Using Design Patterns to Build Web-Based 3D-Collaborative Virtual Environments

Chadwick Carreto[1], Rolando Menchaca[2], Leandro Balladares[1], Rolando Quintero[1]

[1] National Polytechnic Institute, Computer Science Research Center,
07738, Mexico City, México
{ballad, quintero}@cic.ipn.mx; chad_carreto@hotmail.com
[2] University of California Santa Cruz, School of Engineering, 1156 High Street,
95064 Santa Cruz, CA, USA
menchaca@soe.ucsc.edu

Abstract. Although in the past decade a lot of research efforts have been concentrated into building multi-user virtual worlds such as 3D collaborative virtual environments (CVE) and gaming platforms [1], the proprietary nature of such systems and their associated complexity has often resulted in large monolithic applications that are difficult to maintain and extend. At the same time, the openness and scalability of the web let us imagine what could be the next step in the evolution of information sharing and more intuitive collaboration - a web in three dimensions to model the world we live in. Undoubtedly, creating such interactive worlds depends on resolving a number of software engineering challenges, most of them resulting from the conflicting requirements of speed vs. reliability, scalability vs. bandwidth usage, replication vs. consistency or flexibility vs. tight-coupling. In the next sections we propose a framework to build 3D Web-based Collaborative Virtual Environments based on existing and well documented design patterns [2] that will help developers to create this kind of applications. This paper also discusses the foundation for an implementation in Java of such a framework based on an interaction model proposed by the authors of this work.

Keywords: Design Patterns, Software Engineering, 3D-Collaborative Virtual Environments, Human Computer Interaction.

1 Introduction

According to [6] a Collaborative Virtual Environment (CVE) is: "...a computer-based, distributed, virtual space or set of places. In such places, people can meet and interact with others, with agents or virtual objects. CVEs might vary in their representation richness from 3D graphical spaces, 2.5D and 2D environments, to text-based environments". Commonly, those CVEs that use 3D graphical spaces as a representation means are knew as 3D-CVEs. Developing a 3D CVE from scratch can be a tedious task as it involves finding the right compromises to minimize both network bandwidth and latency and yet achieve -real time interaction and appealing visual rendering. This involves solving a number of conflicting requirements: (1) to be scalable with no single point of failure a virtual world needs to be distributed, making synchronization between participants more difficult to achieve. (2) To be meaningful to the application some data such as audible audio streams has to be delivered under severe time constraints. To compensate unpredictable latency on the network or unreliable delivery some techniques can be employed but to the expense of bigger bandwidth, itself resulting in less scalability. (3) Complexity grows when there is a need to support user interfaces on various devices of different memory and processing capabilities such as desktops, PDAs and mobile phones.

In [23] a model for the creation of Distributed Virtual Worlds (DVW) through the use of distributed objects is proposed by Menchaca and Quintero; this model called Soul – Body Model aims to reduce the complexity of developing DVW by separating the communication issues form the virtual reality issues. The work described on this paper, based upon the proposed concepts, extends the use of DVWs for the creation of Collaborative Virtual Worlds (CVW). One of the most important characteristics of the framework architecture presented in this paper is that it encapsulates the algorithms and protocols related to the consistency of the world, making this, transparent for the users and facilitating the implementation of CVE. In this manner, algorithms and protocols can be easily extended, optimized and replaced without affecting other entities in the world. The implementation of the framework, as is described in the following paragraphs, enables the use of different actualization schemes for the entities of the world[1] and facilitates an easy distribution management of processes between clients and a server, or otherwise its centralization within a central server. From the developing process point of view, the goal of the collaboration model we propose is not only to develop CVE faster, but also the resulting CVE have similar structures, they are easier to maintain and eventually to integrate. Design patterns were very useful to fulfill with this characteristics, as we describe in the next sections.

2 Interactive – Entities Model

The proposed CVEs are composed by: individuals, artifacts and decorations. Individuals can be users' avatars or autonomous entities (agents) that interact within the CVE through a service based concept. The individuals define the actions that users or agents are able to perform. Artifacts are elements that individuals can interact with, their services are implemented by software components and they aren't embodied by any one user but, unlike agents, these don't realize some task in autonomous way unless some individual request them through one of the services they offer. These artifacts enable collaboration tools such as shared content editors, blackboards, etc. Decorations are static objects (or animated with deterministic time- based behavior) that are visible within the virtual world but don't have collaboration interface. As in a

[1] Identifying actualization schemes is very valuable because there is not a best actualization protocol for every situation [3], [4], [5].

real world, individuals sharing common characteristics can be classified into social groups. In this case, the characteristics that define the property of a group are the individuals' abilities and the way in which they interact with other components of the CVE (individuals or artifacts). Thus, as it is shown in Fig. 1a, the worlds are populated by sets of individuals pertaining to different social groups and by sets of artifacts that provide some type of service.

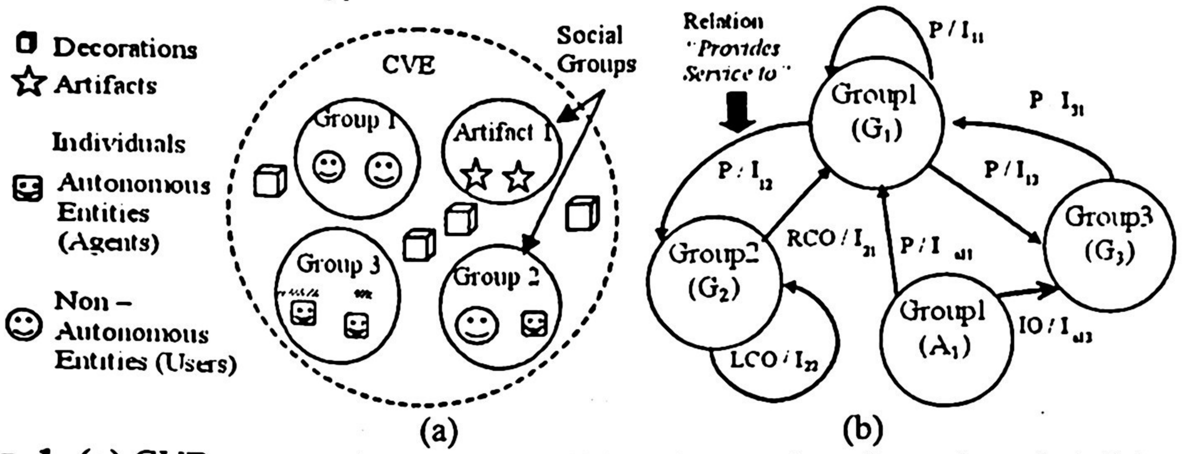


Fig. 1. (a) CVE conceptual model; (b) Collaboration graph: a directed graph defining the collaboration relationships in a CVE.

The set of all existing social groups within a CVE is denoted as *S*, and the set of all types of artifacts as *A*. Each group partially defines the way its elements interact with other group elements. In order to totally define the interactions that can be carried out within the CVE, a directed graph $G=(V,E)$ where $V=S\cup A$ and $E\subseteq S\cup A\times S$ is proposed, and where the edges represent a service relationship labeled with a pair $Event_i$ / $Interface_{a?SG\ TG}$, where: $Event_i$ is the event that will trigger the collaboration between entities from *TG(Target Group)* with entities from *SG (Source Group)*; if *a* is used, then *SG* is an artifacts group; $Event_i$ = type of event = *{Proximity (P), Inside Of (IO), Right Click Over (RCO), Left Click Over (LCO), etc.}*. The relations defined in the graph are translated into graphical interfaces that will appear in the browsers of the users when their avatars (or the avatars of some other individual) produce a collaboration event.

3 Framework and patterns

A framework embodies a generic design, comprised of a set of cooperating classes, which can be adapted to a variety of specific problems within a given domain [6]. A framework is a generic package i.e. one whose contents are intended to be used by "plugging in" specializing elements in place of the parameterized parts of the framework package. Any elements may be substituted for; alternately, distinguished substitutable elements and assumptions about them may be marked explicitly. A framework can be thought of as a "semi-complete" application. Design patterns are a method of codifying design knowledge in separate but interrelated parts [2], [7], [8]. Design patterns can describe the purpose of a framework, can let application programmers use a framework without having to understand how it works in detail, and can teach many of the design details embodied in the framework [9]. We present below the main patterns that make our framework.

3.1 Model-View-Controller (MVC)

The Model-View-Controller is a design pattern largely used and described in the literature that conforms nicely to the previous requirements:

- *The Model*: holds all data relevant to a domain entity or process, and performs behavioral processing on that data.
- *The View*: displays data contained in the Model. It is the view's responsibility to maintain consistency in its presentation when the model changes. This can be achieved by using a push model, where the view registers itself with the model for change notifications, a pull model, where the view is responsible for calling the model when it needs to retrieve the most current data, or a mixture of both models. In the next paragraphs we describe how these models were implemented in our approach.
- *The Controller*: is the glue between View and Model. It reacts to significant events in the View, which may result in manipulation of the Model.

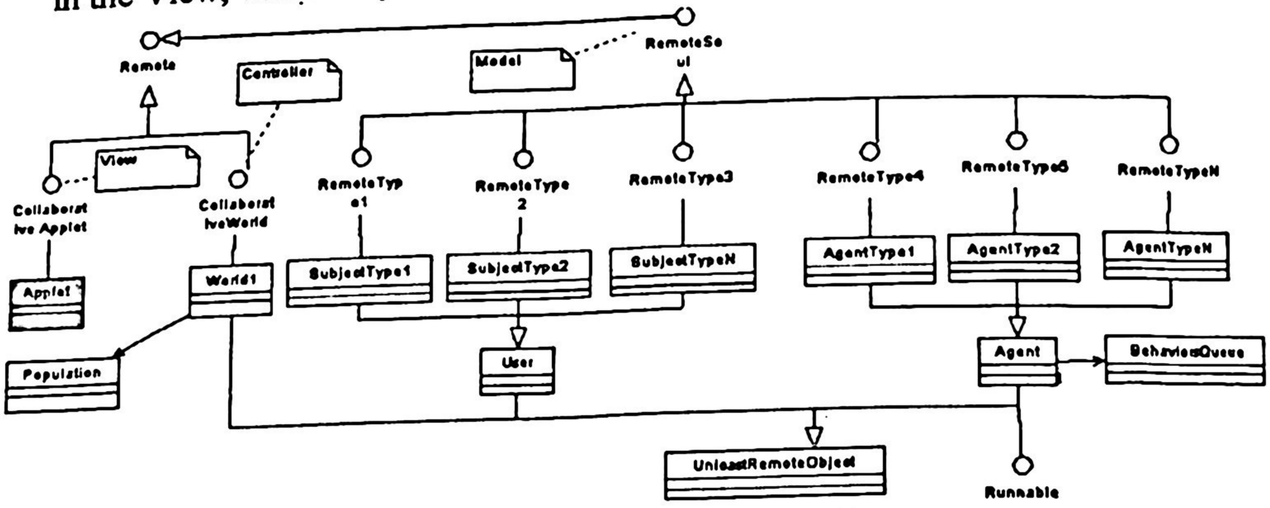


Fig. 2. Main interfaces of the CVEs.

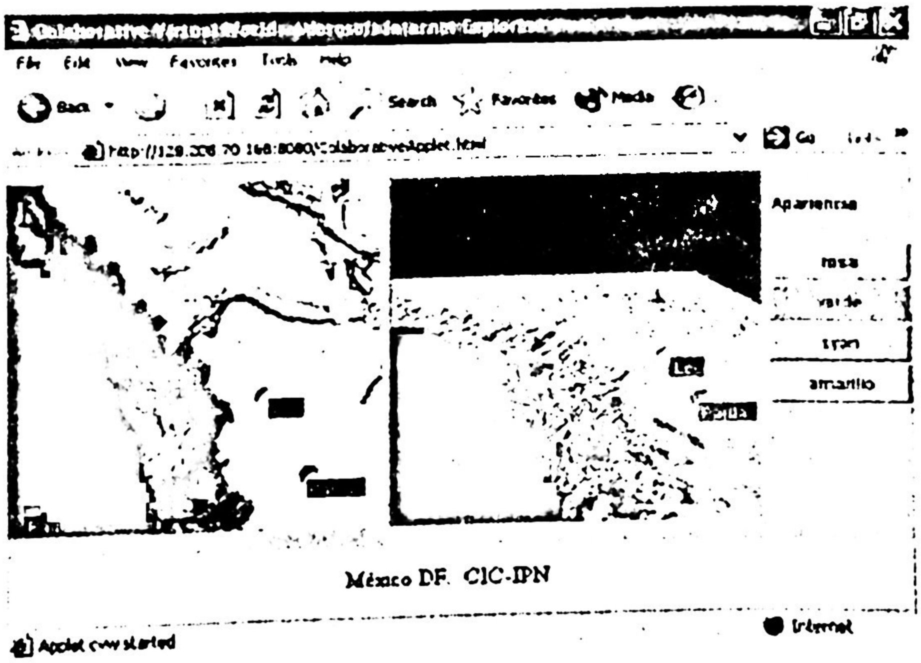
The major interest in this separation of concerns is to minimize the degree of coupling between objects. For example, the same 3D model might be rendered on two different views, one using Java3D [10] and the other OpenGL [11]. In out approach, the CVEs are mainly made up of three classes (corresponding to each one of the components of the Model – View – Controller design pattern): *RemoteSoul* (Model), *CollaborativeWorld* (Controller) and *CollaborativeApplet* (View). The *RemoteSoul* class represents the state and behavior of the individuals and implements the social groups defined in the collaboration graph. A *CollaborativeWorld* class serves as a meeting point for the participants and, as explained further, as a container of souls[2] and references or just as a container of references. The *CollaborativeApplet* class represents the user's environment. It is the class where rendering is carried out and depending on the system architecture (centralized or distributed) the processes associated with the soul of the individual that represents the user or agents are made available. Fig. 2 shows a simplified class diagram of the interfaces that represent the abstractions mentioned. In the same diagram it is possible to observe examples of classes that implement these interfaces. To decouple the Model from the View the *Observer* pattern can be applied [2]. For our purpose, in one of our approaches, the model is the concrete subject and the view is the observer. The concrete subject maintains a list of interested observers. The view registers interest in the model by using the subject's interface. When the model changes it will go through its list of registered views and notify them by calling their `update()` method. Users access the CVE by means of a Web browser. The browser accesses a HTTP server where it obtains the *CollaborativeApplet* (CA) that is in charge of executing all the necessary code to participate in the CVW.

Fig. 3a shows an example of a Collaborative Applet appearance. As it is shown in Fig. 3b, the CA acts as a bridge between the technologies in charge of rendering the CVE and the remote objects that implement the CVE behavior. As was mentioned, the *CollaborativeApplet* is in charge of contacting the *CollaborativeWorld* to get the code necessary to interact within the CVE, meaning references to the souls of all participants and a reference to the soul of the user's avatar (or a factory to construct the soul within the applet). Within the Applet (View), threads are constructed to obtain the state of the objects that implement the souls of subjects and artifacts (Model). We can use different update schemes depending on the nature of the application, such as:

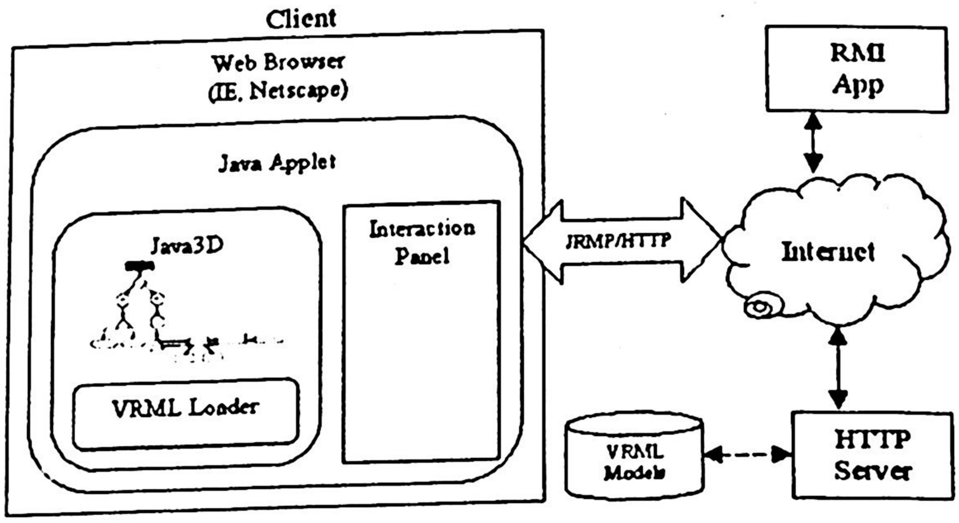
- *Pooling*: a thread is associated with the reference of each active element of the CVE. The thread performs a cycle where it invokes a remote object method to obtain his state. With this information it updates the appearance of the element in the VW.
- *Update*: when the state of a soul (Model) is modified, it makes calls to its references (callbacks) so that clients update the appearance of this element in the View.
- *Both Sides Processing*: the code that is in the browser (View) implements part of the state and behavior of the souls. Local objects make calculations associated with the behavior (Model) of the subjects but they maintain communication between them for synchronization (actually this scheme is not implemented in our approach).

Because the update schemes are implemented between the souls and their references, it is possible to find, in the same CVE, subjects and artifacts that use different update schemes. In other words, each subject has the flexibility to implement the protocol that better covers its particular needs.

[2] From a practical point of view, in the Quintero and Menchaca Model [23] souls are distributed objects which realize all the processing tasks that the subjects need to inhabit the virtual world.



(a)



(b)

Fig. 3. (a) Collaborative Applet GUI; (b) The Collaborative Applet serves as bridge between the technologies in charge of rendering the CVW and the objects that keep their state and implement their behavior.

3.2 Abstract Factory

This pattern documented in [2] provides an interface for creating families of related or dependent objects without specifying their concrete classes. Clients call the `create()` operations to obtain instances but aren't aware of the concrete classes they are using. This enables the factory to encapsulate how objects are created. In out framework, the main task of the *CollaborativeWorld* class (Controller) is to keep a record of the participants of the CVW. These objects generally reside on the same machine as the Web server that hosts the world. When a new user accesses the Web site, he or she downloads a *CollaborativeApplet*. The first operation of the applet is to obtain a remote reference to the object that implements the *CollaborativeWorld* interface. Then, by means of the world's reference, the client requests:

- Registration of a reference to its CollaborativeApplet.
- In the case of the centralized scheme, the *CollaborativeWorld* creates a new soul and returns a reference to it.
- In the case of the distributed scheme, it returns a soul factory (*SoulFactory*) in order to create the soul in the client's machine. In this case, a second call is necessary to register in the *CollaborativeWorld* the reference of the soul that has just been created (see Fig.4).
- References of all the participants, who at the moment, are within the world.

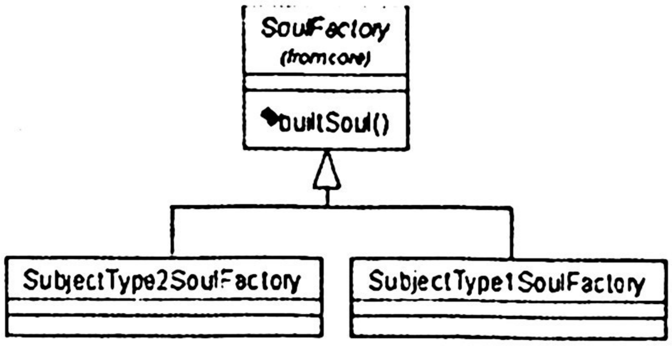


Fig. 4. A fragment of the code that is executed by the *CollaborativeApplet* in order to create the soul in a local way. Depending of the object type, the *CollaborativeApplet* receives the appropriated factory.

Additionally, the server notifies to all other participants that a new individual has entered the world. The *CollaborativeWorld* object produces callbacks to the other participants (*CollaborativeApplets*), sending the reference of the new individual and its geometry as an argument. When an individual leaves the CVW, the *Collaborative-World* is responsible to notify all participants that its reference should be eliminated from their respective containers.

3.3 Prototype

This pattern specifies the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype. The prototype pattern is used when a system should be independent of how its products are created, composed, and represented, and when the classes to instantiate are specified at run-time, for example, by dynamic loading. In our framework, in order to isolate the subjects and the collaborative applets from the details of the interfaces defined by other social groups, all the graphic interfaces must specialize the abstract *GraphicInterface* class.

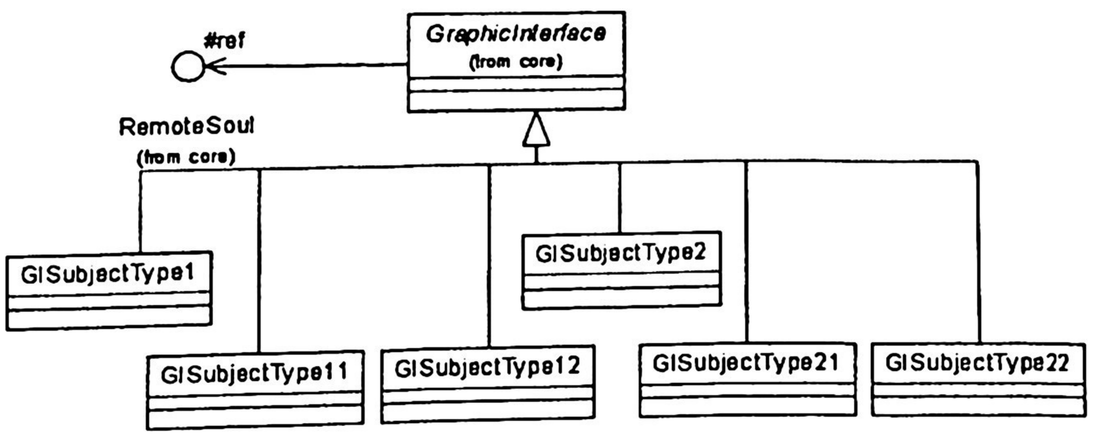


Fig. 5. Diagram of classes of the social groups G1 and G2.

Fig.5 shows the summarized class diagram of the graphic interfaces for two hypothetical social groups: *GISubjectType1* and *GISubjectType2*. It defines the set of graphical components used to access the inherent abilities of the subjects (its respective HS methods). There are two other graphical interfaces for group *G1*: *GISubjectType11* for the interaction with other individuals of the group *G1* (because there is an edge from *G1* to *G1* in the collaboration graph) and *GISubjectType12* for the interaction with individuals of the group *G2* (because there is and edge from *G1* to *G2* in the collaboration graph). There is a similar case for group *G2*.

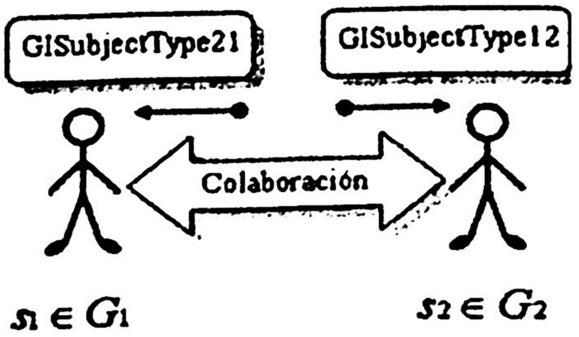


Fig. 6. Interchange of graphical interfaces between two individuals because of a collaborative event.

The collaborative applets don't need to have previous knowledge of the graphical interfaces of the individuals. In the same way, individuals don't need to have previous knowledge of the interfaces that other subjects define. This is because the graphical interfaces are associated with references to the soul of the specific type of subject. When a collaboration event takes place, souls construct the suitable graphical interface (that corresponds to the type of subject with whom it is collaborating) and send it to the target *CollaborativeApplet*. In order to dynamically identify the social group of a subject or the type of an artifact, we use the reflection support within the Java language (see Fig. 6).

3.4 Composite

This pattern composes objects into structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly [2]. In the framework, the *Agent* class represents a common base class for user defined agents. Therefore, from the programmer's point of view, an agent is simply an instance of a user defined Java class that extends the base *Agent* class and implement the corresponding *RemoteTypeN* interface. This implies the inheritance of a basic set of methods that can be called to implement the custom behavior of the agent. A scheduler, internal to the base *Agent* class and hidden to the programmer, automatically manages the scheduling of behaviors. The programmer has to implement the *setup()* method in order to initialize the agent. The programmer should use this initialization procedure to: add tasks to the queue (*BehaviorsQueue* class) of ready tasks using the method *addBehaviour()*. The framework provides ready to use Behavior subclasses that can contain sub-behaviors and execute them according to some policy. For example, a *SequentialBehavior* class is provided, that executes its sub-behaviors one after the other for each *action()* invocation. In order to describe the behaviors of the autonomous entities we define a hierarchy of classes which allows defining simple or composite behaviors (based in the composite design pattern); the types of behaviors are: finite state machine behaviors, sequential behavior, parallel behavior, one shot behavior and cyclic behavior.

4. Area of Interest Management

Area of Interest Management designates a set of algorithms whose aim is to limit the scope of messages received by a participant [12]. Without them the network traffic would be tremendous as a participant would receive every single event and even those of entities that he/she can't see and interact with. As one can adopt many different strategies to control the flow of messages between participants such as based on geographic location, time or functional criteria, the *Strategy* design pattern introduced in [2] can encapsulate each algorithm and make them interchangeable. In our case, the collaborative graph defines some way of area of interest management: one approach is to receive messages and updates only of those individuals of the same social group; another scheme is to receive messages and updates from individual of the same social group and those with which collaboration will be carried out. Actually, we are working in the implementation of these approaches in our framework.

5. Tests

To test the framework, we develop a very simple CVE of a virtual shop in which only will have two types of users: sellers and customers.

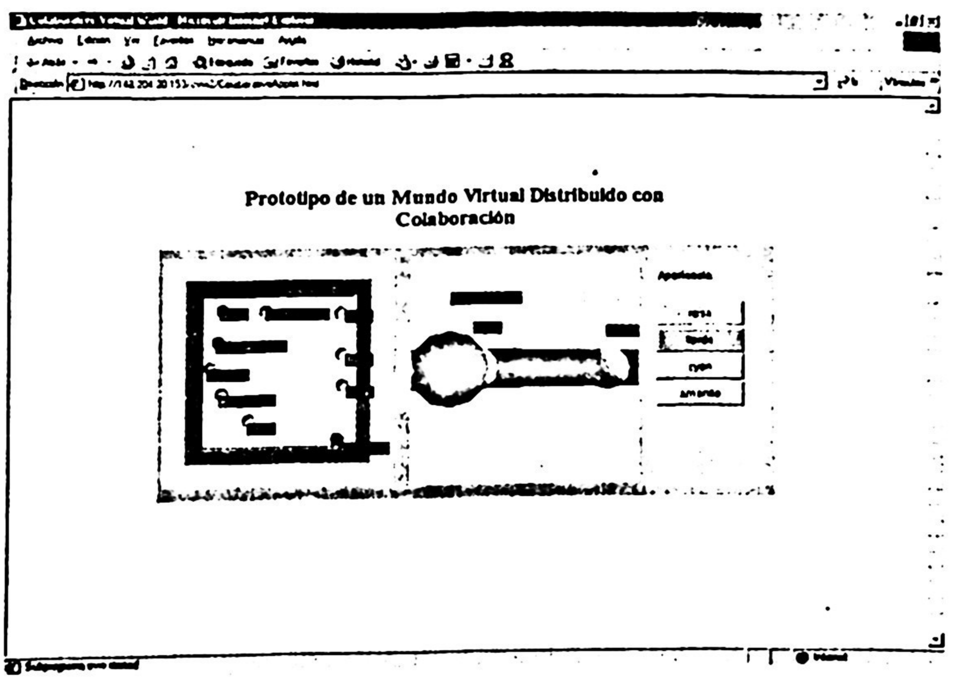


Fig. 7. (a) Client identification to access the CVE; (b) client view of the CVE.

In this first implementation the shop is empty and only sellers and consumers habitat it. We also want that the sellers and consumer can interact according to their roles, that is sellers offer services and products and consumers request services or buy products[3]. Also, we want consumers and seller could communicate through textual messages, like in a chat (in a more complex application could be used the Java Voice XML API in order to support voice communication). For this application we have decided to use the distributed architecture, although centralized architecture could be used too (but supposing the CVE will be populated by many sellers and impulsive consumers, the distributed architecture will be a better option). The updating scheme we have decided to use is update scheme, in this way the view will be updated only when the state of the entities change (when the consumers or sellers change their position in the world). In this case we only have two social groups and the corresponding interactions among them (see Fig. 7).

6. Related Work

CVEs have extensively been studied by various research organizations. However, is being used very little software engineering knowledge in the construction of these applications. Software engineering researchers and people interested in CVW applications are working in this direction, since the methodological and technological point of view [13], [14], [15], [16], [17], [18], [19]. Some related work includes the

[3] In this first implementation we only want to see how to use the tools we have described, so consumers can't buy products and transactions aren't supported.

Java Adaptive Dynamic Environment (JADE) [20] and more recently the NPSNET-V framework [19]. They both specify the notion of a plug-in architecture [21] made of dynamically loadable modules organized into a hierarchy of module containers. In [17] a framework is proposed following the same path, defining a Component Container being able to manage the various components (such as a Network Component for the exchange of messages between machines, or a People Component to manage avatars), the core of that infrastructure follows principles of the OMG's Model Driven Architecture, whose aim is to separate the application logic from the underlying technology [22].

7. Conclusions

From the developing process point of view, the goal of the collaboration model is not only to develop CVE faster, but also the resulting CVE have similar structures. They are easier to maintain and eventually to integrate. One of the most important characteristics of the architecture described is that it encapsulates, inside the souls and its references, the algorithms and protocols related to the consistency of the world. In this manner, algorithms and protocols can be easily extended, optimized and replaced without affecting other entities in the world. As a future work, we must work in extending the model and the framework to support validation of new abilities for entities at run-time; it must be researched some way to give semantics to the virtual worlds. Also we must work in security, quality of service (QoS) and persistence aspects applied to CVE. We are working to improve the model and the framework in order we can develop more interesting applications, such as virtual laboratories, which were one of the reasons we had started this work.

8. Acknowledgements

The authors wish to thank the anonymous referees for their careful reading of the manuscript and their fruitful comments and suggestions. This work was part of a master degree thesis in the Computer Research Center of the IPN [24], and it was supported by: The National Polytechnic Institute of Mexico through CGPI and the COFAA, the Computer Science Research Center and by the Mexican National Council for Science and Technology (CONACyT).

References

1. Frécon, E., Stenius, M.: DIVE: A Scaleable Network Architecture for Distributed Virtual Environments. Dist. Systems Engineering Journal (DSEJ), 5 (1998) 91-100
2. Gamma et al.: Design Patterns: Elements of Reusable Object- Oriented Software. Addison-Wesley (1997)

3. Atul, P., Hyong, S., Jang, H.: Data Management Issues and Trade-Offs in CSCW Systems". IEEE Transactions on Knowledge and Data Engineering, Vol. 11 No. 1 (1999)
4. Hans-Peter, D., Garcia-Luna-Aceves, J.: Group Coordination Support for Synchronous Internet Collaboration. IEEE Internet Computing, March-April (1999)
5. Vidot, N., et-al.: Copies Convergence in a Distributed Real-Time Collaborative Environment. Proceeding of the ACM 2000 Conference on Computer Supported Cooperative work. Philadelphia, Pennsylvania, United States (2000).
6. Sean, C., Potel, M.: Inside Taligent Technology. Reading, MA, Addison-Wesley (1995)
7. Alexander, C. et al.: A Pattern Language: Towns, Buildings, Construction. Oxford University Press (1977)
8. Borcher, J.: A Pattern Approach to Interaction Design. Wiley (2001)
9. Johnson, R.: Documenting Frameworks Using Patterns". Proc. of OOPSLA'92 (1992)
10. Sowizral, H., Rushforth, K., Deering, M.: The Java 3D API Specification. 2nd. Edition, The Java Series, Sun Microsystems, Addison Wesley (2000)
11. Angel, E.: Interactive Computer Graphics: A Top down Approach with Open GL. Second Edition, Addison Wesley (2000)
12. Elizabeth, F., David, N., Alan, J. (Editors): Collaborative Virtual Environments: Digital Places and Spaces for Interaction. Springer-Verlang, ISBN 1-85233-244-1 (2001)
13. Fencott, C.: Towards a Design Methodology for Virtual Environments. Proc. of the Workshop on User Centered Design and Implementation of Virtual Environments. University of Teeside (1999).
14. Duff, J., W., Purtilo, J., Capps, M., Stotts, D.: Software Engineering of Distributed Simulation Environments. IEEE Proceedings of WET ICE '96 (1996)
15. Sanchez, M-I, De Amescua, A., Cuadrado, J-J., De Antonio, A.: Software Engineering and HCI Techniques Joined to Develop Virtual Environments. Proc. of the ICSE'03International Conference on Software Engineering: Bridging the Gaps Between Software Engineering and Human-Computer Interaction, Portland, Oregon, May (2003)
16. Méndez, G., Sánchez, I., De Antonio, A.: Towards a Development Methodology for the Construction of Virtual Environments. Proceedings of the VI Jornadas de Ingeniería de Software y Bases de Datos (JISBD 2001), Almagro (Ciudad Real), (2001)
17. Alexandre, T.; Using Design Patterns to Build Dynamically Extensible Collaborative Virtual Environments. ACM Proc. of PPPJ'03, June 16-18, Kilkenny City, Ireland (2003)
18. Garcia, P., Montala, O., Pairto, C., Rallo, R., Skarmeta, A.: MOVE: Component Groupwar Foundations for Collaborative Virtual Environments. ACM Conference on CVE (2002) 55-62
19. Kapolka, D., Capps, M.: A Unified Component Framework for Dynamically Extensible Virtual Environments. In Proceedings of CVE'02, September (2002)
20. Oliveira, M., Crowcroft, J., Slater, M.: Component Framework Infrastructure for Distributed Virtual Environments, San Francisco, ACM CVE 2000, Sept. (2000)
21. Douglass, B.: Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems, Add.-Wesley, Sept. (2002)
22. Mellor, S., Balcer, M.: Executable UML: A Foundation for Model-Driven Architecture, Addison-Wesley, (2002)
23. Menchaca R., Quintero R.: Distributed Virtual Worlds for Collaborative Work based on Java RMI and VRML, Proceedings of the IEEE 6th International Workshop on Groupware CRIWG (2000)
24. Carreto, C.: Architecture for Collaborative Virtual Worlds, Master Thesis, Computer Science Research Center of the National Polytechnic Institute, Mexico, in Spanish (2004)