The Challenge of Supporting Distributed MOOs
Main Difficulties and their Solutions within the MaDViWorld Software Framework

Patrik Fuhrer and Jacques Pasquier-Rocha

University of Fribourg
Department of Informatics
Rue P.-A. de Faucigny 2
CH-1700 Fribourg
Switzerland
patrik.fuhrer@unifr.ch
WWW home page: http://diuf.unifr.ch/people/fuhrer/

Abstract. MOOs are object-oriented multi-user virtual communities. Although their underlying virtual world paradigm is much richer than the document one traditionally supported by the world wide web, their dissemination on the Internet suffers from two main weaknesses: (1) they are typically based on centralized architectures and do not scale well; and (2) they usually propose a rather closed software environment with limited extension and programming facilities. For example, one cannot populate them with arbitrarily complex objects programmed in a mainstream object-oriented language such as Java, C#, C++ or Smalltalk.

MaDViWorld, the extensible software framework presented in this paper, allows for distributing the rooms of a given MOO on an arbitrarily large number of machines, each running a small server application. It also offers the possibility of easily filling up a room with new Java objects at the condition of respecting a small set of conventions. Finally, it provides the hooks for taking care of some of the most challenging distributed MOO problems such as managing event propagation and securing access to resources.

Keywords: MOO, Virtual World, Distributed Framework, Event Propagation, Security

Contents

1 Introduction

Virtual Community or Inhabited Virtual World finds its roots in the earliest text-based multi-user games: MUDs[1]. Continuing the trend was the development of MOOs (adding object-oriented features to the MUDs), IRC[2] and conferencing systems, as well as the World Wide Web and its many progeny in the 1990s. Finally "inhabited" 2D and 3D virtual spaces have risen by enhancing text-based chat channels with powerful graphical interfaces. Although very different in their concrete application, all these systems have a major similarity: they are based on a *virtual world paradigm*, which allows for much richer interactions than the classical document metaphor traditionally supported by the World Wide Web. Indeed, 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 avatar representation and start a dialog with her. Moreover, it is possible for a user to modify some properties of the world and all the nearby users are immediately made aware of it. Unfortunately, these systems also suffer from the same weaknesses, which hold back their larger dissemination:

- Since many events must be correctly synchronized and propagated in order to maintain a given virtual world consistency, its software architecture is typically very centralized. A single central server contains all the world pertinent data and assumes its accessibility, consistency and persistence, while many clients allow for interaction with the other users and the various objects of the world. This approach has two main weaknesses. First, the whole system depends completely on the central server robustness. Secondly, it *does not scale well.* Just imagine what the web would be today if all its contents would need to be regrouped on a single server having to coordinate all its users.

- A virtual world system usually proposes a rather *closed software environment* with limited extension and programming facilities. For example, an open environment should allow a vi-world developer (i.e. one who is ready to do some real programming in order to extend the world facilities) to add new types of object to the world by using a standard technology[3] and not by resorting to ad-hoc languages. Such an environment should also allow for fully overriding the programming of avatar and/or room standard features, which means much more than merely supporting their parameterization.

MaDViWorld[4], the software framework presented in this paper, represents a successful attempt to propose a software environment able to support the richness of the virtual world paradigm without yielding to the main weaknesses briefly introduced above. It allows for distributing the rooms of a given world on an arbitrarily large number of machines, each running a small server application. It also offers the possibility of populating the rooms with new full-fledged Java objects at the condition of implementing a small set of well-defined interfaces. In other words, MaDViWorld software architecture combines the great advantages (scalability, robustness,...) found in

[1] Multi-User Dungeons, see [15]

[2] Internet Relay Chat

[3] An acceptable solution would be to require her to use a mainstream programming language such as Java, C#, C++ or Smalltalk and to offer her a structured set of abstract and concrete classes that have to be implemented and inherited, respectively, in order for an object to be "vi-world compatible".

[4] MaDViWorld stands for Massively Distributed Virtual World, see [7], [8] and [5].

the document paradigm with the shared virtual world metaphor and contrasts with the "many clients-one server" architectures typically proposed by most virtual world platforms. Some similar approaches are presented by [3] in URBI et ORBI, by [11] in MASSIVE or by [4] in DIVE.
This paper is organized as follows. Section 2 clarifies the terminology and introduces the MaDViWorld model, thus setting up the foundations for the rest of the paper. Section 3 presents an in depth discussion of the software architecture that has been chosen in order to implement the model. The two next sections concentrate on two additional challenging aspects: event distribution and security. Finally, Section 6 summarizes MaDViWorld main achievements and provides an insight into the future of the project.

2 Virtual Worlds Main Concepts

The first part of this section is dedicated to a short 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 and we present the MaDViWorld model.

2.1 An Example 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. Doing so, she comes to a place where she sees two other users playing a battleship game and a little crowd watching them. She joins the observers and, after a while, she says to her neighbour: "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 game and they start to play. When they are finished she puts the battleship object back in her bag and leaves the other user.

2.2 Terminology

The main concepts emerging from the above 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 your favorite fantasy hero.

- *Object*: The objects found in the discussed virtual worlds do not just represent passive data, but are active objects the avatars can execute (e.g. the battleship game). These objects can be multi-user and even have many observers like in our little story. Last, but not least, objects can be copied and/or moved around by the avatar.

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

- *Navigation*: It is the action of going from a given location to another. Avatars can perform this task, either if there is 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 the objects are always *contained* within one given subspace.

- *Event*: The avatar has to be aware of its environment. This awareness is achieved by the concept of events. Another avatar entering one's subspace or a move in the battleship 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".

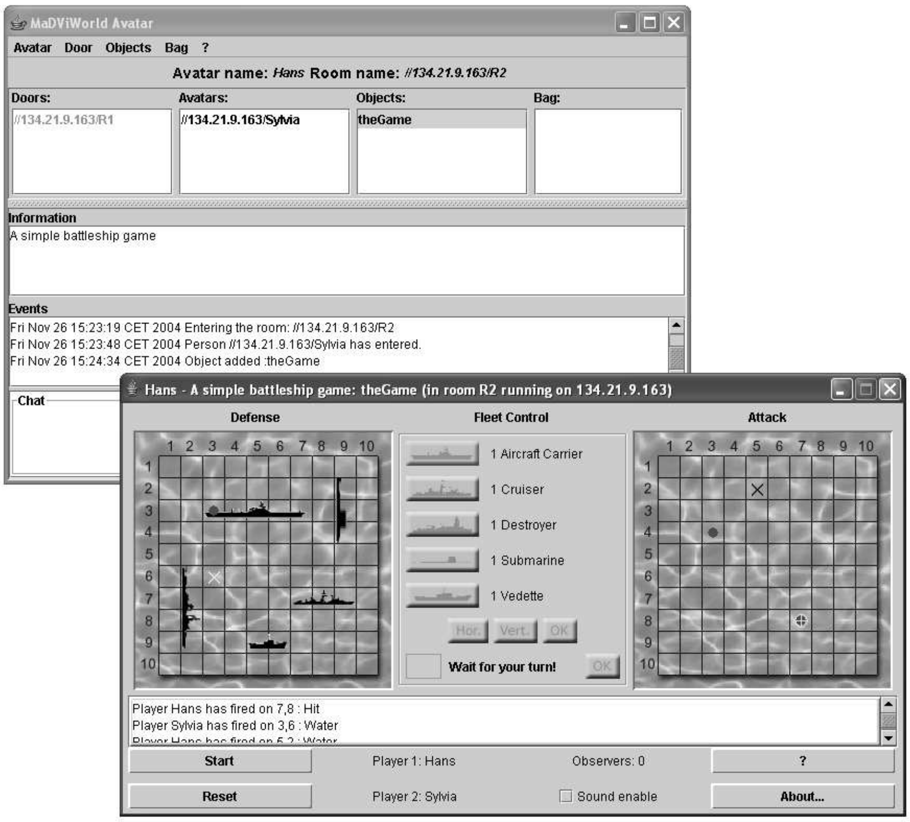
- *Event consumer*: An event can be catched and interpreted by an event consumer, which then reacts properly or simply ignores it. For instance, the players and the audience of a given game understand that the game is finished.

- *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 concepts above have been formalized and integrated within a general theoretical model (see [5]). The rather mathematical formalism used in the latter, however, is out of the scope of the present paper and not necessary for its further comprehension. Therefore, the next subsection provides a more practical and concrete description of the adopted model.

2.3 The MaDViWorld Model

Let us use the metaphor inherited from the early MUDs and MOOs for the subspaces: they are called *rooms* and the links between them are naturally called *doors*.
In order to avoid the dependance on a central server, the rooms have no "geographical" location relative to the entire world. Furthermore, within our actual prototype, the avatars and objects contained in a given room also have no location relative to the latter. In this basic model, when the avatar is in a given room, it only sees three lists containing respectively the available doors to other rooms, the present avatars and the present objects (see Figure 1). This is the simplest topological model supporting immersion, i.e. the feeling of sharing a space with other users and objects.
Figure 2 illustrates the conceptual model of a simple running world composed of four rooms and inhabited by three avatars. One can see that one of the rooms contains an active object. It is also worth noting that, in order to maintain a consistent view of the world from its various components (i.e. avatars, rooms and active objects), a distributed event model is necessary. The basic mechanism is inspired by the observer pattern [10]. There are event producers and event consumers. The event consumers register event listeners to event producers, also called event sources, in order to be notified by them of events of interest. Figure 3 summarizes this mechanism.


MaDViWorld Avatar
Avatar Door Objects Bag ?
Chat
Hans - A simple battleship game: theGame (in room R2 running on 134.21.9.163)
Defense
Fleet Control
Attack
Hor.
Vert.
OK
Start
Reset
?
About...

Avatar name: *Hans* Room name: *//134.21.9.163/R2*

Doors: //134.21.9.163/R1
Avatars: //134.21.9.163/Sylvia
Objects: theGame
Bag:

Information
A simple battleship game

Events
```
Fri Nov 26 15:23:19 CET 2004 Entering the room: //134.21.9.163/R2
Fri Nov 26 15:23:48 CET 2004 Person //134.21.9.163/Sylvia has entered.
Fri Nov 26 15:24:34 CET 2004 Object added :theGame
```

1 Aircraft Carrier
1 Cruiser
1 Destroyer
1 Submarine
1 Vedette

Wait for your turn!

```
Player Hans has fired on 7,8 : Hit
Player Sylvia has fired on 3,6 : Water
Player Hans has fired on 5,2 : Water
```

Player 1: Hans
Player 2: Sylvia
Observers: 0
☐ Sound enable

Figure 1: A MaDViWorld "basic" avatar user interface: *Its name is Hans and it is located in room R2. Avatar Sylvia is also present, as well as an active object called theGame. Hans has nothing in his bag. Sylvia entered and deposited the active object from her bag and Hans just restarted it.*

3 Implementation

This section shows how our concrete solution for the MaDViWorld model is implemented as an object-oriented framework. The first subsection gives a global view of the software architecture, while the second one concentrates on the programming of new objects.

3.1 Global View

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 4 illustrates the global structure of the framework. First, let us recall the roles of each abstraction layer, which alltogether embody the fundamental principle called *separation of interface and implementation* [2]:

- 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

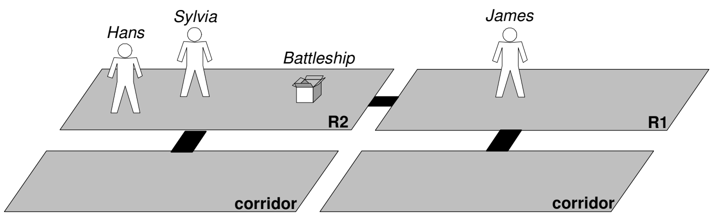


Figure 2: The conceptual view of a simple world

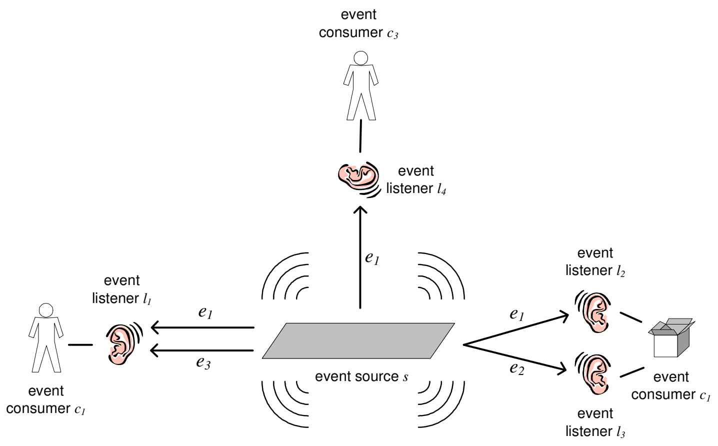


Figure 3: An event source with its listeners and consumers

provides clients with guidelines for using them. The specification of these interfaces could be strengthened by using *Design by Contract* [14]. As MaDViWorld is implemented in the Java language which does not directly support *Design by Contract*, rigorous specification must be provided by a good documentation of the interface methods. 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. The Bridge [10] pattern, for example, addresses this principle.
Second, let us give some details about the vertical tiers which correspond to the three main applications interacting when using virtual worlds.

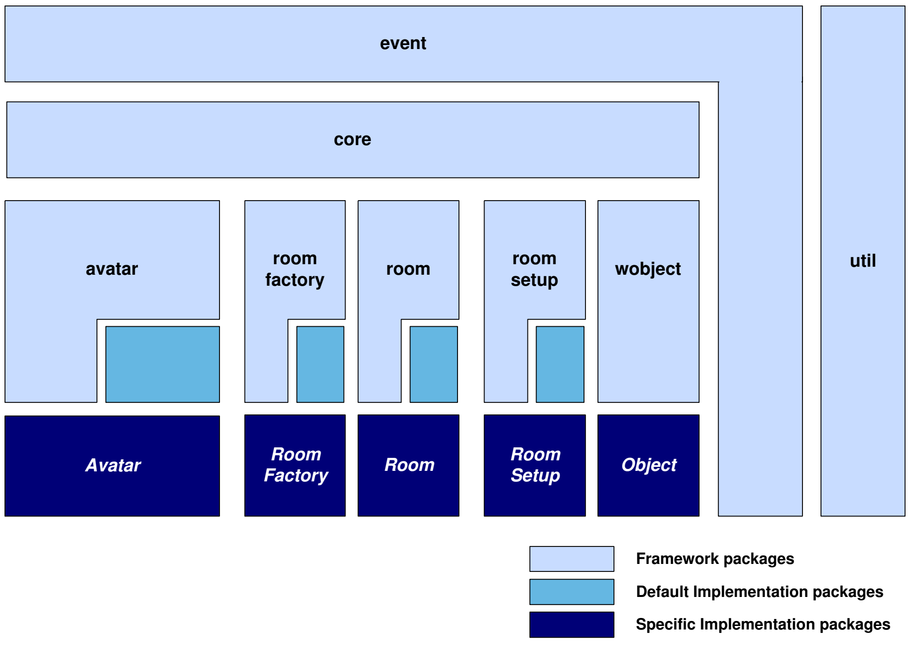


Figure 4: Vertical and horizontal layers of the MaDViWorld framework

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

- *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 [10]. 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.).
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. Indeed, in a massively distributed world, the subspaces are distributed on an arbitrarily large amount of machines. The only requirement is that each machine containing a part of the world runs a small server application and is connected to other machines. With Figure 5 it becomes clear that the simple virtual world of Figure 2 can be sup-

ported by many physical configurations. For instance, four machines interconnected by a network, each hosting one ore several applications (room server and/or avatar application). The most relevant point is, that there is no central server.

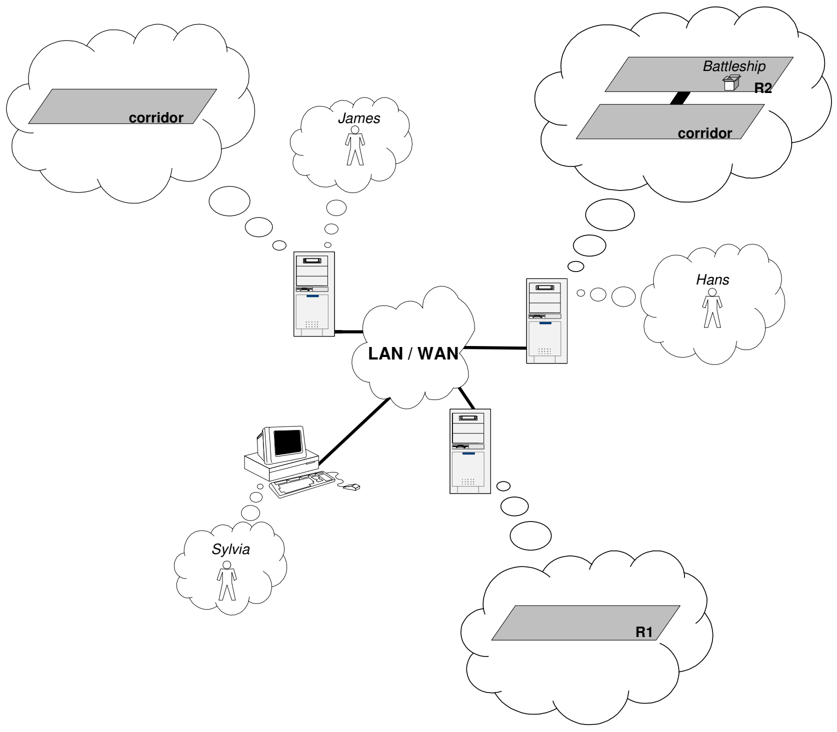


Figure 5: One possible physical configuration for a simple world

3.2 Programming Objects

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.
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, a design pattern fostering a clean separation between presentation and logic is adopted.
Thus, when a developer wants to add a new object NewObj to the framework she has to separately provide[5] the three following pieces of code:

1. the classes supporting the *logic* of the object (see Figure 6). This is done

[5]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 [6].

by implementing a class (NewObjImpl), which extends the abstract WObjectImpl framework class;

2. the classes dedicated to the *presentation*, by extending WObjectGUIImpl (see Figure 7). This graphical class essentially serves as a graphical container of the JPanel subclass NewObjPanel. Hence the latter can directly be designed with any Integrated Development Environment (IDE).

3. 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.

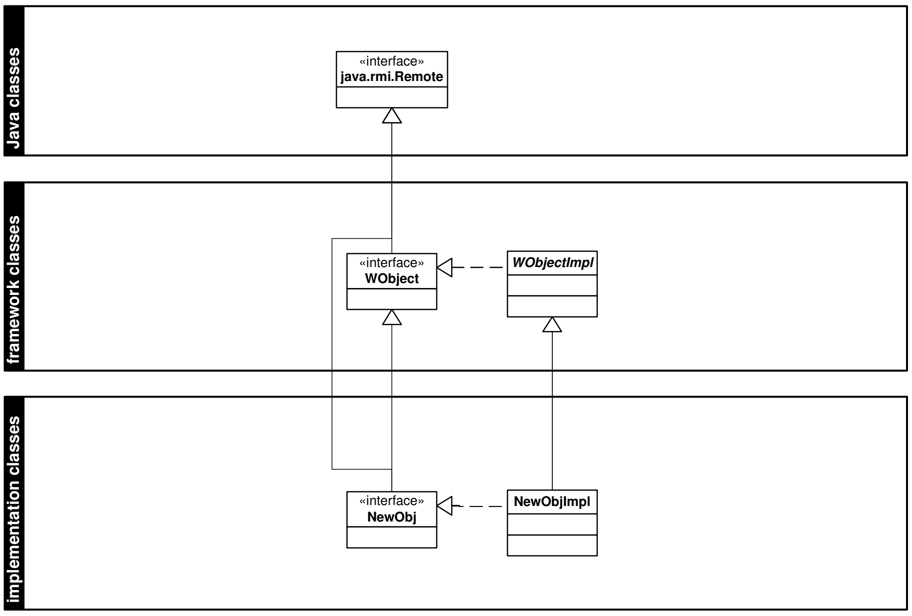


Figure 6: Implementation of the logic part of an object

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 Section 4 fills this gap.
Indeed, the UI registers the NewObjRemoteEventListener depicted on Figure 7 to the logic part of the object, which extends RemoteEventProducer (see Figure 9). 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

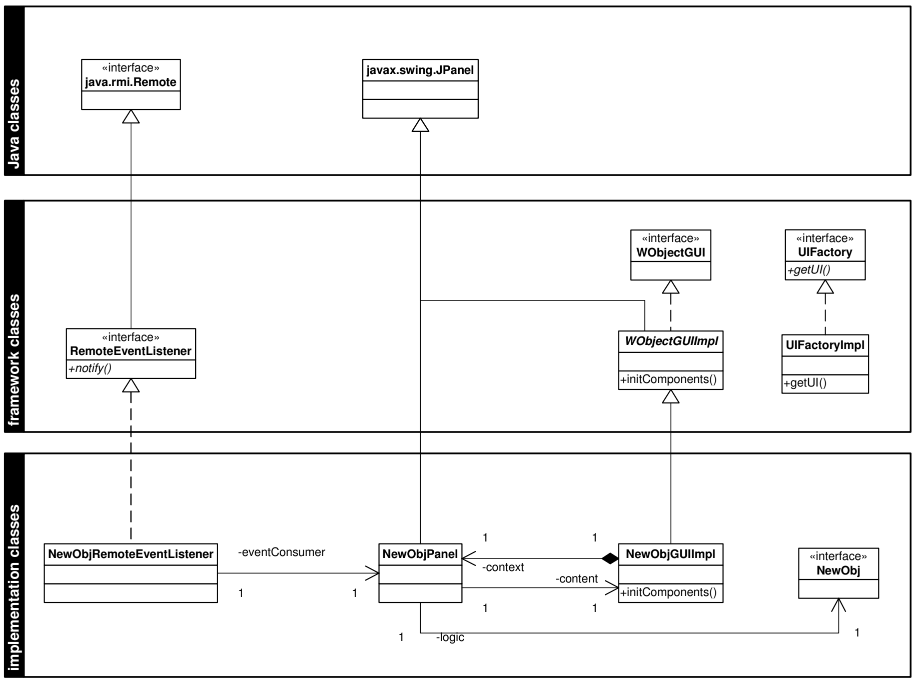


Figure 7: Implementation of the presentation part of an object

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 8 dwells on the mechanism that allows the avatar to get a GUI to a remote object, thus elucidating the role of the UIFactory[6]. This mechanism was inspired by one of the first successes of the Jini.org Jini Community Process, the ServiceUI project [17, 18], 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.

4 The 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. Moreover Subsection 3.2 showed yet another situation where remote events play a central role, namely, offering a communication channel from object logic to its UIs. 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

[6]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 8 does not show how these resources are transferred.

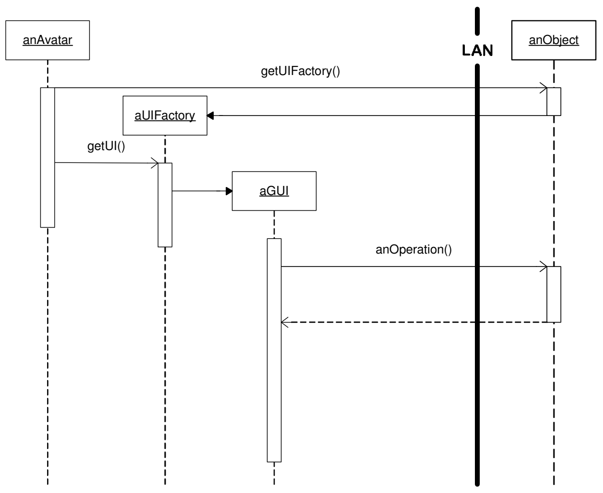


Figure 8: An avatar getting a GUI to an object

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.
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 [9].
It is now time to take a closer look at the framework classes which aim to solve the mentioned problems (see Figure 9):

- 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. There are five possibilities: *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.

[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).

- 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 9 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 [10] 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. Note that the main inspiration of this structure comes from the Obsever [10] pattern and its *publish-subscribe* interaction kind and presents some similarities with the *Jini distributed event programming model*, which is specified in [1] and thoroughly explored in [12].

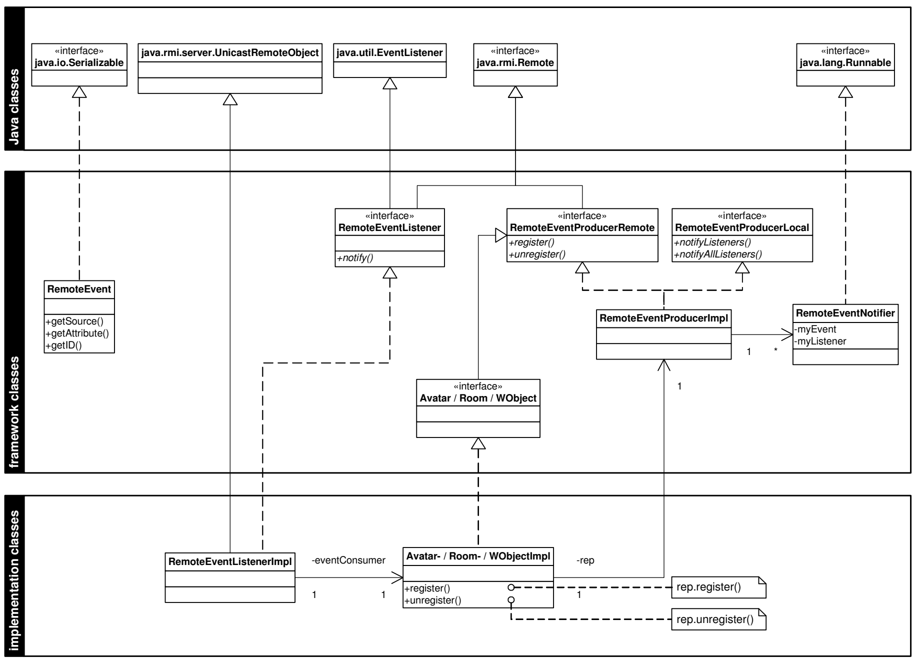


Figure 9: Pattern used for integrating the event model in the framework

5 Security

Security, privacy and trust are crucial elements in virtual world systems. One has to distinguish between two levels of security concerns: *(i)* the system level and *(ii)* the virtual world level. In order to address system level security concerns (e.g., passing through firewalls, encrypted communication protocol, downloaded proxy code trust, etc.), 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 section clarifies how the framework manages security at the virtual world level, i.e. security sensitive *actions inside the virtual world.*

There are several critical actions that objects and avatars may undertake while visiting the rooms of a virtual world: access a given room, use an object, remove or copy an object from a given room, etc. All these interactions concern a room and another entity (an avatar or an object).

Thus the basic principle of MaDViWorld 's security model is that *the subspace grants access rights or privileges to the avatars and objects.* Rooms 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.

The software structure adopted to realize this mechanism adopts the Proxy [10] 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 is solved.

The RoomAccessor's getQuestion() method returns an instance of a Question implementation class. One can see on Figure 10 that the framework offers two default kinds of questions represented by two[8] 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. 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.

The sequence diagram of Figure 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. It is returned to the avatar, which now has a handle for the room.

[8]In fact three subclasses are depicted but the RSAQuestion class is not part of the framework.

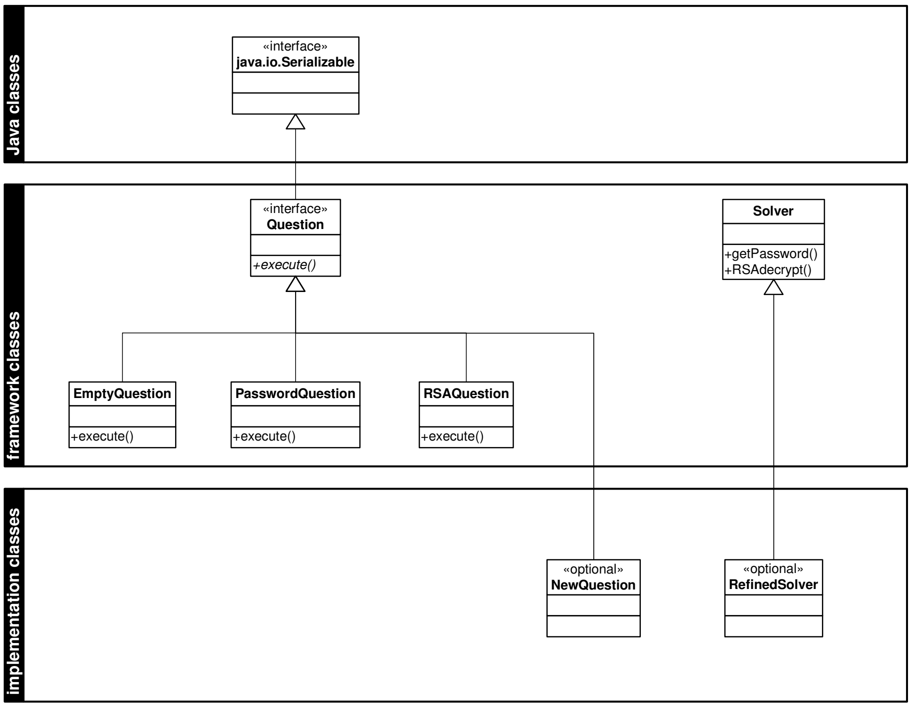


Figure 10: Challenge-response classes relationships

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[9] 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 RSAdecrypt() and *(ii)* provide a corresponding subclass of Question, for instance RSAQuestion. 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, yet flexible and powerful security model.

It will be discussed later.

[9]The RSA algorithm was first described in 1977 by Ronald Rivest, Adi Shamir and Leonard Adleman [16]; the letters RSA are the initials of their surnames. The interested reader can find a comprehensive discussion of this algorithm in [13].

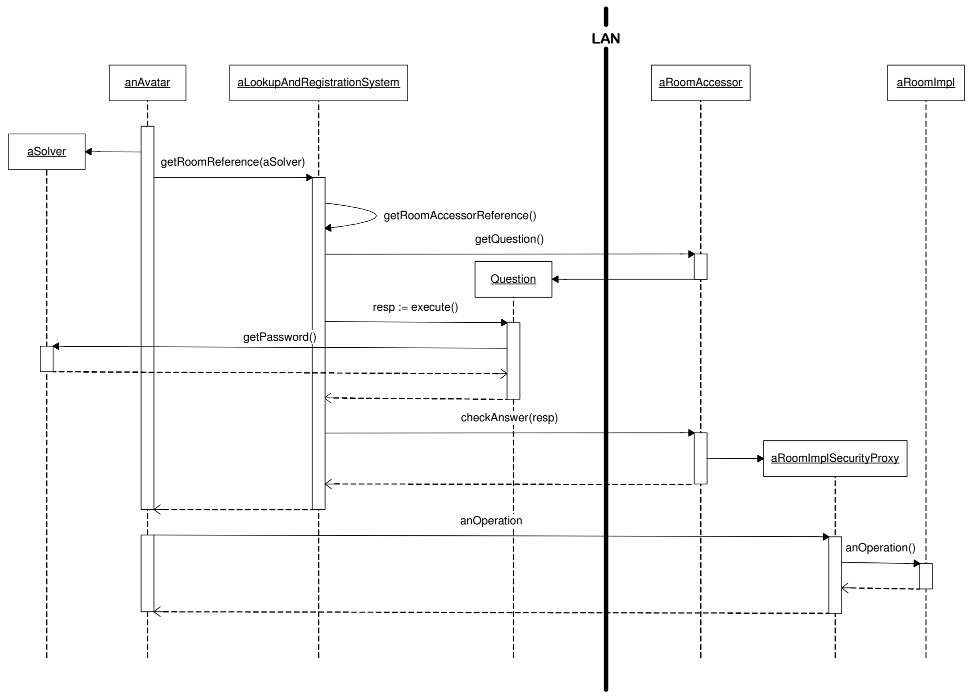


Figure 11: An avatar getting a secure room proxy

6 Conclusion

Designing an extensible and truly decentralized software platform able to support a virtual community based on the MOO paradigm represents a very challenging task at the fringe of today's software engineering technology. In this paper, we have drawn from our experience developing MaDViWorld in order to propose a coherent set of solutions to some of the main questions one must answer in order to embark on such a daunting task. Indeed, the actual version of MaDViWorld is a fully functional framework for creating highly distributed virtual worlds. It has been carefully designed in order to facilitate its enhancement either by extending some of its concrete classes or by implementing the well-documented interfaces of its higher levels.
Although MaDViWorld default avatar (see Figure 1) and room server applications are rather basic (i.e. no immersion into 2D or 3D spaces), the actual implementation is sufficient in order to design interesting virtual worlds by creating a rich enough variety of objects to populate them. This is the reason why we concentrated on facilitating as much as possible the process of programming new types of object with the ultimate goal of instigating a large community of object creators. In order to validate this vision, we launched a series of student projects[10] with the only requirement of enriching the framework by programming new "useful" objects.
Our experience with these projects proved that it is possible for an average Java programmer using the framework to develop her own objects and to test them in a virtual world, with the transparent additional advantages of mobility, remote execution and persistence. Some of the newly created objects are rather specific (e.g. single or

[10]Bachelor or Master level projects realized at the DIUF (see [6]).

multi-user games), while others (e.g. a chat, a whiteboard or a collaborative editor) are generic and could enhance a world by being installed in most of its rooms. A student even took advantage of the object intrinsic mobility and "inter-communication" capabilities in order to program two mobile agents: the first one draws a "map" of the world by exploring it on its own and the second one can arrange appointments for its owner with other users. It is our hope that other programmers will join us in order to continue this "adventure".

References

[1] K. Arnold, B. O'Sullivan, R. W. Scheifler, J. Waldo, and A. Wollrath. *The Jini Specification*. The Jini Technology Series. Addison-Wesley, 1st edition, 1999.

[2] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, 1996.

[3] Y. Fabre, G. Pitel, L. Soubrevilla, E. Marchand, T. Géraud, and A. Demaille. A framework to dynamically manage distributed virtual environments. In J.-C. Heudin, editor, *Virtual Worlds*, volume 1834 of *Lecture Notes in Computer Science*, pages 54–64. Second International Conference, VW 2000, Paris, France, July 2000, Springer-Verlag, 2000.

[4] E. Frécon and M. Stenius. Dive: A scaleable network architecture for distributed virtual environments. *Distributed Systems Engineering Journal (special issue on Distributed Virtual Environments*, 5(3):91–100, June 1998.

[5] P. Fuhrer. *Distributed Virtual Worlds - Abstract Model and Design of the MaDViWorld Software Framework*. PhD thesis, Department of Informatics, University of Fribourg, Switzerland, Nr. 1458, September 2004.

[6] P. Fuhrer. MaDViWorld: Massively Distributed Virtual Worlds. [online], 2005. http://diuf.unifr.ch/softeng/projects/madviworld/index.htm (accessed July 19, 2005).

[7] P. Fuhrer, G. K. Mostéfaoui, and J. Pasquier-Rocha. MaDViWorld : a software framework for massively distributed virtual worlds. *Software - Practice And Experience*, 32(7):645–668, June 2002.

[8] P. Fuhrer and J. Pasquier-Rocha. Massively distributed virtual worlds: A framework approach. In E. A. Nicolas Guelfi and G. 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.

[9] A. Gachet. *A Software Framework for Developing Distributed Cooperative Decision Support Systems*. PhD thesis, Department of Informatics, University of Fribourg, Switzerland, Nr. 1402, February 2003.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley, Massachusetts, 1995.

[11] C. Greenhalgh and S. Benford. MASSIVE: A distributed virtual reality system incorporating spatial trading. In *Proceedings of the 15th International Conference on Distributed Computing Systems (ICDCS'95)*, pages 27–35, Los Alamitos, CA, USA, 1995. IEEE Computer Society Press.

[12] S. Li. *Professional Jini.* Wrox Press Ltd., 2000.

[13] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, October 1996.

[14] B. Meyer. *Object-Oriented Software Construction.* The Object-Oriented Series. Prentice-Hall, 2nd edition, 1997.

[15] E. Reid. Cultural formations in text-based virtual realities. Master's thesis, University of Melbourne, January 1994.

[16] R. L. Rivest, A. Shamir, and L. 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 July 19, 2005, from http://theory.lcs.mit.edu/~rivest/rsapaper.pdf].

[17] B. 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 July 19, 2005, from http://www.javaworld.com/javaworld/jw-10-1999/jw-10-jiniology.html].

[18] B. Venners. *The ServiceUI API Specification (Version 1.1).* Artima Software, October 2002. [Retrieved July 19, 2005, from http://www.artima.com/jini/serviceui/Spec.html].