Journal of Network and Computer Applications (2000) **23**, 291–309
doi:10.1006/jnca.2000.0115, available online at http://www.idealibrary.com on IDEAL®

A pattern language for virtual environments

Alicia Díaz* **and Alejandro Fernández†**

**Lifia, UNLP CC 11, La Plata, Argentina. Also at IUGD, Posadas, Argentina.*
E-mail: alicia@lifia.info.unlp.edu.ar
†*Concert, GMD-IPSI Darmstadt, Germany. Also at Lifia, La Plata, Argentina.*
E-mail: casco@lifia.info.unlp.edu.ar

Designing virtual environments is not an easy task because of the number of activities to coordinate. Describing the virtual space, designing the inhabiting objects, and defining the behaviour of rooms, objects and users according to their surrounding context are some of them. Focusing on MOO environments, this paper describes a simple object-oriented model to design the conceptual aspects of a virtual environment. It then presents a set of design patterns to help VE designers solve recurrent design problems. These patterns provide solutions to the design of virtual space, mobility and behavioural issues. They are named Area, Gate, Locomotion, Transport and Collector. This pattern catalogue represents a first step towards a pattern language for the design of virtual environments.

1. Introduction

Research in virtual reality and computer supported cooperative work has recently evolved into a new area: *Cooperative Virtual Environment* (CVE). A CVE is a network-based virtual space, where distributed users synchronously or asynchronously collaborate and cooperate with one another in real time [1]. A CVE is also characterized by: (1) A *virtual space*. Most CVE are built upon a spatial metaphor such as a building or a city to place users and virtual objects. This metaphor is referred to as the *virtual space*. The virtual space is generally decomposed into smaller virtual spaces (virtual space units), frequently known as rooms or regions. In this paper, we focus on the idea of place and not space. A place is a location where the action happens rather than a physical representation. Harrison and Dourish in [2,3] state that '*space is the opportunity; place is the understood reality*'. This conception of a virtual environment composed of places is consistent with that of MOOs. (2) *Inhabitants*. Objects populate the virtual space. Some common examples of objects are a robot, a shopping basket, a document, a conference table, a bus and a blackboard. Users are also represented as objects. (3) *User embodiment*. In CVEs, users have an appropriate body image representation [4]. In non-textual CVE they are represented by special objects called *avatars* (a 3D representation). In textual CVE such as MOOs [5], users have a textual description and are called *characters* as in LambdaMOO [6]. An avatar is a special kind of virtual object that is manipulated by its owning user.

1084–8045/00/030291 + 19 $35.00/0

In the rest of the paper, we will call these characters *avatars* [7]. (4) *Mobility*. Users and Objects may be able to navigate the CVE by moving around the virtual space. On the way, they can meet and interact with other users, objects, items or goods. (5) The *behaviour*. Object behaviour often depends on context. A context describes a 'circumstance' and is given by the state of the virtual space and the set of surrounding objects and users.

At present, there are many applications of virtual worlds: for work, for meetings, for learning or teaching, for business, for shopping and for entertainment, all of them emulating some aspect of the real world. Most of them use digital world interfaces, virtual reality, and augmented reality. Currently, many are built using VRML, Active Worlds or Viscape technology. In Active Worlds you can meet people from all over the planet, explore virtual worlds, play online games, shop, surf the 2D and 3D web, stake a claim and build your own virtual home on the Net. More details are available at *http://www.activeworlds.com*. VRML, the Virtual Reality Markup Language, is an attempt to extend the web into the domain of three-dimensional graphics. VRML 'worlds' can depict realistic or other-worldly places containing objects that link to other documents or VRML worlds on the web. Another alternative is Viscape, a browser for 3D pages (see superscape.com at *http://204.162.100.100/*).

On the other hand, MOO environments are simply textual [5]. A MOO (Multi-user Domain/Dungeon Object-Oriented) is a multi-user, programmable, text-based virtual reality. MOOs are easy to customize and have powerful networking abilities. These characteristics have made them a popular choice for social virtual realities and conferencing systems, as well as for the more traditional games such as their predecessors MUDs. In a MOO environment, multiple users can communicate with each other in real-time, and move around within a set of linked virtual *rooms*. Each room is associated with a textual description, and may contain one or more objects, each with their own textual description. MOOs grew from the efforts of their citizens using the possibility to build rooms, connect them, create objects, write programs and participate in the organization of the community. Special programming languages are used for that purpose. For example, LambdaMoo uses the LambdaMoo language that is a relatively small and simple object-oriented language, designed to be easy to learn for non-programmers; however most complex tasks still require some significant programming abilities.

The design of a virtual world is not trivial. Modelling and design techniques must deal with architecture, spatial navigation or mobility, objects and users inhabiting the virtual space, and they must also take into account the dynamic behaviour of the virtual environment, because an object's behaviour may depend on the context.

The purpose of this work is to shed light on the design of object-oriented virtual environments. Our goal is to record the experience of designing virtual environment applications in a set of design patterns. As far as we know, design

patterns for virtual environment applications remain unexplored. As a more ambitious goal we plan to develop a pattern language for designing virtual environment applications.

Design Patterns [8] describe problems that occur repeatedly, and present the core of the solution to a problem in such a way that they can be used many times in different contexts and applications. Patterns enable widespread reuse of software architectures and improve communication within and across development teams by providing a concise shared vocabulary. They explicitly capture knowledge that designers use implicitly. Pattern descriptions provide a framework for recording tradeoffs and design alternatives. A Pattern Language is a partially ordered set of related design patterns that work together in the context of a certain application domain.

The collection of patterns presented here aims at the design of the space, mobility and behavioural issues of virtual environments. It includes the patterns Area, Gate, Locomotion, Transport and Collector, representing a set of commonly found interaction patterns and the principles that rule them. This first approach to a virtual environment pattern language could be improved by considering cooperation capabilities (an interesting approach to design patterns for collaborative systems is presented in [9]). Our virtual environment patterns are specially suited for MOO virtual environments because of their underlying object model, although this proposal could be broadened to include non-text-based environments as well.

The rest of this paper is organized as follows. In Section 2, inspired by MOO environments, we present an object-oriented description of virtual environment that serves as a case study. Section 3 presents an overview of design patterns and pattern languages. Section 4 presents a catalogue of design patterns for object-oriented virtual worlds. Finally, Section 5 contains conclusions and topics for future work.

2. Designing object-oriented virtual worlds: A case study

The design of a virtual world is not easy because of the number of activities to coordinate. Any design technique or methodology must support the description of a virtual space, the description of objects and their capabilities, the definition of users and their functionality and privileges, and the population of the virtual space and cooperation spaces.

Taking ideas from existing virtual environments, we next shape an OO design model for virtual worlds. This model provides primitives to describe the structure and behaviour of objects, rooms and users.

From an object-oriented point of view, rooms, avatars, and objects populating the world are all ‘objects’. They interact with one another by sending messages. We will refer to all of these objects as *virtual objects*. Any *virtual object* exhibits behaviour and has an internal structure. Depending on its nature (being a room,

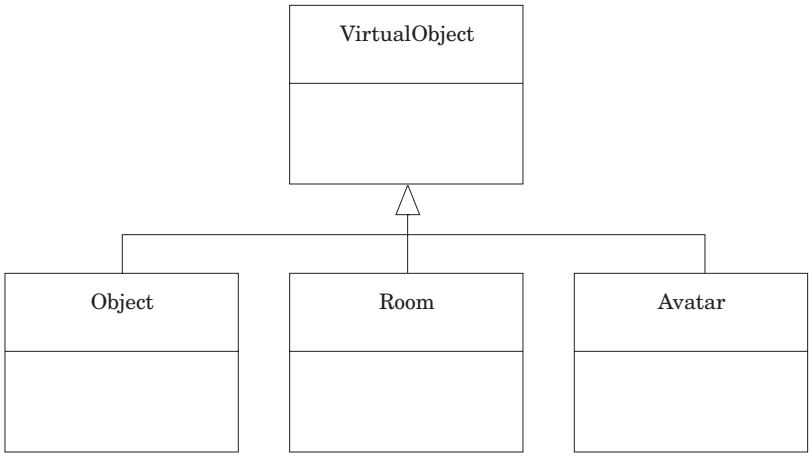

Figure 1. The virtual object hierarchy.

an object or an avatar) specific design constructions are needed. For example, rooms have a description of their space, objects can move and so can avatars, avatars can communicate with each other, etc.

Properties shared by all *virtual objects* can be encapsulated in class VirtualObject. This class is specialized in its subclasses: *Object, Room* and *Avatar* (see Fig. 1).

VirtualObjects are designed following well-known object-oriented design techniques [10–12]. Inheritance and aggregation can be used to model complex objects, and delegation can be used to distribute behaviour.

VirtualObjects can possess or collect other objects, and *VirtualObjects* that collect other objects work as containers. Container objects model real-world objects like bags, shopping carts and pockets. A container object has to describe a proper interface to handle its collected objects. It also has to define how its behaviour affects the behaviour of the objects that it collects. For example, if a container object moves, the objects it collects must move too. Another example is a sale container object affecting the price of the objects it comprises.

The virtual space is a set of connected *rooms — smaller* virtual spaces. *Rooms* represent places and are the minimal unit to describe the virtual space. Each room has a particular purpose, for example, many simple rooms can describe the virtual space for a learning environment, one for each activity involved in the learning/teaching process (e.g. to hold lectures, to do homework, or to participate in a cooperative activity).

Objects populate rooms. For example a blackboard is in a classroom, a slide projector in a conference room. The presence of objects in a room can be transient or permanent. The blackboard and the slide projector are permanent objects in the classroom, but the slides shown by a slide projector are occasional; the teacher brings them.

There are many kinds of objects. They can be classified by taking into account their behaviour and the relationships they establish with the space they inhabit, the user and other objects. We categorize objects as:

- *passive object*: does not exhibit behaviour (e.g. a wall in a room);
- *reactive object*: exhibits behaviour; activated by a user action (e.g. by a click), by the room or by another object;
- *active object*: exhibits an autonomous behaviour;
- *movable* or *non-movable object*: whether the location of the object can change or not, respectively.

In this work, only passive, reactive and movable or non-movable objects will be concentrated on. The only active feature considered is here in relation to autonomous movable objects. Other active objects like agents are out the scope of this paper.

Rooms are connected with other rooms by exits. Exits allow users and objects to move from one room to another. Two connected rooms share the same exit object. A door plays a double role. From one room it is seen as an exit and from the other as an entrance.

A virtual object behaves dynamically. This means it changes its behaviour dynamically. This could happen because it changes its properties over time, or its awareness changes because it changes its location, or other surrounding virtual objects change their location, or other virtual objects change their properties.

In many cases, objects behave depending on context. *Context* is an abstract concept that describes a particular situation in the virtual environment. The neighbour objects and their state, the object's location, the room and its states, the room inhabitants, the role-played by each avatar and the 'virtual environment history' characterize a context. The virtual environment history records every event that has occurred in the environment.

3. Design patterns and pattern languages

Design patterns [7] are being increasingly used in software design but, as far as we know, they remain unexplored in the field of virtual worlds.

A software design pattern describes a family of solutions to a software design problem. It consists of one or several software design elements such as modules, interfaces, classes, objects, methods, functions, processes, threads, etc., relationships among the elements, and a behavioural description. Example design patterns are Model/View/Controller, Blackboard, Client/Server and Process Control.

The purpose of design patterns is to capture software design know-how and make it reusable. Design patterns can improve the structure of software, simplify maintenance and help avoid architectural drift. Design patterns also improve

communication among software developers, and empower less experienced personnel to produce high-quality designs. One of the most important types of reuse is design reuse, and design patterns are a good means for recording design experience.

A design pattern systematically names, explains and evaluates an important and recurrent design in software systems. Design patterns make it easier to reuse successful designs and architectures. They describe problems that occur repeatedly, and describe the core of the solution to that problem in such a way that we can use this solution many times in different contexts and applications. It is important to emphasize that patterns are not invented, but discovered in existing software or working environments.

A design pattern is described by stating the context in which the pattern may be applied, the problem and interacting forces that bring it to life, and the collaborating elements that make up the reusable solution. These elements are described in an abstract way because patterns are like templates that can be applied in many different situations. The important elements of a pattern are the responsibilities that must be assigned to each component and the thread of collaborations among them that solve the recurrent problem in the specific context. The consequences and tradeoffs of applying the pattern are also important because they allow evaluation of design alternatives.

The main advantages of the use of patterns can be summarized as follows, as discussed in [13]:

- *Patterns enable widespread reuse of software architectures*. Reusing architectures is more valuable than reusing algorithms or data structures. In the case of virtual realities this is obvious because it is quite difficult to reuse concrete components.
- *Patterns improve communication within and across software development teams as they provide a shared concise vocabulary*. Using patterns, the level of discourse among team members has a higher level of abstraction.
- *Patterns explicitly capture knowledge that designers use implicitly*. Though expert designers usually make good decision, they do not document what problem they are solving and the rationale for that solution. Explicit design patterns are helpful for training new developers as they can learn from others' experience.
- *Pattern descriptions provide a framework for recording tradeoffs and design alternatives*. This complements the previous statement as we have a clear understanding for adopting a particular solution.

To capture patterns in a useful manner, certain information needs to be provided, such as: including concrete examples to help understand how to implement a pattern when it is too abstract; choosing pattern names carefully and

using them consistently; carefully documenting the contexts where patterns apply or not; and using a fixed, uniform and self-explanatory format to communicate an integrated set of patterns clearly and concisely.

The Patterns movement began in the area of architectural design almost 20 years ago with the work of Christopher Alexander [14]. Recently, the object-oriented community has begun to discuss the subject, and an impressive corpus of work has been developed. This work has been mainly focused on two different areas: design patterns and pattern languages. Design patterns are general enough to solve recurrent design problems in different domains. An example is the Observer design pattern that describes one-to-many dependency between objects so that when one object changes its state, all its dependents are notified and updated automatically. Another design pattern—the Strategy design pattern—deals with families of algorithms, encapsulating each one and making them interchangeable. Patterns are now considered by many to be the very basic building blocks of a software designer's daily work.

There is no fixed format to describe patterns, although the essential elements must include the name, the problem, the solution and the consequences. Design patterns are usually found in the format proposed by [7], named the 'GOF' format (Gang-Of-Four, the authors of the book). This format is very complete and detailed, which makes it almost straightforward to implement the described pattern in software. The GOF format allows recording the rationales for the use of the pattern in the 'Motivation' section and the tradeoffs involved in its use in the 'Consequences' section.

More abstract patterns, i.e. those that encompass more abstract solutions that can be implemented in many different ways, as well as organizational patterns, are usually represented in the 'Alexandrian' format that contains name, context, problem, solution and related patterns [14].

In some domains such as, for example, communications or real-time systems, design problems may be more specific, and therefore more specialized patterns arise [13]. They are usually the basis of a pattern language, a partially ordered set of related patterns that work together in the context of a certain application domain. Pattern languages can be used as a mechanism for describing procedures with many steps or a complex solution to a complex problem [13]. Each pattern solves a specific problem within the context of the language, and patterns in a pattern language are usually related to each other. Relationships express that some patterns usually appear together, that some patterns are mutually complementary or exclusive.

Pattern languages have been developed for different areas of program design, for shaping complex organizations and their development processes [16] and even for writing patterns [15]. Pattern languages are powerful tools for transmitting experience, and the objective of our work is to establish the basis for a pattern language for virtual reality applications.

4. Pattern language for virtual worlds

In Section 2, an example of an object-oriented model was presented, where the base principles of object technology have been used for building virtual environments. Although designers can be helped by GOF patterns for designing virtual worlds, this section presents a pattern catalogue that complements object-oriented design of virtual environments. Section 4.1 introduces the pattern catalogue. The template used to document the patterns in this catalogue is close to that in GOF. The solution section is specified using the Unified Modeling Language (UML), as described in [17]. Sections named *Structure*, *Collaborations* in the original GoF format have been replaced by one section called *Solution*.

4.1 *Pattern catalogue for virtual worlds*

The patterns in this catalogue are specific to virtual worlds and comprise most of the problems/design aspects that we have made reference to. Patterns *Area* and *Gate* deal with structural design of virtual worlds (places). *Locomotion* and *Transport* deal with navigation. Finally, *Collector* reflects commonly-found behaviour of virtual objects.

Patterns Area and Gate are closely related, as are Locomotion and Transport. Area deals with modelling the special structure of the virtual world, places where actions take place; Gates allow connecting Areas to shape meaningful and consistent structures. Locomotion establishes the principles for objects to move around the virtual world. The use of Gates and Transports depends on Locomotion. Transports simplify moving objects though complex structures or following complex paths. After this introduction, we will now present the individual patterns.

Name: Area

Also known as:
Place

Intent:
Build objects capable of containing/holding other objects. Represent perception boundaries. Represent the place where actions take place. Areas are the simplest units of space.

Motivation:
You are building a virtual Museum. You have already created the pictures and statues for your visitors to enjoy. Now it is time to arrange them in different areas according to their historical period and source to make museum visits simpler and more pleasant. These areas must be objects capable of containing other objects: pictures, statues and eventually the avatars visiting them. They must be disjoint, making it impossible for an object to be in two places at the same time. When

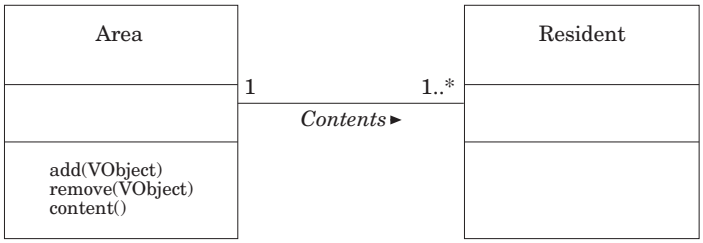

Figure 2. An UML specification of Area pattern. *Area* and *Resident* classes represent its participants.

visitors enter an area, they perceive all objects inside it and nothing else. Areas act as perception boundaries.

Considering the model presented in Section 2, space is discrete. This means that there is no distinction between the location of two objects in the same area. This model does not support relative location of objects with reference to the space they inhabit. All pictures, statues and avatars in an area are in the 'same' place. Areas are the unit of space.

Rooms are examples of Areas. Rooms are considered to be primitive constructions of the framework described in this paper, but they are instances (examples) of the Area pattern. You can use rooms to organize your museum.

Solution:

Area pattern has two participant objects: Area and Residents as is shown in Figure 2.

- *Area*. Areas know all the objects they contain. Area implements a protocol for adding/removing objects.
- *Resident*. Resident is the object contained in the area. Initially, it can be instance of any class.

Consequences:

1. Rooms are defined as disjoint structures for holding objects. This restriction contributes towards enforcing unique location of objects.
2. To ensure uniqueness of location, Resident must implement Locomotion.

Implementation:

1. *In-area navigation.* As previously mentioned, our model considers space to be discrete with areas as unit. Therefore it is not possible to move (or change location) within an area. Combining many areas together can simulate *larger* areas. Connecting two or more areas in a line, for example, can simulate a corridor. With some help from the virtual programming environment, areas in the corridor can be 'decorated', to look as a whole. However, a single area will define perception boundaries for an object in the corridor it inhabits.

2. *Areas within areas*. As depicted by the class diagram in the solution section of this pattern, an area can contain any virtual object. In fact, areas can also contain other areas. A pocket carrying a wallet with coins is an example. The pocket is an area holding the wallet and maybe other objects. The wallet itself is also an area containing, at least, coins.

Known uses:
Room, Bag, Pocket.

Related patterns:
Locomotion. Locomotion defines the basic mechanisms for moving objects between Areas. Both patterns cooperate to ensure objects inhabit one place at a time (location uniqueness).
Gate. Gates are used to connect Areas.

Name: Gate

Also known as:
Entrance, Exit

Intent:
Communicate areas. Suggest navigation paths and enforce spatial distribution.

Motivation:
You are building a virtual world, defining rooms and populating them with avatars and objects. In this context, objects and avatars can only interact with peers in the same room, which limits their possibilities. A mechanism for navigation between rooms is necessary to fully exploit virtual world facilities and take profit of objects located elsewhere. Any navigation mechanism must ensure world consistency (as regards metaphor properties, spatial distribution, etc.) and enable the enforcement of restrictions such us access control.

A gate is a particular type of object that lets avatars and other objects navigate between rooms. At the same time gates serve as an access restriction mechanism and assist shaping the world. Gates enforce spatial distribution of rooms narrowing the distance between connected rooms.

A real-world door connecting two rooms is located exactly between those rooms. It is visible from both rooms. When someone tries to open the door it is clear where this person is located and the direction he/she wants to go. Once the door in opened, the person moves towards the other room. In our object-oriented model the gate is in charge of moving the object from one room to the other. Suppose a virtual objects tries to go through a door. It is represented as the object making a request on the door (*cross()*). What the door must then do is to move the requestor from the room it is located in to the other room the door connects.

Structure:
Gate pattern has three participants as is shown in Figure 3; they are Requestor, Gate and Area.

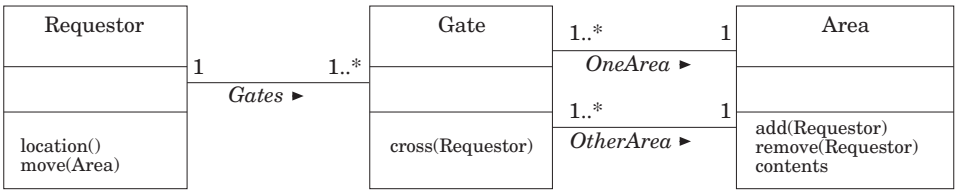

Figure 3. An UML specification of Gate pattern. Requestor, Gate and Area are its participants.

- *Requestor*. The Requestor is the object that wants to cross the door. It sends *cross()* to the door and the door does the rest of the work. The Requestor must implement Locomotion. The Requestor may know many doors at a time. The diagram represents this relationship as an object reference, (Gates). However, it is usually given by context perception—a virtual object knows all the objects at scope.
- *Gate*. The Gate knows the rooms it connects. When an object sends *cross()* the door finds the Requestor's location, decides whether the object can pass or not and then moves the object to the other room.
- *Area*. Implements Area. Gates are visible in both areas they connect.

Consequences:

1. Using Gate enforces a navigation path. Objects can enter a room only if they can get to a door that connects to it.
2. Only objects implementing Locomotion can use doors.

Implementation:

1. *Lock and key*. Gates can be implemented with locks and keys. Only an object with a corresponding key can cross the Gate. Locks can be implemented in many ways: simulating real locks where you need a key; checking identity, etc. Semaphore door is another flavour of Lock and key doors. It only allows objects up to a maximum number to go through it, then it closes until someone exits. There are also doors where you need to pay to go through.
2. *One-way doors*. Gates can be implemented so that they can be crossed in only one direction. Whether the gate is 'visible' from both sides is up to the developer.
3. *Entrance, exit*. Gates are usually called entrances or exits, but these terms have different semantics. Suppose you are in a room and you are about to cross a Gate that leads to a corridor. You would say that you are about to exit the room (instead of entering the corridor). From inside the room that door is an exit but if you look the door from the corridor it is an entrance to the room. Entrances generally wind your path toward the inside of the spatial structure you are navigating, away from the place where you come from. In some cases, giving entrance/exit semantics to gates helps navigation.

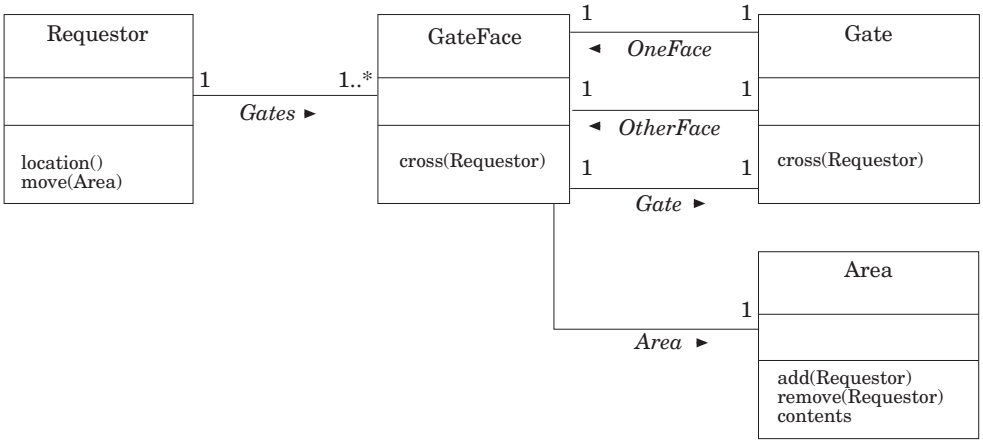

Figure 4. The design of Gates with Faces.

4. *Faces*. Gates must be visible in both rooms that they connect. Most virtual world models define perception for a given subject in terms of the objects located in the same room or Area. According to that, a Gate must be contained in both rooms it connects, which actually goes against the restrictions of our model. In that case the faces of the Gate can be modeled as objects. The overall design is depicted in the Figure 4.

Known uses:
Door.

Related patterns:
Area. Gates connect areas. There must be a mechanism to make the Gates visible in Areas.
Locomotion. Gates rely on the requestor implementing Locomotion.

Name: Locomotion

Also known as:
Movable object, Mobility

Intent:
Provide a safe mechanism for moving objects around the virtual world. Locomotion defines an abstract protocol for movable objects.

Motivation:
The beauty of virtual worlds relies heavily on the ability to move objects around. The simplest example is an avatar navigating a virtual museum. Navigation is the activity that avatars perform when they move from one room in the museum

to another. This activity involves moving at least one object: the avatar itself. It can also be the case of an expensive or scarce tool that needs to be shared and that must be moved to the place where it is needed. These examples and many others involve changing the location of an object or group of objects.

Moving is an operation on objects that takes a room (or container object) as an argument, removes the subject object from its current location, changes its location to be the new room, and adds it to the contents of the new room. These three steps should be performed atomically to ensure the object is not located in two places at the same time, and that it is always somewhere. As regards movements and location, virtual objects in this model behave like real objects: they cannot be in two places at the same time.

Solution:

```
Interface Movable
    location(): Area;
    Boolean move(DestinationArea): Boolean;
```

Objects implementing Locomotion must implement the movable interface. They must have a location operation that returns the 'area' where the object is located. They must also implement *move()* that takes an 'destination area' as an argument. This operation moves the object from its current position to that indicated by the argument. It returns a Boolean to indicate whether the move operation was successful or not.

This solution is particularly simple and rare because it proposes an interface as solution. Notice that a solution proposing a particular structure where a *Mobile Object* class is used to describe mobile virtual object is very restricted because it does not allow one to classify virtual objects by another feature different from mobility.

Consequences:

1. Locomotion provides a safe mechanism for moving objects around the virtual world.

Implementation:

1. *Query destination for acceptance*. The move operation can query the destination (and possibly the origin) room to see if it accepts the object. This is the default implementation in LambdaMOO [6].
2. *Lock objects*. In some cases, objects need to be locked to their position so no one can move them. The move operation can query the object for acceptance. Acceptance can also depend on factors that are external to the objects such as the identity of the requestor.

3. Some objects remember their home location. When they are away for a long time they return automatically. This functionality is useful, for example for managing books in a virtual library. *Note*: the underlying VR environment usually implements Locomotion facilities.

Known uses:
Characters or Avatars, Cars, Animals

Related patterns:
Area. Areas define the mechanism for building objects capable of holding other objects. Locomotion makes it possible to move objects between areas, ensuring that objects inhabit one place at a time.

Gate. Gates rely on their requestors to implement Locomotion.

Transport. Transports rely on Locomotion to accomplish their task. The Transport itself can implement Locomotion.

Name: Transport

Intent:
Encapsulate an algorithm for moving one or many objects. Transports hide details of the underlying spatial distribution, thus simplifying navigation.

Motivation:
Moving objects by using Locomotion facilities is simple. However, it may be undesirable under certain circumstances, for example, when it is important to restrict navigation. Some virtual worlds rely on Gates to enforce access restrictions and spatial distribution. However, when the final destination is located many doors away—maybe on the other end of a complex path—navigating door by door is awkward. The solution is to build an object that knows the algorithm for getting from origin to destination, and make this object transport all the others. It will act as a bus, transporting all the other objects to their final destination.

Solution:
The participants of the Transport pattern are Driver, Transport and Passenger, as is shown in Figure 5.

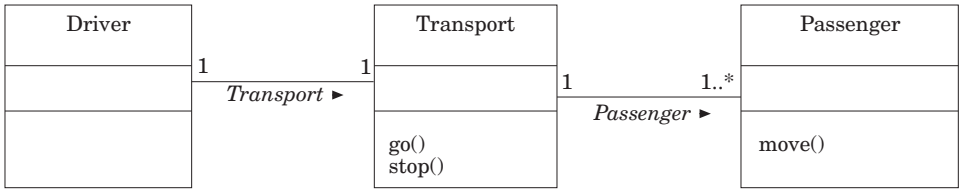

Figure 5. An UML specification of Transport pattern. *Driver*, *Transport* and *Passenger* classes represent its participants.

- *Driver*. Requests services of the transport. It is possibly in charge of configuring and loading the transport.
- *Transport*. Knows how to get to the destination, moving all its passengers as it goes. It knows all its passengers.
- *Passenger*. Implements 'Locomotion' operations.

Consequences:

1. Transports hide the details of the underlying spatial distribution, thus simplifying navigation.
2. Transports let the navigation algorithm and its underlying structure vary without affecting the 'passengers' or the driver.
3. Passengers must implement Locomotion.

Implementation:

1. *Drivers*. The driver is usually a third object that requests the transport to do its work. However it is possible for the transport to be self-driven (e.g. escalators) or to be driven by its passengers (e.g. lift, car).
2. *Fixed route*. The most common example of a transport object has one origin and one destination. It moves other objects back and forth between those places. More complex transport paths can be implemented by defining terminal stations and intermediate stops. If the goal is to hide the navigation path, then fixed route with predefined stops is encouraged. Passengers will depend on the availability of stops and not on the path followed to get to those stops.
3. *Loose route*. Transport objects do not need to have a fixed route. In fact, there are cases where the passengers choose the navigation path, for example cars. Using loose route gives flexibility on destinations and navigation paths.
4. *Containment and location*. There are two final aspects in implementing a Transport—whether the transport contains its passengers, and the transport's location. They are both related. In cases where the transport must travel with its passengers it is better to make it contain the passengers. That is the case of bus, lift and car. When the Transport must stay in a single place it is better not to make the passenger to be the content. That is the case of the teleportation beams.

Known uses:

Train, Bus, Lift, Teleportation beams.

Related patterns:

Areas. Transport moves objects between Areas. Objects implementing Transport may also be Areas or/and located inside Areas.

Locomotion. Passengers implement Locomotion. Transport may also implement Locomotion.

Name: Collector

Also known as:
Propagator

Intent:
To propagate property changes to a group of surrogate objects. A collector object is used to tie an aspect of objects in a group to the same aspect of another distinguished object.

Motivation:
There are cases where changes in aspects of an object must be propagated to a group of surrogate objects. As an example, consider a bag. Location changes are propagated to all the objects inside the bag. We say that a bag is a *collector object*. It is interested in location therefore we call it a *location collector*.

A collector is defined by a group of surrogate objects, a property or aspect, and a mechanism to propagate property changes to its surrogates. The group members can be added and removed dynamically. The property has to be meaningful for the collector object and every surrogate object. A collector is built based on one or more aspects. Collector and collectables can have other aspects independent of the collector behaviour.

Solution:
A diagram representing the Collector structure is shown in Figure 6. Its participants are:

- *Client*. Makes requests to the collector.
- *Collector*. Knows all its collectables and the aspects they are interested in. Propagates changes.

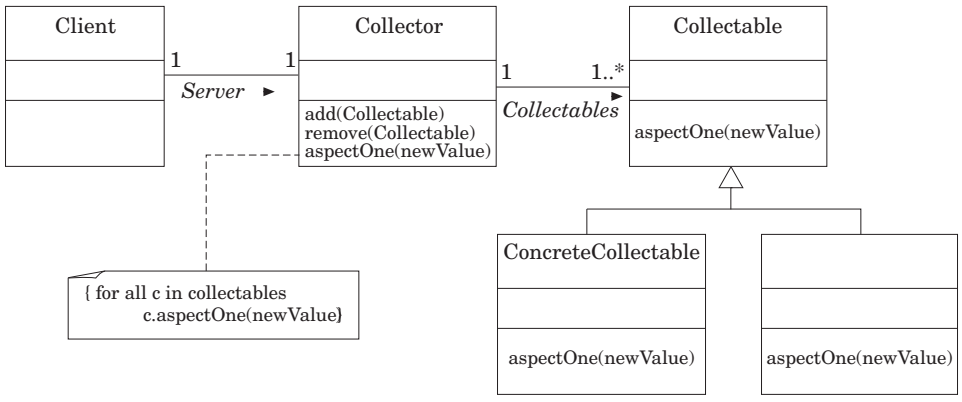

Figure 6. Collector pattern and its participants: Client, Collector and Collectable.

- *Collectable*. Abstract class (or interface). The aspect is meaningful for it. It implements an operation to update the aspect.

Consequences:

1. Collectables can be added and removed dynamically.
2. Collectables can evolve independently.
3. Client does not care about the collectables.

Implementation:

1. *Propagation mechanism*. The propagation mechanism can be different for each surrogate.

Known uses:

A shopping basket is a container object collecting products chosen by the user in a shopping center. Location is broadcasted from the shopping basket object to its collected products.

A sale object collects all objects that are on sale. All objects on sale will have their price reduced according the discount percentage in the sale object.

Related patterns:

Transport. Trivial transports, with no other purpose than propagating location changes can be seen as location collectors. However, it is important to notice that the real power of transport comes from encapsulating complex navigation behaviour.

5. Conclusions and future work

Design patterns appear to be an outstanding tool for managing the complexity of design. They favor reuse, design communication and correctness. Patterns may help developers of virtual environments from many areas to share their experiences. Our catalogue presents patterns to address some problematic aspects of virtual environment design: the space of the virtual world (Area and Gate patterns), mobility (Locomotion and Transport) and change propagation (Collector pattern).

The basis for a pattern language presented in this paper is far from complete. There are aspects of the design of virtual environments that must still be explored, for example active objects, such as agents. Also unexplored are other behavioural issues like subjective behaviour [19] (an object behaves according to the context), decentralized systems as those presented in [20] and virtual world mutability [21] (the world changes over time). Design patterns for communication awareness in cooperative systems have to be further studied. Existing models, such as the spatial models presented in [22,23] and extended in a third-party-object model [24] are a first approach to design patterns for awareness. Efforts are now directed towards assessing the results of applying this pattern catalogue to

non-textual virtual environments. Existing virtual environments will be explored for new patterns. Studies will also consider instances of object-oriented design patterns [7]. As regards the architecture of virtual environments, special attention will be put on architectural patterns such as those presented by Alexander [14].

References

1. C. Greenhalgh 1999. *Large Scale Collaborative Virtual Environment*. Distinguished Dissertation, Springer Verlag.
2. S. Harrison & P. Dourish 1996. Re-Place-ing space: The roles of places and spaces in collaborative systems. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSWC'96)*, Vol. I. *Boston MA*: ACM Press, 67–76.
3. P. Dourish & M. Chalmers 1994. Running out of space: Models of information navigation. In *Proceedings of the ACM Conference on Human Computer Interaction, HCI'94, Glasgow, Scotland* November 16–20. ACM Press, Vol. I.
4. S. Benford, J. Bowers, L. Fahlen, C. Greenhalgh & D. Snowdon 1995. User embodiment in collaborative virtual environments. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI'95), May 7–11, Denver, Colorado, USA*. ACM Press, Vol. I.
5. C. Haynes & J. Rune Holmevik (eds.) 1998. *High Wired: On the Design, Use, and Theory of Educational MOOs*. Michigan: The University of Michigan Press.
6. LambdaMOO: One of the oldest, most diverse, and largest MOOs in existence. At Telnet: *telnet://lambda.moo.mud.org:8888*
7. B. Damer 1998. Avatars! Exploring and Building Virtual Worlds on the Internet. Peachpit Press.
8. E. Gamma, R. Helm, R. Johnson & J. Vlissides 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison Wesley Publishing Company.
9. L. Guerrero & D. Fuller 1999. Design patterns for collaborative systems. In *Proceedings of Fith International Workshop on GroupWare CRIWG'99*. IEEE CS Press.
10. R. Wirfs-Brock, B. Wilkerson & L. Wiener 1990. *Designing Object-Oriented Software*. Prentice Hall.
11. A. Riel 1996. *Object-Oriented Design Heuristics*. Reading, MA: Addison Wesley.
12. K. Beck 1996. *Smalltalk Best Practice Patterns*. Volume 1: Coding. Prentice Hall.
13. D. Schmidt 1995. Using design patterns to develop reusable object-oriented communication software. *Communication of the ACM* **38**(10), 65–74.
14. C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson & I. Fiksdahl-King 1977. *A Pattern Language: Towns, Buildings, Construction*. Oxford: Oxford University Press.
15. G. Meszaros & J. Doble 1997. A pattern language for pattern writing. In *Pattern Languages of Program Design 3*. (R. Martin, D. Riehle & F. Buschmann, eds). Reading, MA: Addison-Wesley.
16. J. Coplien 1995. Development process generative pattern language. In *Pattern Languages of Program Design*. (J. O. Coplien & D. C. Schmidt, eds). Reading, MA: Addison-Wesley, 183–237.
17. G. Booch, I. Jacobson & J. Rumbaugh 1998. *The Unified Modeling Language User Guide*. The Addison-Wesley Object Technology Series. Reading, MA: Addison-Wesley.
18. I. Tomek, A. Diaz, A. Rito da Silva, R. Melster, M. Haahr, A. Fernandez, Z. Choukair, M. Antunes, A. Ohnishi & W. Wieczerzycki 1999. *Multi-User Object-Oriented Environments. Object-Oriented Technology*. ECOOP'99 Workshop Reader, Lecture Notes in Science, Springer Verlag, Vol. 1743. 80–96.
19. M. Prieto & P. Victory 1997. Subjective object behavior. *Object Expert* **2**(3), March/April.
20. M. Resnick 1994. Turtles, termites, and traffic jams. In *Explorations in Massively Parallel Microworlds*. MIT Press.
21. J. C. Heudin 1999. Virtual worlds. In *Synthetic Universes, Digital Life and Complexity*. New England Complex Systems Institute, Series on Complexity. Perseus Books.

22. S. Benford & L. Fahlen 1993. A spatial model of interaction in virtual environments. In *Proceedings of the Third European Conference on Computer Supported Cooperative Work (ECSCW'93), Milano, Italy*.
23. T. Rodden 1996. Populating the application: A model of awareness for cooperative applications. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSWC'96), November 16–20, Boston, MA*: ACM Press.
24. S. Benford, C. Greenhalgh & D. Lloyd 1997. Crowded collaborative virtual environments. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI'97), Atlanta*, GA, 59–66.

Alicia Díaz is Assistant Professor at the Facultad de Informática and has held a research position at the Lifia Research Laboratory at the Universidad Nacional de La Plata since 1987. Her work areas include: Virtual Environments, design models, process and reuse design; hypermedia applications and hypermedia design based on models; computer-aided support for hypermedia design and development (in particular, she has designed a CASE tool called RMCase, based on RMM Methodology to design applications. RMCase generates WWW applications), and using query language as a way to retrieve hypermedia information and to design hypermedia applications. She has presented research project results at many computer science conferences.

Alejandro Fernández has held a research position at the Concert Department of the GMD-IPSI in Germany since February 2000. He has also been a member of Research Laboratory at the Universidad Nacional de La Plata in Argentina since 1993. He has been involved in teaching object technology in industry and academia since 1994. He has published in international conferences in areas related to object technology, design and learning. He participated in several international cooperation projects during the years 1998 and 1999. His research interest include object technology, software design, groupware and learning.