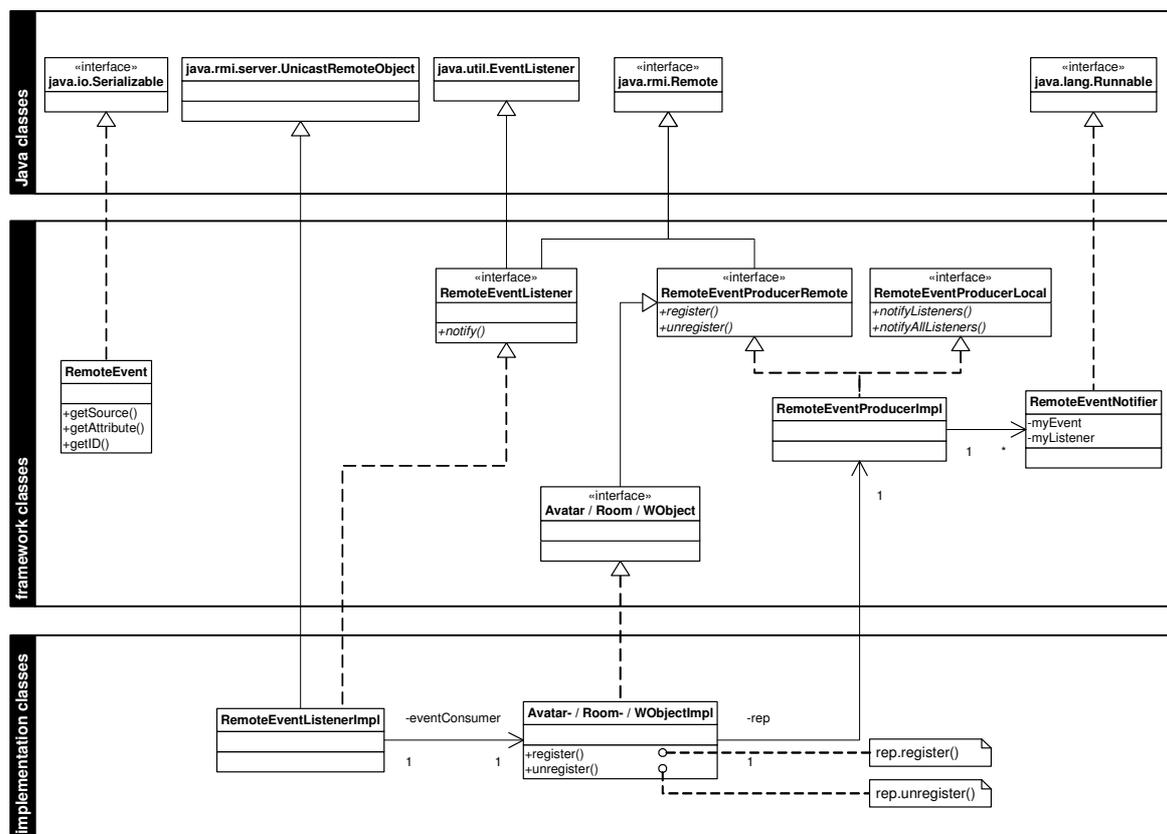
Figure 6.7: Pattern used for integrating the event model in the framework

6.2.3 Security

In Subsection 3.5 two levels of security concerns have been distinguished: *(i)* the system level and *(ii)* the virtual world level. In order to address system level security concerns, facilities offered by the Java and Jini technology can be used. In the actual version of the **MaDViWorld** project, system level security is not the first priority, and some further configuration would be necessary prior to large scale deployment. This subsection clarifies how the framework manages security at the virtual world level where actually two kind of security sensitive actions can be identified.
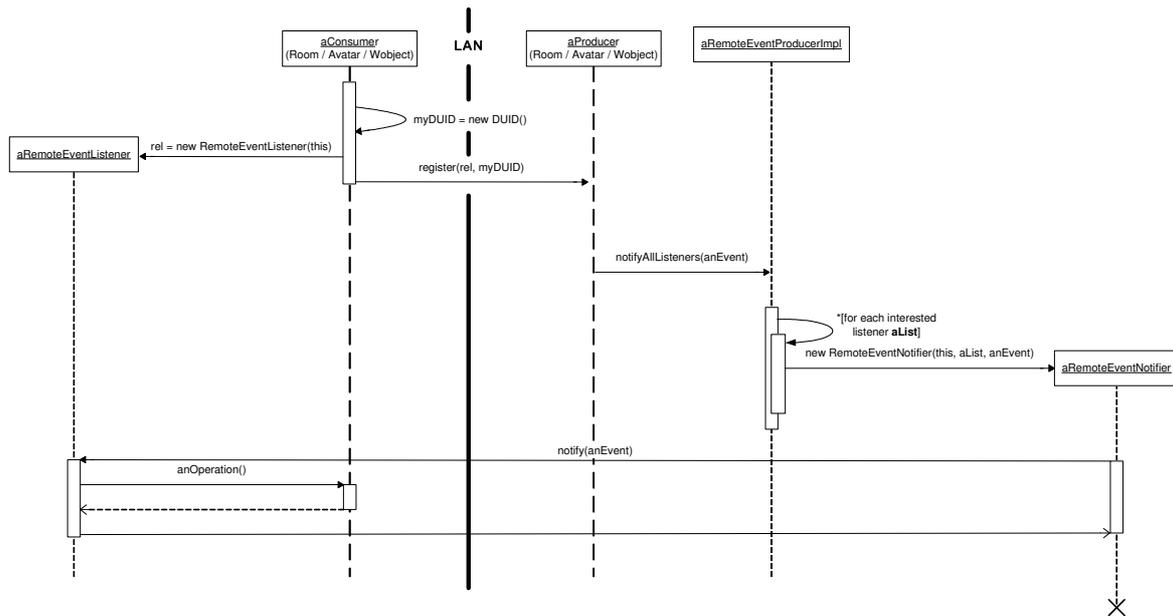
Figure 6.8: Setup of the event model and notification of an event

- *Virtual world applications administrative actions.* The room server application may have restricted access and prevent anyone to manage rooms it is hosting. This means that in order to create a new room or to remove it, authentication may be required. It may also be desirable to encrypt persistent data concerning rooms or avatar. Although these features would be of great utility, they have not been tackled by the current version of the MaDViWorld framework. But since there exist well-known techniques (user authentication, password protection, digital signatures, cryptography) to address these issues, those features could be effortlessly added to the room server or to the avatar applications.

- *Actions inside the virtual world.* Section 4.5 identifies the critical actions that objects and avatars may undertake while visiting the rooms of a virtual world. The conditions under which this actions are allowed were theoretically defined. An implementation of the proposed model is part of the MaDViWorld framework and is documented below. Note that the adopted approach is mainly concerned with authorization[8], and less with authentication[9]. The latter aspects should be reinforced in order to support identity critical applications such as e-commerce, virtual e-banking, secured chats or e-learning.

It is worth recalling the security model's basic principle: the room grants access rights to the avatars and objects. Rooms will achieve this task by using *challenge-response tests*. A challenge-response test is a test involving a set of questions (or "challenges"), that the other entity has to answer in order to pass the test. If the entity provides a satisfactory response to the challenges then it is deemed that the entity has passed the test. The question often relies on the possession of a secret of some sort. A simple example challenge is asking for a password, and the adequate response is the correct password.

---

[8]Authorization is the process of giving individuals access to system objects based on their particular level of security clearance.

[9]Authentication is any process by which you verify that someone is who they claim they are. This usually involves a username and a password.

The software structure adopted to realize this mechanism is illustrated by the UML class diagram of Figure 6.9 and adopts the Proxy [73] design pattern. Indeed, the **RoomAccessor** provides a factory for room proxies. For each existing room there is exactly one corresponding **RoomAccessor** registered in a remote lookup registry or service. The **RoomAccessor**'s **checkAnswer()** method provides clients of the room it represents with an appropriate **RoomSecurityProxy** depending on how the challenge was solved.

Figure 6.9: Pattern used for the security mechanism

The **RoomAccessor**'s **getQuestion()** method returns an instance of a **Question** implementation class. One can see on Figure 6.10 that the framework offers two default kinds of questions represented by two[10] lightweight classes: *(i)* **EmptyQuestion** is an empty implementation of the **Question** interface whose **execute()** method simple returns **null**; *(ii)* **PasswordQuestion** represents the simple challenge asking for a password. It fulfills its task by invoking the **getPassword()** method of the solver it receives as parameter.

The framework also contains a **Solver** class, which contains one method per challenge supported by the security system. This class simply provides dummy implementations of each method, i.e. simply returning **null**. This class is intended to be refined, and some methods overridden in order to provide correct solutions to the proposed challenges.

---

[10]In fact three subclasses are depicted but the RSAQuestion class is not part of the framework. It will be discussed later.

Typically the avatar will need a smart **Solver** class which either asks the human user to type a password or provides the solution of the question autonomously.
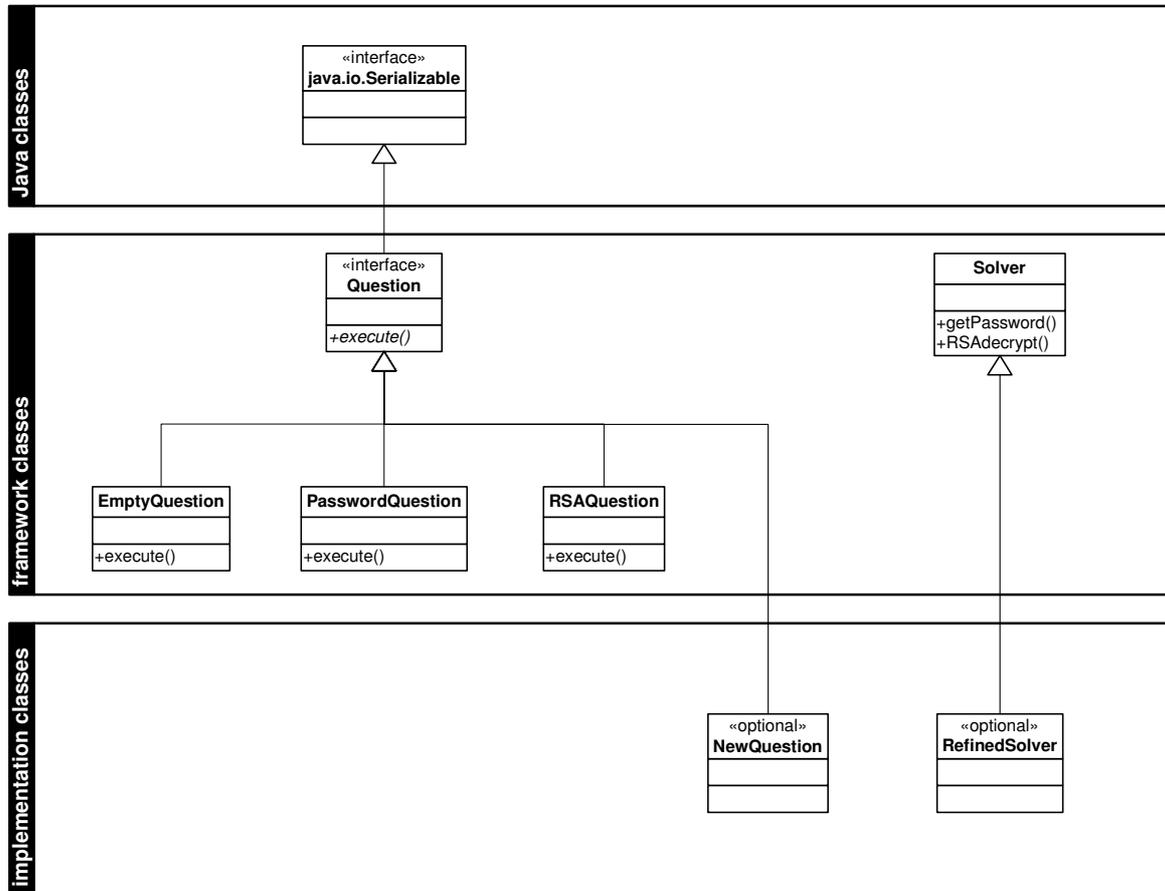
Figure 6.10: Challenge-response classes relationships

The sequence diagram of Figure 6.11 illustrates in greater detail the different steps an avatar has to pass to gain access to a room. The room accessor sends a **Serializable Question** to the avatar. The avatar locally solves the question through its **Solver** and receives an answer. The answer is serialized and sent back to the room accessor, which can check it for correctness and create a proxy for the room with the corresponding access rights. This proxy is actually a remote-secure proxy for the room, and is returned to the avatar, which now has a handle for the room.

Note that the communication channel between the avatar and the room accessor may not be secure and some malicious individual could intercept the answer sent by the avatar. Thus sending a password in plain text over this channel clearly represents a security hole. To thwart such kind of attacks a more sophisticated challenge-response must be proposed. An asymmetric (public key - private key) cryptographic algorithm like RSA[11] could be employed to achieve this goal.

Enhancing the **MaDViWorld** framework with such a new authentication process can be done in two simple steps: *(i)* add a new method to the **Solver** which could be named

---

[11]The RSA algorithm was first described in 1977 by Ronald Rivest, Adi Shamir and Leonard Adleman [160]; the letters RSA are the initials of their surnames. The interested reader will find a comprehensive discussion of this algorithm in [133].
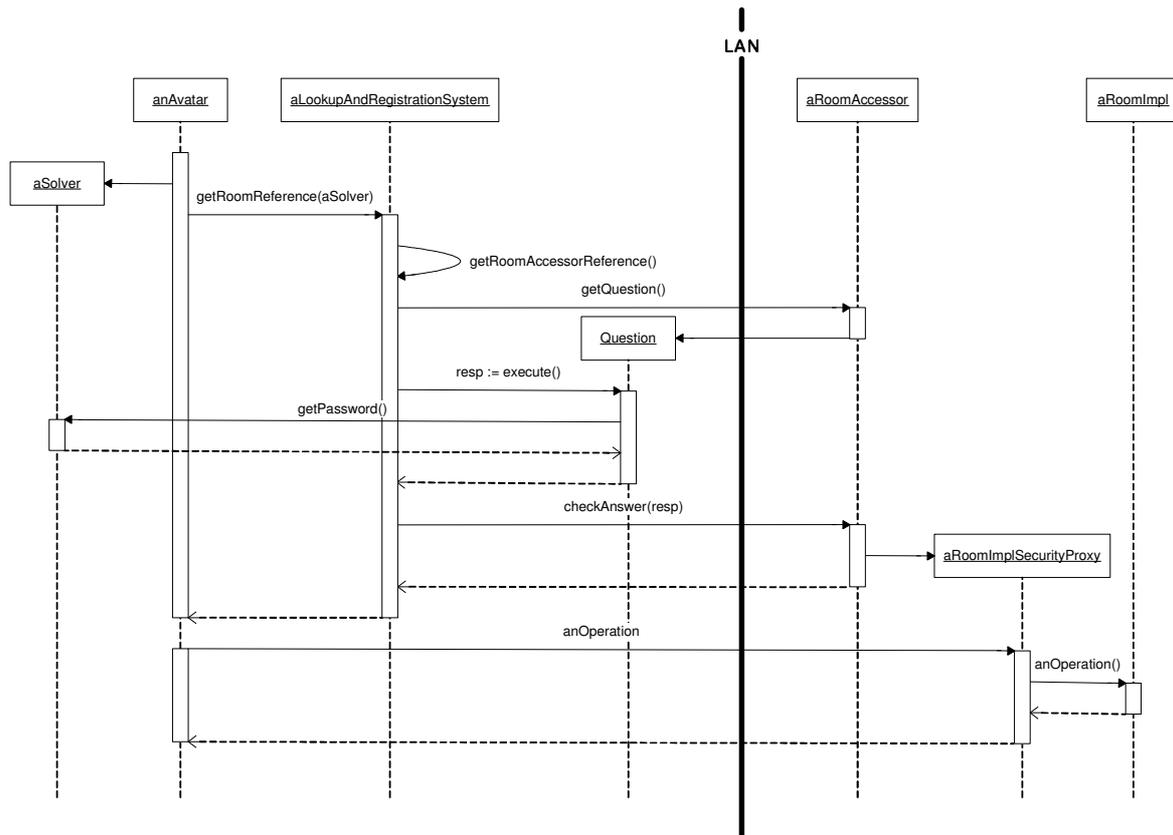
Figure 6.11: An avatar getting a secure room proxy

RSAdecrypt() and *(ii)* provide a corresponding subclass of **Question**, for instance **RSAQue-stion**. The new **RSAdecrypt()** method should be able to manage a key ring to successfully pass the challenges proposed by the different rooms.

Because the security is a difficult topic that may require some experimentation to get right, the security policy of a room is centralized in a single subclass of **Question**. This allows the framework user to easily try different policies if the existing proves inadequate. Another benefit of the explained architecture is that each room manages its security policy independently allowing for a completely distributed implementation with no central security authority. At installation time, the user who creates the room can choose and parameterize its security policy. Thus we have a simple but flexible and powerful security model.

6.2.4 Object Structure

Objects occupy a special place in the distributed virtual world. At the user level, they aim to resemble as much as possible objects of the real world in terms of mobility. At the programmer level, objects are the main hot spot of the framework, since adding a new type of object is the most obvious way to customize an existing virtual world. This subsection explains the extension mechanism and the software design of the object related classes. The next subsection concentrates on the problem of the mobility.

Objects must offer a graphical user interface (GUI) to the avatar who wants to use them. As the avatar and the object generally run on different computers, the GUI of the object

must be executed on the avatar's host and remotely interact with the application logic of the object. To achieve this, the design pattern of Figure 6.5 is adopted. This design pattern fosters a clean separation between the presentation and the application logic.

Thus, when a developer wants to add a new object **NewObj** to the framework she has to separately provide[12]:

- the classes supporting the *logic* of the object (see Figure 6.12). This is done by offering a class (**NewObjImpl**) implementing the abstract **WObjectImpl** framework class;

- the classes dedicated to the *presentation*, by extending **WObjectGUIImpl** (see Figure 6.13). This graphical class essentially serves as a graphical container of the **JPanel** subclass **NewObjPanel**. Hence the latter can directly be designed with any IDE[13].

- The object's pure functionality, expressed via the methods of its **NewObj** interface. This interface is the coupling point between UI code and functionality code.

One advantage of this architecture, in which UI and functionality are loosely coupled, is that multiple UIs can be associated with the same object. Associating multiple UIs with one object lets you tailor different UIs for clients that have particular UI capabilities, such as Swing or speech. Clients can then choose the UI that best fits their user interface capabilities. In addition, you may want to associate different UIs that serve different purposes, such as a main UI or an administration UI, with an object.

However this clean separation does not provide a two-way communication channel between these two parts. The aggregation relationship between the **NewObjPanel** class and the **NewObj** class provides a one-way communication channel (from the UI to the logic), but the logic cannot send information back to the UI. The distributed event model presented in Subsection 6.2.2 fills this gap.

Indeed, the UI will register the **NewObjRemoteEventListener** depicted on Figure 6.13 to the logic part of the object, which extends **RemoteEventProducer** (see Figure 6.7). This allows the object logic to easily notify the remote event listeners of the object's presentations. In this way, an object's logic part does not have to care about the presentation's implementation details. Furthermore, an arbitrarily number of UIs can be attached to a single logic simultaneously. Thus, one has a solution which allows a given object to be shared by several avatars using it at the same time.

The sequence diagram of Figure 6.14 dwells on the mechanism that allows the avatar to get a GUI to a remote object, thus elucidating the role of the **UIFactory**[14]. This mechanism was inspired by one of the first successes of the Jini.org Jini Community Process, the ServiceUI project [188, 191], led by Bill Venners of Artima Software. The ServiceUI API enables multiple user interfaces to be associated with a single Jini service, allowing the service to be accessed by users with varying preferences and accessibility requirements on computers and devices with varying user interface capabilities.

---

[12]For detailed instructions about how to create a new type of object the reader is invited to consult the MaDViWorld Object Programmer's Guide on the project's web site [64].

[13]IDEs (Integrated Development Environment) are graphical user interface programming environments (often called GUI Builders) that allow you to interactively build graphical interfaces, that help you to edit the code and execute your applications.

[14]To allow the UIFactory to return a concrete GUI, some resources (e.g., sound files, icons, etc.) may need to be downloaded. For sake of simplicity, Figure 6.14 does not show how these resources are transferred.
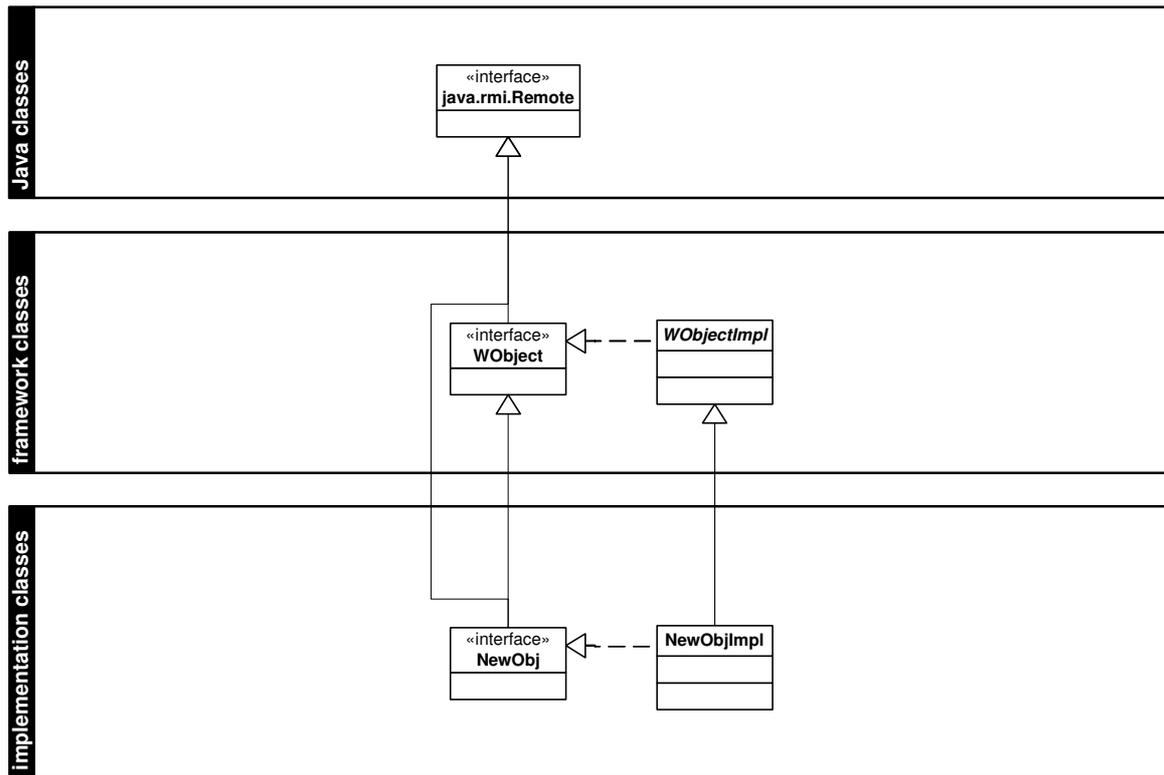
Figure 6.12: Implementation of the logic part of an object

6.2.5 Object and Code Mobility

In order to support mobile objects in a distributed virtual world the underlying technology must offer code mobility. Mobile code is one of the main characteristics of the Java language, which was conceived for the network since its beginnings. Java code mobility is achieved by dynamic classloading, which is used for applets and which is the cornerstone of the RMI mechanism. The main idea is that the platform independent bytecode of an object is downloaded from a remote HTTP server whose reference, called codebase annotation, is obtained thanks to the serialization[15] process [181, 85, 56]. This elegant solution has the drawback that during the whole object lifetime, the codebase annotation remains the same and always points to the original HTTP server. This becomes a serious issue in the context of virtual worlds where objects may frequently hop from room to room.

Indeed the object mobility depends on the availability of the original HTTP server (see Figure 6.15). The latter may crash or simply not exist any more or be connected with a slow network and the object can no longer move. There would be a single point of failure.

A solution better suited for a distributed virtual world architecture should enforce that there is a running HTTP server per object host and should allow the codebase annotation of an object to change during its lifetime, thus always pointing to the HTTP server of its current host. This is the strategy adopted by MaDViWorld and the resulting object

---

[15]Serialization is the process of converting a set of object instances that contain references to each other into a linear stream of bytes. Serialization is the mechanism used by RMI to pass objects between JVMs, either as arguments in a method invocation from a client to a server or as return values from a method invocation.
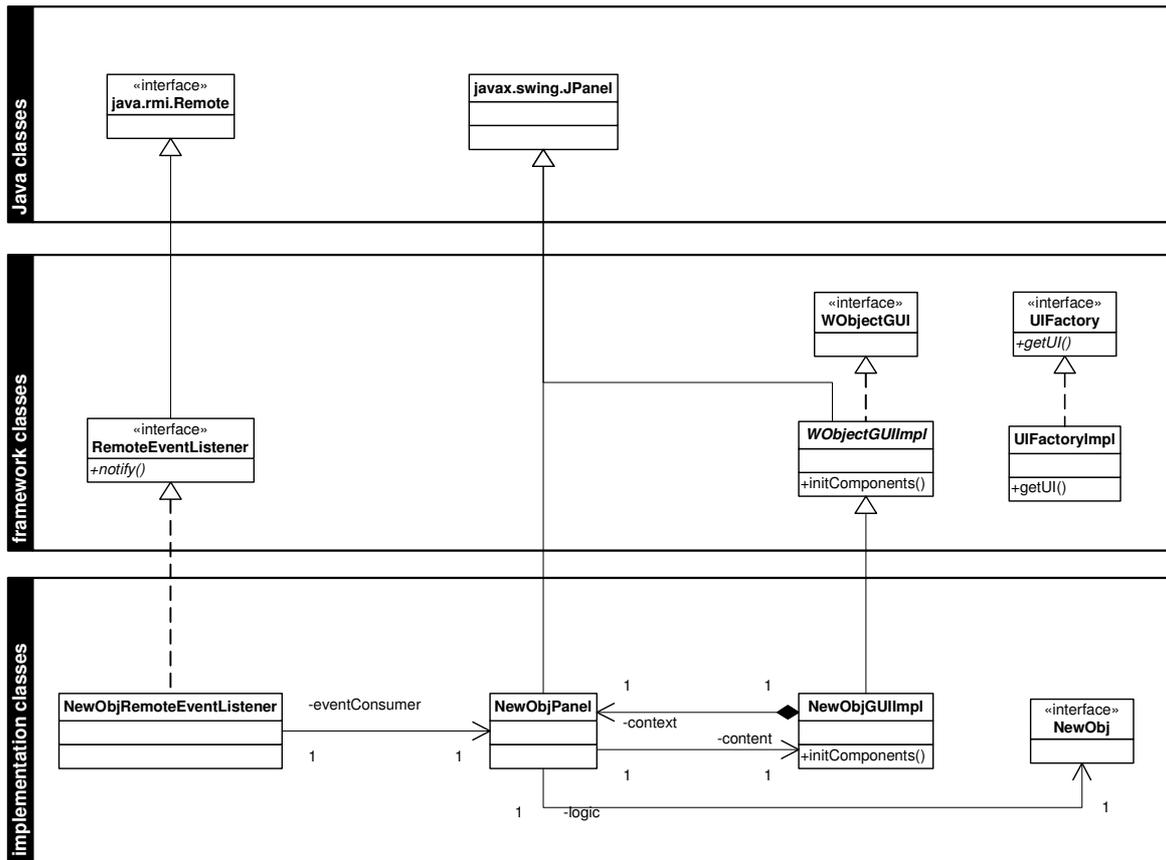
Figure 6.13: Implementation of the presentation part of an object

mobility is depicted in Figure 6.16.

In order to be deployed, each object is bundled in a single JAR[16] file containing all of its class files and auxiliary resources[17]. Packaging all files in a single archive allows for transmitting all needed stuff in a single transaction and reduces bandwidth requirements (see Subsection 3.2). The JAR file is downloaded to the destination host where it is decompressed. As the classes are now available on the object's new host, a new instance can be created with the appropriate codebase annotation. Special care has to be taken to copy the state of the original object into the new one. These operations are done under the hood and the framework user does not need to care about them at all. The details about how the transfer of the objects' files is achieved and how the state is preserved are out of the scope of this thesis but the interested reader can directly consult the source code from the project's official website [64].

6.2.6 Persistence

As already mentioned in Subsection 3.7 persistence is a key feature in distributed virtual world environments. Indeed, in every distributed application plenty of runtime errors can occur and there must be a mechanism to gracefully recover from them. The MaD-ViWorld framework includes a lightweight albeit satisfactory persistence mechanism for

---

[16] JAR (Java ARchive) is a platform-independent file format based on the popular ZIP file format.
[17] A very similar approach is used to deploy EJB [140] components on an application server.
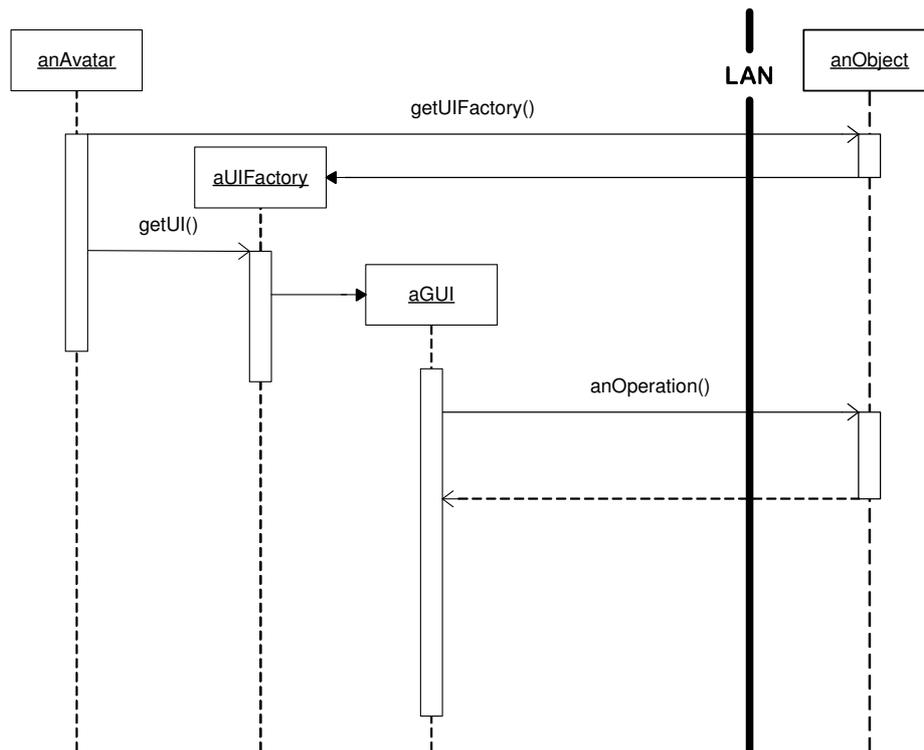
Figure 6.14: An avatar getting a GUI to an object

the room servers. In order to manage the data relevant to the state of the virtual world, these applications essentially take advantage of the activation system and of the object serialization facilities offered by Java.

The class diagram of Figure 6.17 shows that the room factory and the room accessors 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. The activation daemon, **rmid**, manages the execution of activatable remote objects. For each activatable object or RMI service, we need a setup class whose job is to to create all the information necessary for the corresponding activatable class, without necessarily creating an instance of the remote object. The setup class for the **RoomFactory** is the **ActiveSetup** class and the **RoomFactory** class is responsible for the setup of the rooms it will manage. These setup classes have to carefully set the properties of the activation group and the mechanism to preserve state information between activations has to be done properly [181]. The room server and each room store their state in local files, using the Java Serialization API [81]. Persistent data for rooms include the objects they contain with their respective states as well as a list of the avatars currently visiting them. After a software or hardware failure, the room server recovers by simply restarting the **rmid** activation daemon. Thus **MaDViWorld** adopts a distributed data architecture (see Figure 3.3) with data distributed among the different room servers hosting the rooms forming the virtual world. There is no need for a single host to have full knowledge of the world, and each one ensures the persistence of the part of the world it handles.

Persistence for the room server and rooms is encouraged and supported by the framework classes. On the other hand, there are no constraints for the persistence of avatars. Each framework user is free to implement her own persistence mechanism for her avatar
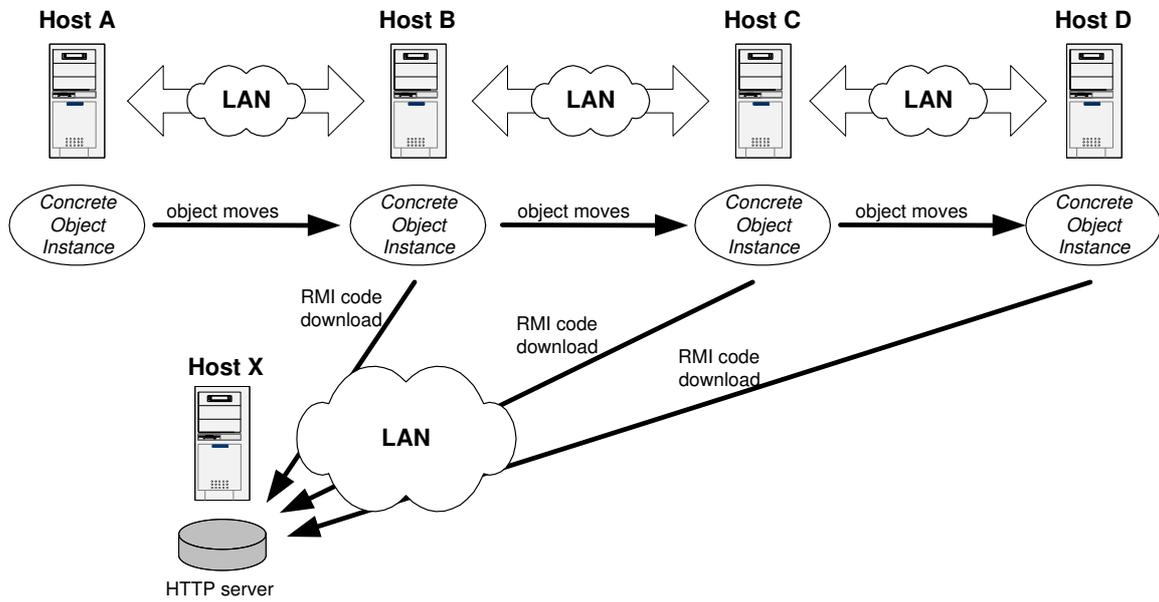
Figure 6.15: Classic Java code mobility
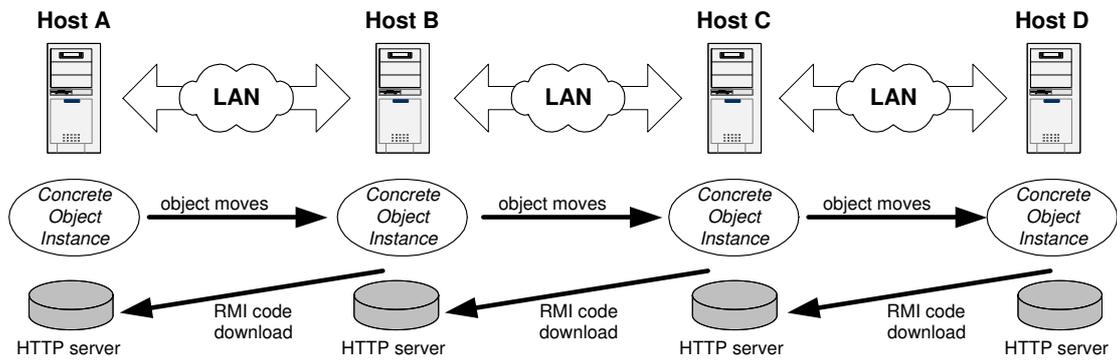
Figure 6.16: Code mobility for objects in MaDViWorld
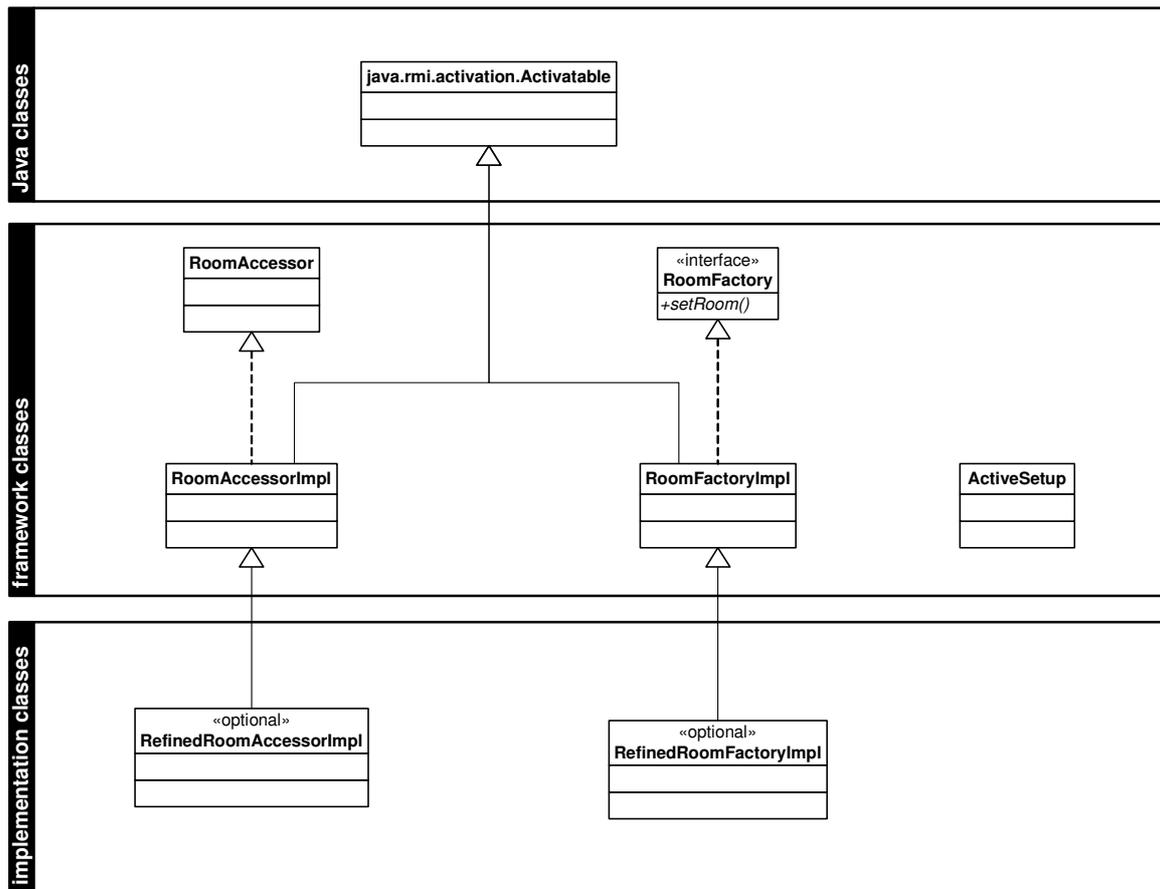
Figure 6.17: Persistence of room servers and rooms

applications. An avatar walks through the world, collects some objects that it puts in its bag and meets other avatars. For evident commodity reasons, the avatar's history and the objects contained in its bag are data the user should be able to store and retrieve afterwards. The default avatar application simply serializes the objects and stores this information to a local file on the user's demand.

7

More about Objects

A successful [software] tool is one that was used to
something undreamed of by its author.
—S. C. Johnson

This chapter is entirely devoted to the framework's most important hot spot, namely the objects. MaDViWorld's open architecture allows a developer with average Java knowledge to create new objects following some simple design principles and accepting only a few restrictions. In counterpart the developer benefits of all the advantages of mobility, remote execution and virtual 'plug-and-play' of her objects. Until now the projects lead by the MaDViWorld community (see Appendix B) contributed to create a collection of more than twenty different objects as well as a detailed object programmer's guide. The first section of this chapter contains some general reflections about objects and their utility. The middle part briefly describes some existing objects and the concluding section compares MaDViWorld objects with software agents. Thus this chapter shows the potentialities of the framework and hints promising applications for the future.

7.1 General Aspects

First, a simple scenario aims to recall the reader of the strengths of the objects in a virtual world and of the central role they play. Second, a short reflection allows to identify some technical and functional characteristics objects may have.

7.1.1 A Typical Scenario

Let us consider a typical utilization scenario of the MaDViWorld applications. This example is less general and more related to objects than the introductory scenario discussed in Subsection 4.1.1. The starting point is a virtual world composed of two rooms, R1 and R2, hosted on two different machines. Let us comment, step by step, the scenario illustrated by Figure 7.1:
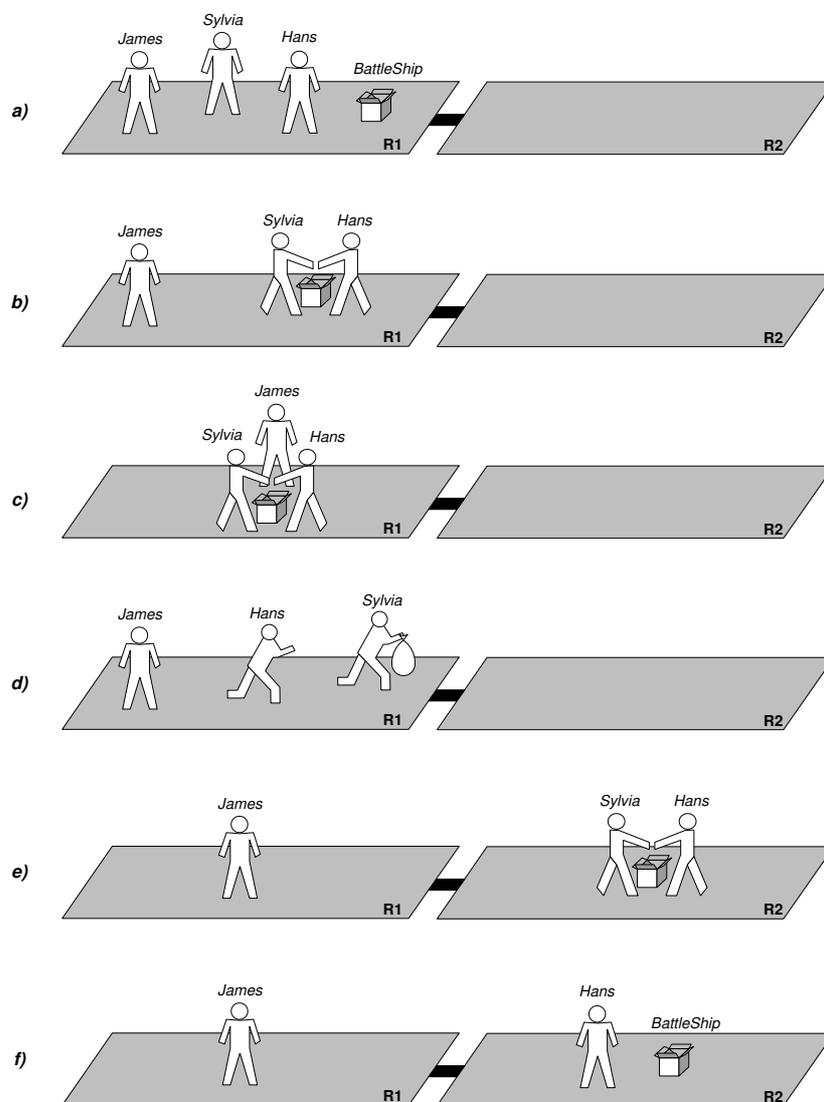
Figure 7.1: Example of object usage in MaDViWorld

a) Figure 7.1a): The virtual world is shared by three avatars: James, Sylvia and Hans, all present in the same room R1. There is a battleship game in this room.

b) Figure 7.1b): Sylvia and Hans both launch the battleship game and start playing it (see Figure 7.4).

c) Figure 7.1c): James also launches the battleship game. As it is a two players game, he becomes an observer of the game and can only watch how his two roommates play.

d) Figure 7.1d): Sylvia and Hans decide to finish their game in room R2. Sylvia takes the battleship object and puts it in her bag.

e) Figure 7.1e): Sylvia and Hans move to the empty room R2. Sylvia puts the game she had in her bag into the room. Then both Hans and Sylvia launch the game again and go on from the point they stopped before. James is now alone in room R1.

f) Figure 7.1f): The game is finished and Sylvia logged off the world. James and Hans are still inhabiting the world, each in a different room.

Although very simple, the preceding story reveals several interesting points:

- The remote event mechanism plays an important role at two levels in the scenario. On the one hand, thanks to it, the avatars are aware of their environment. James immediately knows that Sylvia and Hans left the room. Hans sees when Sylvia puts the battleship object in room R2. On the other hand, the event mechanism is used to update the graphical user interface of the objects. This allows each move to be displayed immediately on each logged avatar's board, player or observer (see Figure 7.4).

- The battleship object has "physically" been carried from room R1 to room R2 by the avatar Sylvia. It is worth noticing that R2 is hosted by another machine than R1 and that the machine hosting R2 had no prior knowledge of this kind of object.

- The state of the game has not been lost during its transfer from R1 to R2.

7.1.2 Lessons Learned

The battleship object used in the scenario of Subsection 7.1.1 has several interesting characteristics but does not demonstrate all the potentialities of the framework. Indeed, the object developer has a lot of possibilities and when programming an object she is free to decide if it will:

- be *cloneable or not*, deciding if the object can be copied or moved by avatars or not.

- be *statefull or stateless*, determining if the internal state of the object is carried around when the objects moves or not.

- be *single-user or multi-user*.

- take advantage of the distributed event mechanism to have a reactive GUI (*intra-communication*) or to communicate with the other objects located in the same room by multicasting events (*inter-communication*).

The already existing objects that are portrayed in the following section help illustrating these different options and can further be classified into three categories:

- *Collaborative Objects*: The framework offers all what is needed in order to build collaborative objects. Indeed, objects are automatically shared by several users and the events are transparently broadcasted. This allows the creation of collaborative editors (see Subsection 7.2.1) or "chat" utilities (see Subsection 7.2.2). Multi-player games are also part of this category of objects. Existing examples of multi-user games are the battleship game (see Subsection 7.2.3) or the tic-tac-toe game. The minesweeper game shown in Figure 7.2.3 is the typical example of a single-user game. It ranges in the collaborative objects category if one considers the avatars watching how someone else plays.

- *Resource Sharing Objects*: Since the objects are executed on the machine hosting their containing room, resource sharing is a possible use of objects. Objects needing a lot of computing power are put in a room hosted by a powerful computer and remotely started and driven by thin avatar clients running the object's GUI. A little example illustrating this feature is the Fibonacci number calculator (see Subsection 7.2.4). A lot of other examples can easily been imagined, for example from mathematical topics such as numerical linear algebra, fractal calculation, cryptography or linear programming solvers.

- *Communicating Objects*: The remote event mechanism model can also be used in order to make objects located in the same room communicate with each other. A possible application consists in creating so-called "social" objects (see Subsection 7.2.6). Other applications of the communication between objects are, for instance, an audio player accessing a music rack containing several music files (see Subsection 7.2.7).

7.2 Concrete Examples

This section presents in a few words a selection of existing MaDViWorld objects. Some of these examples are quite elaborated, others should rather be considered as prototypes, but they all illustrate the main capabilities of the distributed virtual world framework. As most of these objects have been developed by programmers who did not write the framework[1], they additionally validate the framework's design and documentation.

7.2.1 Paint

This object ranges in the category of collaborative software, also known as groupware[2]. Indeed, the 'Paint' [121] object implements a prototypical collaborative editor allowing

---

[1]Actually they were mainly students programming for the first time a complex Java application.

[2]The term groupware is often related with the term CSCW (Computer Supported Collaborative Work). Some people consider that both are synonyms but some authors identify a difference between these two concepts. Ellis [50] defines groupware as "computer-based systems that support groups of people engaged in a common task (or goal) and that provide an interface to a shared environment." According to Wilson [207] CSCW is a generic term, "which combines the understanding of the way people work in groups with the enabling technologies of computer networking, and associated hardware, software, services and

multiple people–represented by avatars–to remotely edit a single document simultane-
ously. The document cannot only contain text but graphic components as well (see
Figure 7.2). This basic editor can be moved by avatars from one room to another and
its state–the actual content of its documents–is conserved. An improved version of this
object, would be a full fledged collaborative text editor with coordination-aiding features
such as those found in SubEthaEdit [185] for example.
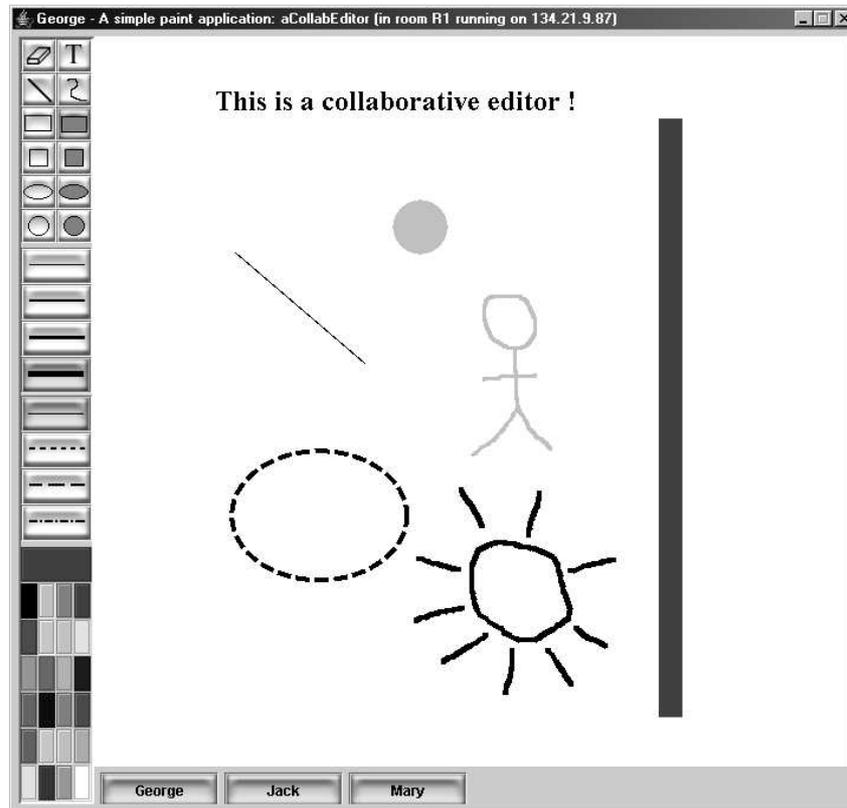
Figure 7.2: A graphic collaborative editor simultaneously used by three avatars

7.2.2 Chat

Real-time communication between avatars is facilitated with the chat object [96] which
manages connected users and presents a friendly GUI (see Figure 7.3). All basic func-
tionalities of classic IRC[3] chats clients are supported by this object. For instance, the
avatars can create discussion themes or use private chats. The avatar which launches the
chat object first, is promoted to the role of master and consequently obtains the privilege
to kick undesirable users out of the chat.

---

techniques". Thus groupware refers to real computer-based systems, while CSCW focus rather on their
psychological, social and organizational effects.

[3]Internet Relay Chat (IRC) is an open plaintext protocol designed for group communication in discussion
forums called channels, but also allows one-to-one communication. IRC is specified by the following four
RFCs [158]: 2810, 2811, 2812 and 2813.
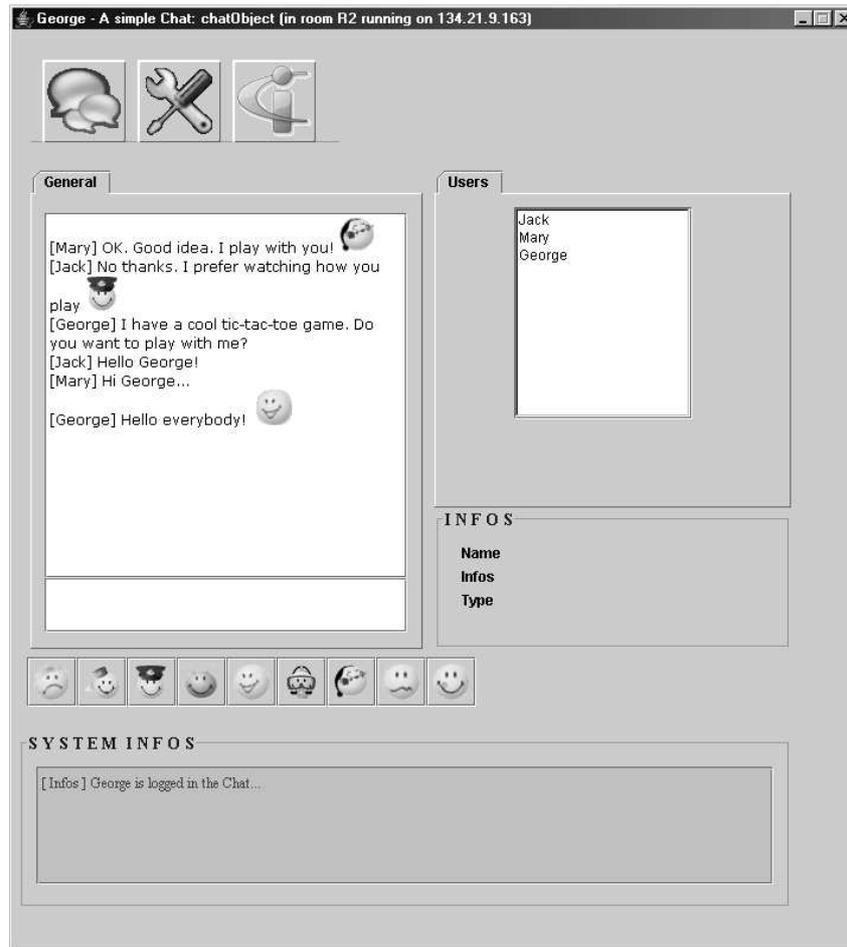
Figure 7.3: A chat object

7.2.3 Battleship

The battleship object [129] provides a MaDViWorld version of the classic two player game. It demonstrates that almost any standard Java application can be adapted in order to be integrated in a virtual world. This object has a complete GUI (see Figure 7.4) with graphic icons and some actions are accompanied by sounds. The application logic copes with players and observers administration and controls the flow of the game. An improved version of this object with some intelligence, could even let a single human user play against the computer. An analog existing MaDViWorld object offers avatars a tic-tac-toe game. Single-user games are also available, as for instance the minesweeper game [129] illustrated in Figure 7.5.

7.2.4 Fibonacci

The mobility of the MaDViWorld objects can be used to achieve sharing of computing power. An object which does intensive work and a lot of calculations can be simply deposed in a room hosted by a 'supercomputer' or a machine specially dedicated for this use. The avatar can provide input, launch the desired operations and come back later to retrieve the result. The fibonacci number calculator is a very simple object aiming at illustrating this idea: the avatar enters the index of the number of the series it wants to
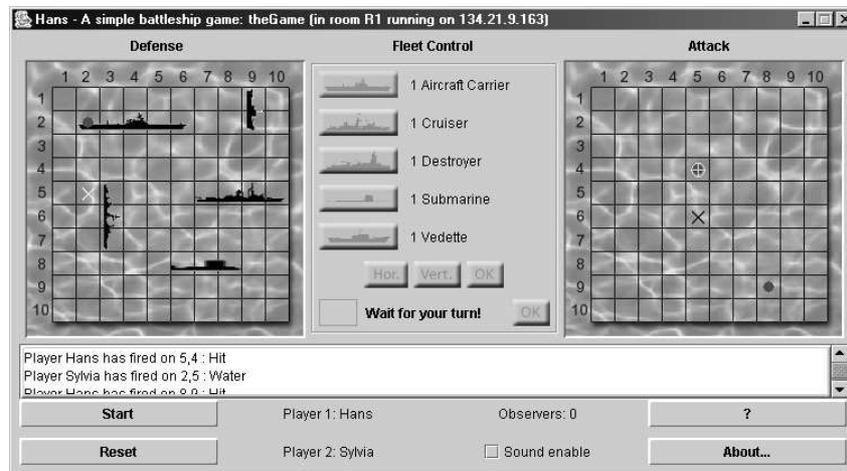
Figure 7.4: A battleship game
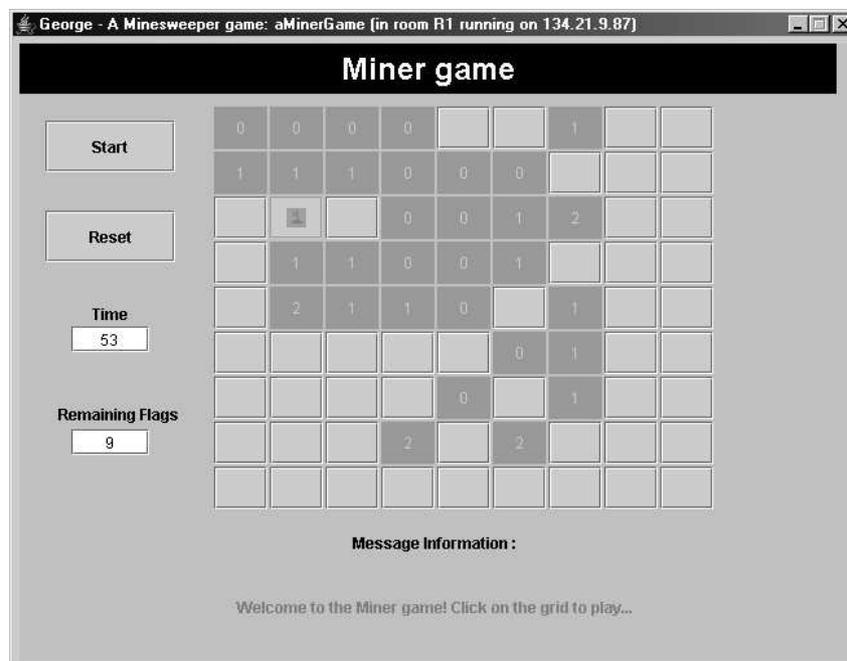
Figure 7.5: The single-user minesweeper game

know, and the result is displayed as soon as the remote object logic has computed it.

7.2.5 Clock

A typical example object demonstrating the fact that the object is using the resources of
the distant machine hosting the room in which the object is contained in, is the clock.
Indeed, if an avatar launches a MaDViWorld clock object, she will see the time of the
remote machine and not the time of the machine hosting the avatar which just runs
the clock's graphical interface. This example also highlights well that one single object
implementation can have several graphical user interfaces, since the avatar can choose
between a digital or an analogical display. Figure 7.6 illustrates a clock running on a
remote host and indicating a different time than the local system.
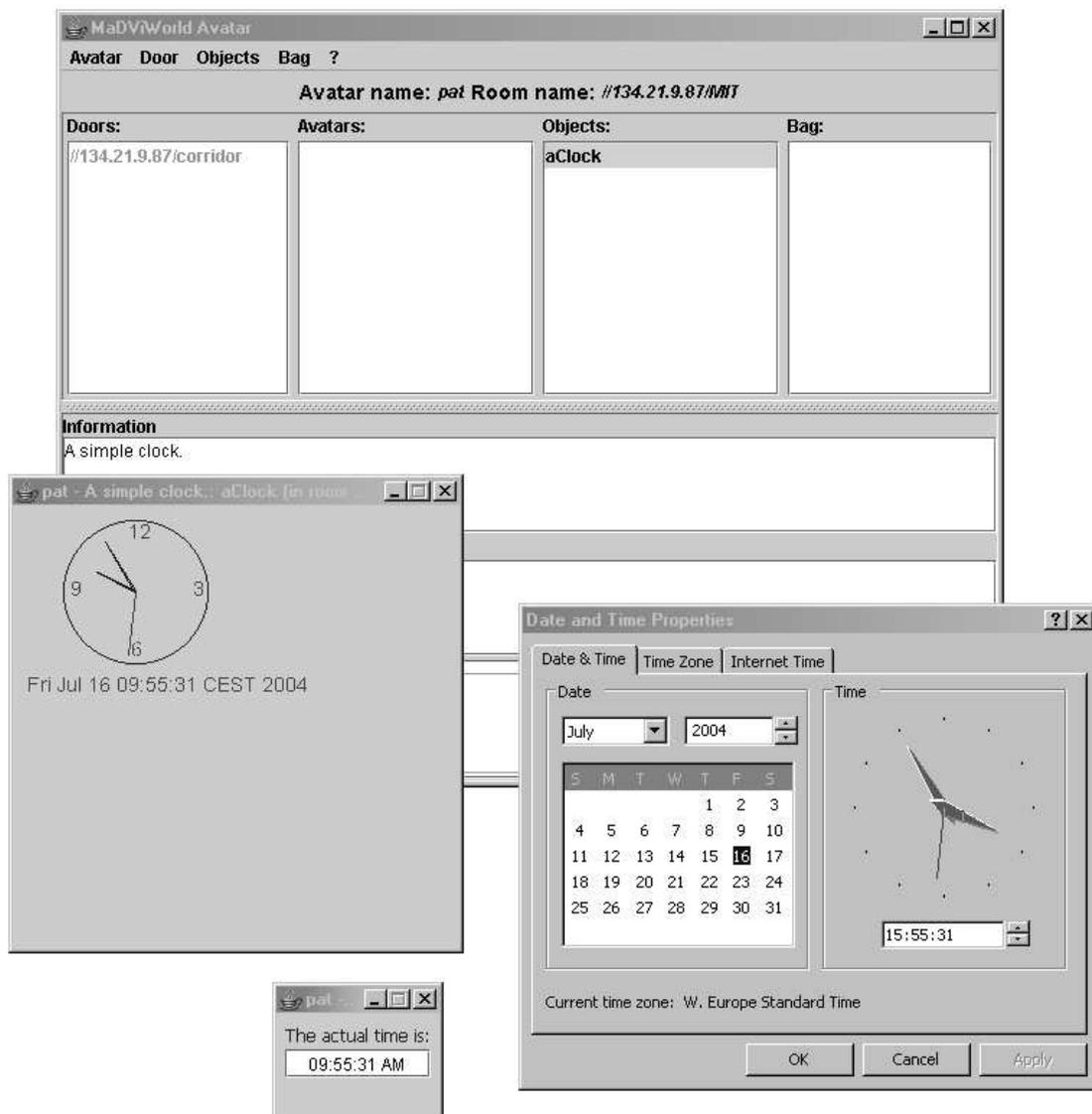
Figure 7.6: A clock object on a remote host and its two open GUIs

7.2.6 Tamagotchi

The tamagotchi object [92] is the MaDViWorld adaptation of the famous Japanese hand-held virtual pet[4]. The avatars owning these pets have to play with them, clean or feed them in order to keep them healthy and happy. These objects take advantage of the fact that objects located in the same room can communicate with each other. Indeed, if a tamagotchi dies, the other pets living in the same room are affected by the death of their friend and their "life capital" decreases in consequence. The GUI of such an object is illustrated by Figure 7.7.

Figure 7.7: The GUI of the virtual pet object

7.2.7 Musicrack and Madtunes

Another example of communication between objects is provided by the musicrack and madtunes objects [46]. The first is a little music files server and the second is a simple audio player. Madtunes asks the musicrack for the list of available songs and proposes them to the avatars which are using it. When an avatar selects an item, all avatars can hear it simultaneously. The user interface of the player is just responsible for the rendering of the sound, while the sound file is stored and decoded on the remote machine hosting the object's implementation part. This prototypical music player illustrates the multimedia capabilities of MaDViWorld. Thus objects allowing for spoken chat and videoconferencing are possible enhancements of the virtual world.

---

[4]A 31-year-old Japanese woman, Aki Maita, came up with a virtual pet idea and sold it to Bandai Corporation, one of the biggest toy manufacturers in Japan. Bandai named their product "tamagotchi" and first released it in fall of 1996. These interactive digital pets were small, plastic eggs containing a tiny computer with a simple black and white screen with three buttons below it. The egg was attached to a keychain, to encourage owners to always keep their tamagotchi close by. The word literally means "loveable egg" (it derives from Japanese "tamago" meaning "egg" and "chi" as a term of endearment).

7.2.8 Matchmaker

All the other objects, even if they present some degree of autonomy, do not move by themselves from one room to another. The feasibility of this feature is proven by the matchmaker object [55]. It also shows a practical application of such objects in a distributed virtual world. Indeed, a matchmaker works on behalf of a given avatar trying to fix an appointment with another avatar. To achieve its goal, a matchmaker object travels from room to room, searching for another matchmaker object. As soon as it finds one, it communicates with it and negotiates a time and a place (i.e. a room) the avatars they are working for agree to meet at. If the negotiations succeed, the matchmaker objects will inform their respective employers of the fixed meeting. Otherwise, they continue travelling around the world looking for other candidates. This can be very helpful when an avatar wants to find someone to play with. Objects that move autonomously through a distributed virtual world can also help collecting useful information in order to draw a map for instance. The explorer object, which is part of the MaDViWorld project, serves this goal.

7.3 Comparison with the Agent Paradigm

At this point, we will disgress a little bit and delve into the world of agents. Indeed, the reader could ask herself if the objects in our distributed virtual world cannot be qualified as software agents. Software agents represent a broad domain and there are entire books dedicated only to them. Therefore this section only gives some key elements helping to compare virtual world objects and software agents.

7.3.1 What Is an Agent?

The word *agent* has found its way into a number of technologies. It has also been applied early to constructs, which were developed for improving the experience provided by collaborative online social environments like MUDs and MOOs.

There is a plethora of definitions for software agents (see for instance [62], [209], [128] or [167]) and there is no consensus on a generally accepted precise definition. A thorough, well-thought-out classification scheme is given by [62], where one can also find a very broad definition[5], which is unfortunately too large to be useful as is. But even if there is no general agreement as to what constitutes an agent, or as how agents differ from programs, we can provide a list of characteristics that have been proposed as desirable agent qualities (see [26], [62], [210] and [52]):

- reactive: responds in a timely fashion to changes in the environment. In other words, an agent senses its environment and acts upon it,

- autonomous: is able to take initiatives and exercises a non-trivial degree of control over its own actions,

---

[5] "An *autonomous agent* is a system situated within and a part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to effect what it senses in the future."

- goal-oriented: does not simply act in response to the environment (pro-active),

- temporally continuous: is a continually running process, not a "one-shot" computation that maps a single input to a single output, then terminates,

- communicative: communicates with other agents, perhaps including people, in order to obtain information or to enlist their help in accomplishing its goals (socially able),

- collaborative and/or competitive: cooperates and/or competes with other agents in pursuit of common goals,

- adaptive: changes its behavior based on its previous experience and automatically adapts to changes in its environment (learning),

- mobile: able to transport itself from one machine to another (possibly across different system architectures and platforms),

- flexible: it is able to dynamically choose which actions to invoke, and in what sequence in response to the state of its external environment,

- character: has a well-defined believable "personality" and emotional state.

Every agent, by the definition of [62] satisfies the first four properties. Adding other properties produces potentially useful classes of agents, for example, mobile, learning agents.

Of course, other classifying schemes are possible. For example, software agents are often classified according to the tasks they perform and following kinds can be identified: *interface agents*, supporting the metaphor of a personal assistant, which is collaborating with the user and which employs artificial intelligence [127]; *information agents*, which have access to at least one source of information and are able to collate and manipulate information obtained from these sources to answer queries posed by users and other information agents [210]; *conversational agents* or chatterbots, which attempt to maintain a human-like[6] conversation with a person (e.g., Eliza [205] which parodies a psychotherapist, the ALICE[7] open source project [7]); *entertainment agents* like Julia [57], a chatterbot living in MUDs where it converses in natural language with other players, integrating reactivity, goals and emotion; *commerce agents* providing commercial services (e.g., selling, buying, prices' advice) for a human user or another agent; or even *computer viruses*.

Finally, from the preceding considerations it becomes clear that (software) agents do not exist by themselves but that they "live" in an environment, that we will call an *agent host*. The basic requirements of an agent host, are identified in [182]:

- An agent host must allow multiple agents to co-exist and execute simultaneously.

- An agent host must allow agents to communicate with each other and itself.

- It must be able to negotiate the exchange of agents.

- It must be able to freeze an executing agent and transfer it to another host.

---

[6]Since 1990, the Loebner Prize [120] annual competition awards prizes to the chatterbot considered most human-like for that year. The format of the competition is much that of a standard Turing test [186].

[7]A.L.I.C.E. is an acronym and stands for Artificial Linguistic Internet Computer Entity.

- It must be able to thaw an agent transferred from another agent host and allow it to resume execution.

- It must prevent agents from directly interfering with each other.

7.3.2 Are the Virtual World Objects Agents?

As explained in Subsection 7.1.2, the MaDViWorld framework allows for the creation of objects of several types. According to the criteria discussed in Subsection 7.3.1, it is also possible to create software agents.

As a matter of fact, a framework user could develop an object that is reactive, autonomous, temporally continuous, communicative and mobile by directly and effortlessly using the offered features. The object developer is then free to add goal-orientation, flexibility or learning capabilities to her object in order to build a more interesting software agent.

Among the already existing objects, the virtual pet example (see Subsection 7.2.6) could be classified as a basic agent and the matchmaker (see Subsection 7.2.8) as a more elaborated mobile agent. Indeed they both satisfy the properties required by agents.

- The tamagotchi:

  - is reactive, as it senses the death of another pet and is affected by this event.
  - is autonomous, as it starts "living" automatically.
  - is goal-oriented, with the most basic goal of just "living".
  - is mobile, as an avatar can move it from one room to another. It does not move autonomously, but it was shown that it is possible to develop objects with such a behavior (see Subsection 7.2.8).
  - is temporally continuous, as when it moves from one room to another, it continues to live at the point it was before moving.
  - is communicative, as it can inform other pets that it is dying.

- The matchmaker:

  - is reactive, as it detects the presence of another matchmaker and acts in consequence.
  - is autonomous, as it does its work automatically.
  - is goal-oriented, with the mission to fix an appointment with another avatar.
  - is mobile, as it autonomously moves from one room to another in order to achieve its goal.
  - is temporally continuous, as when it moves from one room to another, it preserves the information already collected.
  - is communicative, as it 'talks' to other objects.
  - is collaborative, as it cooperates with other matchmakers in pursuit of common goals.

The MaDViWorld platform offers several features that have to be supported by an agent architecture. However, the framework does not supply facilities helping object programmers to add cognitive characteristics, such as adaptation, learning and goal-orientation to their objects. And it seems that these are the areas (the aspect related to intelligence) that receive the most attention in the agent community.

To conclude, one could say that the main difference is the intent of the MaDViWorld objects. At the beginning, they were not developed to act like agents, but the framework happens (it is a side effect) to potentially support–at least to a certain extent–these particular kind of objects as well!

8

Conclusion

It is not the strongest of species that survive, nor the most intelligent, but the one most adaptable to change.
—Charles Darwin

This concluding chapter presents the summary of the contributions made in this dissertation and points to directions for future work.

8.1 Summary

This thesis describes the design and implementation of a software solution supporting the virtual world paradigm. To reach this goal, we first went over the origins of the Internet and the early shared virtual environments. Then, a review of some actual projects facing analog challenges lead to the identification of the main problems virtual world developers have to face.

Prior to delving into implementation tasks, a fully theoretical and general model clearly defining the concept of virtual world was provided. The main novelty of this model is a bottom-up composition of the virtual world, which contrasts with the top-down decomposition adopted by other projects.

In order to validate the model, one of its possible instantiations has been implemented from scratch. The adopted software architecture is a distributed framework, MaDViWorld, programmed in the Java language and based on the Jini distributed technology. The preferred framework approach offers high customization possibilities and allows developers to create new mobile objects intended to populate the rooms and offer appealing services to the virtual world occupants.

Special care was granted to the design and the documentation of the framework. These efforts made possible the creation of a user and developer community around the project which permitted the exploration of possible applications of the young framework such as multimedia, gaming, collaborative work or mobile agents.

8.2 Future Research

The heart of the framework can be considered as mature; it provides all the abilities to build running virtual worlds. Now before its transition to larger scale deployment, some utility classes deserve some optimizations and some new specialized services should be added.

For example, the framework could use an improved, more efficient persistence mechanism based on open technologies combining Java and XML. The state of the applications could be represented by a standard human readable persistent state format rather than in a more fragile binary representation.

Objects supporting communication between avatars in structured or non-structured form, in synchronous or asynchronous way should be supplied. These objects would go beyond the simple chat, offering facilities for file exchange or for multimedia-based interaction for example.

Other important notions one can mention are low-level network security, authentication, authorization, confidentiality, integrity and trust concerns. These problems should be carefully investigated and the new features of Jini 2.0 and its Jini JERI[1] infrastructure should help the lookup and registration utilities of the framework to properly address the issues related with security.

The major technological evolution of MaDViWorld would probably be its ability to run on the Internet. Making RMI calls across firewalls is known to be a hassle and several projects tackle this issue with variable success. But a new promising solution which is worth being investigated is proposed by the Jxta-JERI project [180], aiming to integrate Jini and JXTA[2] technologies. This approach enables services and clients to use the familiar RMI programming model, but take advantage of the JXTA peer to peer network which enables communication through firewalls and NAT[3].

Besides the technological future work, we think that the next most important step consists in creating a full fledged virtual world–a kind of 'killer application'. A concrete example demonstrating the advantages gained by using the different features of MaDViWorld, tested by a group of people in a real-world context should be developed. To reach this goal, an application domain has to be chosen and a world topology has to be appropriately defined. Two possible directions such research could take are sketched below: the first ranges in the area of entertainment and the second deals with distance learning.

Gameworld   This virtual environment consists of a set of rooms full of active collaborative game objects, ranging from single user arcade games to sophisticated multi-user ones (card games for instance). After having paid a fee, the users are allowed to visit the rooms; to watch other users play; to try out some demo versions of the games; or even to join a game and to exchange their impressions about it. Later, if she is interested, a user can even copy a given game object onto her own machine by getting the right to clone it.

---

[1]JERI is Jini Extensible Remote Invocation. JERI is a new implementation of the RMI programming model which allows high customization based on runtime configuration information. More information about JERI can be found in [190, 144].

[2]JXTA™ (*Juxtapose*) is Sun Microsystems' open source-based peer-to-peer (P2P) infrastructure. The official web page is http://www.jxta.org/ (accessed December 28, 2004).

[3]NAT is the acronym for Network Address Translation.

A slightly modified version of this world would be to replace the cloneable game objects by active pieces of art that would be unique in the sense that one could only move them around, not copy them.

Eduworld A more ambitious project is to build up a *distributed learning environment* on the top of the MaDViWorld framework. While Figure 8.1 sketches the conceptual model of such a world, its key elements are enumerated below.

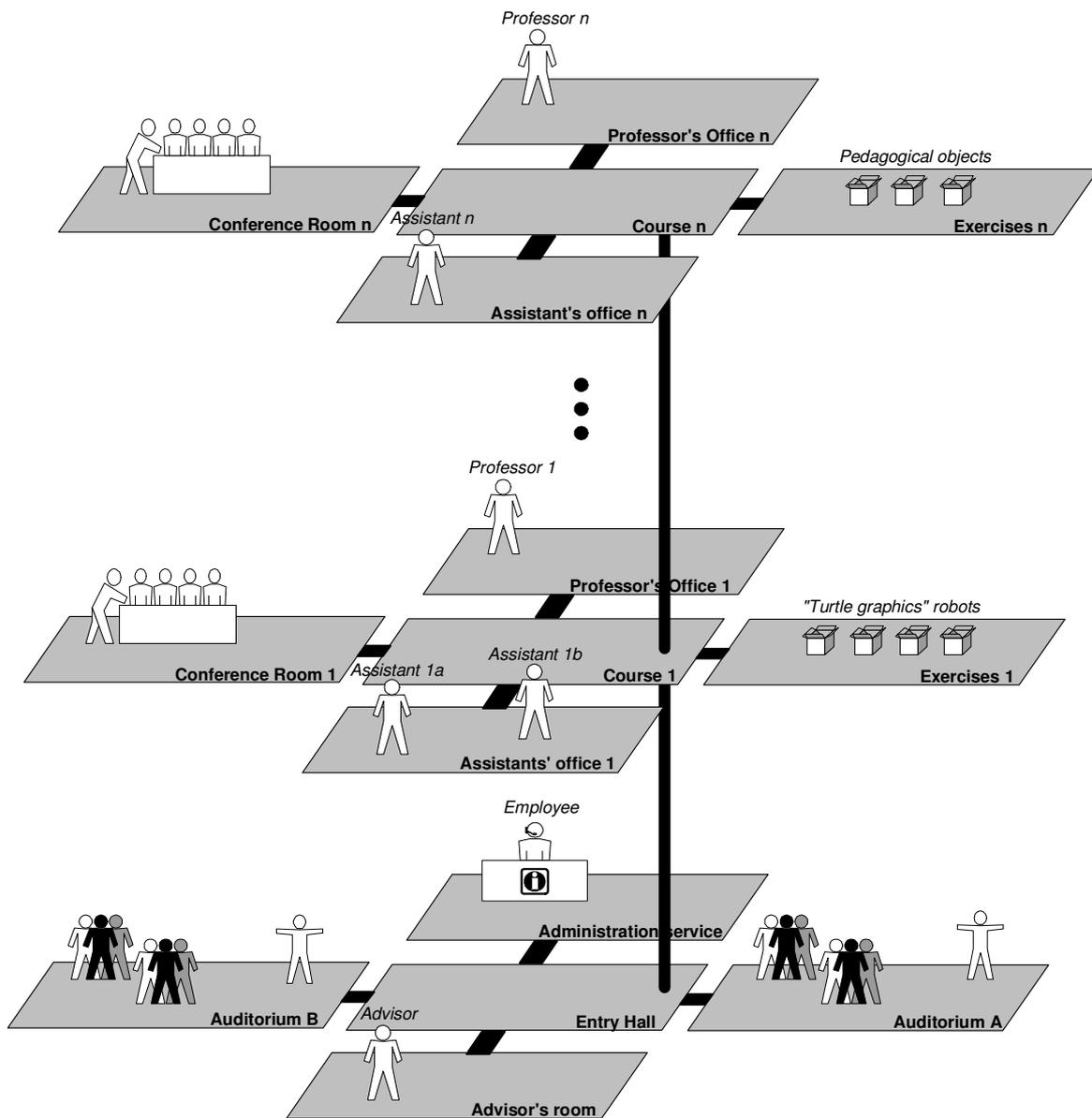Figure 8.1: Distributed learning environment conceptual model

- *Individual professors' offices* are used in order to receive students for private discussions. We propose to physically decentralize them on the professors' private machines.

- *Assistants' offices* are rooms used by the assistants of a given professor in order to receive individual students for questioning about their on-going homework. The

functions of these rooms are close to the former ones and we also propose to decentralize them.

- *Conference rooms* are associated to a professor's group and are used by both the professor and his assistants in order to have an open discussion with several students at once. They can also serve for more classical *ex cathedra* courses. These rooms can either be decentralized on a machine associated with a given professor's group or put on a larger department's server.

- *Exercises room*s are the most interesting ones, since they contain the *active pedagogical objects* associated with a given course. For instance, programmable drawing robots could be used in order to teach algorithmic concepts. This idea is analogous to the turtle graphics methodology adopted by Logo [148, 1]. Adapted to a virtual world environment such a learning strategy would lead to the following scenario. Each student clones the 'exercise of the day' robot and takes it into her virtual office, running on her own physical machine. She then tries to instruct the robot to do a given drawing. Once she is finished, the student puts her programmed robot in another room for correction (the assistants' office for instance). A reasonable solution is to put these rooms on the same server as the conference ones. They will not overload this machine, since the real work will always take place on the students' individual machines.

- Administrative rooms provide various central services (registration, accreditation, etc.) and would typically run on a larger department (or even university) server.

It is our hope that such a world or another will be built in the near future thus further validating our framework.

A

Class Structure of the Framework

This appendix provides the class structure of the MaDViWorld framework. The main classes and interfaces are enumerated and shortly described. For an exhaustive and up-to-date technical documentation the interested reader is invited to consult the official javadoc on the MaDViWorld project website [64].

A.1 Overview (MaDViWorld Framework API Documentation)

A.1.1 Packages (see Figure 6.1)

ch.unifr.diuf.madviworld.core	Central classes and interfaces defining the abstract communication protocol between the different components of a virtual world.
ch.unifr.diuf.madviworld.avatar	Sample implementation classes for the avatar.
ch.unifr.diuf.madviworld.room	Sample implementation classes for the room.
ch.unifr.diuf.madviworld.roomfactory	Sample implementation classes for the roomfactory.
ch.unifr.diuf.madviworld.setup	Sample implementation classes for the setup application.
ch.unifr.diuf.madviworld.wobject	Abstract classes dedicated to the implementation of objects.
ch.unifr.diuf.madviworld.event	Classes and interfaces dedicated to the distributed event mechanism provided by the framework.
ch.unifr.diuf.madviworld.util	Miscellaneous utility packages. The substructure of this package organizes the classes according to their role (class loaders, network administration, HTTP servers, security, logging,...). These are all system-wide, or crosscutting, concerns that span multiple classes of the framework.

A.2 The ch.unifr.diuf.madviworld.core Package

A.2.1 Interface Summary

Constants	This interface defines all the constants used throughout the framework.

WContainer	This remote interface defines the container concept of MaDViWorld. It defines a few methods that are common to avatars and rooms, which both can contain objects. It is a remote interface
Avatar	This interface extends the WContainer interface but has no methods or fields and serves only to identify the semantics of being an avatar.
Room	This remote interface extends the WContainer interface and defines all the methods a room must provide.
RoomAccessor	This remote interface provides methods to get access to a room with the appropriate access permissions (see Subsection 6.2.3).
RoomFactory	This remote interface defines all the methods a roomfactory must offer to allow a client setup rooms.
WObject	This is the interface for the implementation part of any object developed with the MaDViWorld framework (see Subsection 6.2.4).
WObjectGUI	This is the interface for the graphical user interface part of any object developed with the MaDViWorld framework (see Subsection 6.2.4).
UIFactory	This is the minimal interface of a graphical user interface factory for MaDViWorld objects (see Subsection 6.2.4).

A.2.2 Class Summary

DUID	Distributed Unique ID. Pair an instance of InetAddress with an instance of UID.

A.2.3 Exception Summary

MaDViWorldException	General purpose subclass of java.lang.Exception used to wrap generic exceptions that can occur while working with the framework.

A.3 The ch.unifr.diuf.madviworld.avatar Package

A.3.1 Interface Summary

LAvatar   This interface defines all basic operations
                                        that avatars need to implement.

A.3.2 Class Summary

AvatarImpl   This class provides a default implementation
                                        for an avatar.

AvatarSolver   This subclass of the generic Solver is respon-
                                        sible for answering the questions granting se-
                                        curity permissions for the avatar (see Subsec-
                                        tion 6.2.3).

A.3.3 Exception Summary

RoomDoesNotExistException   This exception is thrown when a room is re-
                                        quired but cannot be found.

RoomAccessException   This exception is thrown when an avatar
                                        tries to enter into a room and it does not
                                        have sufficient access permissions.

A.4 The ch.unifr.diuf.madviworld.room Package

A.4.1 Class Summary

RoomImpl   This class provides a default implementation
                                        for a room.

RoomRemoteEventListener   This class implements RemoteEventListener
                                        and reacts appropriately to the received
                                        events.

RoomAccessorImpl   This class implements the methods to get ac-
                                        cess to a room and grants the appropriate
                                        permissions.

RoomImplSecurityProxy   This class implements a security proxy con-
                                        trolling the access of a given room's methods.

RoomSolver   This subclass of the generic **Solver** is responsible for answering the questions granting security permissions for the room (see Subsection 6.2.3).

A.4.2 Exception Summary

MethodAccessException   This exception is thrown when a method of the room is invoked and the client does not have enough access rights.

A.5 The ch.unifr.diuf.madviworld.roomfactory Package

A.5.1 Class Summary

RoomFactoryImpl   This class provides a default implementation for a room factory.

ActiveSetup   This is the setup class for the activatable **RoomFactoryImpl** (see Subsection 6.2.6).

RoomFactorySolver   This subclass of the generic **Solver** is responsible for answering the questions granting security permissions for the room factory (see Subsection 6.2.3).

A.6 The ch.unifr.diuf.madviworld.setup Package

A.6.1 Class Summary

RoomSetupLogic   This class provides a default implementation part for a room setup application.

RoomSetupSolver   This subclass of the generic **Solver** is responsible for answering the questions granting security permissions for the room setup application (see 6.2.3).

A.6.2 Exception Summary

RoomSetupException   This generic exception is thrown whenever an error with the room setup application occurs.

A.7 The ch.unifr.diuf.madviworld.wobject Package

A.7.1 Class Summary

WObjectImpl   This abstract class defines the general imple-
                                     mentation part of any object (see Subsection
                                     6.2.4).

WObjectGUIImpl   This abstract class provides a general user
                                     interface part for an object (see Subsection
                                     6.2.4).

UIFactoryImpl   This class is responsible for the creation of
                                     WObjectGUI for the objects (see Subsection
                                     6.2.4).

A.7.2 Exception Summary

NotCloneableException   This exception is thrown when one tries to
                                     copy an object which is not cloneable.

A.8 The ch.unifr.diuf.madviworld.event Package

A.8.1 Interface Summary

RemoteEventListener   This interface defines all the methods an
                                     avatar must offer (see Subsection 6.2.2).

RemoteEventProducerLocal   This interface defines all the methods an
                                     event producer must provide for notify-
                                     ing the interested listeners (see Subsection
                                     6.2.2).

RemoteEventProducerRemote   This remote interface defines all the methods
                                     allowing interested clients to register event
                                     listeners (see Subsection 6.2.2).

RemoteEventProducerTemporal   This interface defines a method for getting
                                     the time in the event producer's referential
                                     (see Subsection 4.2.4).

A.8.2 Class Summary

RemoteEvent

This is the base class for remote events (see Subsection 6.2.2).

RemoteEventNotifier

This class is responsible for notifying an event to a listener on behalf of an event producer in a short-lived thread (see Subsection 6.2.2).

RemoteEventProducerImpl

This class manages the registered listeners and is responsible for notifying them with events (see Subsection 6.2.2).

A.9   The ch.unifr.diuf.madviworld.util Package

A.9.1   Interface Summary

federation.LookupAndRegistrationStrategy

This interface defines all the methods a lookup and registration strategy must provide (see Subsection 6.2.1).

security.Question

This interface represents a question or challenge of the security mechanism (see Subsection 6.2.3).

A.9.2   Class Summary

administration.Administration

This class allows to programmatically setting up a Jini federation. It provides static methods to start HTTP server, shared VM, lookup services, and other framework specific services.

classloader.DownloadedClassLoader

This class extends the standard java.lang.ClassLoader and allows to load java classes from a directory on the local disk which is not in the java classpath, in order to avoid the "lost codebase annotation problem" of RMI [132].

federation.JiniLookupAndRegistration

This class is a Jini-based implementation of the LookupAndRegistrationStrategy interface (see Subsection 6.2.1).

federation.RMILookupAndRegistration

This class is a classic RMI-based implementation of the LookupAndRegistrationStrategy interface (see Subsection 6.2.1).

federation.LookupAndRegistrationSystem	This class represents a composite of several LookupAndRegistrationStrategy implementation (see Subsection 6.2.1).
federation.JiniManager	This utility class allows to register and lookup remote objects in a Jini lookup service (see Subsection 6.2.1).
federation.RMIManager	This utility class allows to register and lookup remote objects in the standard rmiregistry (see Subsection 6.2.1).
logging.LoggingService	This class implements a standard java logging service.
security.Solver	This class defines all the methods that the Question implementation can invoke (see Subsection 6.2.3).
security.EmptyQuestion	This class is a trivial implementation of the Question interface (see Subsection 6.2.3).
security.PasswordQuestion	This class implements the Question interface asking for a password (see Subsection 6.2.3).
wclassserver.ClassFileServer	This class provides a simple HTTP server for java class files.
fileserver.AllFileServer	This class provides a simple HTTP server for any kind of files. In the framework it is used to provide access to resources files (sounds, images,...).

B

The MaDViWorld Community

The **MaDViWorld** project (see Figure B.1) started in the fall of 1999 and in the spring of 2001 the first beta version of the framework was released. This initial version of the software architecture got a lot of improvements but the hot spots were clearly identified from the beginning and allowed for the creation of a small developer's community aiming to use and extend the framework. The **MaDViWorld** community is exclusively academic for the time being, since only computer science students are involved. Indeed, between fall of 2001 and summer 2004, two bachelor projects and five master or diploma theses were concerned with the **MaDViWorld** project.

Involving other people in the project presents several benefits to the **MaDViWorld** framework:

- A framework needs acceptance and validation tests in order to gain in maturity. The several projects helped improve the design and the documentation of the framework as well as adding some missing facilities.

- The portability of the framework on different platforms could be tested. Indeed, the students used several possible platforms such as Windows, Mac OS X or Linux. **MaDViWorld** proved to work well on any of them.

- The creation of a collection of objects showing the capabilities of the framework, provides a good starting point to the elaboration of a complete virtual world.

MaDViWorld's official website [64] contains all the related material including:

- A platform independent installation guide;

- The complete javadoc of the **MaDViWorld** API and of all developed objects;

- An object programmer's guide, describing step by step the task of creating a new virtual world object;

- The entry point for the **MaDViWorld**'s CVS[1] tree;

---

[1]Concurrent Versions System (CVS) is an open-source network-transparent version control system. Its official website address is **https://www.cvshome.org/** (accessed December 28, 2004).
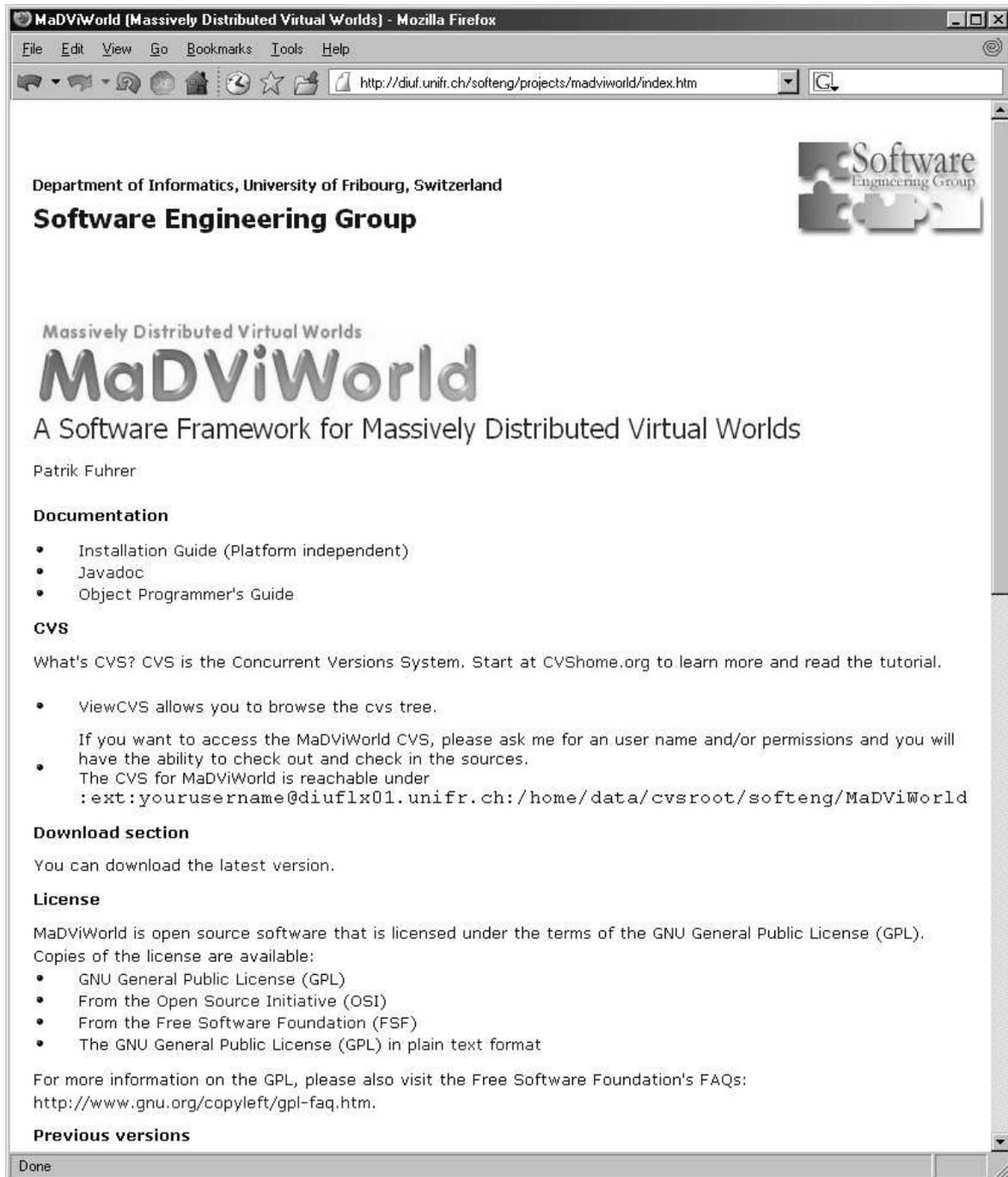
Figure B.1: **MaDViWorld** project's official website

- A download section providing the binaries and source code of the most recent version of the framework;

- All the related publications;

- References to all the student projects information.

After several development cycles, the actual version of the framework, even if some technical improvements are still possible, has reached a good level of maturity. Table B.1 concludes this appendix enumerating some simple metrics of MaDViWorld.

Criterion	Value
Total Lines of Code	74'000
Commented Lines of Code	21'000
Non-commented Lines of Code	53'000
Number of Classes	650

Table B.1: Some simple metrics of the MaDViWorld framework

C

Abbreviations

This appendix gathers some abbreviations found in this dissertation and gives their corresponding meanings:

Abbreviation	Meaning
CORBA	Common Object Request Broker Architecture
CSCW	Computer Supported Collaborative Work
CVS	Concurrent Versions System
D&D	Dungeons and Dragons
EJB	Enterprise Java Beans
FTP	File Transfer Protocol
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
IP	Internet Protocol
IRC	Internet Relay Chat
IVW	Inhabited Virtual World
JAR	Java Archive
MCG	Multi-Player Computer Games
MOO	MUD Object-Oriented
MUD	Multi User Dungeon (or Dimension or Domain)
NAT	Network Address Translation
NVE	Networked Virtual Environment
OCL	Object Constraint Language
OMG	Object Management Group
RFC	Request For Comments
RMI	Remote Method Invocation
RUP	Rational Unified Process
SOAP	Simple Object Access Protocol
TCP	Transmission Control Protocol
UDDI	Universal Description, Discovery, and Integration
UML	Unified Modeling Language
VRML	Virtual Reality Modeling Language
WSDL	Web Services Description Language
WWW	World Wide Web
XML	eXtensible Markup Language
XP	Extreme Programming
XSLT	Extensible Stylesheet Language Transformations

References

[1] Harold Abelson and Andrea A. diSessa. *Turtle Geometry: The Computer as a Medium for Exploring Mathematics*. MIT Press, September 1986.

[2] Parker Abercrombie and Murat Karaorman. jContractor: Bytecode instrumentation techniques for implementing design by contract in Java. In Klaus Havelund and Grigore Roşu, editors, *In the Proceedings of Second Workshop on Runtime Verification (RV 02), Copenhagen, Denmark, July 26, 2002*, volume 70 of *Electronic Notes in Theoretical Computer Science*, pages 56–80. Elsevier, 2002.

[3] Active Worlds: Home of the 3D Internet, Virtual Reality and Community Chat. [online], 2004. http://www.activeworlds.com/ (accessed December 28, 2004).

[4] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms*. Computer Science and Information Processing. Addison-Wesley, 1983.

[5] Christopher Alexander. *The Timeless Way of Building*. Oxford University Press, New York, 1979.

[6] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, New York, 1977.

[7] The A. L. I. C. E. Artificial Intelligence Foundation. [online], 2004. http://www.alicebot.org/ (accessed December 28, 2004).

[8] Andrea L. Ames, David R. Nadeau, and John L. Moreland. *The VRML Sourcebook*. John Wiley & Sons, 2nd edition, January 1997.

[9] Peter Amstutz, Reed Hedges, and Karsten Otto. *Creating Interreality: The Virtual Object System*, 2004. [Retrieved December 28, 2004, from http://www.interreality.org/static/docs/manual-html/index.html].

[10] Krzysztof R. Apt. Ten Year's of Hoare's Logic: A Survey - Part I. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 3(4):431–438, October 1981.

[11] Ken Arnold, Ann Wollrath, Bryan O'Sullivan, Robert Scheifler, and Jim Waldo. *The Jini Specification*. Addison-Wesley, Reading, MA, USA, 1999.

[12] Detlef Bartetzko, Clemens Fischer, Michael Möller, and Heike Wehrheim. Jass - Java with Assertions. In Klaus Havelund and Grigore Roşu, editors, *In the Proceedings of First Workshop on Runtime Verification (RV'2001), Paris, France, July 23, 2001*, volume 55 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2001.

[13] Imran Bashir and Amrit L. Goel. *Testing Object-Oriented Software: Life Cycle Solutions*. Springer-Verlag, May 2000.

[14] Kent Beck. *Extreme Programmning Explained*. The XP Series. Addison-Wesley Professional, October 1999.

[15] Kent Beck and Ralph E. Johnson. Patterns Generate Architectures. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'94)*, Lecture Notes in Computer Science, pages 134–149. Springer-Verlag, July 1994.

[16] Boris Beizer. *Software Testing Techniques*. Van Nostrand Reinhold Computer, 2nd edition, June 1990.

[17] Steve Benford, John Bowers, and Lennart E. Fahlén. Managing Mutual Awareness in Collaborative Virtual Environments. In G. Singh, S. K. Feiner, and D. Thalmann, editors, *Proceedings ACM SIGCHI Symposium on Virtual Reality Software and Technology (VRST'94)*, pages 223–236, Singapore, August 1994. ACM Press.

[18] Steve Benford, John Bowers, Lennart E. Fahlén, John Mariani, and Tom Rodden. Supporting Cooperative Work in Virtual Environments. *Computer Journal*, 37(8):635–668, 1994.

[19] Steve Benford, Chris Greenhalgh, Tom Rodden, and James Pycock. Collaborative Virtual Environments. *Communications of the ACM*, 44(7):79–85, 2001.

[20] Eric J. Berglund and David R. Cheriton. Amaze: a Multiplayer Computer Game. *IEEE Software*, 2(3):30–39, May 1985.

[21] Tim J. Berners-Lee, Robert Cailliau, and Jean-François Groff. The World Wide Web. *Computer Networks and ISDN Systems*, 25:454–459, 1994.

[22] Andreas Birrer and Thomas Eggenschwiler. Frameworks in the Financial Engineering Domain: An Experience Report. In *Proceedings of the Seventh European Conference on Object-Oriented Programming (ECOOP'93)*, pages 21–35. Springer-Verlag, July 1993.

[23] Grady Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin/Cummings, Redwood City, CA, 2nd edition, 1994.

[24] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Object Technology Series. Addison-Wesley, 1999.

[25] Laurence Bradley, Graham Walker, and Andrew McGrath. Shared Spaces. *British Telecommunications Engineering Journal*, 15:162–167, July 1996.

[26] Jeffrey M. Bradshaw. *Software Agents*. AAAI Press, 1997.

[27] Wolfgang Broll. DWTP: An Internet Protocol For Shared Virtual Environments. In *Proceedings of the 3rd International Symposium on the Virtual Reality Modeling Language*, pages 49–56, February 1998.

[28] Wolfgang Broll. SmallTool: a Toolkit for Realizing Shared Virtual Environments on the Internet. *Distributed Systems Engineering*, 5(3):118–128, 1998.

[29] Lauren P. Burka. The MUDdex. [online], 1993. `http://www.linnaean.org/~lpb/muddex/` (accessed December 28, 2004).

[30] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns.* John Wiley & Sons, 1996.

[31] Tolga K. Capin, Igor S. Pandzic, Nadia Magnenat-Thalmann, and Daniel Thalmann. *Avatars in Networked Virtual Environments.* John Wiley & Sons, July 1999.

[32] Tolga K. Capin and Daniel Thalmann. A Taxonomy of Networked Virtual Environments. In *Proceedings of International Workshop on Synthetic - Natural Hybrid Coding and Three Dimensional Imaging (IWSNHC3DI'99)*, September 1999.

[33] Vinton G. Cerf and R. Kahn. *Innovations in Internetworking*, chapter A protocol for packet network intercommunication, pages 10–21. Artech House, Inc., 1998.

[34] Tzi-Cker Chiueh. Distributed Systems Support for Networked Games. In *Proceedings of the Sixth Workshop on Hot Topics in Operating Systems*, pages 99–104, Cape Cod, MA, May 1997. IEEE Computer Society Press.

[35] Zièd Choukair and Damien Retailleau. A QoS Model for Collaboration through Distributed Virtual Environments. *Journal of Network and Computer Applications*, 23(3):311–334, July 2000.

[36] Gérald Collaud, Jacques Monnard, and Jacques Pasquier. Du livre de cours traditionnel au support de cours informatisé: une perspective historique. *Sciences et Techniques Educatives*, 5(4):319–342, December 1998. From traditional to electronic textbooks: a historical perspective.

[37] Apple Computer. *MacApp Programmer's Guide.* Apple Computer, Inc., 1986.

[38] Apple Computer. *MacAppII Programmer's Guide.* Apple Computer, Inc., Cupertino, CA, 1989.

[39] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* MIT Press, 2nd edition, 2001.

[40] Sean Cotter and Mike Potel. *Inside Taligent Technology.* Inside Taligent Series. Addison-Wesley, 1st edition, March 1995.

[41] Pavel Curtis. LambdaMOO Programmer's Manual - For LambdaMOO Version 1.8.0p6, March 1997.

[42] United States Department of Defense. Defense Modeling and Simulation Office. [online], 2004. http://www.dmso.mil/ (accessed December 28, 2004).

[43] United States Department of Defense. Defense Modeling and Simulation Office. [online], 2004. http://www.dmso.mil/public/transition/hla (accessed December 28, 2004).

[44] Peter Deutsch. The Eight Fallacies of Distributed Computing. [online], 1992. http://today.java.net/jag/Fallacies.html (accessed December 28, 2004).

[45] Stephan Diehl. *Distributed Virtual Worlds: Foundations and Implementation Techniques Using VRML, Java and CORBA.* Springer, New York, 2001.

[46] Bruno Dumas. MadTunes: des objets audio pour le framework MaDViWorld. Bachelor thesis, Department of Informatics, University of Fribourg, Switzerland, April 2004. http://diuf.unifr.ch/people/fuhrer/studproj/dumas/stream.html (accessed December 28, 2004).

[47] Jean-Claude Heudin (ed.). *Virtual Worlds: Synthetic Universes, Digital Life, and Complexity.* Perseus Books, Reading, Massachusetts, USA, 1999.

[48] Thomas Eggenschwiler and Erich Gamma. ET++ Swaps Manager: Using Object Technology in the Financial Engineering Domain. *OOPSLA'92, Special Issue of SIGPLAN Notices*, 27(10):166–178, 1992.

[49] Anders Eliasson. Implement Design by Contract for Java Using Dynamic Proxies: write Bug-free code with the DBCProxy framework. *JavaWorld How-To-Java*, February 2002. [Retrieved December 28, 2004, from http://www.javaworld.com/javaworld/jw-02-2002/jw-0215-dbcproxy.html].

[50] Clarence A. Ellis, Simon J. Gibbs, and Gail L. Rein. Groupware: Some Issues and Experiences. *Communications of the Association for Computing Machinery*, 34(1):38–58, January 1991.

[51] Véronique Normand et al. The COVEN Project: Exploring Applicative, Technical, and Usage Dimensions of Collaborative Virtual Environments. *Presence*, 8(2):218–236, 1999.

[52] Oren Etzioni and Daniel Wedd. A Softbot-Based Interface to the Internet. *Communications of the ACM*, 37(7):72–79, 1994.

[53] Yoann Fabre, Guillaume Pitel, Laurent Soubrevilla, Emmanuel Marchand, Thierry Géraud, and Akim Demaille. A Framework to Dynamically Manage Distributed Virtual Environments. In Jean-Claude Heudin, editor, *Virtual Worlds*, volume 1834 of *Lecture Notes in Computer Science*, pages 54–64. Second International Conference on Virtual Worlds, VW 2000, Paris, France, July 2000, Springer-Verlag, 2000.

[54] Yoann Fabre, Guillaume Pitel, Laurent Soubrevilla, Emmanuel Marchand, Thierry Géraud, and Akim Demaille. An Asynchronous Architecture to Manage Communication, Display, and User Interaction in Distributed Virtual Environments. In J. D. Mulder and R. van Liere, editors, *Proceedings of the 6th Eurographics Workshop on Virtual Environments (EGVE'2000)*, Computer Science / Eurographic Series, pages 105–113, Amsterdam, The Netherlands, June 2000. Springer-Verlag.

[55] Nicola Fankhauser. Mobile Agents in MaDViWorld. Master thesis, Department of Informatics, University of Fribourg, Switzerland, November 2004. http://diuf. unifr.ch/people/fuhrer/studproj/fankhauser/agentmadviworld.html (accessed December 28, 2004).

[56] David Flanagan, Jim Farley, and William Crawford. *Java Enterprise in a Nutshell: a Desktop Quick Reference*. O'Reilly & Associates, Inc., April 2002.

[57] Leonard N. Foner. Entertaining Agents: A Sociological Case Study. In *Proceedings of the First International Conference on Autonomous Agents (AA '97)*, 1997.

[58] Brian Foote. *Designing to Facilitate Change with Object-Oriented Frameworks*. Master thesis, University of Illinois at Urbana-Champaign, USA, 1988.

[59] Otto Forster. *Analysis 2: Differentialrechnung im $\mathbb{R}^n$ - Gewönliche Differentialgleichungen*. Vieweg Studium, 5th edition, 1984.

[60] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[61] Martin Fowler and Kendall Scott. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Object Technology Series. Addison-Wesley, 3rd edition, 2003.

[62] Stan Franklin and Art Graesser. Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents. In *Proceedings of the Third International Workshop on Agent Theories, Architectures and Languages*. Springer-Verlag, 1996.

[63] Emmanuel Frécon and Mårten Stenius. DIVE: A Scalable Network Architecture for Distributed Virtual Environments. *Distributed Systems Engineering*, 5(3):91–100, September 1998.

[64] Patrik Fuhrer. MaDViWorld: Massively Distributed Virtual Worlds. [online]. http://diuf.unifr.ch/softeng/projects/madviworld/index.htm (accessed December 28, 2004).

[65] Patrik Fuhrer, Ghita Kouadri Mostéfaoui, and Jacques Pasquier-Rocha. MaDViWorld : a Software Framework for Massively Distributed Virtual Worlds. *Software —Practice and Experience*, 32(7):645–668, June 2002.

[66] Patrik Fuhrer and Jacques Pasquier-Rocha. Massively Distributed Virtual Worlds: A Framework Approach. In Nicolas Guelfi, Egidio Astesiano, and Gianna Reggio, editors, *Scientific Engineering for Distributed Java Applications*, volume 2604 of *Lecture Notes in Computer Science*, pages 111–121. International Workshop, FIDJI 2002 Luxembourg-Kirchberg, Luxembourg, November 2002, Springer-Verlag, March 2003.

[67] Alex Fukanaga, Wolfgang Pree, and Takayuki Dan Kimura. Functions as Data Objects in a Data Flow Based Visual Language. In *Proceedings of the ACM Computer Science Conference, Indianapolis*, February 1993.

[68] Thomas A. Funkhouser. RING: A Client-Server System for Multi-User Virtual Environments. In *Proceedings of the 1995 Symposium on Interactive 3D Graphics*, pages 85–92, Monterey, CA, April 1995.

[69] Thomas A. Funkhouser. Network Topologies for Scalable Multi-User Virtual Environments. In *Proceedings of the Virtual Reality Annual International Symposium*, pages 222–228, Santa Clara, CA, March 1996.

[70] Alexandre Gachet. *A Software Framework for Developing Distributed Cooperative Decision Support Systems*. Doctoral thesis (no. 1402), University of Fribourg, Switzerland, Department of Informatics, February 2003.

[71] Alexandre Gachet and Pius Haettenschwiler. A Jini-based Software Framework for Developing Distributed Cooperative Decision Support Systems. *Software—Practice and Experience*, 33(3):221–258, February 2003.

[72] Erich Gamma. *Objektorientierte Software-Entwicklung am Beispiel von ET++: Design-Muster, Klassenbibliothek, Werkzeuge*. Doctoral thesis, University of Zürich, Switzerland, 1991. published by Springer-Verlag, 1992.

[73] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley, Massachusetts, 1995.

[74] Martin Gardner. The Fantastic Combinations of John Conway's New Solitaire Game Life. *Scientific American*, 223(120):120–123, October 1970.

[75] Laurent Gautier and Christophe Diot. Design and Evaluation of MiMaze, a Multi-Player Game on the Internet. In *Proceeding of IEEE International Conference on Multimeda Computing and Systems*, pages 233–236, Austin, TX, July 1998. IEEE Computer Society Press.

[76] Laurent Gautier, Emmanuel Lety, and Christophe Diot. MiMaze, a 3D Multi-Player Game on the Internet. In *Proceedings of the 4th International Conference on Virtual System and Multimedia*, volume 1, pages 84–89, Gifu, Japan, November 1998.

[77] David Geary. Simply Singleton: Navigate the Deceptively Simple Singleton Pattern. *JavaWorld How-To-Java*, April 2003. [Retrieved December 28, 2004, from http://www.javaworld.com/javaworld/jw-04-2003/jw-0425-designpatterns.html].

[78] William Gibson. *Neuromancer*. Ace Books, 1984.

[79] Adele Goldberg. *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley, January 1984.

[80] Danny Goodman. *The Complete HyperCard Handbook*. Bantam Books, 1987.

[81] Todd M. Greanier. Flatten your Objects: Discover the Secrets of the Java Serialization API. *JavaWorld How-To-Java*, July 2000. [Retrieved December 28, 2004, from http://www.javaworld.com/javaworld/jw-07-2000/jw-0714-flatten.html].

[82] Chris Greenhalgh. Awareness-based communication management in the MASSIVE systems. *Distributed Systems Engineering*, 5(3):129–137, September 1998.

[83] Chris Greenhalgh and Steve Benford. MASSIVE: a Distributed Virtual Reality System Incorporating Spatial Trading. In *Proceedings of the 15th IEEE International Conference on Distributed Computing Systems (ICDCS'95)*, pages 27–34, Vancouver, Canada, May 1995. IEEE Computer Society.

[84] Chris Greenhalgh and Steve Benford. MASSIVE: A Virtual Reality System for Tele-conferencing. *ACM Transactions on Computer Human Interfaces (TOCHI)*, 2(3):239–261, September 1995.

[85] William Grosso. *Java RMI*. O'Reilly & Associates, Inc., October 2001.

[86] The Object Management Group. Common Object Request Broker Architecture: Core Specification, December 2002. [Retrieved December 28, 2004, from http://www.omg.org/docs/formal/02-12-02.pdf].

[87] Frank G. Halasz. Reflections on NoteCards: Seven Issues for the Next Generation of Hypermedia Systems. *Communications of the ACM*, 31(7):836–852, July 1988.

[88] Paul R. Halmos. *Measure Theory*. Springer Verlag, 1974.

[89] J. C. Herz. *Surfing on the Internet: a Nethead's Adventure on-line*. Little, Brown and Company, 1st edition, 1995.

[90] Charles Antony Richard Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[91] id Software. The DOOM PC Computer Game (1993) and The QUAKE PC computer game (1997). [online], 2004. http://www.idsoftware.com/ (accessed December 28, 2004).

[92] Alex-Ivan Invernizzi. MaDViWorld : Etude du framework et communication inter-objet. Diploma thesis, Department of Informatics, University of Fribourg, Switzerland, April 2004. http://diuf.unifr.ch/people/fuhrer/studproj/invernizzi/metalpanic.html (accessed December 28, 2004).

[93] Ivar Jacobson, Grady Booch, and James Rumbaugh. *The Unified Software Development Process*. Object Technology Series. Addison-Wesley, 1999.

[94] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Overgaard. *Object-Oriented Software Engineering – A Use Case Driven Approach*. Addison-Wesley, Wokingham, England, 1992.

[95] Using Design by Contract to Automate Java Software and Component Testing. Technical report, Parasoft Corporation, 2002. [Retrieved December 28, 2004, from http://www.parasoft.com/jsp/printables/_111.pdf?path=/jsp/products/article.jsp].

[96] François Jimenez. MaDViWorld : Etude du Framework - Implémentation d'un Objet Chat - Implémentation XML. Master thesis, Department of Informatics, University of Fribourg, Switzerland, January 2004. http://diuf.unifr.ch/people/fuhrer/studproj/jimenez/chat.html (accessed December 28, 2004).

[97] William A. Jindrich. Foible: a Framework for Visual Programming Languages. Master thesis, University of Illinois at Urbana-Champaign, USA, 1990.

[98] Design by Contract for Java Using JMSAssert. Technical report, Man Machine Systems. [Retrieved December 28, 2004, from http://www.mmsindia.com/DBCForJava.html].

[99] Ralph E. Johnson. Documenting Frameworks using Patterns. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'92) Conference Proceedings*, pages 63–76. ACM Press, October 1992.

[100] Ralph E. Johnson. Components, Frameworks, Patterns. In *ACM SIGSOFT Symposium on Software Reusability*, pages 10–17, 1997.

[101] Ralph E. Johnson and Brian Foote. Designing Reusable Classes. *Journal of Object-Oriented Programming*, 1(2):22–35, June/July 1988.

[102] Ralph E. Johnson, Carl McConnell, and J. Michael Lake. *Code Generation–Concepts, Tools, Techniques. Proceedings of the International Workshop on Code Generation*, chapter The RTL system: A framework for code optimization, pages 255–274. Springer-Verlag, 1992.

[103] JUnit, Testing Resources for Extreme Programming. on-line, 2004. http://junit.org/ (accessed December 28, 2004).

[104] Murat Karaorman and Parker Abercrombie. jContractor: Bytecode Instrumentation Techniques for Implementing Design by Contract in Java. [online], March 2003. [Retrieved December 28, 2004, from http://jcontractor.sourceforge.net/doc/jContractor_FMSD03.pdf].

[105] Rick Kazman. Making WAVES: On the Design of Architectures for Low-end Distributed Virtual Environments. In *Proceedings of IEEE Virtual Reality Annual International Symposium (VRAIS'93)*, pages 443–449. IEEE Computer Society Press, September 1993.

[106] Rick Kazman. Load Balancing, Latency Management and Separation of Concerns in a Distributed Virtual World. In Albert Y. Zomaya, editor, *Parallel Computing: Paradigms and Applications*. International Thomson Publishing, November 1995.

[107] Hans-Ulrich Kiel and Joerg Czeranski. Softwarepraktikum Netzwerkprogrammierung unter Unix am Beispiel des Spiels. [online], 1994. http://home.tu-clausthal.de/student/iMaze/ (accessed December 28, 2004).

[108] David Kirsch. *Cooperative Buildings-Integration Information, Organization, and Architecture*, chapter Adaptive rooms, virtual collaboration, and cognitive workflow, pages 94–106. Lecture Notes in Computer Science. Springer, Heidelberg, 1998.

[109] Reto Kramer. The Java Design by Contract Tool. In Raimund Edge, Bertrand Meyer, and Madhu Singh, editors, *In the Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS) USA'98, Santa Barbara, California, August 3-7, 1998*, pages 295–307. IEEE Press, 1998.

[110] Glenn E. Krasner and Stephen T. Pope. A cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26–49, August/September 1988.

[111] Philippe Kruchten. *The Rational Unified Process*. Addison-Wesley, 1999.

[112] S. Ilango Kumaran. *Jini Technology: An Overview.* P T R Prentice-Hall, 2002.

[113] Christopher G. Langton. *Artificial Life I*, proceedings volume Artificial Life. SFI (Santa Fe Institute) Studies in the Sciences of Complexity. Addison-Wesley, 1988.

[114] Craig Larman. *Applying UML and Patterns - An Introduction to Object-Oriented Analysis and Design and the Unified Process.* Prentice Hall PTR, 2nd edition, 2002.

[115] Rodger Lea, Yasuaki Honda, and Kouichi Matsuda. Virtual Society: Collaboration in 3D spaces on the Internet. *Computer Supported Cooperative Working*, 6(2-3):227–250, 1997. Special issue on groupware and the World Wide Web.

[116] Rodger Lea, Yasuaki Honda, Kouichi Matsuda, and Satoru Matsuda. Community Place: Architecture and Performance. In *Proceedings of the 2nd Symposium on Virtual Reality Modeling Language (VRML'97)*, pages 41–50, February 1997.

[117] Rodjer Lea, Yasuaki Honda, Kouichi Matsuda, Olaf Hagsand, and Mårten Stenius. Issues in the Design of a Scalable Shared Virtual Environment for the Internet. In *Proceedings of the 30th Hawaii International Conference on System Sciences (HICSS'97)*, January 1997.

[118] Dave Lebling, Mark Blank, and Tim Anderson. Zork: A Computerized Fantasy Simulation Game. *IEEE Computers Magazine*, pages 51–59, April 1979.

[119] Sing Li. *Professional Jini.* Wrox Press, 2000.

[120] Hugh Loebner. Loebner Prize Home Page. [online], 2004. http://www.loebner.net/Prizef/loebner-prize.html (accessed December 28, 2004).

[121] Sanaa Maati. Un éditeur collaboratif pour le framework MaDViWorld. Bachelor thesis, Department of Informatics, University of Fribourg, Switzerland, December 2003. http://diuf.unifr.ch/people/fuhrer/studproj/maati/collab.html (accessed December 28, 2004).

[122] Michael R. Macedonia and Michael J. Zyda. A Taxonomy for Networked Virtual Environments. *IEEE Multimedia*, 4(1):48–56, 1997.

[123] Michael R. Macedonia, Michael J. Zyda, David R. Pratt, Paul T. Barham, and Steven Zeswitz. NPSNET: A Network Software Architecture for Large-Scale Virtual Environment. *Presence*, 3(4):265–287, 1994.

[124] Peter W. Madany, Roy H. Campbell, Vincent F. Russo, and Douglas E. Leyens. A Class Hierachy for Building Stream-Oriented File Systems. In Stephen Cook, editor, *Proceedings of the Third European Conference on Object-Oriented Programming (ECOOP'89)*, pages 311–328. Cambridge University Press, 1989.

[125] Per Madsen. Testing by Contract - Combining Unit Testing and Design by Contract. In Kasper Østerbye, editor, *Proceedings of the Tenth Nordic Workshop on Programming and Software Development Tools and Techniques (NWPER'2002)*, pages 111–116, Copenhagen, August 2002.

[126] Per Madsen. Unit Testing Using Design by Contract and Equivalence Partitions. In *Proceedings of the Fourth International Conference on Extreme Programming and Agile Processes in Software Engineering (XP2003)*, volume 2675 of *Lecture Notes in Computer Science*, pages 425–426. Springer-Verlag, August 2003.

[127] Pattie Maes. Social Interface Agents: Acquiring Competence by Learning from Users and other Agents. In *Proceedings of the 1994 AAAI Spring Symposium on Software Agents*, pages 71–78. AAAI Press, March 1994.

[128] Pattie Maes. Artificial Life meets Entertainment: Interacting with Lifelike Autonomous Agents. *Communications of the ACM, Special Issue on New Horizons of Commercial and Industrial AI*, 38(11):108–114, November 1995.

[129] Fabrice Marchon. MaDViWorld : Etude du framework et création dun guide du programmeur d'objets MaDViWorld. Diploma thesis, Department of Informatics, University of Fribourg, Switzerland, November 2002. http://diuf.unifr.ch/people/fuhrer/studproj/marchon/index.html (accessed December 28, 2004).

[130] Thomas Marill, Daniel Edwards, and Wallace Feuerzeig. DATA-DIAL: Two-way Communications with Computers from Ordinary Dial Telephones. *Communitcations of the ACM*, 6(10):622–624, 1963.

[131] Martin Mauve, Volker Hilt, Christoph Kuhmuench, and Wolfgang Effelsberg. RTP/I - Towards a Common Application Level Protocol for Distributed Interactive Media. *IEEE Transactions on Multimedia*, 3(1):152–161, 2001.

[132] John McClain. A Short Presentation on Codebase. [online], 2004. http://user-jmcclain.jini.org/codebase/codebase.htm (accessed December 28, 2004).

[133] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, October 1996.

[134] Bertrand Meyer. *Eiffel: The Language*. The Object-Oriented Series. Prentice-Hall, 1992.

[135] Bertrand Meyer. *Object-Oriented Software Construction*. The Object-Oriented Series. Prentice-Hall, 2nd edition, 1997.

[136] Nelson Minar. Distributed Systems Topologies: Part 1. *O'Reilly & Associates, Inc.*, December 2001. [Retrieved December 28, 2004, from http://www.openp2p.com/pub/a/p2p/2001/12/14/topologies_one.html].

[137] Nelson Minar. Distributed Systems Topologies: Part 2. *O'Reilly & Associates, Inc.*, August 2002. [Retrieved December 28, 2004, from http://www.openp2p.com/pub/a/p2p/2002/01/08/p2p_topologies_pt2.html].

[138] Jacques Monnard and Jacques Pasquier. An Object-Oriented Scripting Environment for the WEBSs Electronic Book System. In *Proceedings of the ACM Conference on Hypertext ECHT'92*, pages 81–90, 1992.

[139] Jacques Monnard and Jacques Pasquier. *Virtual Worlds and Multimedia*, chapter WEBSs : an Electronic Book Shell with an Object-Oriented Scripting Environment. John Wiley and Sons, New York, 1993.

[140] Richard Monson-Haefel. *Enterprise JavaBeans*. O'Reilly & Associates, Inc., 3rd edition, September 2001.

[141] Chip Morningstar and Randall Farmer. The Lessons Learned of Lucasfilm's Habitat. In Michael Benedikt, editor, *Cyberspace: First Steps*, pages 273–302, Cambridge, Massachussets, 1990. MIT Press.

[142] Katherine L. Morse, Lubomir Bic, and Michael Dillencourt. Interest Management in Large-Scale Virtual Environments. *Presence*, 9(1):52–68, 2000.

[143] The Mud Connector. [online], 2004. http://www.mudconnector.com (accessed December 28, 2004).

[144] Jan Newmarch. *A Programmer's Guide to Jini Technology*. APress, 1st edition, November 2000.

[145] Scott Oaks and Henry Wong. *Jini: a Desktop Quick Reference*. In a nutshell. O'Reilly & Associates, Inc., 981 Chestnut Street, Newton, MA 02164, USA, 2000.

[146] Rickard Öberg. *Mastering RMI: Developing Enterprise Applications in Java and EJB*. John Wiley & Sons, 2001.

[147] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. Doctoral thesis, University of Illinois at Urbana-Champaign, USA, Department of Computer Science, 1992.

[148] Seymour A. Papert. *Mindstorms: Children, Computers and Powerful Ideas*. Basic Books, 2nd edition, March 1999.

[149] PARADISE Project Web site. [online], 2004. http://www.dsg.stanford.edu/paradise.html (accessed December 28, 2004).

[150] Jacques Pasquier, Ed Grossman, and Gérald Collaud. Prototyping an Interactive Electronic Book System Using an Object Oriented Approach. In *ECOOP'88 Proceedings*, pages 177–190, August 1988.

[151] Jacques Pasquier and Jacques Monnard. *Livres électroniques - De l'utopie à la réalisation*. Presses Polytechniques et Universitaires Romandes, 1995.

[152] Patrick Pauchard. Anwendung zur Darstellung einer verteilten virtuellen Welt. Diploma thesis, Department of Informatics, University of Fribourg, Switzerland, April 2004. http://diuf.unifr.ch/people/fuhrer/studproj/pauchard/xmlmadviworld.html (accessed December 28, 2004).

[153] Simon Powers, Mike Hinds, and Jason Morphett. DEE: An Architecture for Distributed Virtual Environment Gaming. *Distributed Systems Engineering*, 5(3):107–117, 1998.

[154] Wolfgang Pree. Meta Patterns - A Means for Caputring the Essentials of Reusable Object-Oriented Design. In *Proceedings of ECOOP'94*, pages 150–162, 1994.

[155] Wolfgang Pree. *Design Patterns for Object-Oriented Software Development*. Addison Wesley, Wokingham, 1995.

[156] Jeff Prosise. *Programming Windows with MFC.* Microsoft Press, 2nd edition, May 1999.

[157] Elizabeth Reid. *Cultural Formations in Text-Based Virtual Realities.* Master thesis, University of Melbourne, English Department, January 1994.

[158] RFC-Editor Webpage. [online], 2004. http://www.rfc-editor.org/ (accessed December 28, 2004).

[159] Howard Rheingold. *The Virtual Community: Homestanding on the Electronic Frontier.* Perseus Book, October 1993.

[160] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21(2):110–126, February 1978. Previously released as an MIT "Technical Memo" in April 1977, [Retrieved December 28, 2004, from http://theory.lcs.mit.edu/~rivest/rsapaper.pdf].

[161] David J. Roberts and Paul M. Sharkey. Maximising Concurrency and Scalability in a Consistent, Causal, Distributed Virtual Reality System, Whilst Minimising the Effect of Network Delays. In *IEEE Proceedings of 6th Workshop on Enabling Technologies Infrastructure for Collaborative Enterprises (WET-ICE'97)*, pages 161–167. IEEE Computer Society Press, June 1997.

[162] David J. Roberts, Paul M. Sharkey, and P. D. Sandoz. A Real-time, Predictive Architecture for Distributed Virtual Reality. In *Proceedings of 1st Conference Simulation and Interaction in Virtual Environments*, pages 72–81, July 1995.

[163] Walter Rudin. *Principles of Mathematical Analysis.* McGraw-Hill, Tokio, 3rd edition, 1976.

[164] James Rumbaugh. The Life of an Object Model: How the Object Model Changes During Development. *Journal of Object-Oriented Programming*, 7(1):24–32, March/April 1994.

[165] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorenson. *Object-Oriented Modeling and Design.* Prentice Hall, Englewood Cliffs, NJ, 1991.

[166] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual.* Object Technology Series. Addison-Wesley, 1999.

[167] Stuart Russel and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Series in Artificial Intelligence. Prentice-Hall, 2nd edition, December 2002.

[168] Jacques Savoy. *Le livre élécronique EBOOK3*, volume 852 of *Publications universitaires européennes: Series V - Sciences économiques.* Peter Lang, 1987.

[169] Jacques Savoy. Les sources des hypertextes: une bibliographie commentée. *T.S.I. - Technique et Sciences Informatiques*, 9(6):515–524, 1990.

[170] Kurt J. Schmucker. *Object-Oriented Programming for the Macintosh.* Hayden Books, 1986.

[171] Chris Shaw and Mark Green. The MR Toolkit Peers Package and Experiment. In *Proceedings of IEEE Virtual Reality Annual International Symposium (VRAIS'93)*, pages 463–469. IEEE Computer Society Press, September 1993.

[172] Chris Shaw, Mark Green, Jiandong Liang, and Yunqi Sun. Decoupled Simulation in Virtual Reality with the MR Toolkit. *ACM Transactions on Information Systems*, 11(3):287–317, July 1993.

[173] Gurminder Singh, Luis Serra, Willie Png, and Hern Ng. BrickNet: A Software Toolkit for Networked-based Virtual Environments. *Presence: Teleoperators an Virtual Environments*, 3(1):19–34, 1994.

[174] Gurminder Singh, Luis Serra, Willie Png, Audrey Wong, and Hern Ng. BrickNet: Sharing Object Behaviors On The Net. In *Proceedings of the IEEE Virtual Reality Annual International Symposium (VRAIS'95)*, pages 19—25. IEEE Computer Society Press, 1995.

[175] Sandeep Singhal and Michael Zyda. *Networked Virtual Environments: Design and Implementation*. Addison-Wesley, 1999.

[176] Jouni Smed, Timo Kaukoranta, and Harri Hakonen. A Review on Networking and Multiplayer Computer Games. Technical Report Technical Report 454, Turku Centre for Computer Science, April 2002.

[177] Jouni Smed, Timo Kaukoranta, and Harri Hakonen. Aspects of Networking in Multiplayer Computer Games. *The Electronic Library*, 20(2):87–97, 2002.

[178] Bjørn Stabell and Ken Ronny Schouten. The Story of XPilot. *ACM Crossroads*, 3(2), 1996. [Retrieved December 28, 2004, from http://www.acm.org/crossroads/xrds3-2/xpilot.html].

[179] Neal Stephenson. *Snow Crash*. Bantam Spectra, New York, 1992.

[180] Warren Strange. Jxta-JERI Programmers Guide. [online], 2004. http://user-wstrange.jini.org/jxtajeri/JxtaJeriProgGuide.html (accessed December 28, 2004).

[181] Sun Microsystems, Inc. *Java Remote Method Specification*, October 2002. Revision 1.8, Java 2 SDK, Standard Edition, v1.4, [Retrieved December 28, 2004, from ftp://ftp.java.sun.com/docs/j2se1.4/rmi-spec-1.4.pdf].

[182] Todd Sundsted. An introduction to agents: Find out what agents are and what they can do for us, and take the first steps toward building your own simple agent architecutre in java. *JavaWorld How-To-Java*, June 1998. [Retrieved December 28, 2004, from http://www.javaworld.com/javaworld/jw-06-1998/jw-06-howto.html].

[183] Un-Jae Sung, Jae-Heon Yang, and Kwang-Yun Wohn. Concurrency Control in CIAO. In *1999 IEEE Virtual Reality Conference*, pages 22–28. IEEE Computer Society Press, March 1999.

[184] The TecfaMOO. [online], 1995. http://tecfa.unige.ch/moo/tecfamoo.html (accessed December 28, 2004).

[185] TheCodingMonkeys. SubEthaEdit. [online], 2004. http://www.codingmonkeys.de/subethaedit/ (accessed December 28, 2004).

[186] Alan M. Turing. Computing Machinery and Intelligence. *MIND*, 59(236):433–460, October 1950.

[187] Manny Vellon, Kirk Marple, Don Mitchell, and Steven Drucker. The Architecture of a Distributed Virtual Worlds System. In USENIX, editor, *Proceedings of the fourth USENIX Conference on Object-Oriented Technologies and Systems (COOTS): April 27–30, 1998, Santa Fe, NM*, Berkeley, CA, USA, 1998. USENIX.

[188] Bill Venners. How to attach a user interface to a Jini service: An in-depth look at the serviceui project from the Jini community. *JavaWorld How-To-Java*, October 1999. [Retrieved December 28, 2004, from http://www.javaworld.com/javaworld/jw-10-1999/jw-10-jiniology.html].

[189] Bill Venners. Design Principles and Code Ownership - A Conversation with Martin Fowler, Part II. *The Artima Developer Community*, November 2002. [Retrieved December 28, 2004, from http://www.artima.com/intv/principles.html].

[190] Bill Venners. Jini Extensible Remote Invocation - A Conversation with Bob Scheifler, Part VI. *The Artima Developer Community*, August 2002. [Retrieved December 28, 2004, from http://www.artima.com/intv/jeri.html].

[191] Bill Venners. *The ServiceUI API Specification (Version 1.1)*. Artima Software, October 2002. [Retrieved December 28, 2004, from http://www.artima.com/jini/serviceui/Spec.html].

[192] Didier Verna, Yoann Fabre, and Guillaume Pitel. Urbi et Orbi: Unusual Design and Implementation Choices for Distributed Virtual Environments. In H. Thwaites, editor, *VSMM 2000: Sixth International Conference on Virtual Systems and Multimedia*, pages 714–724, Gifu, Japan, October 2000.

[193] John M. Vlissides. *Generalized Graphical Object Editing*. Doctoral thesis, Stanford University, USA, Department of Electrical Engineering, June 1990.

[194] John M. Vlissides and Mark A. Linton. Unidraw: A Framework for Building Domain-Specific Graphical Editors. *ACM Transactins on Information Systems*, 8(3):237–268, July 1990.

[195] Juergen Vogel and Martin Mauve. Consistency Control for Distributed Interactive Media. In *Proceedings of ACM Multimedia 2000*, pages 259–267, October 2001.

[196] John von Neumann and A. W. *Theory of Self–Reproducing Automata*. University of Illinois Press, 1966. edited and completed by A. W. Burks.

[197] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. A Note on Distributed Computing. In *Mobile Object Systems: Towards the Programmable Internet*, pages 49–64. Springer-Verlag: Heidelberg, Germany, 1997.

[198] Qunjie Wang, Mark Green, and Chris Shaw. EM - An Environment Manager for Building Networked Virtual Environments. *Proceedings of IEEE Virtual Reality Annual Symposium (VRAIS'95)*, pages 11–18, March 1995.

[199] Jos Warmer and Anneke G. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Object Technlolgy Series. Addison-Wesley Professional, March 1999.

[200] Richard C. Waters, David B. Anderson, John W. Barrus, David C. Brogan, Michael A. Casey, Stephan G. McKeown, Tohei Nitta, Ilene B. Sterns, and William S. Yerazunis. Diamond Park and Spline: Social Virtual Reality with 3D Animation, Spoken Interaction and Runtime Extendability. *Presence*, 6(4):461–481, August 1997.

[201] André Weinand. *Objektorientierte Entwurf und Implementierung portabler Fensterumgebungen am Beispiel des Application-Frameworks ET++*. Doctoral thesis, University of Zürich, Switzerland, 1991. published by Springer-Verlag, 1992.

[202] André Weinand and Erich Gamma. *Computer Science Research at UBILAB, Strategy and Projects; Proceedings of the UBILAB'94 Conference, Zürich*, chapter ET++ - a Portable, Homogenous Class Library and Application Framework, pages 62–92. Universitaetsverlag Konstanz, September 1994.

[203] André Weinand, Erich Gamma, and Rudolf Marty. ET++ - An object-oriented application framework in C++. *OOPSLA'88, Special Issue of SIGPLAN Notices*, 23(11):46–57, 1988.

[204] André Weinand, Erich Gamma, and Rudolf Marty. Design and Implementation of ET++, a Seamless Object-Oriented Application Framework. *Structured Programming*, 10(2):63–87, 1989.

[205] Joseph Weizenbaum. ELIZA - A Computer Program for the Study of Natural Language Communication between Man and Machine. *Communications of the Association for Computing Machinery*, 9:36–45, 1966.

[206] David A. Wilson, Larry S. Rosenstein, and Daniel G. Shafer. *Programming with MacApp*. Addison-Wesley, 1990.

[207] Paul Wilson. *Computer Supported Cooperative Work: An Introduction*. Intellect, 1991.

[208] Rebecca J. Wirfs-Brock and Ralph E. Johnson. Surveying Current Research in Object-Oriented Design. *Communications of the ACM*, 33(9):104–124, September 1990.

[209] Michael Wooldridge. Agent-Based Software Engineering. *IEEE Proceedings on Software Engineering*, 144(1):26–37, 1997.

[210] Michael Wooldridge and Nicholas R. Jennings. Intelligent Agents: Theory And Practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.

[211] Nichole Yankelovich, Bernard J. Haan, Norman K. Meyrowitz, and Steven M. Drucker. Intermedia: The Concept and the Construction of a Seamless Information Environment. *IEEE*, 21(1):81–96, January 1988.

[212] Amr F. Yassin and Mohamed E. Fayad. Application Frameworks: A Survey. In Mohamed E. Fayad and Ralph Johnson, editors, *Domain-Specific Application Frameworks*. New York: John Wiley & Sons, 1999.

[213] Robert H. Zakon. Hobbes' Internet Timeline v7.0. [online], 2004. `http://www.zakon.org/robert/internet/timeline/` (accessed December 28, 2004).

Index

Curriculum Vitae

Personal Data

Name	:	Patrik FUHRER
Date of Birth	:	March 17th, 1975 in São José dos Campos (Brasil)
Nationality	:	Swiss (from Signau/BE)
Marital Status	:	married, one son

Education

since 1999	:	PhD Student, Informatics, University of Fribourg, Switzerland
1998	:	**Diploma in Mathematics**
1994 - 1998	:	Mathematics and Informatics Studies, University of Fribourg, Switzerland
1994	:	**Maturity diploma (type economics)**
1992 - 1994	:	Maturity studies, Collège du Sud, Bulle
1990 - 1992	:	Maturity studies, Ecole Supérieure de Commerce, Neuchâtel

Languages

- French (mother tongue)
- German
- English
- Portuguese (spoken)

Publications

Referred Publications

[1] Patrik Fuhrer and Jacques Pasquier-Rocha. Massively Distributed Virtual Worlds: A Framework Approach. In Nicolas Guelfi, Egidio Astesiano, and Gianna Reggio,

editors, *Scientific Engineering for Distributed Java Applications*, volume 2604 of *Lecture Notes in Computer Science*, pages 111–121. International Workshop, FIDJI 2002 Luxembourg-Kirchberg, Luxembourg, November 2002, Springer-Verlag, March 2003.

[2] Patrik Fuhrer, Ghita Kouadri Mostéfaoui, and Jacques Pasquier-Rocha. MaDViWorld : a Software Framework for Massively Distributed Virtual Worlds. *Software– Practice and Experience*, 32(7):645-668, June 2002.

Internal Publications

[1] Patrik Fuhrer, Ghita Kouadri Mostéfaoui, and Jacques Pasquier-Rocha. The MaDViWorld Software Framework for Massively Distributed Virtual Worlds: Concepts, Examples and Implementation Solutions. Internal Working Paper 01-23, Departement of Informatics, University of Fribourg, Switzerland, July 2001. http://diuf.unifr.ch/people/fuhrer/publications/internal/madviworld.pdf (accessed December 28, 2004).

[2] Patrik Fuhrer and Jacques Pasquier-Rocha. Massively Distributed Virtual Worlds: a Framework Approach. Internal Working Paper 02-16, Departement of Informatics, University of Fribourg, Switzerland, December 2002. http://diuf.unifr.ch/people/fuhrer/publications/internal/madObjEvt.pdf (accessed December 28, 2004).

[3] Patrik Fuhrer and Jacques Pasquier-Rocha. Massively Distributed Virtual Worlds: a Formal Approach. Internal Working Paper 03-14, Departement of Informatics, University of Fribourg, Switzerland, August 2003. http://diuf.unifr.ch/people/fuhrer/publications/internal/madModel.pdf (accessed December 28, 2004).

[4] Patrik Fuhrer and Jacques Pasquier-Rocha. MaDViWorld Objects: Examples and Classification. Internal Working Paper 03-15, Department of Informatics, University of Fribourg, Switzerland, August 2003. http://diuf.unifr.ch/people/fuhrer/publications/internal/madObjAgents.pdf (accessed December 28, 2004).

[5] Patrik Fuhrer and Jacques Pasquier-Rocha. Virtual worlds: From Concepts to a Distributed Implementation Framework. Internal Working Paper 04-04, Department of Informatics, University of Fribourg, Switzerland, May 2004. http://diuf.unifr.ch/people/fuhrer/publications/internal/madimpl.pdf (accessed December 28, 2004).