AN ARCHITECTURE FOR THE IMPLEMENTATION AND DISTRIBUTION OF MULTIUSER VIRTUAL ENVIRONMENTS

A Thesis
Submitted to the Graduate Faculty
of the
North Dakota State University
of Agriculture and Applied Science

By

Benjamin James Dischinger

In Partial Fulfillment of the Requirements
for the Degree of
MASTER OF SCIENCE

Major Department:
Computer Science

May 2010

Fargo, North Dakota

North Dakota State University
Graduate School

Title

AN ARCHITECTURE FOR THE IMPLEMENTATION AND

DISTRIBUTION OF MULTIUSER VIRTUAL ENVIRONMENTS

By

BENJAMIN JAMES DISCHINGER

The Supervisory Committee certifies that this ***disquisition*** complies with North Dakota State University's regulations and meets the accepted standards for the degree of

MASTER OF SCIENCE

ABSTRACT

Dischinger, Benjamin James, M.S., Department of Computer Science, College of Science and Mathematics, North Dakota State University, May 2010. An Architecture for the Implementation and Distribution of Multiuser Virtual Environments. Major Professor: Dr. Brian M. Slator.

JavaMOO is an architecture for creating multiuser virtual environments focusing on domain-specific design and rapid development. JavaMOO components use best practices and extensible design for system configuration, client-server communication, event handling, object persistence, content delivery, and agent control. Application dependencies such as database and web servers are embedded, promoting wide dissemination by decreasing management overhead. The focus of this thesis is the design and implementation of the JavaMOO architecture and how it helps improve the state of multiuser virtual environments.

The formatting of the thesis has been done in accordance with the Guidelines for the Preparation of Disquisitions.

ACKNOWLEDGEMENTS

Thank you to everyone who worked with me on this project including Bradley Vender, Robert Cosmano, Christopher Imdieke, John Opgrande, Kellie Erickson, and Jacob Halvorson.

Special thanks is deserved to Dr. Brian Slator and Dr. Harold Chaput; without their help, I may not have found the motivation required to complete this endeavor.

I am very grateful to my wife, Cheri, for her constant support, feedback, and patience during life's balancing act.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

INTRODUCTION

JavaMOO is an architecture for creating multiuser virtual environments that helps developers focus on domain-specific design and rapid development. JavaMOO components use best practices and extensible design for system configuration, client-server communication, event handling, object persistence, content delivery, and agent control. Application dependencies such as database and web servers are embedded, promoting wide dissemination by decreasing management overhead.

This thesis describes the design and implementation of the JavaMOO architecture and how it helps improve the state of multiuser virtual environments.

Virtual Environments

At North Dakota State University (NDSU), the World Wide Web Instructional Committee (WWWIC) is engaged in research aimed at developing virtual environments to assist in the education and growth of students (Slator, 1999).

Some of the key factors that lead to the success of these environments are: the theory of role-based environments on which they are based (Brandt, 2006), the use of graduate and undergraduate students in the development process (Borchert, 2001), the use of the environments in actual classes (Mack, 2003; Slator, 2001), and the application of knowledge from a virtual environment to the real world (Slator, 1996).

The virtual environments developed at NDSU cover a variety of disciplines. Two of the most mature and successful environments teach topics in cellular biology and geology. Table 1 lists the environments that have been or are currently being developed by WWWIC.

Environment	Description
Geology Explorer	Geologic exploration of the fictitious planet Oit.
Virtual Cell	3D environments teaching cellular biology concepts
Blackwood	19th century town exploring American history and microeconomics
On-a-Slant Village	Simulation of a real-world archeology excavation
Dollarbay	Virtual shop-keeping teaching basic marketing and microeconomics

Table 1: NDSU Virtual Environments

We examine the *On-a-Slant Village* simulation to illustrate the typical educational goals of these environments.

On-a-Slant models a Native American village in North Dakota, an important historic site that was occupied by the Mandan along the Missouri Valley until the late 18th century. Archeology students learn about the village by studying and interpreting the excavation documentation and maps, but it is difficult to visit the site to see it for themselves.

This project presents a unique opportunity to experience the village "to the inch" as a three-dimensional environment. Students can login to the system and be immersed in a realistic environment that not be easily visited in real life. Students take on the role of an archaeologist in order to tackle problems scientifically. The goal is for the students to begin thinking as an actual archaeologist would, and to "learn by doing". It is emphasized that "this is NOT a museum piece where people come to wander around and passively look at things. Visitors will be engaged in geology, botany, and archeology of the excavation site." (NDSU Archeology Technologies Laboratory, 2008)

As the On-a-Slant example shows, NDSU educational environments intend to place students in situations where authentic learning can take place which would otherwise not be possible because of physical, economic, or other limiting factors.

Implementation Challenges

Each virtual environment has unique characteristics, but is implemented using the same fundamental architecture with servers written using LambdaMOO and clients written using Java. This hybrid architecture has created many implementation challenges in current virtual environments.

LambdaMOO is a network accessible multi-user programmable system that can be used to construct text-based virtual realities. Users interact with LambdaMOO by typing in a command-line based client such as telnet. The user provides commands in an intuitive syntax such as "look in box". No alternative communication mechanism exists other than the text-based system that is intended for direct human interaction (Curtis, 1997).

Graphics based environments have intrinsic challenges to overcome when being implemented using the text-based LambdaMOO. For graphical clients, where the user interacts with the virtual environment using a mouse, a translation from the Java representation of user actions to the LambdaMOO text representation must be performed. As with any communication protocol these actions must be translated into a format which both the client and server understand.

For large data structures this text based representation is inefficient. The maximum length of a message for LambdaMOO is 256 characters, so large messages must be broken into smaller segments. In complex interactions, such as animation, the number of messages may increase dramatically. This further exacerbates scalability issues and the complexity of the communication.

Any non-text content must be handled externally or implemented along side of LambdaMOO. No delivery system exists natively within LambdaMOO to send graphics or other content to clients. An external solution creates increased system complexity, additional points of failure, and maintenance cost. A solution implemented within LambdaMOO must either extend the core system, which is complex and no longer regularly maintained, or be written within the MOO language itself and subject to the same communication challenges discussed earlier.

Because of the need for a content delivery system, external dependencies are created, making the distribution of these educational simulations complex. A web server may be needed to deliver images, or an external database could be used to store learning assessment results. These types of dependencies require a high degree of knowledge to install correctly.

LambdaMOO also features a fail-safe mechanism for canceling runaway processes that depends on monitoring processes for their 'tick count' and killing those that exceed a quota. Of course, in a continuous simulation, some routines are intended to run perpetually, and this is orchestrated with a 'suspend' mechanism, which yields execution to another process in exchange for resetting the process's execution limits.

It is necessary for some LambdaMOO routines processing large datasets to pause even though they are not intended to be running forever. Because LambdaMOO was designed to be an open environment these execution limits are essential, but the overhead involved in monitoring is considerable and creates obvious inefficiencies in production systems.

The LambdaMOO execution environment is purely interpreted. Unlike Java environments, LambdaMOO is a basic byte-code interpreter without any support for just-in-time compilation or similar improvements. All execution is performed serially in the system to avoid concurrency issues. This further amplifies performance problems.

Lastly, objects in LambdaMOO are memory resident, meaning that all objects must fit into available memory. The persistence strategy is to checkpoint all objects to disk periodically. This approach is inefficient if the number of objects is large, and if the cumulative size of all objects is greater then available memory, this approach fails. There is also danger in losing critical changes which were made to the system since the last checkpoint. Therefore, for a sufficiently large simulation, this persistence approach is not feasible.

Goals and Constraints

The goal of this thesis is to create a client-server architecture that addresses the above challenges while simplifying the design and implementation of virtual environments, and improving their scalability and performance. This architecture is called JavaMOO.

We have the following goals:

1. Existing virtual environments can be implemented using JavaMOO.
2. Create a self-sufficient server with minimal external dependencies.
3. Develop a client-server communication model that is customizable.
4. Design incremental object persistence to avoid memory residency of objects.
5. Ensure overall acceptable performance of JavaMOO.
6. Identify JavaMOO best practices for future developers.

LITERATURE REVIEW

There is a long lineage of programming architectures appropriate for developing multiuser environments. Like most useful technologies, these tools have been reused in many new and unforeseen ways. We examine relevant technologies from the past and present that have influenced the design of JavaMOO. The goal in this section is to provide relevant background and describe the technological gap that JavaMOO fills.

LambdaMOO

The original multi-user dungeon (MUD) was called just that; MUD was the title of the game, not a game genre. The first version was developed by Roy Trubshaw and Richard Bartle at Essex University on a DECsystem-10 mainframe in the spring of 1979. It supported multiple players, and employed the text-based adventure style that we still associate with MUDs today (Bartle, Nov. 1990).

By 1990 there were many different MUD systems: AberMUD, LPMUD, TinyMUD, TinyMUCK, UberMUD, among others. These systems were differentiated by the level of user programmability, the programming language used, and what features were available. They all are descendants of the original MUD (Bartle, Dec. 1990). A new MUD called MOO (Mud-Object-Oriented) was created by Stephen White and released May 1990. MOO was developed with the same style and conventions as TinyMUD, but with an underlying object-oriented programming model and database (Wikipedia, 2008).

Pavel Curtis, a programming language researcher at Xerox PARC, began playing MOO in September 1990, fixing bugs and adding a number of features to the language. In October he forked a version of MOO, called LambdaMOO, after his nickname lambda since it was "Lambda's MOO" (Curtis, 1998).

Although research on LambdaMOO began from a technical standpoint, it quickly became clear that the social aspects of mudding were worth research on their own. Curtis makes observations of social phenomena which occurred during the early life of LambdaMOO. This emergence of a collaborative community, and also the handling of trouble makers, is important to consider when developing environments for education (Curtis, 1992).

LambdaMOO's strongest benefit is the ability of users to modify programs while the server is executing, allowing for rapid prototyping and real-time creation of virtual environments. This has been a major factor in the successful creation of WWWIC environments because of the immediate feedback that is available when modifying the system. In addition, LambdaMOO has shown itself to be remarkably stable over the years, with system failures resulting from machine crashes more often than software failure.

To aid in rapid prototyping, an intuitive verb based programming model exists where a user is able to type commands such as "put ball in box". LambdaMOO parses this command and calls the *put* verb on the *ball* object with the arguments "in, box", where box is a container object. This command structure allows for a natural text-based interaction and typically makes training new programmers for LambdaMOO easier than most modern programming languages (Curtis, 1997).

Commands are executed serially in the system to avoid concurrency or locking issues. This isolates programmers from complex issues that arise in multi-threaded programming at the cost of performance. As a result, LambdaMOO programs do not run quick enough to handle real time events such as mouse-motion or real-time network processing (Curtis, 1998).

MMOG Architectures

The need to build scalable persistent multi-user virtual environments makes massively-multiplayer-online-games (MMOGs) relevant to this thesis, including the background of MMOGs and the available literature surrounding their architecture and implementation.

MMOGs are descendants of MUDs such as LambaMOO. Like MUDs, MMOGs create persistent worlds in which thousands of players can interact simultaneously. The main difference is how the state of the world is created and communicated to the user; instead of users creating world content through textual descriptions and programming, MMOGs display a graphically rich three-dimensional world geography. It is uncommon for MMOGs to permit user created content, whereas for MUDs it is essential to their existence (Mortensen, 2006).

MMOGs have grown rapidly in popularity since the first, *Meridian 59*, was launched in 1996 (Kent, 2003). By the year 2000, MMOGs as a whole had over one million monthly subscriptions and by April 2008 over sixteen million, with more than ninety percent in the fantasy role-playing genre. Blizzard's World of Warcraft, the most successful MMOG to date, currently has by far the largest market share with over ten million subscribers (Woodcock, 2008).

Because of this popularity, there is a wide interest in using MMOGs for such purposes as military training (Bonk, 2005), education (Big World, 2008), or distributed business meetings (Sun Press, 2006). Bonk and Dennon find an absence of research surrounding

MMOGs and suggest additional research to help determine the cognitive and cultural impact of these games on adults. They propose fifteen research experiments to help determine whether MMOGs could be useful for military training and education in general. These experiments could answer questions about the impact that MMOGs have on decision making skills, community building, and leadership skills (Bonk, 2005).

Companies such as BigWorld have developed commercial middleware with the goal of reducing the average time to market for new MMOGs. Middleware implements the low-level functionality and difficult distributed system concepts so that development effort can be focused on game specific development. The middleware includes a server framework, server monitoring tools, client libraries, and content creation tools. A company wishing to create a new MMOG can license BigWorld tools for a fee made available under a non-disclosure agreement (Big World, 2008).

The central problem in creating an MMOG is one of scale: how can high quality service be provided to thousands, or perhaps millions, of simultaneous players? In the following sections we will discuss useful or novel MMOG architecture techniques that help address this problem: *service distribution, sharding*, *region-based clustering*, and *peer-to-peer distribution.*

Details of the system architectures for commercial MMOGs such as World of Warcraft are not generally available to the public. We can only speculate on what the architecture is for many games, however reference material exists for open source projects such as Second Life and many papers have been written researching MMOG architectures.

Service Distribution

A successful MMOG must provide many services such as authentication, region simulation, content management, game state management, and inter-player communication. OpenSim, an open source implementation of the SecondLife server, refers to these services as UGAIM or User, Grid, Asset, Inventory, and Messaging. An example distribution of these services are shown in Figure 1. Each service fulfills a specific purpose and can be implemented as an independent process on separate networked machines (Open Simulator, 2009).

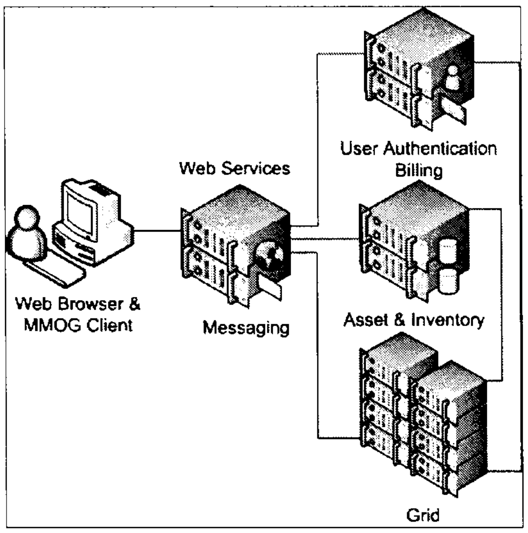


Figure 1: UGAIM Service Distribution

The scalability of the system can be improved by simply dividing services into separate processes and distributing the workload across multiple machines. Splitting these services into separate cooperative networked processes allows them to scale to increasing numbers of users by adding new processes and machines to the system. Coordination is needed to route events between services. For example, events can be distributed to the appropriate grid server based on a player's location within the game world. The fact that dozens or hundreds of servers are processing user actions is transparent to the user connected to the MMOG, hopefully resulting in a seamless experience.

Sharding

Sharding distributes users across many identical copies of a world on geographically separated clusters. The player chooses, or is automatically directed to, the copy to connect with. In this way a large population of users, greater than what can be handled by service distribution alone, can be spread across many servers without expensive coordination because each copy is independent. An example of sharding is shown in Figure 2. If existing shards are nearing capacity, new shards can be added without affecting existing users.

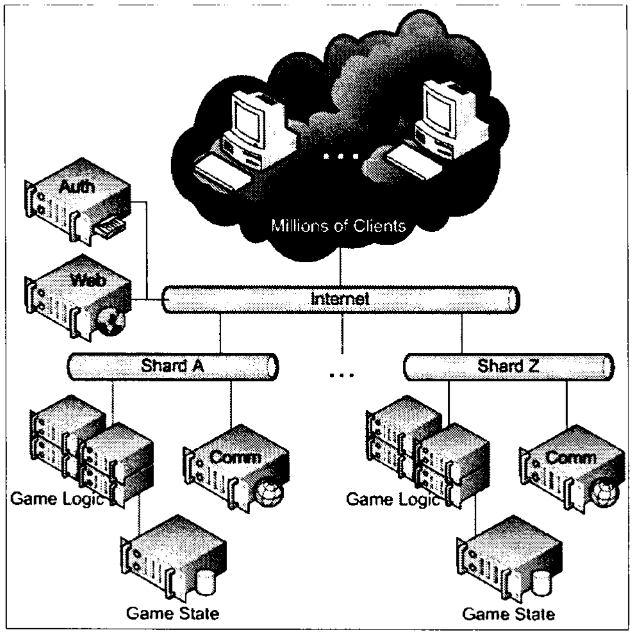


Figure 2: MMOG Sharding Example

World of Warcraft currently has 241 realms, or shards, named from *Aegwynn* to *Zuluhed* (Blizzard, 2008). Each realm has an associated game play style: Normal, PvP (player vs. player), RP (role playing), or RPPvP. Since players are allowed to select which realm they wish their character to belong, the distribution among realms is not evenly spread. Assuming ten million subscribers, there is an average of around 41,000 subscribers per realm. Thus a single realm must support thousands of simultaneous connections but not millions, helping to reduce the scale and complexity of the system.

This intuitive approach helps reduce the number of players in each copy of the world substantially, but at a cost: since each shard is independent, players who wish to collaborate must ensure that they are on the same shard. Certain shards may become more popular due to a network effect as more players join where their friends play, resulting in a 'hot shard'. When the shard reaches quality of service thresholds future players will be forced to choose a different shard, which may be perceived negatively by the player.

Region-based Clustering

The large geographies present in MMOGs are no accident: a large world increases the potential distance between players, decreasing interaction and increasing scalability. Much like how mountain ranges or oceans separated ancient civilizations and limited how they could interact, distance within an MMOG can limit players interactions. This allows the geography to play a direct role in the processing of events, especially in relation to each player's sphere of interest and influence (Lu, 2006).

Geography can be distributed across a cluster of servers using a variety of strategies, but most often each node is responsible for processing events that occur within a specific assigned geographic region. Either regions are statically determined as part of the world design, or can be dynamically allocated depending on the current system load. In the distributed MMOG architecture developed by Assiotis, a dynamic approach splits congestion hotspots where too many users have congregated. The region is divided in two, and objects in the new region are transferred to the new server over an extended period of time. Once the transfer is complete the new server takes control of the split-off region (Assiotis, 2006).

Figure 3 shows an example network topology of a cluster of machines distributed by game geography. In this example the game might be an American Revolutionary War simulation in which many users participate. The servers are each assigned responsibility for a unique region.

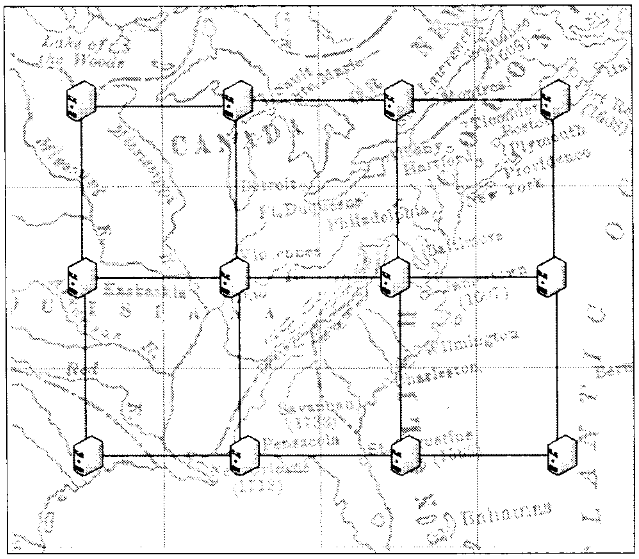


Figure 3: Region Based Grid Distribution

The amount of communication between regions is important for the transparency of the distribution. For example, Second Life assigns each 256x256 meter region to a different process running on its own processor core. Each region has a maximum number of simultaneous users, but users seamlessly travel between regions because of region process communication. To contrast, regions in the MMOG *EverQuest* are totally isolated from each other so that no region communication is needed. In this approach, region event management is simplified, but a noticeable connection delay occurs when traveling between regions.

Peer-to-Peer Distribution

It is possible to create an MMOG in which no servers exist and the world state is distributed across clients. The previously discussed MMOG architectures are based on a centralized server architecture where clients connect to a server farm with potentially hundreds of dedicated machines. It is the responsibility of the organization managing this system to ensure the quality of service is met for the number of clients connected.

It is very expensive to create and manage an MMOG server farm. Enterprise servers and storage with built-in redundancy features can cost tens of thousands of dollars per unit, easily adding up to millions of dollars. Development and administration labor costs can run millions of dollars per year.

Peer-to-peer distribution has potential for reducing deployment costs and increasing scalability, allowing huge community hosted MMOGs to be created. Overlay networks can dynamically grow to thousands of participating nodes in a single system, but applying this to MMOG games has serious problems to overcome such as consistency, security, and reliability. No one has yet created a successful commercial peer-to-peer MMOG.

Distributed hash tables (DHT) have become a key technology in peer-to-peer systems by solving how to distribute a large amount of data across a large number of systems and lookup that data quickly based on a key value (Wiley, 2007). The idea is simple: take a normal hash table, partition the key space of that table and assign responsibility of partitions to nodes in a peer-to-peer network. The assignment of key-space should be done in such a way to allow quick traversal across the overlay network to retrieve the data. Many different approaches exist to implement DHT and the original implementations from 2001 are still in use today; these include CAN, Tapestry, Pastry and Chord (Ratnasamy, 2001; Zhao, 2002; Rowstron, 2001; Brunskill, 2001).

Research has attempted to utilize DHT in order to implement peer-to-peer MMOGs. SimMud, MOPAR, and a prototype developed by Hampel, use Pastry to build a MMOG on top of a peer-to-peer overlay network (Knutsson, 2004; Yu, 2005; Hampel, 2006). SimMud uses a DHT to replicate game state and interest management across nodes to scale to approximately 4000 users. The authors note the possibility of a catastrophic data failure if multiple nodes containing the main and replicated data leave the network simultaneously. With the test workload and a single replicated copy, a failure occurred on average once every 20 hours (Knutsson, 2004).

Non-DHT implementations of peer-to-peer MMOGs have also been developed. Hydra is a novel architecture that presents to the game developer standard client-server interfaces, but translates these requests to an underlying peer-to-peer network (Chan, 2007). All peers participating in the game also act as servers. In order to scale the system, it is up to the game developer to implement their world into separate explicit regions that can be replicated for fault tolerance.

Solipsis is another non-DHT implementation of a peer-to-peer MMOG that is very similar in concept to Second Life. Solipsis attempts to implement the concept of a metaverse, where users can travel to an unlimited number of user-generated worlds through a single navigator application. It uses the Raynet overlay network which subdivides the game geometry into an approximation of a Voronoi tessellation (Beaumont, 2007). This assigns responsibility of game state and physics calculations to each individual client and optimizes event routing (Frey, 2008).

APPROACH

JavaMOO has evolved in concept and design over time. We describe the work chronologically as we remember, culminating in the current design which is the main focus of this thesis.

To give background and highlight differences from LambdaMOO, we first describe our beginning attempts at creating the JavaMOO framework. We then describe a prototype of an existing environment called Dollarbay which was created as a proof of concept to show the benefits of a pure Java implementation. From this prototype the requirements for JavaMOO were determined. Finally, we describe the design of the JavaMOO architecture in detail and show how new environments can be implemented using the framework.

First Attempt

We first envisioned a translation approach in designing JavaMOO. We wanted to take environments that were programmed in LambdaMOO and convert them to Java automatically with clever programming or using some form of semi-automatic human assistance.

It did not take many design meetings to see that a translation approach would be impractical. Although LambdaMOO and Java are both object oriented, the language semantics are incompatible. LambdaMOO uses *prototype inheritance*, meaning that the class and instance of an object are identical (Taivalsaari, 1996).

To illustrate the differences between prototype and class inheritance, let's consider a GenericBall object in LambdaMOO. The ball has properties such as color, weight, or bounciness and has verbs like bounce, throw, or hit. Even though the GenericBall is an abstraction from which other balls will be created, it can still be a physical object in the environment; it can exist in a room, be picked up, and looked at.

To create a new ball in LambdaMOO we call the *create* verb on the GenericBall to make a BouncyBall. The new object is said to be a child of GenericBall and inherits its properties and functionality. If we change the parent's color to orange, the BouncyBall's color, and more importantly the color of all children, will also change to orange. This dynamic property sharing is the essence of prototype inheritance.

In LambdaMOO the inheritance tree of an object is a dynamic graph of object instances. In contrast, a Java class defines a structure that cannot be modified during a program's lifetime, including the fields, methods, and inheritance tree of the class. Each Java object is an instance of a single class whose class structure remains static for the lifetime of program execution. Whereas an object's inheritance structure could be changed in LambdaMOO dynamically at runtime, no such mechanism exists in Java.

Prototype inheritance is not natively possible in Java. This hampers a direct LambdaMOO to Java translation because any occurrence of prototype inheritance would need to be implemented programmatically, essentially emulating LambdaMOO language constructs. This would hurt the performance and simplicity of the JavaMOO object model.

In addition to inheritance model incompatibilities, LambdaMOO code relies heavily on the semantics of the MOO language. The language has unique features that are not easily translated into Java code. For example, the notion of object gender is built into LambdaMOO so that gender appropriate pronouns are used in messages to the user. Another example is the use of prepositions in user interaction that resolve to object verbs.

This allows players to specify "put ball into box" or "put box on ball". These natural language facilities do not exist in native Java and would need to be written for a conversion of LambdaMOO.

Lastly, Java programs generated from LambdaMOO would not be well written. LambdaMOO contains a number of utility classes that have an overlap of functionality with existing Java libraries. To automatically convert the MOO code we would need to determine for every function whether the functionality should be ported or existing Java functions used. The resulting code would be difficult to follow and debug, and would not benefit from Java programming best practices.

For these reasons we determined that neither automatic nor semi-automatic conversion of LambdaMOO code to Java would be efficient, accurate, or desirable. Instead we decided to follow an iterative prototype approach.

Dollarbay Prototype

Instead of attempting to automatically convert LambdaMOO implementations into Java, it was decided to implement a Java prototype of an existing environment. This allowed us to determine the best way to implement existing virtual environments entirely in Java, and extract the necessary design for JavaMOO from that prototype. We decided to create the prototype using a relatively stable instance of a LambdaMOO environment: the NDSU Dollarbay educational game.

The Dollarbay educational game teaches students different economic topics such as supply and demand, pricing, market research, advertising, and business strategy (Mack, 2003). The original version, called SELL, was a turn-based simulation developed at

Northwestern University in the mid-nineties. Players set prices on their products and made decisions about their store during each turn. Between turns the simulation would calculate each store's sales based upon all players' decisions and consumer motivations (Slator, 1996).

Dollarbay has had many iterations and improvements since its SELL origins. It is now a real-time simulation with shoppers implemented as autonomous agents. A case-based tutor monitors play and helps players make better decisions (Regan, 2002). Players can interact with other players and with the agents in the game, which helps increase collaboration and engagement. There are now 286 different types of products and 63 shoppers representing 20 different consumer groups. This relatively simple object structure, manageable amount of data, and rich client/server interaction makes Dollarbay a good choice for a pure-Java based prototype and the basis of an iterative prototype project.

As players join the Dollarbay game they are assigned a location and must decide what to sell, what level of service to offer, how much to spend on advertising, how much to stock, who to buy from, and what prices to set in order to attract customer agents.

In order to simulate an economic environment, time is divided into 'virtual weeks'. A virtual week is simulated game time, not to be confused with a real week of actual game play. At the beginning of each virtual week, simulated customer agents are given a shopping list representing a week's worth of demand for various products representing an economic group. These agents spend the virtual week purchasing the items on the list from the stores in Dollarbay. After each virtual week has concluded and the shopping lists are exhausted, each agent assigns new attractiveness ratings to each store based upon the past week's experience (Slator, 2001).

Players are also charged for their weekly expenses, such as rent and advertising, and their case files are updated with a record of the week's activity. Customer agent motivations are recalculated based on this new information, and new shopping lists are created for the upcoming week.

At the end of a player's life, they are retired to the Hall of Fame. The Hall of Fame is a place where players are moved when they graduate from the game either by reaching a profit goal, going bankrupt, or being inactive for a long period of time. Upon a player's retirement from Dollarbay, their store's active case file is archived to the historical cases for future reference by the case-based tutor (Regan, 2002).

The pedagogical goal of Dollarbay is to teach a wide set of skills associated with running a retail business by allowing the student to control a simulated store in a simulated economy. Therefore, the economic simulation must be authentic and complex, not only to effectively teach shop-keeping concepts but also to preserve the player's interest over an extended period (Slator, 2006).

We chose Dollarbay because its design problems had already been solved. Dollarbay represented a fixed target where development of the JavaMOO architecture could be focused. Therefore, the purpose of the JavaMOO translation of Dollarbay was to demonstrate a proof of concept – a LambdaMOO simulation could be transformed into a JavaMOO simulation, with the additional goal that it work better than the original.

JavaMOO Architecture

The JavaMOO architecture has six components that together help developers implement virtual environments. The following functional areas are provided: system configuration, client-server communication, event handling, object persistence, content delivery, and agent control. Developers are largely isolated from the underlying details of these areas, instead being directed to focus on the domain specific design of each virtual environment. In this section, we first describe the architecture at a high-level and afterward examine each component separately with a focus on design decisions, showing the utility of each component.

Overview

Figure 4 shows JavaMOO server component data flow. Clients communicate with the server using event objects, and retrieve content such as images, 3-D models, or dynamic web-pages from the content delivery component. The communication component receives events from the client, manages the client's session, and passes the events to the event handling component.

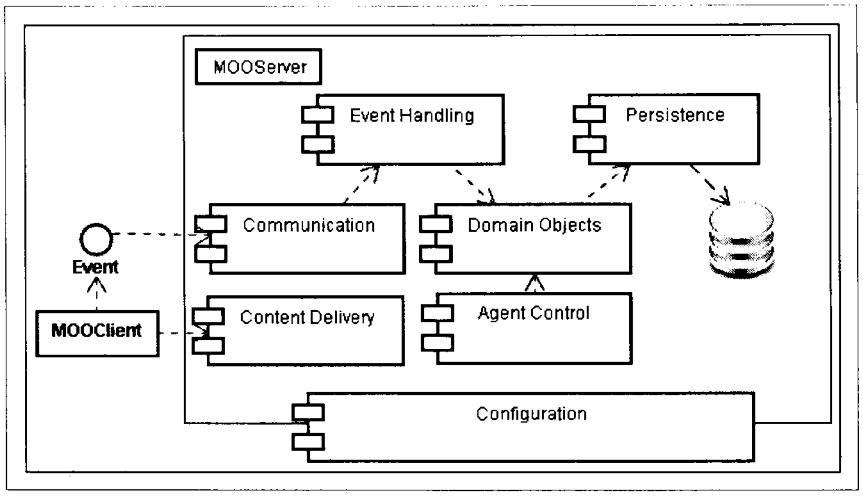


Figure 4: JavaMOO Component Diagram

Depending on the type of events received, event handlers modify domain objects that comprise the state of the virtual environment. Changes to the environment must persist between invocations of the server. The persistence component ensures that the domain objects are saved. The objects are stored in backing storage, such as a relational or object database.

Autonomous agents may also modify the state of the environment. Agents act within environments, either fulfilling a simulation role or providing entertainment value. The agent control component provides a way to organize and manage agents that exist within an environment. This component helps limit resource usage and reduce complex concurrency problems.

The configuration component is responsible for the overall configuration of a virtual environment. This component connects dependent components together, and specifies the implementation that should be used for each component. The JavaMOO framework provides the ability to swap out different implementations for the components, increasing flexibility to meet environment requirements. Administrators of deployed servers can modify the configuration.

Configuration

It became clear after developing the Dollarbay prototype that configuration played an important role in building a successful virtual environment. Without a well-defined configuration model, the system becomes difficult to use and modify.

In the Dollarbay prototype, we created a configuration system based on a key-value pairs in a single flat file called *javamoo.conf.* This contained the type of database, web server port, RMI URLs, logging configuration, and client content url. Configuration values were used by classes by passing a lookup key to the *Config.getValue()* utility method.

This approach worked, but had limitations and drawbacks. Using this type of configuration creates an explicit dependency between the configured class and the Config class. Only text-based values can be used, meaning that more complex data types such as objects or even numbers, have to be parsed from the text. This made it difficult to factor out the dependencies between classes into the configuration. We have addressed these limitations by utilizing the Spring application framework, specifically for the *Inversion of Control* container it provides.

Inversion of Control (IOC). The idea behind IOC is to move dependency creation away from component classes and make it a part of the application configuration. This principle is also known as *Dependency Injection* because of the way class dependencies are set automatically by the IOC container. (Spring Framework, 2010)

For example, consider a hypothetical Car class which depends on an Engine implementation. Without IOC the Car might create a new Engine inside of the constructor. This would be a hard coded explicit dependency between the engine used and the car. If it was decided that an electric engine was needed instead of gasoline, the code would need to be changed and recompiled.

In contrast, IOC keeps the configuration of the Car in an XML file. The configuration file would specify that the Car depended on the electric engine, and the container would create the car, and then *inject* the Engine into the Car. Now if the developer wanted to switch to an electric engine from a gasoline, they change the configuration file and restart the application. This is much more flexible as not only simple values can be specified but entire object trees.

Configuration Example. Figure 5 shows an example configuration of the JavaMOO persistence component. Each object that is configurable in the IOC is called a *bean*. Top-level beans, the children of the root *beans* element, have unique ids. The application can use the ids to lookup beans created by the IOC container.

```
<beans>
  <bean id="persistManager"
class="javamoo.persist.sql.SQLPersistManager">
    <constructor-arg>
      <bean
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName"
                  value="org.apache.derby.jdbc.EmbeddedDriver"/>
        <property name="url" value="jdbc:derby:db;create=true"/>
      </bean>
    </constructor-arg>
    <constructor-arg>
      <bean class="javamoo.persist.sql.DerbySQLDialect"/>
    </constructor-arg>
  </bean>
</beans>
```
Figure 5: Persistence Configuration Example

In this example, a new *persistManager* bean will be created using the class *SQLPersistManager.* This class has two constructor arguments. The first is a *java.sql.DataSource*, an interface to create SQL database connections. We chose a Spring implementation for the DataSource, using an embedded Derby database. The second is a *javamoo.persist.sql.SQLDialect,* a JavaMOO interface containing specific implementation details for different database implementations. Since we are using a Derby database we use the Derby specific SQL dialect we have developed.

Configuration becomes extremely flexible using IOC. We could change the type of persistence manager from an SQL to an object-oriented database if needed. As long as the chosen implementation implements the common *PersistManager* interface, all existing dependencies are satisfied in the application and the environment does not need to be recompiled. The configuration files are shown in Table 2.

Configuration File	**Description**
config/javamoo-beans.xml	Central component IOC configuration. Contains implementation choices and values for persistence, communication, environment, and plugins. Imports javamoo-events.xml, javamoo-persist.xml and javamoo-plugins.xml.
config/javamoo-events.xml	Event handler configuration. Specifies the mapping between events and their handlers. This makes the actions taken for each event configurable.
config/javamoo-persist.xml	Persistence configuration. Defines the persistence implementation and its particular configuration.
config/javamoo-plugins.xml	Defines a list of plugins for the environment. Plugins are started during the server startup sequence.
config/log4j.properties	Logging properties. Defines where logging messages are sent and what log level to use for packages.
bin/launcher.xml	Configuration file for Apache commons launcher. This is required to be in bin along side of LauncherBootstrap.class. This file does not require changes for each environment.

Table 2: JavaMOO Configuration Files

Configuration Files. JavaMOO has six configuration files. These configure the server components, logging, and launching of the server. The four Spring xml configuration files are meant to be customized for each environment and define the implementation and configuration of the server components. The other two files define logging and architecture independent startup configuration. The logging configuration can be modified to change debug level, or to send log messages to an alternate location, such as an email account for errors. The launcher configuration does not need to be modified.

Logging. The lack of a formal logging facility was a major problem in the Dollarbay prototype. When the prototype was created we were unaware of the industry standard logging facilities provided by the Java API or the open source log4j library. On multiple occasions when the prototype server was generating thousands of error logging messages, the log files grew to over a gigabyte in size. We had not thought of rolling logs or cleanup functionality when we implemented our own logging.

We now have standardized JavaMOO logging using the Apache log4j logging library. This provides a configurable logging facility where log messages have multiple levels of severity and can be routed to different destinations. It is a best practice to instrument the code with many log messages, from debugging messages to fatal error messages. This provides very useful information to help debug problems without the use of a debugger, especially problems at installation sites. The configuration can filter which level messages are sent to the actual log file.

Figure 6 shows our adopted usage of the Log4j logging facilities. Each class wishing to use logging has a static final variable that contains a logger object. This logger is used by all instances of the class for logging messages. Loggers are organized into a hierarchy that follows the package hierarchy. So, for example. the logger for javamoo.persist is a parent logger for javamoo.persist.sql.SQLPersistManager. Children loggers inherit the parent's log level and behaviors. This makes it possible to configure the logging based on the high-level parents, or control logging at a fine class-level granularity.

```
public class LoginEventHandler implements EventHandler {
      private static final Logger LOGGER =
            Logger.getLogger(LoginEventHandler.class);

      public void handle(Event e, Session s) throws EventException {
          LOGGER.info("Handling event[" + e + "], session[" + s + "]" );
      }
}
```

Figure 6: Example Logger Usage

Server Startup. Our goal is for JavaMOO to be platform independent, being able to run on Windows, Linux, Mac OS X or any other platform that supports Java. This gives us flexibility in distributing environments.

Previously the Dollarbay prototype used custom shell and batch scripts in order to start the server. A variety of issues became clear with this approach, such as maintainability and portability. Every different architecture required a new non-trivial script to be written from scratch. There was no ability to leverage common code between the scripts since they were each written specially for a particular operating system. We have addressed this problem by designing a well-defined start up sequence for the JavaMOO server independent of underlying architecture.

Figure 7 shows the sequence of steps that occur during the server start up. First an external utility starts the launcher. This could be a very simple couple-of-lines shell script, an application shortcut in Windows, or an operating system service. Launcher is an Apache commons component that "eliminates the need for a batch or shell script to launch a Java class" (Apache Launcher, 2009). This component reads a startup configuration from an xml file and starts the JavaMOO server. The same configuration file is used independent of the architecture we are currently running. This means JavaMOO will run on all operating systems where a supported VM is available.

The launcher calls MOOServer's main method, initializing the server object from the Spring IOC container. All components are specified in the configuration files and instantiated by IOC so that the server now has all of the environment components and plugins loaded. After initialization, the server is started by calling the start method on all dependent components and plugins in a specific order.

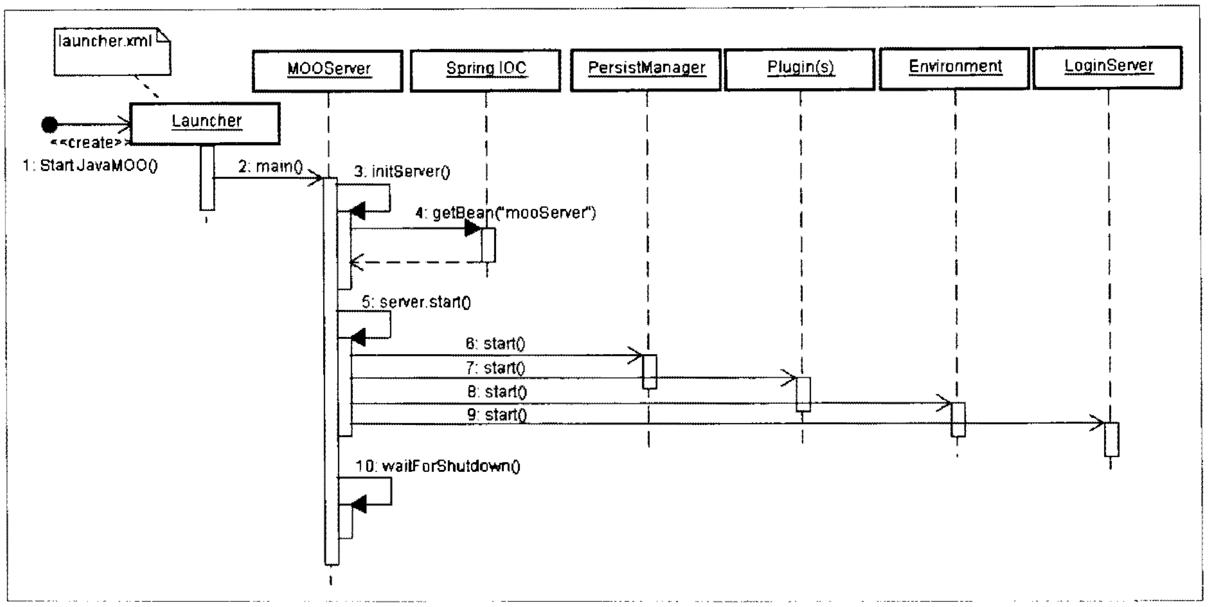


Figure 7: MOOServer Startup Sequence

First started is the persistence manager, which depending on the implementation, may open a connection to the database and perform other implementation specific startup logic. Persistence is started first so that any subsequent components are guaranteed to have access to persistent storage.

Started next are plugins, which are thin wrappers to components that extend the functionality of the JavaMOO server. Examples of plugins are the content server, or a network discovery service such as JmDNS. Plugins need to be notified when to start, and have explicit life-cycle management so that plugins are notified when the server is stopping.

The Environment component is started next. This is the entry point to start and initialize the specific environment implementation. For example, in Dollarbay the environment startup would initialize the map, refresh the scoreboard, and start the shoppers.

Finally the login server component is started. The login server is the entry point for clients to connect and is started last so that the initialization of the server has finished before clients are able to connect.

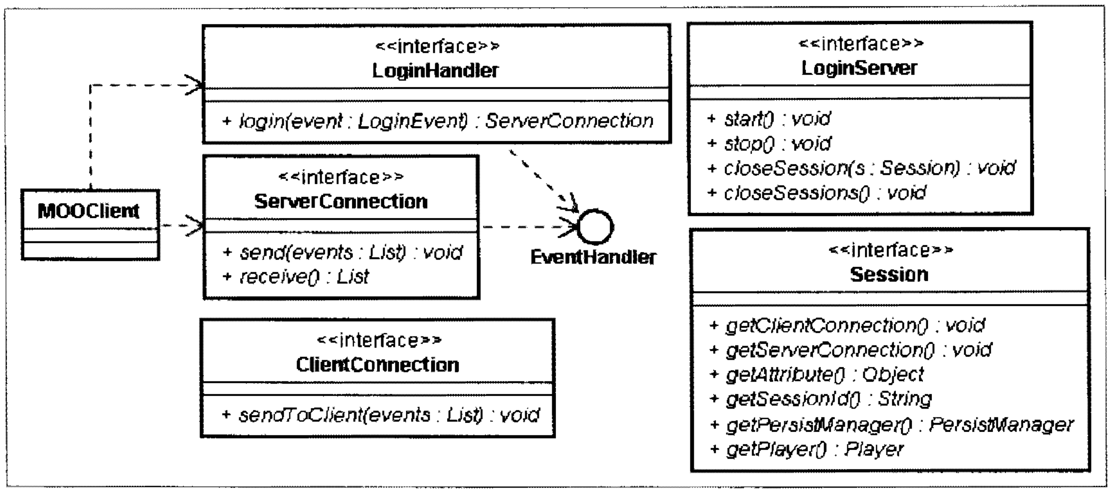
Plugins. JavaMOO allows new functionality to easily be extended through plugins. Different environments will have varying requirements, and may need to provide additional server functionality to fulfill these requirements. Environment developers configure plugins in the *javamoo-plugins.xml* configuration file and initialized by the server using a entry-point provided by the *Plugin* interface.

The content delivery component is an example of the type of functionality that can be provided through plugins. This component allows an environment to contain images and dynamic JSP web pages that any web client can access when connected to the server. The plugin simply starts up the Tomcat application server in embedded mode, which then allows JSP scripts to be written that have access to JavaMOO domain objects. For instance, the Dollarbay scoreboard can be dynamically generated through a JSP script.

Communication

JavaMOO uses an event based communication model where the server and client exchange small objects. These small objects are called events and correspond to specific environment actions. For example, a SayEvent with the data "Hello World!" may be passed from the client to the server to indicate the player spoke within a room. The server might then relay the SayEvent to clients who handle the event by displaying a speech balloon saying "Hello World!".

We designed communication with the five high-level interfaces shown in Figure 8. Interfaces are used in our design so that future environment developers can create their own implementation if the default implementation provided does not match their requirements.



<<interface>>
LoginHandler
+ login(event : LoginEvent) : ServerConnection
<<interface>>
LoginServer
+ start() : void
+ stop() : void
+ closeSession(s : Session) : void
+ closeSessions() : void
MOOClient
<<interface>>
ServerConnection
+ send(events : List) : void
+ receive() : List
EventHandler
<<interface>>
Session
+ getClientConnection() : void
+ getServerConnection() : void
+ getAttribute() : Object
+ getSessionId() : String
+ getPersistManager() : PersistManager
+ getPlayer() : Player
<<interface>>
ClientConnection
+ sendToClient(events : List) : void

Figure 8: Communication Interfaces

Our provided implementation uses Java RMI (discussed in the next section), which may incur too much overhead if real-time positional updates are required. A developer could implement an efficient low-level network protocol underneath our provided interfaces and configure it through the IOC container to fulfill their real-time requirement.

Figure 9 shows the sequence of client-server communication operations. Clients first connect to the server by calling the login function provided by LoginHandler. The login handler accepts a LoginEvent containing the data required to authenticate with the server. Depending on the environment, this data may be a simple username and password, or can be more complex as needed, such as a symmetric key encryption.

If the LoginHandler authenticates successfully, a new ServerConnection is returned to the client. The ServerConnection contains all of the state and logic necessary to send and receive events from the server, and since the connection is an interface, an appropriate underlying implementation can be chosen for each environment.

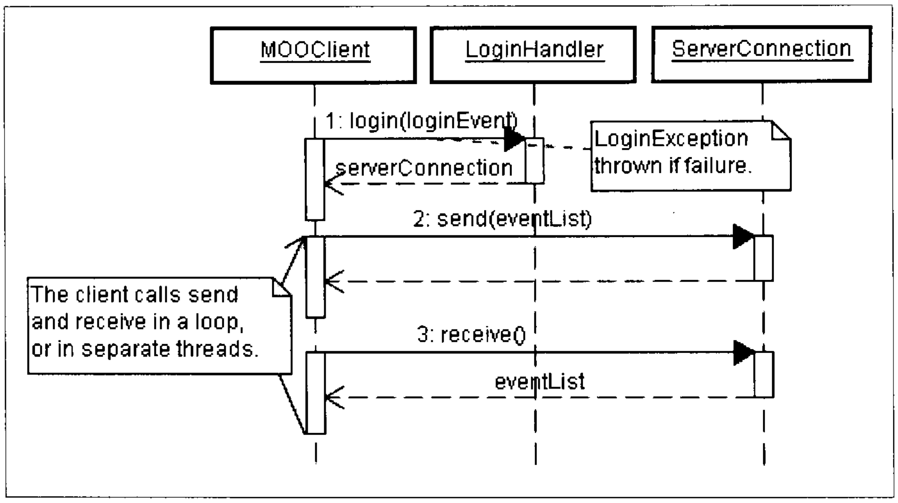


Figure 9: Client Event Communication Sequence

After connecting with the server, the client can begin sending and receiving events. User actions generate events that are sent to the server which handles them according to their type. The event handlers may generate new events that need to be sent back to clients through the sendToClient method. The client receives events from the server by calling the connection's receive method.

In order to process events properly and uniquely to each client, every connection is associated with a session object. The session keeps track of state information that is unique to each client, including the server and client connections, persistence manager, and generic session attributes. Session attributes are used to keep track of data that may be needed during event processing for the duration of the user's session. For example, the player object in Dollarbay is stored as a session attribute. This allows event handlers to access the player object, and through it, the player's store inventory and other state information.

Once the client is closed, a disconnect event is sent to the server. This causes the server to cleanup any resources that may have been created for the client.

Remote Method Invocation. We chose to provide an implementation of the communication interfaces based on Java's *remote method invocation* (RMI). RMI provides a transparent mechanism to invoke methods on objects residing on remote machines with method arguments automatically marshaled using serialization. By using RMI we avoid the need to implement our own socket protocol and event serialization, instead utilizing existing infrastructure. Environment developers may also benefit from additional features such as dynamic class loading which allows the implementation of objects to be transmitted along side of its data.

For clients to call remote methods they must obtain a remote object reference using a name lookup command, shown in Figure 10.

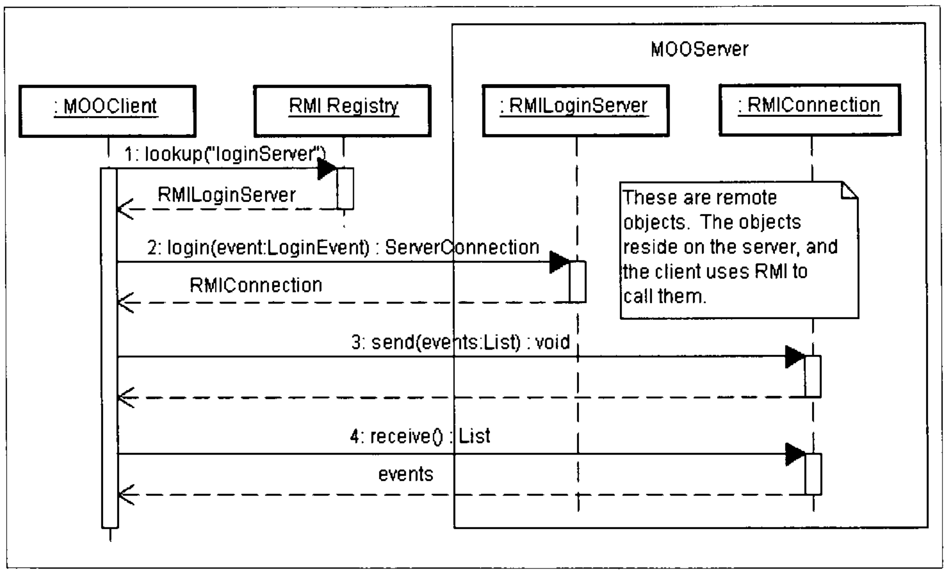

MOOServer
: MOOClient
RMI Registry
: RMILoginServer
: RMIConnection
1: lookup("loginServer")
RMILoginServer
2: login(event:LoginEvent) : ServerConnection
RMIConnection
These are remote objects. The objects reside on the server, and the client uses RMI to call them.
3: send(events:List) : void
4: receive() : List
events

Figure 10: RMI Communication Implementation

Each environment has one remote bootstrap object implementing the LoginHandler interface. The client initially attempts to lookup the LoginHandler from the RMI server by its classname. If found, the client can proceed by calling the remote login method and the server will attempt authorization, resulting in new RMIConnection and Session objects. The new connection is exported using its session id and is returned to allow the client to send and receive events.

Event Handling

Both client and server contain event handlers that are defined by the environment developers. Each event class has a handler associated that updates the environment state appropriately. Client side event handlers update the state of the user interface, such as displaying images or text, and server side event handlers update the state of the environment, such as changing the position of an item from one location to another.

JavaMOO provides a composite event handler called EventDispatcher that can handle events of any type by delegating responsibility to children handlers. The EventDispatcher is configurable through the javamoo-events.xml configuration file by defining a mapping between event classes and event handler classes.

Polymorphic Map. The underlying data structure for EventDispatcher is a *polymorphic map*. This allows EventDispatcher to handle any type of event passed to it by finding an appropriate child handler through the polymorphic map. An example event handler usage is shown in Figure 11.

Although an easy to derive and understand concept, we have not found previously existing research mentioning this usage of a map or hashtable. A standard map is defined as *map(k)->v,* where k is a lookup key that determines value v. In a polymorphic map, key values are object classes and defined to be in the map if any of its superclasses exist in the map. The most specific class in the inheritance hierarchy existing in the map takes precedence. A lookup in a polymorphic map can be implemented using a standard map as in Figure 12.

```
private PolymorphicMap<EventHandler> handlerMap =
    new PolymorphicMap<EventHandler>();

@Override
public void handle(Event e, Session s) throws EventException {
        EventHandler handler = handlerMap.get(e.getClass());
        if (handler == null) {
                throw new EventException(e,
                  "No handler for '" + e.getClass().getName() + "'.");
        } else {
                handler.handle(e, s);
        }
}
```

Figure 11: EventHandler Polymorphic Map Usage

```
poly_lookup(k) :
   val := NULL
   cur_key := k
   while (cur_key != NULL && val == NULL)
        val := map.lookup(cur_key)
        cur_key := parent_class(cur_key)
   return val
```

Figure 12: Polymorphic Map Implementation

The polymorphic map is useful whenever an action needs to take place based on the type of an object. Because of object inheritance and polymorphism, any valid class may have an entry in the map. For example, a default entry could be created for the base class so that any class without an explicit entry would use the default entry.

This enables us to assign a new set of polymorphic crosscutting behaviors to a preexisting class of objects. For example, in the event handling case, we define a new class called EventHandler which defines a handleEvent method. We want to be able to execute this method on any type of Event class, but be able to change its behavior based on the Event's class. We do this by creating EventHandlers for each Event class. When the event is received, the handler is found and executed. The handleEvent method can not exist as part of the specific Event because it may have server or client specific logic.

Having the dispatcher be configurable though a text file allows the behavior of the system to be modified without recompiling. The configuration determines how the various objects in the system are wired together. Figure 13 shows an example of how the event dispatcher is configured.

```
<bean id="eventDispatcher"
class="javamoo.event.handler.EventDispatcher">
  <constructor-arg>
    <map>
<!-- Add new event handler definitions here -->
      <entry key="javamoo.event.SimpleLoginEvent">
        <bean class="javamoo.event.handler.SimpleLoginEventHandler"/>
      </entry>

      <entry key="javamoo.event.DisconnectEvent">
        <bean class="javamoo.event.handler.DisconnectEventHandler"/>
      </entry>

      <entry key="javamoo.domain.event.ClickEvent">
        <bean class="javamoo.domain.event.handler.ClickEventHandler"/>
      </entry>
<!-- END event handler definitions -->
    </map>
  </constructor-arg>
</bean>
```

Figure 13: EventDispatcher Configuration Example

It can even be possible to modify the configuration at runtime though Java management extensions called JMX. Figure 14 shows a specific example the event dispatcher usage of a polymorphic map. In this example a player's click causes an explosion.

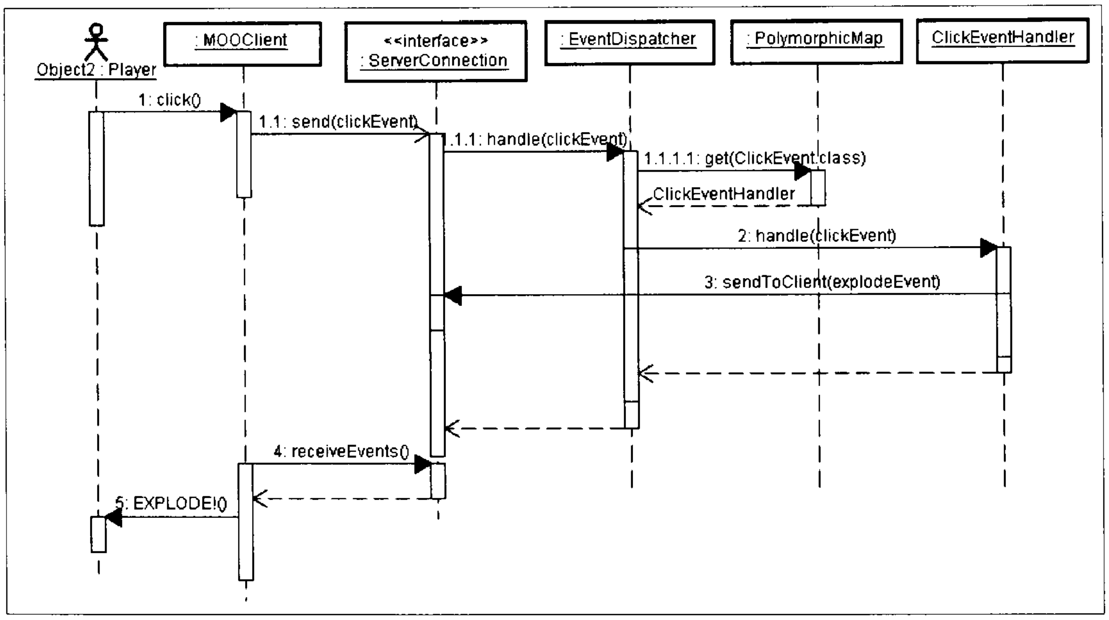


Figure 14: Click Causes Explosion Example

Persistence

Persistence ensures that environmental data, such as players or rooms, will survive multiple executions of a program. To achieve this, objects are written to a permanent data store. Our goals for creating an effective persistence mechanism are transparency, extendability, data integrity, and performance.

Transparency is the hiding of whether a resource is in local memory or if it is stored on disk (Tanenbaum, 2002). Having transparency of persistence will help make JavaMOO's programming model intuitive. We do not aim to achieve complete persistence transparency in JavaMOO, because previous research has attempted this with great effort and small return (Atkinson, 2000). Our goal is to balance the amount of programmer action required to store and retrieve persistent objects with simplicity and efficiency.

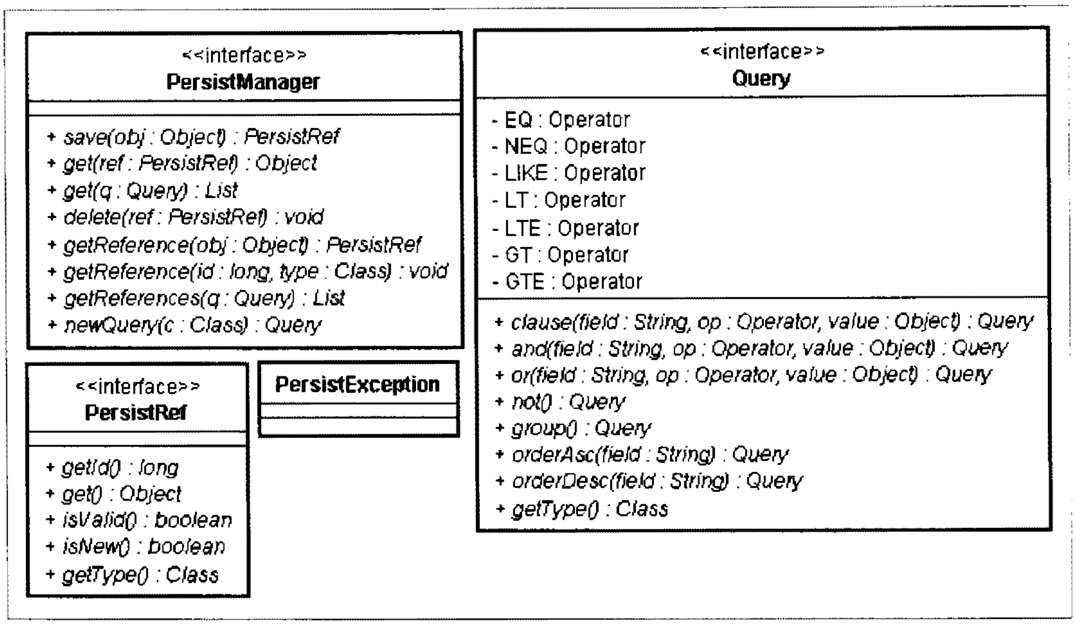
Domain Objects. Domain objects are environment specific; the Virtual Cell contains cell organelles, Geology Explorer has rocks and minerals, and Dollarbay has shoppers and products. The objects and their behaviors contribute to the uniqueness of each environment and will define the environments, their objectives, and the experience to be had from them.

The domain class library is created by the developers of each educational environment, as classes necessary to each specific game. It is the responsibility of the developers to ensure their domain objects are thread safe. JavaMOO is inherently multi-threaded, as domain objects are shared among all connected clients whose events are processed by a thread pool.

To assist in the creation of domain objects JavaMOO provides a set of reference classes including Player, Agent and Room. These classes can be used as is, or extended by the developers to create domain specific classes. As these objects are created, they are saved to persistent storage for later retrieval.

Interface Design. In order to meet our goal of extendability, we have designed a set of interfaces that facilitate easy to use persistence for storage, retrieval, and querying of objects. These interfaces, shown in Figure 15, make no assumptions on what types of data can be made persistent; all objects can potentially be persistent. They also make no assumptions as to what the underlying storage is, whether a database, binary file, or some other mechanism.

The PersistManager interface is the main entry point for object persistence. It provides the methods that save, get, and query objects. An implementation of this interface would provide the logic to take an object, break it into appropriate pieces, and write it out to storage. It also must be able to retrieve and reconstitute the object based on an implementation specific PersistRef, which is a unique reference to the object that was stored.



<<interface>>
PersistManager
+ save(obj : Object) : PersistRef
+ get(ref : PersistRef) : Object
+ get(q : Query) : List
+ delete(ref : PersistRef) : void
+ getReference(obj : Object) : PersistRef
+ getReference(id : long, type : Class) : void
+ getReferences(q : Query) : List
+ newQuery(c : Class) : Query
<<interface>>
PersistRef
+ getId() : long
+ get() : Object
+ isValid() : boolean
+ isNew() : boolean
+ getType() : Class
PersistException
<<interface>>
Query
- EQ : Operator
- NEQ : Operator
- LIKE : Operator
- LT : Operator
- LTE : Operator
- GT : Operator
- GTE : Operator
+ clause(field : String, op : Operator, value : Object) : Query
+ and(field : String, op : Operator, value : Object) : Query
+ or(field : String, op : Operator, value : Object) : Query
+ not() : Query
+ group() : Query
+ orderAsc(field : String) : Query
+ orderDesc(field : String) : Query
+ getType() : Class

Figure 15: Persistence Interface Design

Any implementation of these interfaces must follow a set of rules to ensure data integrity.

- **Storage Referential Integrity** – One copy of the object is saved to storage and consecutive save calls on the same object return equivalent PersistRefs, i.e. save(obj).equals(save(obj))
- **Retrieval Referential Integrity** – Two independent gets on equivalent PersistRefs must return the exact same object reference, i.e. ref1.equals(ref2) => get(ref1) == get(ref2)
- **Deletion Referential Integrity** – If a persistent object is deleted, all subsequent calls to retrieve the object return null, i.e. delete(ref) => get(ref) == null

Querying. A novel approach to querying for persistent objects has been designed as part of JavaMOO. This provides an intuitive way to formulate queries using Java code through chained function calling. Queries are created through a factory method on the persistence manager. A factory method allows different implementations of the persistence manager to choose the query implementation. At this point if the resulting Query is executed, would return all the objects of the specified type.

Query results are filtered by specifying clauses on the Query. In Figure 16, we want to query all people who have brown hair and weigh at least 200 pounds. When this query is executed it will return all Person objects that meet the specified restrictions.

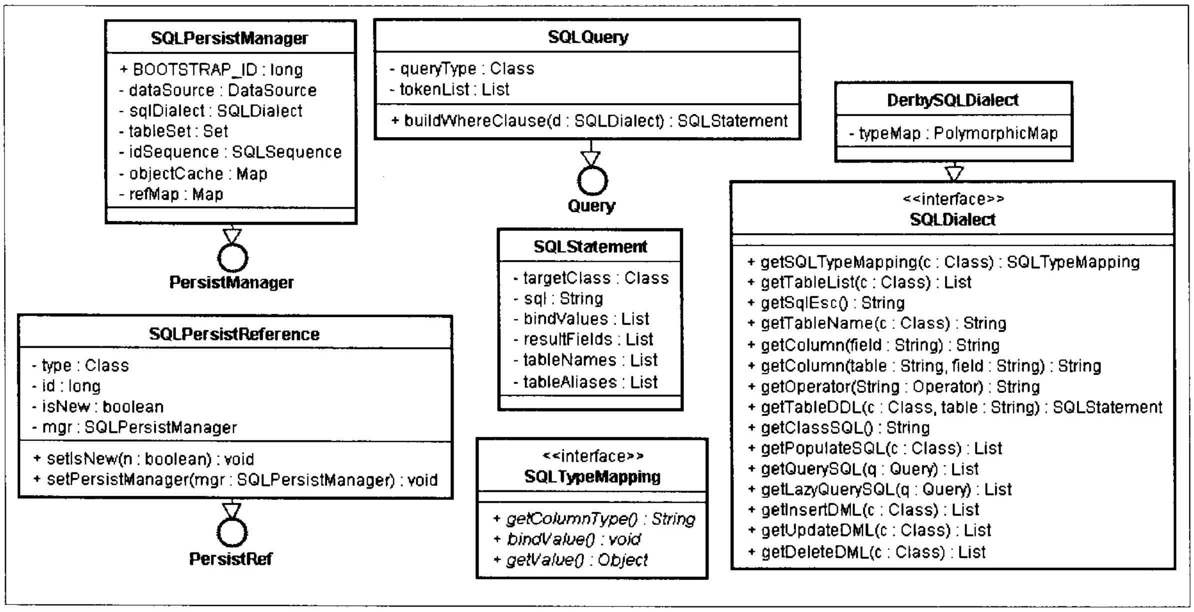
```
Query<Player> query = persistManager.newQuery(Person.class)
        .clause("hairColor", Query.EQ, "brown")
        .and("weight", Query.GTE, 200);

for (Person p : persistManager.get(query)) {
        System.out.println(p.getName());
}
```

Figure 16: Persistence Query Example

A current limitation of this query model is the string representation of field values such as “hairColor”. These names must correspond exactly with a field name in the class being queried. Although our queries provide an easy to use query interface, if a field is renamed, automatic code refactoring of queries currently cannot be performed. Currently there is no Java language construct to obtain class fields such as Person.class.hairColor without specifying a string to allow for automatic refactoring.

SQL Implementation. We have provided a SQL persistence implementation shown in Figure 17 using the interfaces discussed earlier. Our goals of extendability and performance have been considered as part of this implementation to allow for different SQL dialects, for example Derby, MySQL or Oracle. Caching techniques are used to aid in performance and referential integrity.

SQLPersistManager implements all of the methods defined by the PersistManager class. For example, the save method will result in a set of statements being generated in a particular dialect of SQL The successful execution of these statements will save the content of the object to the database. This sequence of operations is also true for the get, delete and query methods.



SQLPersistManager
+ BOOTSTRAP_ID : long
- dataSource : DataSource
- sqlDialect : SQLDialect
- tableSet : Set
- idSequence : SQLSequence
- objectCache : Map
- refMap : Map
PersistManager
SQLQuery
- queryType : Class
- tokenList : List
+ buildWhereClause(d : SQLDialect) : SQLStatement
Query
DerbySQLDialect
- typeMap : PolymorphicMap
<<interface>>
SQLDialect
+ getSQLTypeMapping(c : Class) : SQLTypeMapping
+ getTableList(c : Class) : List
+ getSqlEsc() : String
+ getTableName(c : Class) : String
+ getColumn(field : String) : String
+ getColumn(table : String, field : String) : String
+ getOperator(String : Operator) : String
+ getTableDDL(c : Class, table : String) : SQLStatement
+ getClassSQL() : String
+ getPopulateSQL(c : Class) : List
+ getQuerySQL(q : Query) : List
+ getLazyQuerySQL(q : Query) : List
+ getInsertDML(c : Class) : List
+ getUpdateDML(c : Class) : List
+ getDeleteDML(c : Class) : List
SQLStatement
- targetClass : Class
- sql : String
- bindValues : List
- resultFields : List
- tableNames : List
- tableAliases : List
SQLPersistReference
- type : Class
- id : long
- isNew : boolean
- mgr : SQLPersistManager
+ setIsNew(n : boolean) : void
+ setPersistManager(mgr : SQLPersistManager) : void
PersistRef
<<interface>>
SQLTypeMapping
+ getColumnType() : String
+ bindValue() : void
+ getValue() : Object

Figure 17: SQL Persistence Classes

SQLDialect abstracts all of the database implementation details into separate implementation classes. SQLPersistManager uses the configured dialect to generate the SQL required to persist objects. All that is required for JavaMOO to work with a new database is to implement a new type of SQLDialect and configure it to be used.

There are three requirements for a class to be persisted using SQLPersistManager:

- Empty constructor to ensure that JavaMOO can instantiate through reflection.
- All fields must have valid data mappings for data translation to and from the database.
- Field names must be case-insensitive unique to prevent database column name

collision.

Object Relational Mapping. During a save the first time a new class of object is encountered new tables may need to be created in the database. How many tables to create and how to map data into those tables is an implementation specific concern. The main problem is how to map the class inheritance tree into one or more database tables. There are three usual approaches to this problem: one table per hierarchy, one table per concrete class, or one table per class (Ambler, 1997).

In one table per hierarchy, each class as a new table created with all of its fields from all levels in the hierarchy as columns in the table. This has performance advantages of requiring only a single execution of insert, update, or select statements when updating or retrieving object data from the database. The disadvantage is performing queries across a common class in the hierarchy may require many separate queries.

In one table per concrete class a table is created for each level of the inheritance hierarchy which is a concrete class (not an interface or abstract class). This provides a more natural mapping between levels in the hierarchy and tables in the database. Multiple execution of statements may need to be required to store and retrieve object data. The number of tables is reduced because not all types are stored in the database.

Finally, in one table per class, every encountered object type has a corresponding table in the database, including interfaces. This is the most natural mapping between the inheritance hierarchy and the database tables. This approach allows queries on the interface level, finding all objects that implement a certain interface. This comes at a price; many more statements will be executed to extract and insert object data.

Independent of the table mapping chosen, one column will exist in the database for each field in an object. The database column data type must be decided for each object data type which depends on the specific database implementation. We have provided the SQLTypeMapping interface to implement these mappings and data conversions. Currently implementations exist for String, Numbers (int, float, etc.), Date, PersistRef, and Serializable. These mappings cover most types of Java objects and new mappings can be easily defined if needed.

Derby SQL Dialect. We have chosen the open source Derby database and one table per class object relational mapping for our reference implementation. Derby can be packaged along with JavaMOO and executed embedded in the running virtual machine so that no external database is required for the environment.

Every persistent class has a corresponding Derby table whose name is generated based on a combination of the hash code of the fully qualified class name and the simple class name. For example, the class dollarbay.domain.Product has the table name '1415003060_product'. The reason for using the hash code is to prevent long table names, which might grow beyond the maximum table name length allowed by Derby.

Since the value of Java's string hash code is consistent across virtual machine implementations, this table name will be portable. The name will be nearly guaranteed to be unique as the hash code is based on the full class name and appending the simple class name. In order for a naming collision to occur between two classes, the simple class names must be equal and the unique full class name must generate the same hash code.

The first time a class is encountered the SQLDialect's *getTableDDL* method is called

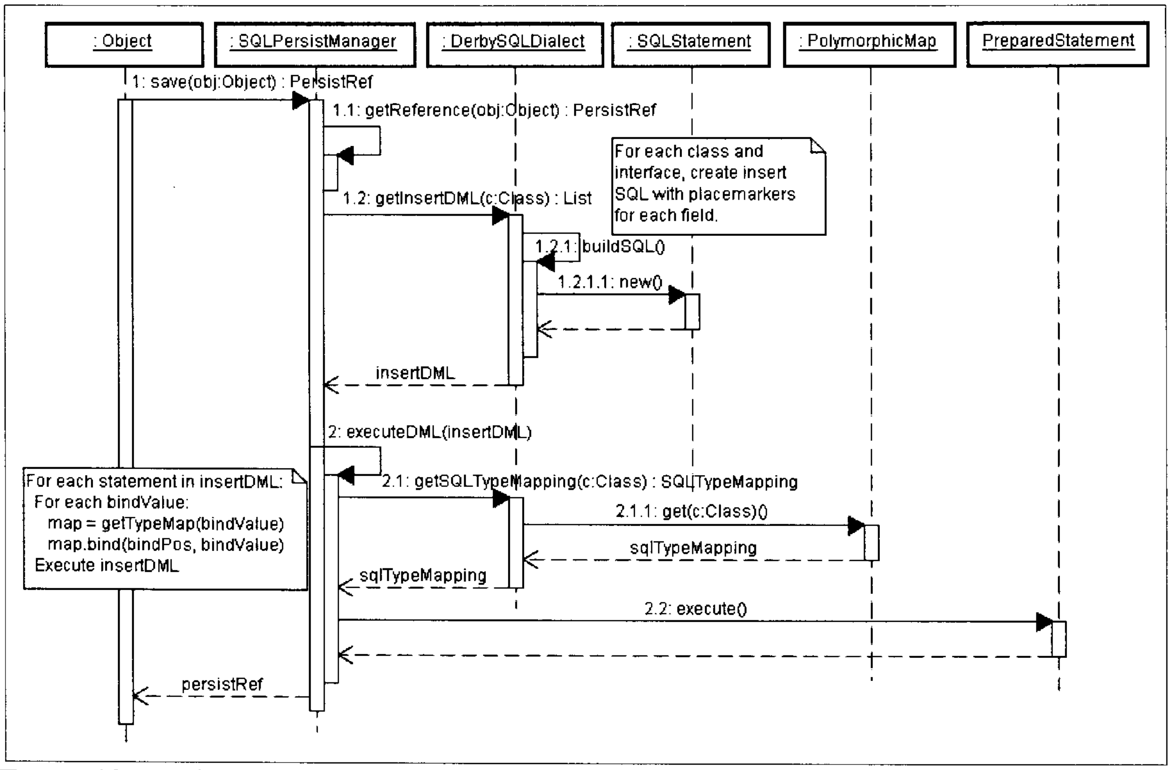
(DDL stands for Data Definition Language). This generates the SQL necessary to create a table with all of the fields in the class and a unique object identifier (OID). Figure 18 shows examples of generated DDL. The OID is the primary key for each table which can be used for more efficient data retrieval. The combination of OID and fully qualified classname, such as in a SQLPersistReference, allows an object to be reconstructed.

```
CREATE TABLE "1602218791_moobject"(oid BIGINT NOT NULL,
  "persistreference" VARCHAR(512), PRIMARY KEY (oid))

CREATE TABLE "604275198_tangible"( oid BIGINT NOT NULL,
  "xpos" DOUBLE, "ypos" DOUBLE, "zpos" DOUBLE, "name" VARCHAR(30000),
  "description" VARCHAR(30000), "imagefile" VARCHAR(30000), "location"
VARCHAR(512),
  "isdraggable" INT, "isclickable" INT, PRIMARY KEY (oid))

CREATE TABLE "1415003060_product"(oid BIGINT NOT
NULL,"annualdemandindex" DOUBLE,
  "msrp" INT,"size" INT,"seasonal" VARCHAR(30000), "singularname"
VARCHAR(30000),
  "nountype" VARCHAR(30000), "generictype" VARCHAR(30000), "iscollectible" INT,
  "numberleft" INT, "consumertable" BLOB, PRIMARY KEY (oid))
```

Figure 18: DDL for MOObject, Tangible, and Product

A save operation when using DerbySQLDialect with SQLPersistManager is shown in Figure 19. First, SQLPersistManager gets the PersistRef for the object it is saving to see whether the object is being updated, or is new and needs to be inserted into the database. In this example the object needs to be inserted and the dialect's getInsertDML method is called on the object's class (DML stands for Data Modification Language).

The getInsertDML method iterates through all of the superclasses and interfaces belonging to the target class and creates SQL insert statements for each one. It does this by using the Java Reflection API to introspect the class field types and names to build the query.



: Object
: SQLPersistManager
: DerbySQLDialect
: SQLStatement
: PolymorphicMap
PreparedStatement
1: save(obj:Object) : PersistRef
1.1: getReference(obj:Object) : PersistRef
For each class and interface, create insert SQL with placemarkers for each field.
1.2: getInsertDML(c:Class) : List
1.2.1: buildSQL()
1.2.1.1: new()
insertDML
2: executeDML(insertDML)
For each statement in insertDML: For each bindValue: map = getTypeMap(bindValue) map.bind(bindPos, bindValue) Execute insertDML
2.1: getSQLTypeMapping(c:Class) : SQLTypeMapping
2.1.1: get(c:Class)()
sqlTypeMapping
sqlTypeMapping
2.2: execute()
persistRef

Figure 19: DerbySQLDialect Save Example

PolymorphicMap is used as part of the implementation to lookup mappings based on the type of the persistent object. This allows us to find the most specific data mapping for an object and specify very general mappings based on interfaces such as Serializable or List. The specific data type mappings are configured in javamoo-persist.xml and can be changed without recompiling. Default mappings are shown in Table 3 and new data mappings can be added for any Java class.

Figure 20 shows the SQL resulting from getInsertDML for a Dollarbay product. There is an insert statement for each table of the inheritance hierarchy. Each insert value is assigned a bind variable positional placeholder, a question mark '?', which indicates where data should be placed for the statement. These bind variables are used in prepared statements, which is an efficient mechanism to execute the same SQL statements repeatedly. Since we are saving multiple objects of the same type, we will be executing identical statements many times.

Java Type	Derby Type
Long, long	BIGINT
Date	BIGINT
Integer, int	INT
Short, short	INT
Byte, byte	INT
Boolean, boolean	INT
Double, double	DOUBLE
Float, float	DOUBLE
Character, char	CHAR(1)
String	VARCHAR(30000)
PersistRef	VARCHAR(512)
List, Map, Set, Collection	BLOB
Serializable	BLOB

Table 3: Default Derby Data Type Mappings

```
INSERT INTO "1415003060_product"
(oid,"annualdemandindex","msrp","size","seasonal","singularname","nountype","generict
ype","iscollectible","numberleft","consumertable") VALUES (?,?,?,?,?,?,?,?,?,?,?)

INSERT INTO "604275198_tangible"
(oid,"xpos","ypos","zpos","name","description","imagefile","location","isdraggable","iscli
ckable") VALUES (?,?,?,?,?,?,?,?,?,?)

INSERT INTO "1602218791_moobject" (oid,"persistreference") VALUES (?,?)

INSERT INTO "3211360658_object" (oid,CLASSNAME) VALUES (?,?)
```
Figure 20: Generated Inserts for a Dollarbay Product

The update and delete cases are similar to the insert example.

Caching Strategies. We have developed various caching strategies to aid performance and comply with the referential integrity requirements of the PersistManager interface. These cache approaches include the implementation of CachedSQLDialect, QueuedPersistManager, and two weak reference maps internal to SQLPersistManager.

CachedSQLDialect is a wrapper class for SQLDialect that caches results from calls to various dialect methods. The premise is that SQL statements used for creating and modifying database tables for a particular class do not change and should only be generated once to save cycles. CachedSQLDialect uses maps to keep track of which classes have already had the SQL generated for certain methods and instead of calling the child dialect, returns the cached data. This is done for the getInsertDML, getUpdateDML, getDeleteDML, getPopulateDML,getTableList and getClassSQL methods. This saves much redundant processing.

QueuedPersistManager is another wrapper class that queues save requests and forwards them in FIFO order to its child PersistManager. It consolidates redundant requests that already exist on the queue, effectively collapsing consecutive requests into a single save operation. An object may have all of its fields modified in rapid succession. If save is called after each modification, collapsing all these requests into a single operation will greatly help reduce the number of database round trips.

We use a new data structure for QueuedPersistManager called a QueueSet. QueueSet is a FIFO queue that follows set semantics. We have implemented this data structure by extending AbstractMap, implementing the Set interface, and using a TreeMap whose main operations (add, remove, contains) have *O(log n)* complexity. Therefore, the QueueSet will have comparable complexity for offer and poll operations.

QueueSet has an option to use identity instead of equals comparison for set semantics. Identity comparison means that two objects are equal only if they are exactly the same object, i.e. their memory addresses are equal. This provides an identity preserving data structure which is useful for persistence requests. QueueSet is a general purpose data structure in the javamoo.util package used by QueuedPersistManager, but can be used by anywhere found useful.

Finally, we have utilized two reference maps to cache persistent objects and their corresponding PersistRefs. A reference map is a lookup table whose keys or values are stored using Java References, and depending on configuration, may be garbage collected and disappear from the map. This type of map is typically used to create a memory sensitive cache allowing garbage collection in the case of memory pressure.

The first cache in SQLPersistManager is a ReferenceMap that uses PersistRefs for keys and persistent objects as values. The objects are stored in the map using weak references so that if a persistent object is garbage collected, it will disappear from the cache. We conform to the retrieval referential integrity requirement by checking this cache when a get operation is performed on a PersistRef. If a corresponding object exists in the cache we return it immediately. Otherwise we create a new object, populate it from the database, enter into the cache, and return the object. This ensures that only one valid persistent object is loaded into memory for any PersistRef.

The second cache is the inverse of the first by using the objects as keys pointing to PersistRef values. Again, the objects are stored using weak references to allow garbage collection, with one important distinction. We have used a ReferenceIdentityMap so that key comparison is identity based, meaning that the exact object must be used for lookup in order to succeed.

The ReferenceIdentityMap helps solve a very important transparency problem. In the original Dollarbay prototype we required all persistent objects to extend a common base class called MOObject. We did this to assign a unique object identifier and keep track of whether an object was newly created and therefore requiring an insert or already persisted and requiring an update. This caused several design problems, and prevented us from directly persisting any class that was not explicitly written for JavaMOO.

Using an identity comparison allows us to factor out the object identifier to an external map, therefore making a common base class unnecessary and enabling us to store preexisting Java classes external to JavaMOO. This makes our persistence solution transparent to the implementation of classes and increases its extendability. Most classes can be stored using existing data mappings, but new data mappings can be written and configured with no modifications to JavaMOO or the preexisting class.

When an object is first saved, a call is made to getReference that attempts to lookup the corresponding PersistRef using the identity reference map. If no PersistRef exists one is created having an incremented unique object identifier and its *isNew* variable set to true. This value indicates an insert instead of an update statement must be executed

We also add an entry to the cache when an object is first loaded from the database. This ensures compliance with storage referential integrity, meaning that we will not attempt to insert duplicate entries into the database.

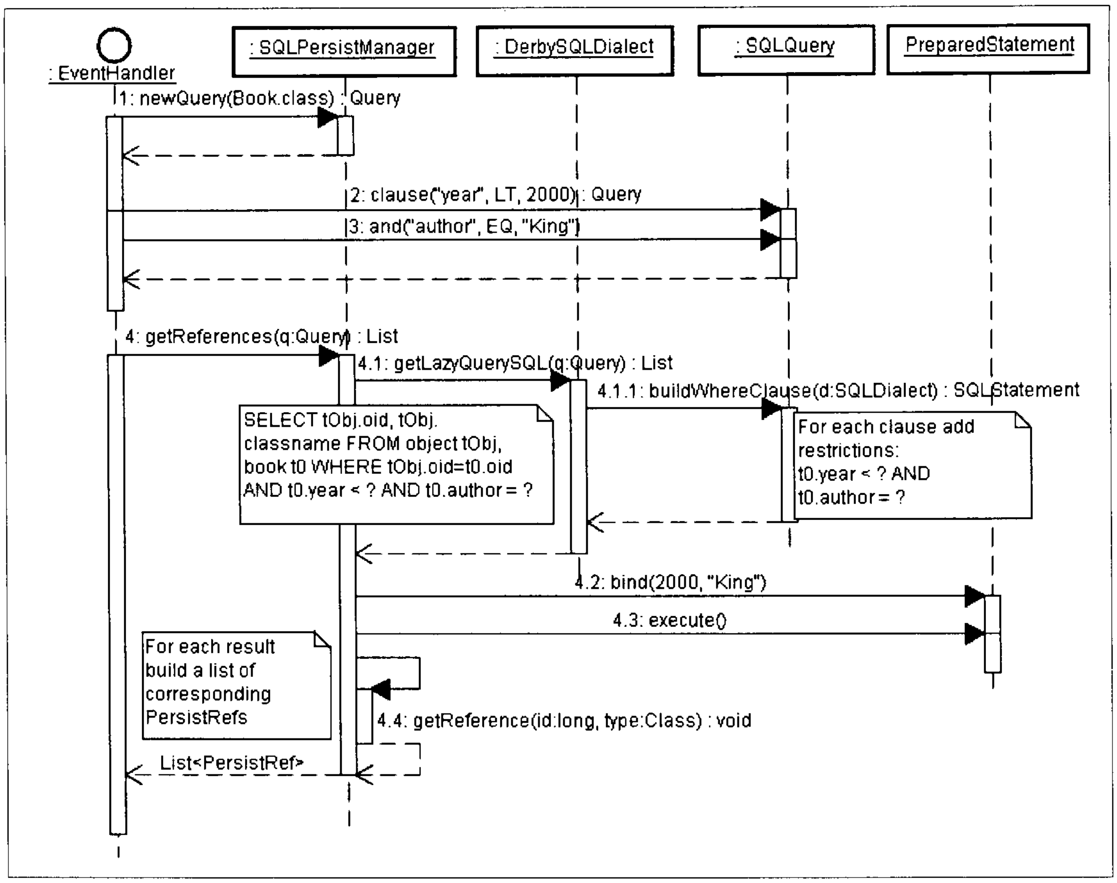
Query Implementation. SQLPersistManager provides the SQLQuery class which is a query implementation that generates a where clause to be used during select statements. End users will not be aware of the specific implementation because query instantiation is abstracted through the *newQuery* factory method of PersistManager.

SQLQuery keeps a record of all clauses and operations requested for the query, and constructs an equivalent SQL where-clause with bind values. The database tables used in the query are assigned table aliases so to ensure shorter statement text and enhance readability.

The example shown in Figure 21 builds and executes a query for books written before the year 2000 by author King. First the *newQuery* factory method is called on the PersistManager. Because the programmer is using the PersistManager interface, the underlying SQLQuery implementation is hidden. Next the returned Query is used to add clauses.

The event handler calls the clause method which takes the field name, an operator, and a comparison value. This adds an internal internal to the SQLQuery that holds the information (CLAUSE, “year”, LT, 2000). Next the event handler adds another clause by calling the *and* method. This adds another token holding the information (AND, “author”, EQ, “King”). If a field name is typed incorrectly a *FieldNotFoundException* will be thrown. The query is now built to the event handler's satisfaction.

The event handler can now get objects satisfying the query. It does this by calling *getReferences* who in turn calls *getLazyQuerySQL* to build a statement to query objects matching the query. SQLQuery loops over all of the tokens in FIFO order and builds a where clause from the field names. The where clause is used to build the select statement joining multiple tables together to perform the query. PersistRefs are created from the results of the query, inserted into a list, and returned to the event handler.



: EventHandler
: SQLPersistManager
: DerbySQLDialect
: SQLQuery
PreparedStatement
1: newQuery(Book.class) : Query
2: clause("year", LT, 2000) : Query
3: and("author", EQ, "King")
4: getReferences(q:Query) : List
4.1: getLazyQuerySQL(q:Query) : List
4.1.1: buildWhereClause(d:SQLDialect) : SQLStatement
SELECT tObj.oid, tObj. classname FROM object tObj, book t0 WHERE tObj.oid=t0.oid AND t0.year < ? AND t0.author = ?
For each clause add restrictions: t0.year < ? AND t0.author = ?
4.2: bind(2000, "King")
4.3: execute()
For each result build a list of corresponding PersistRefs
4.4: getReference(id:long, type:Class) : void
List<PersistRef>

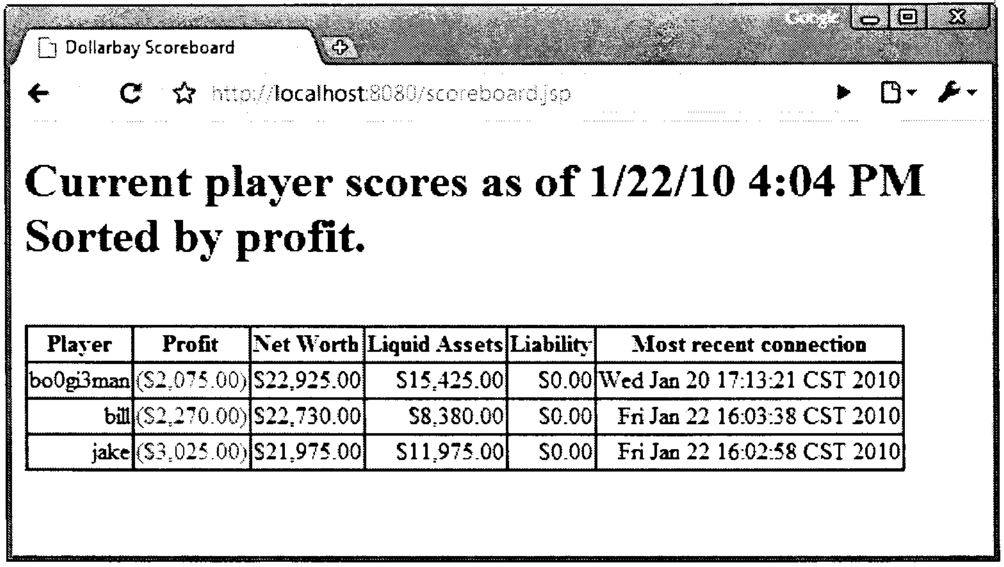
Figure 21: SQLQuery for Books Written Before 2000 by Author King

Content Delivery

Delivery of game content to clients can greatly influence end-user perceptions of performance and quality. Content may include images, three-dimensional models, sounds, or movies whose reliable transfer is necessary for game functionality. LambdaMOO environments rely on a web server separate from the server program to deliver content to clients. This introduces an external system dependency that must be installed and maintained by an administrator.

To address this, JavaMOO includes an embedded web server so content can be packaged along side of the class library. This enables the educational simulation to be distributed as a complete application instead of requiring external dependencies.

```
<%@ page import="javamoo.persist.*" %>
<%@ page import="dollarbay.event.handler.*" %>
<%
PersistManager mgr = (PersistManager)getServletContext()
  .getAttribute(PersistManager.class.getName());
%>
<html><head><title>Dollarbay Scoreboard</title></head>
<body>
<% out.print(InformationEventHandler.getScoreboard(mgr)); %>
</body>
</html>
```



Current player scores as of 1/22/10 4:04 PM Sorted by profit.

Player	Profit	Net Worth	Liquid Assets	Liability	Most recent connection
bo0gi3man	($2,075.00)	$22,925.00	$15,425.00	$0.00	Wed Jan 20 17:13:21 CST 2010
bill	($2,270.00)	$22,730.00	$8,380.00	$0.00	Fri Jan 22 16:03:38 CST 2010
jake	($3,025.00)	$21,975.00	$11,975.00	$0.00	Fri Jan 22 16:02:58 CST 2010

Figure 22: Dollarbay Scoreboard JSP script

JavaMOO provides a plugin that starts an embedded web application server called Apache Tomcat (tomcat.apache.org). Dynamic web pages called Java Server Pages (JSPs) can be authored for Tomcat which are accessible through standard web browsers.

Domain objects are available to JSPs through the JavaMOO persistence manager. When the Tomcat plugin is first started, the persistence manager is set as a servlet attribute using its class name as a lookup key. To obtain the persistence manager a JSP script must call *ServletContext.getAttribute()*.

Queries can then be run against the persistence manager to obtain references to objects and build web pages from them. New objects can be created and persisted, for instance displaying all of the current player's scores or executing learning assessments to test the knowledge acquired by the students. Any data gathered through the web server can be stored in the standard JavaMOO persistence mechanism. Figure 22 shows the Dollarbay scoreboard as a JSP script.

Autonomous Agents

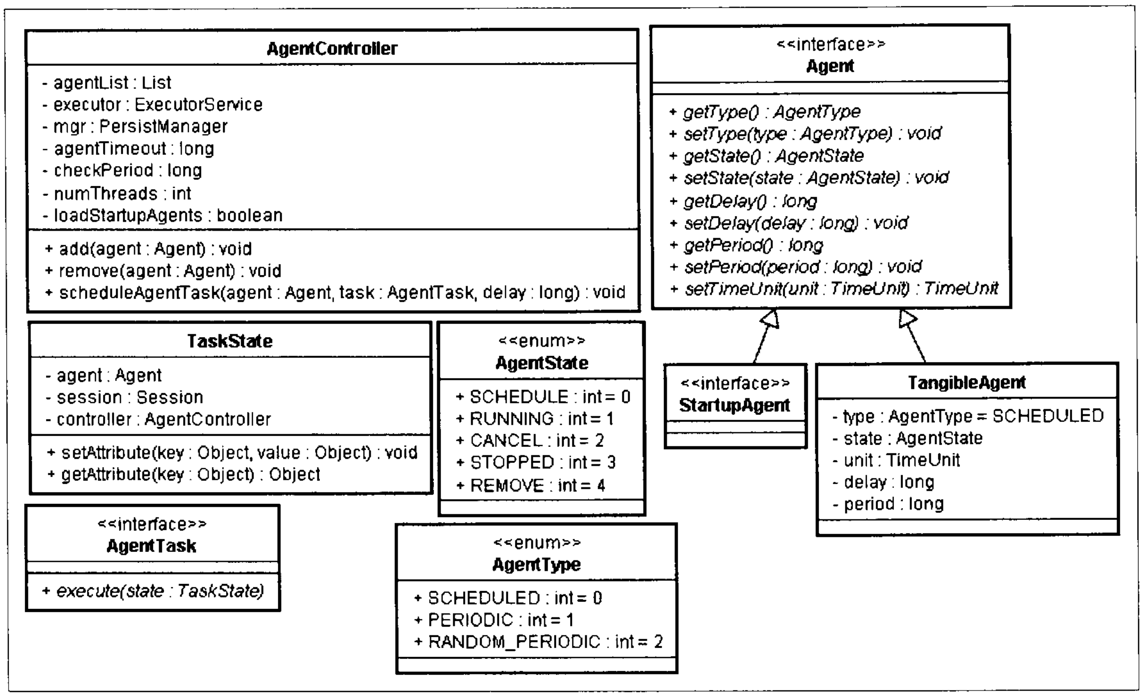
Franklin and Graesser define an autonomous agent as "a system situated within and a part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to affect what it senses in the future" (Franklin, 1996).

Agents are naturally implemented using threads. In Dollarbay, customers and employees are agents that participate autonomously in simulated commerce. There are also many "atmosphere agents", some notable examples being a beat cop, a fortune teller, and a political candidate running for office. These types of agents help bring life to an environment.

In the original Dollarbay prototype each agent had their own thread of execution even though they may have only executed once every hour. This model complicated multi-threaded programming because it was difficult to debug behavior between agents. We have created a lightweight framework in JavaMOO to solve these problems by formalizing the execution of autonomous agents within environments.

The lifecycle and execution of agents is managed by the agent controller component shown in Figure 23. The AgentController is implemented as a Plugin and so is started during the JavaMOO startup sequence. It can be configured to load all persistent agents implementing the StartupAgent marker interface. This behavior, enabled by the *loadStartupAgents* variable, allows Agents to automatically start where they left off before the server stopped.

Our framework provides a scheduling capability to agents within an environment, but does not provide the facilities to implement true agents which learn from their environment. This responsibility for creating true agents is left to the environment developer, which can use the agent framework in order to help schedule tasks which the agent must perform.


AgentController
- agentList : List
- executor : ExecutorService
- mgr : PersistManager
- agentTimeout : long
- checkPeriod : long
- numThreads : int
- loadStartupAgents : boolean
+ add(agent : Agent) : void
+ remove(agent : Agent) : void
+ scheduleAgentTask(agent : Agent, task : AgentTask, delay : long) : void
<<interface>>
Agent
+ getType() : AgentType
+ setType(type : AgentType) : void
+ getState() : AgentState
+ setState(state : AgentState) : void
+ getDelay() : long
+ setDelay(delay : long) : void
+ getPeriod() : long
+ setPeriod(period : long) : void
+ setTimeUnit(unit : TimeUnit) : TimeUnit
TaskState
- agent : Agent
- session : Session
- controller : AgentController
+ setAttribute(key : Object, value : Object) : void
+ getAttribute(key : Object) : Object
<<enum>>
AgentState
+ SCHEDULE : int = 0
+ RUNNING : int = 1
+ CANCEL : int = 2
+ STOPPED : int = 3
+ REMOVE : int = 4
<<interface>>
StartupAgent
TangibleAgent
- type : AgentType = SCHEDULED
- state : AgentState
- unit : TimeUnit
- delay : long
- period : long
<<interface>>
AgentTask
+ execute(state : TaskState)
<<enum>>
AgentType
+ SCHEDULED : int = 0
+ PERIODIC : int = 1
+ RANDOM_PERIODIC : int = 2
Figure 23: Agent Framework Classes

AgentController uses a *scheduledThreadPoolExecutor* to start runnable objects at either periodic or scheduled times in the future. Underlying the executor is a thread pool responsible for the actual execution of AgentTasks. Using a pool helps reduce thread pressure, and simplifies agent implementation by using a scheduling framework instead of a series of sleep function calls. A consequence is that agents can now be naturally implemented as state machines, with scheduled transitions between states.

All agents must implement the Agent interface. This provides the methods that the AgentController examines to determine when and how to execute the agent's tasks. There are three types of agents: scheduled, periodic, and random_periodic. The agent can switch between these types dynamically if needed to meet the desired agent behavior.

Scheduled agents' tasks are explicitly scheduled to run once at a predefined time in the future using the *delay* Agent attribute. The delay units are defined by the agent's *getTimeUnit* method which specifies whether the unit is seconds, minutes, etc. This agent is useful for scripting actions taking place at scheduled intervals. For example an agent that enters a room, a second later says 'Hello', and two seconds later leaves. This can be implemented as a sequence of tasks that schedule the next task a certain amount of time into the future.

Periodic agents are executed indefinitely using a predefined period. Again the time unit is defined by *getTimeUnit.* This type of agent is useful for scheduling actions that must take place on a fixed schedule. A simple example would be a character that updated a scoreboard every minute, or an agent that was responsible for managing an area and needed to monitor activity periodically.

Finally, random periodic agents are executed indefinitely using a predefined period with a random delay. This can be used for modeling natural behaviors that must occur at random times. The next scheduled execution is guaranteed to occur randomly uniform within the time interval [period-delay, period+delay]. We use this type of agent to implement the atmosphere agents of Dollarbay, who should make an appearance once in a while, but not always at the same time.

An agent can exist in five states relative to the agent controller: scheduled, running, cancel, stopped, and remove. Scheduled means that an agent has been scheduled to run at some time in the future. Running agents are currently executing a task. The agent controller implements a timeout value to monitor running agents, and interrupt them in the event they become unresponsive. The cancel state means that any scheduled agent tasks should be canceled by the controller and moved to stopped state. Stopped agents still exist in the agent controller, but have not been scheduled for execution. Finally the remove state means to remove the agent from the controller all together.

Environment Development

One of the goals for JavaMOO is to make environment development easier while following programming best practices. With these considerations in mind, this section provides the necessary information to help developers create new JavaMOO environments.

Project Structure

JavaMOO is separated into three independent folders: client, common, and server. The client and server folder contains code which is only used on either the client or the server respectively. For example, graphical user interface classes are in the client folder while object persistence classes exist in the server folder. The common folder contains classes that are shared between the server and client. For example, event classes are shared because they are sent between the client and server. The folders are built into three jar files: javamoo-client.jar, javamoo-common.jar, and javamoo-server.jar.

The client and server folders have a similar substructure. Both contain the following folders: bin, config, lib, log, and src. The bin folder contains files necessary to execute the programs using Apache Commons launcher. Configuration information is kept in the config folder and log files generated by Apache log4j are kept in the log folder. All jar file dependencies necessary to run the client or server are kept in the lib folder. Finally, and most importantly, the Java source files are organized into packages, which is a folder hierarchy inside of the src folder.

The project structure of an environment should mirror that of JavaMOO. This is not required but a recommended practice that we have found useful in organizing code and keeping a clear separation between client and server. A JavaMOO environment template is built as part of the distribution which contains all of the folder structure, jar files, and configuration files necessary to start a project.

We have found Eclipse to be a very productive integrated development environment (IDE). A new project can be quickly created within Eclipse by going to *File->Import->Existing Project* and selecting the template zip file provided by JavaMOO. Figure Error: Reference source not found shows the Dollarbay project in Eclipse which was created from the JavaMOO project template.

JavaMOO Package Structure. JavaMOO organizes its classes into packages that provide a grouping by functional area. The client jar has two packages, javamoo.client and javamoo.client.handler that contain convenience classes for GUI development The most interesting classes to examine exist in the common and server jars.

The following sub packages currently exist in the top level javamoo package: agent, domain, event, net, persist, plugin and server. The package names correspond to the components discussed in the Overview section, except that the folder named *net* corresponds to the communication component, and content delivery is implemented as a *plugin*.

An important Java feature is that the same package can be found in multiple source folders, and hence multiple jar files. This allows a programmer to add a new class to a package without modifying source trees or jar files. Unit tests can be created in a separate source tree and still belong to the same package as the target class being tested, increasing the visibility of package private and protected functions for testing. Although a seemingly simple concept, knowing that the same package may exist in multiple folders is very useful.

An example of shared package names is javamoo.event.handler which exists in both the common and server jars. This allows us to have a generic handler called EventDispatcher which can be used on both the client and server, but at the same time have server specific handlers on the server source folder. When the environment client and server are built, they include the javamoo-common.jar, and so both contain the EventDispatcher class.

Environment Package Structure. To help organize classes we recommend environment projects follow a similar package structure to JavaMOO. The top level package should reflect the project name. Figure 24 shows the package structure of Dollarbay.

You can see that Dollarbay follows JavaMOO's package structure: the project name dollarbay as the top-level package with appropriate sub packages. In Dollarbay, the only package in the common folder is event because communication events are the only Dollarbay specific classes shared between the client and server.

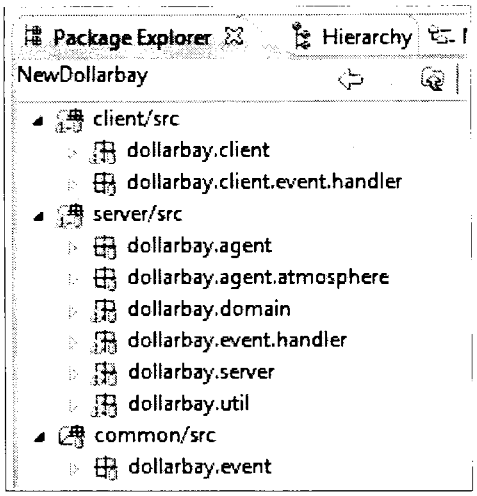

Package Explorer
Hierarchy
NewDollarbay
client/src
dollarbay.client
dollarbay.client.event.handler
server/src
dollarbay.agent
dollarbay.agent.atmosphere
dollarbay.domain
dollarbay.event.handler
dollarbay.server
dollarbay.util
common/src
dollarbay.event

Figure 24: Dollarbay Packages

Domain Objects

Every environment has domain specific objects that define what things can exist in a particular environment and how they are related to each other. Some domain objects should be persistent, like players and their inventory, other objects are transitory and do not need to persist across invocations.

JavaMOO provides a set of basic domain classes in the javamoo.domain package that can be used or extended to create new domain objects. They are modeled in the spirit of the original MOO room metaphor. These domain classes include room, exit, player, and two generic classes called Tangible and MOObject. Tangible is an object which can be physically observed in the environment and has properties such as position and an image.

MOObject is a convenience class to provide persistence that other domain objects can extend. It provides the methods save, delete, and getRef and keeps track of its persistence information. A PersistManager must be passed into the MOObject constructor and the save method be called in order for the object to be made persistent. Once the save method is called, the object will be stored using the provided persistence manager. For any method that modifies the object the save method should be called again to update the data store with the modified data. It is up to the developer of the class to decide whether the object should call save when modified, or whether it is the caller's responsibility to call the save method.

Figure 25 shows an example domain class named Car that extends MOObject. The car has three fields which will be persistent and one field, isRunning, which will not be saved because it is marked transient. Transient fields should always have an initializer, unless the default initializer is appropriate. There are two constructors for Car. The first, the required empty constructor, allows the persistence manager to reconstruct the object through reflection when being retrieved from the data store.

The second constructor is the one called by the consumer of this class. The first line of this constructor calls its super constructor on MOObject to pass the persistence manager. It then initializes its member variables, and saves itself. It is not required to use the MOObject class for persistence, and it may not be possible if the developer must extend a different class that does not stem from MOObject. The standard persistence manager interface can be used to persist such classes.

Environment developers need to identify all of the possible entities that will exist in the environment. This can be done in a brain-storm session, along with finding all of the entities' properties and relationships between entities. These relationships may help with organizing classes or determining where dependencies or inheritance might exist. These entities will correspond to the domain classes that are implemented for the environment.

Design with Interfaces. Where a common generalization between entities is identified a Java interface should be defined. Interfaces help create a flexible design environment and should be used liberally. Most entities should result in a corresponding interface as it helps so much in reducing dependencies and unit testing (Gamma, 1995).

```
package racing.domain;

import javamoo.domain.MOObject;
import javamoo.persist.PersistManager;

public class Car extends MOObject {
	private String mfg;
	private String model;
	private int year;
	private transient boolean isRunning = false;

	public Car() {}

	public Car(PersistManager persistManager,
		String mfg, String model, int year) {
		super(persistManager);
		this.mfg = mfg;
		this.model = model;
		this.year = year;
		save();
	}
	...
}
```

Figure 25: Example MOObject Domain Object

Interfaces allow developers to implement an is-a relationship without requiring an inheritance structure. This helps to reduce dependencies between components because interface usage will protect against changes in the underlying components. Interface types should be used as fields and parameters to constructors or methods, then no explicit dependencies are created between classes. When doing this, the entire underlying implementation can be swapped without breaking code.

For convenience, a developer may wish to create an abstract class that implements the interface. In the JDK, for example, AbstractMap exists to make implementing the Map interface easy, only requiring the extending class to implement one method instead of fourteen. However, it is still possible to implement the Map directly if required.

JavaMOO uses interfaces extensively to support generic implementation of components. The persistence manager and communication components all have interfaces defined, so if a different technology other than SQL or RMI is appropriate for an environment, a new implementation can be created and configured to be used without the need to recompile the environment.

Composition versus Inheritance. Many times a decision exists during implementation whether to extend an existing class or to consume it's behavior as a member variable. When faced with this decision it is wise to "favor composition over inheritance" (Gamma, 1995).

Composition allows us to take multiple separate components and utilize their behavior to create a new component without using inheritance. In order to use composition we need to be able to implement desired interfaces, so the previously discussed practice of designing with interfaces is important.

An example of the use of composition in JavaMOO is the QueuedPersistManager. This is a persistence manager that forwards object save requests to its child persistence manager. This class could have been implemented as a specific extension to SQLPersistManager by extending the SQLPersistManager and overriding the save method but this approach would not have been reusable.

Using composition allows us to create a general concept surrounding persistence managers that can be used for any persistence manager, not a specific extension. In general, whenever an extension of functionality can be abstracted to an interface, composition results in more reusable code that can be applied to a large variety of problem sets.

Events

Events generated by clients and servers form the basis of each environment's communication protocol. Events sent from the server to the client typically carry domain object state changes, or are used to query the user for additional information. Events sent from the client usually indicate a user performed some action, such as clicking a button or typing a message. When either the client or server receives an event, the appropriate event handler is executed.

JavaMOO provides a small set of general events in the javmoo.event package that can be used for any environment This includes the base Event interface that all JavaMOO events must implement. Other general events include create player, login, and disconnect events.

JavaMOO also provides a set of domain events that can be used in the implementation of environments using the MOO room metaphor and are available in the javamoo.domain.events package. These include entering a room, clicking a tangible object, going home, saying something, and being notified of object changes.

If the existing JavaMOO events do not fit the requirements for a particular environment, the events must be created by the developer. Events should be kept very simple in design, serving mostly as a data container to communicate required information between the client and server. Events should not contain any client or server specific objects so that a clean separation of code can be maintained. All member variables of an

Event must be serializable so that they can be marshaled across the network.

Login Events. All events are sent to the server through a server connection except for the first which must initiate the server connection. This first event is called a login event, and must implement the LoginEvent interface.

LoginEvent is a marker interface meaning that it does not have any methods defined and is only used to mark a class as having special behavior or following an implicit contract. In this case an event marked as a login event is able to be sent to a JavaMOO server without authentication. A successful handling of the login event will result in a ServerConnection and corresponding context to send and receive future events.

JavaMOO provides an implementation of LoginEvent called SimpleLoginEvent that only contains the username and password of the player wishing to connect to the server. A client uses the login handler to send a login event to the server and the server will call the registered event handler. SimpleLoginEventHandler attempts to authenticate the user using the provided username and password by querying the current persistence manager.

Users must somehow be able to create players in the environment. This could be accomplished differently depending on what is appropriate for the particular environment. For example an environment could use a JSP page from the content delivery component and have a user fill out a web form. The result of submitting this web form would be to create a new player with a username and password that can be used for future authentication.

Another possible approach is for the game client to send a special login event that creates a player. JavaMOO provides a login event class called SimpleCreatePlayerEvent to create player objects. The corresponding event handler, SimpleCreatePlayerEventHandler, creates a new player of a configurable type, for example, a DbayPlayer in Dollarbay.

Once a player successfully logs in, a set of attributes are registered on the newly created session. These include the environment objects Player, Environment, and LoginServer. Each of these attributes use the interface class as keys, for instance, Player.class. Other attributes needed to handle events could be registered by custom login handlers.

Event Handlers

Events cause actions to take place on either the client or server. Handlers are objects that modify server or client state based on event data and perform a set of operations to successfully process specific types of events. Event handlers will almost always be tightly coupled to either the server or client because they contain references to client or server specific classes.

Event handlers implement the EventHandler interface that defines the sole method *handle(Event e, Session s).* Handlers should specify the targeted Event classes in the Javadoc for its handle method. This is an implicit contract that the consumer of the class must follow; it is not enforced through type checking using generics or by explicit class checks. If a consumer calls an event handler with an unsupported event class the result is undefined, but likely that a runtime exception such as ClassCastException would be thrown.

The session parameter provides connection specific context for handling events. This allows a handler to understand which session caused the event and determines the set of objects that will be modified. Attributes can be registered on sessions to propagate global or session specific data. For example, the persistence manager and authenticated player are attached to the session object.

Server side event handlers will tend to follow a similar pattern: cast an event to a specific class, get the desired session variables, query for domain objects based on event data, modify the domain objects based on event data, and send result events back to client. Although a very simple logical flow, creating very specific event handlers that follow this pattern encourages small easy to follow code that is loosely coupled and scalable.

Client side event handlers are less apt to follow a set pattern because they are more likely to be generic, handling many different event classes. A typical client event handler will update GUI elements such as images or labels, create new windows, or query the user for information. Any class can be an event handler, which means that a client window frame class could be an event handler to change the content in the window.

Configuration

JavaMOO configuration is very flexible and includes not only simple parameters, but also the ability to choose the implementation of specific components and how they are wired together. This is accomplished through a set of Spring IOC xml files that describe the initial set of objects to create and their properties. The structure of the configuration files is pure convention, the developer can choose to arrange the configuration any way they choose, and initialize any beans they want.

The only requirement is that a bean exists with an id of *mooServer* that is an instance of MOOServer. Figure 26 shows an example MOOServer bean configuration. All other configuration requirements will stem from this single constraint.

```
<bean id="mooServer" class="javamoo.server.MOOServer">
  <constructor-arg><ref bean="loginServer"/></constructor-arg>
  <constructor-arg><ref bean="persistManager"/></constructor-arg>
  <constructor-arg><ref bean="environment"/></constructor-arg>
  <constructor-arg><ref bean="pluginList"/></constructor-arg>
</bean>
```
Figure 26: MOOServer Configuration in javamoo-beans.xml

Persistence. The environment developer has the ability to choose the persistence manager to use as part of the server configuration in javamoo-persist.xml. Each implementation of a persistence manager will have their own set of configuration options. Currently one wrapper class (QueuedPersistManager) and one base class (SQLPersistManager) exist in JavaMOO.

The QueuedPersistManager, as discussed earlier, queues save requests and executes them against a child persistence manager in a FIFO order. There are two configurable parameters, the child persistence manager and the type of queue to use.

In Figure 27 we are creating a new bean named *persistManager* of the class type QueuePersistenceManager. The first constructor argument creates a nameless child bean that is also a persistence manager. The queued manager's second argument specifies that a QueueSet should be used as the underlying queue. Any class which implements the Queue interface can be used as the underlying queue, so that the behavior or performance requirements of the environment can determine the type of queue needed. We see the highly hierarchical structure of object initialization reflected in the XML configuration, which is naturally good at representing hierarchies.

A data source contains the information needed to connect to the database. An example data source configuration is shown in Figure 28. The first property specifies the driver to use for the connections; in this case we are connecting to an embedded Derby database.

The only other property needed to connect to an embedded derby database is the url.

```
<bean id="persistManager" class="javamoo.persist.QueuedPersistManager">
   <constructor-arg>
      <bean class="javamoo.persist.sql.SQLPersistManager">
            <constructor-arg><ref bean="dataSource"/></constructor-arg>
            <constructor-arg><ref bean="sqlDialect"/></constructor-arg>
      </bean>
   </constructor-arg>
   <constructor-arg value="javamoo.util.QueueSet"/>
</bean>
```

Figure 27: PersistManager Configuration in javamoo-persist.xml

```
<bean id="dataSource"
   class="org.springframework.jdbc.datasource.DriverManagerDataSource">
   <property name="driverClassName"
      value="org.apache.derby.jdbc.EmbeddedDriver"/>
   <property name="url" value="jdbc:derby:db;create=true"/>
</bean>
```

Figure 28: Data Source Configuration in javamoo-persist.xml

The next constructor argument for the SQL persistence manager is the dialect. All SQL databases speak the SQL language, but variations and deviations from standards exist, forming different dialects. The dialect parameter factors these differences for common operations that can be customized for each type of database. Since we are connecting to a Derby database, we are using the Derby dialect.

One responsibility of the dialect is defining the mapping between Java data types and the appropriate database data types. This aspect of dialect is completely configurable so that the environment developer can create entirely new data mapping classess. A data mapping describes a Java and SQL data type pair, and the operations required to transform and store the data to a database column and retrieve and transform the data back into the corresponding Java data type.

As an example, Figure 29 shows the configuration of a Derby dialect that only knows how to map strings and dates to the database. Only one constructor argument is needed, which is a map with Java classes as keys and SQLMappings as values. The keys must be valid Java classes when they are translated into a polymorphic map which uses the class inheritance tree for key traversal.

```
<bean id="sqlDialect" class="javamoo.persist.sql.DerbySQLDialect">
  <constructor-arg>
    <map>
      <!-- Add new data mapping definitions here -->
      <entry key="java.lang.String">
        <bean class="javamoo.persist.sql.mapping.StringMapping">
          <constructor-arg value="VARCHAR(30000)"/>
        </bean>
      </entry>
      <entry key="java.util.Date">
        <bean class="javamoo.persist.sql.mapping.DateMapping"/>
      </entry>
    <map>
</bean>
```

Figure 29: SQL Dialect Configuration in javamoo-persist.xml

Default mappings exist in the provided javamoo-persist.xml file but are not shown here for brevity. These include mappings for all of the primitive data types, for example int and double, and their wrapper classes Integer and Double. Also included are all serializable implementations of the Java Collection interfaces: List, Map, Set, and Collection.

Event Handlers. The association between events and their handlers is configurable in the javamoo-events.xml file which exists on both the client and server. The base event handler is called an EventDispatcher, which executes child handlers based on the current type of event. This mapping of child handlers needs to be defined and is similar to the SQL data mapping configuration.

Figure 30 shows a simplified example of the configuration of an event dispatcher. The event dispatcher has a single constructor argument that is a map of event classes to event handlers. The keys of the map must be valid classes that implement the Event interface and the values must be EventHandler classes. This map completely describes the set of events the server (or client) expects and the behaviors that result when receiving these events.

```
<bean id="eventDispatcher"class="javamoo.event.handler.EventDispatcher">
  <constructor-arg>
    <map>
      <entry key="javamoo.event.SimpleLoginEvent">
        <bean class="javamoo.event.handler.SimpleLoginEventHandler"/>
      </entry>
      <entry key="javamoo.domain.event.ClickEvent">
        <bean class="javamoo.domain.event.handler.ClickEventHandler"/>
      </entry>
      <entry key="javamoo.event.DisconnectEvent">
        <bean class="javamoo.event.handler.DisconnectEventHandler"/>
      </entry>
    </map>
  </constructor-arg>
</bean>
```

Figure 30: EventDispatcher Configuration in javamoo-events.xml

Plugins. The plugin configuration file, javamoo-plugins.xml, contains a list of plugin components that have lifecycle management at the application level. Any initialization dependencies between plugins must be manually determined by the environment developer, but such dependencies are likely to be rare.

The named bean *pluginList* is created in Figure 31 with beans for tomcat web server, agent controller, and shopper controller.

```
<bean id="pluginList" class="java.util.LinkedList">
  <constructor-arg>
    <list>
      <bean class="javamoo.plugin.web.TomcatPlugin">
        <constructor-arg><ref  ean="persistManager"/></constructor-arg>
        <!---<constructor-arg value="8081"/>-->
      </bean>
      <ref bean="agentController" />
      <ref bean="shopperController" />
    </list>
  </constructor-arg>
</bean>
```

Figure 31: Dollarbay Plugin Configuration in javamoo-plugins.xml

These beans are implemented as plugins which will be started and stopped as the MOOserver is started and stopped. They are started in the order specified before the login server but after the persistence manager and stopped in the reverse order. Any component that needs to be notified or manage state at server startup and shutdown can be registered as a plugin.

Content Delivery

Clients may require access to images, sounds, three dimensional models, or web pages. To efficiently store and update this information a content server is typically used.

Web-based. Content such as images and multimedia can be stored on a file server for retrieval by the client. This enables content to be updated without the need to update the client software. As soon as new content is available, it can be uploaded to the webserver.

Other content like web pages may be dynamically generated using environment domain objects and user data. These types of pages or web applications are typically served by an application server such as Websphere, Tomcat, or Spring. JavaMOO bundles the Tomcat application server as a plugin that can be used to serve this type of content through JSP pages and the http protocol.

The Tomcat configuration, as shown in Figure 31, has two constructor arguments. The first argument is a persistence manager bean that allows JSP scripts to query for domain objects. The second optional constructor argument, serverPort, determines which network port to start the Tomcat server on. If the port is not specified the default port 8080 is used.

Content in the webroot server directory will be available through the Tomcat server. The location of webroot is also configurable through the configuration property contentRootPath. All files placed into this directory hierarchy are publicly accessible through a web browser. JSP scripts will be compiled on the first access attempt.

Bundled content. Another approach to distributing content is to bundle it with the client. This has a number of advantages including perceived improved game play and reduced network requirements. It also has drawbacks such as larger client distribution size and difficultly updating content.

JavaMOO includes support for a resource directory on the client. Any files placed into the resource directory will exist in the virtual machine classpath and can be retrieved through calls to the method Class.getResource. The Dollarbay images files are bundled with the client in this manner to reduce the load times of rooms.

Build

JavaMOO uses Apache Ant for its build process. Ant provides a syntax for describing steps needed for different targets such as compiling, building jars, and building distribution files. If an action has already occurred during a previous build, Ant will detect this and skip to the next action. For example, if ten Java files were compiled and then one was modified, only the modified file would need to be recompiled.

In both the JavaMOO and environment projects the build file is named build.xml and exists in the project root directory. Five targets exist in the provided build configuration: init, clean, compile, jar, and dist. Init creates the build and dist directories which are required for the build. Clean removes any files and directories generated from a previous build. Compile creates class files for all Java files found in the client, server and common source directories. The jar target generates jar files using the compiled class files. Finally, dist will create a complete directory structure in the dist folder, copy the required files including the generated jars, and build a zip file that can be distributed.

The build can be started in two ways. Typically Ant is embedded into IDEs such as Eclipse. To execute the build in Eclipse simply right click on build.xml and select *Run As->Ant Build.* This will initiate the ant process with the output displaying in the console panel. Ant can also be used manually to execute the build. Once Ant is installed a build can be started by executing the ant command from the project's root directory (Apache Ant, 2010).

Installation

An environment will need to create an initial set of domain objects. JavaMOO has a defined installation process to perform such initializations. When the install comand line argument is used to start the server, the install method for the Environment implementation is invoked. This method is responsible for initializing all of the objects which must exist for the environment.

Specifying the initial objects to create poses a problem. This could be solved by hard coding all of the object initialization and breaking down each object into its constituent parts and dependencies. This approach would not be very flexible and requires recompilation after every change. A different strategy was taken in the first Dollarbay prototype. A flat dump file contained all of the initial object state in a special format, but this was difficult to modify and parse. It was especially difficult when new fields were added to domain objects.

We solved the problem in JavaMOO Dollarbay by using the same IOC mechanism used for configuration. When Dollarbay's install method is executed, a new Spring context is created from a special IOC configuration file containing entries for all of the domain objects that should be created during installation. This approach proved to be very easy to implement, as shown in Figure 32.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
      <import resource="dollarbay-products.xml"/>
      <import resource="dollarbay-shoppers.xml"/>
      <import resource="dollarbay-hoods.xml"/>
      <import resource="dollarbay-warehouses.xml"/>
      <import resource="dollarbay-admedia.xml"/>

<!-- Atmosphere Agents -->
      <bean id='Art_Salesman'
class='dollarbay.agent.atmosphere.ArtSalesman'>
            <constructor-arg ref='persistManager'/>
            <constructor-arg value='Art Salesperson'/>
            <property name='imageFile' value='/images/titania.gif'/>
            <property name='clickable' value='true'/>
      </bean>
...
</beans>
```

```
@Override
public void install(ApplicationContext parentContext, String[] args) {
      ApplicationContext appContext = new
      ClassPathXmlApplicationContext(
            new String[] { "dollarbay-install.xml" }, parentContext);
}
```

Figure 32: Fragment of dollarbay-install.xml and the Dollarbay Install Method

Using IOC enables the developer to specify additional object creation without requiring a recompilation, only a reinstall. Creating a new ClassPathXmlApplicationContext will read the xml install file and create all of the beans listed. Since the bean types are persistent domain objects, the beans will be saved to the database so that the installation only needs to be ran once.

The previously mentioned Dollarbay prototype dump file was automatically converted into xml install files by a custom utility class, dollarbay.util.DumpFileTransform. This allowed us to quickly create the files necessary for our new installation strategy. For a sense of scale, the dollarbay-products.xml installation file is 2.8 MB and the installation typically takes less than a two minutes.

EVALUATION

Test results and analysis show how successful we have been in achieving our goals of performance, functionality and ease of use. The implementation of JavaMOO successfully fulfills the goals of this thesis but with room for future improvements, work and research.

Performance

We've written test utilities for testing JavaMOO and LambdaMOO performance. The tests measure the time of each operation to create and delete 1000 simple objects. Four machines were used for the test setup, as shown in Figure 33, to reduce the chance of a hardware bottleneck.

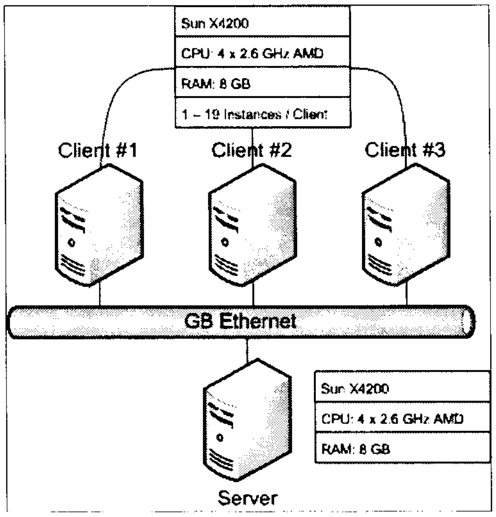


Figure 33: Test Hardware Setup

Multiple instances of the test are ran on each client machine starting with one instance. The number of instances increment by two until a maximum of 19 instances are running per machine. For each level we run the test is executed three times to help eliminate bad data. The test executions result in 60 data sets of performance data.

The tests measure how much time each operation takes end to end, whether create or delete. The results provide an estimate of how much work a server can handle as the number of connected clients increase. Figure 34 shows the performance of create operations for both JavaMOO and LambdaMOO.

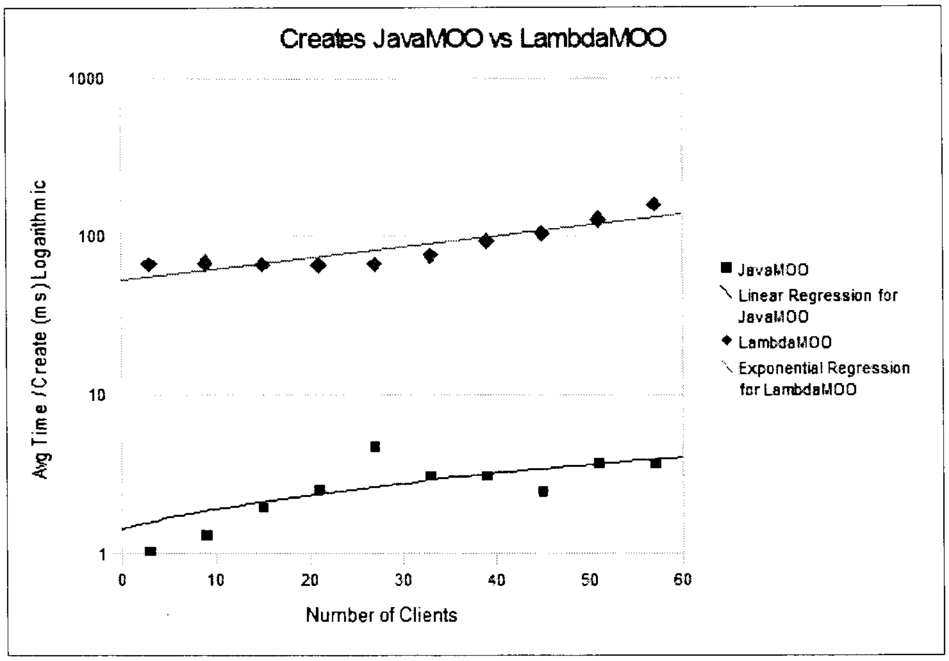


Figure 34: Create Performance

The average time taken per create event is better with JavaMOO than LambdaMOO. There are a few reasons for this. In JavaMOO events are batched together so that multiple events are sent as one message across the network. This will cause a delay in sending a message to the server, but will reduce the amount of network overhead overall. Although it took JavaMOO 120 ms versus 60 ms for LambdaMOO to respond to the first create request, JavaMOO batched one hundred events together. By the time LambdaMOO had created two objects, JavaMOO had created one hundred.

As the number of clients increase we notice that JavaMOO response time increases linearly and that LambdaMOO becomes more and more unresponsive. The graph shows that LambdaMOO tests starts to degrade at about 30 clients. This is most likely due to

LambdaMOO being single threaded.

Figure 35 shows the comparison of delete performance. JavaMOO delete performance is database bound. To process a delete event, a database statement must be executed across multiple tables. This is a synchronous event, so the event is not successfully processed until the database delete is finished.

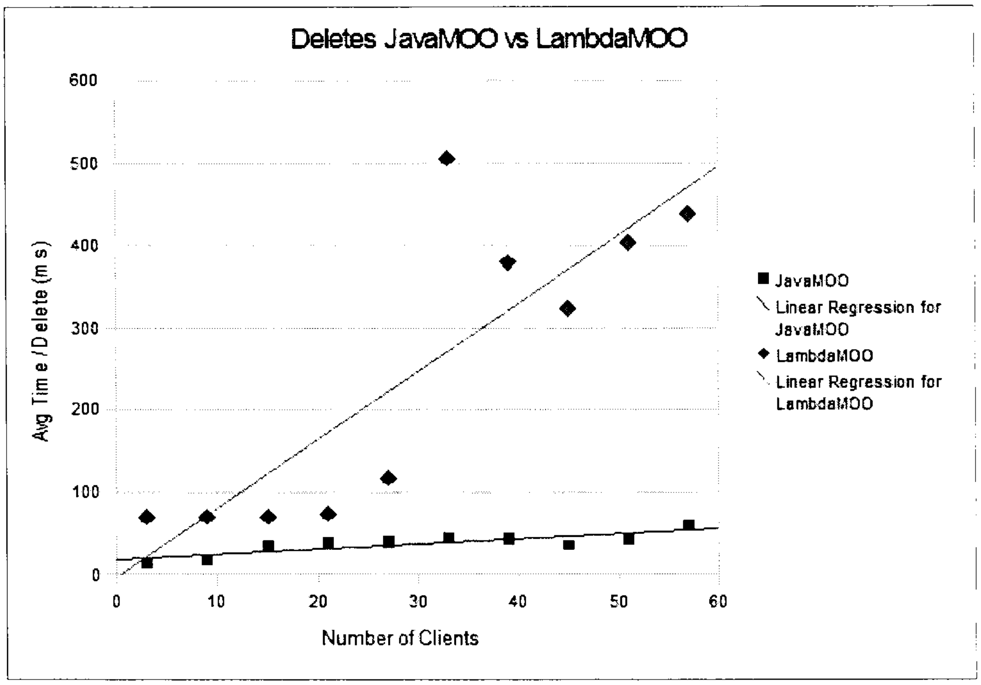


Figure 35: Delete Performance

An interesting behavior of LambdaMOO was observed as part of the delete testing. When the number of clients increased beyond twenty, multiple "unable to fork" messages appear. Examination of the MOO code and online research found recycle is performed as a background task. This means LambdaMOO returns immediately after scheduling the background task without waiting for the recycle to complete.

The LambdaMOO error messages were a result of hundreds of recycle tasks backing up until the global background tasks threshold was reached. After setting the background_tasks option to infinity we no longer experienced this error message. An unfortunate side-effect of recycling this many objects in LambdaMOO is excessive CPU consumption. Long after the tests had completed, one CPU was utilized 100% in effort to catch up with the thousands of recycle background tasks.

The performance comparison of JavaMOO to LambdaMOO is best illustrated using total time taken for test completion, shown in Figure 36. The LambdaMOO time do not include the extra time needed to recycle objects after the test program had finished. The graph uses a logarithmic scale and shows that JavaMOO finishes the tests an order of magnitude faster than LambdaMOO. This is the result of multi-threading and batching requests across the network. For 50 clients, JavaMOO took an average of ten seconds to finish whereas LambdaMOO took over seven minutes.

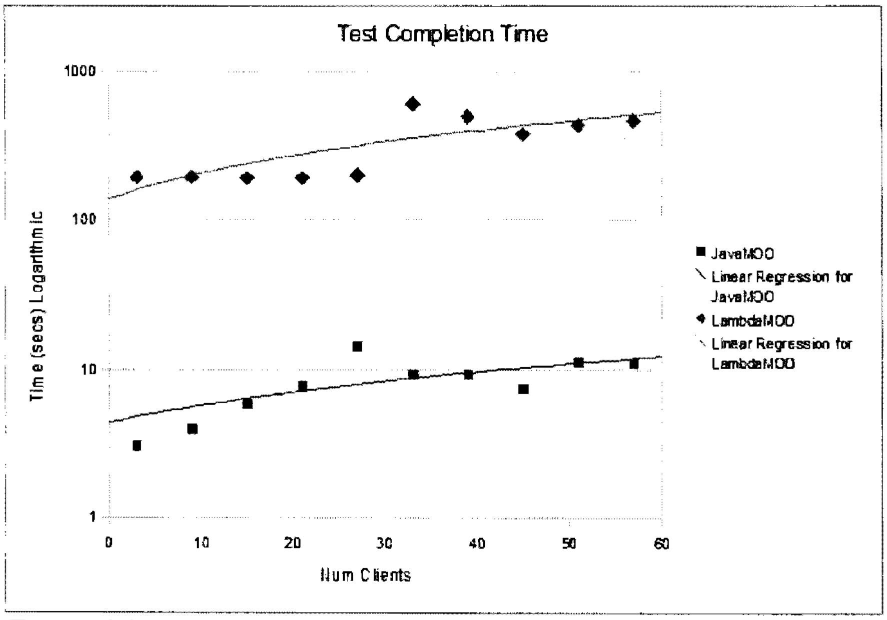


Figure 36: Test Time to Completion

Finally, LambdaMOO experienced errors when the number of clients increased as shown in Figure 37. These errors were related to the CPU problems discussed earlier with recycling.

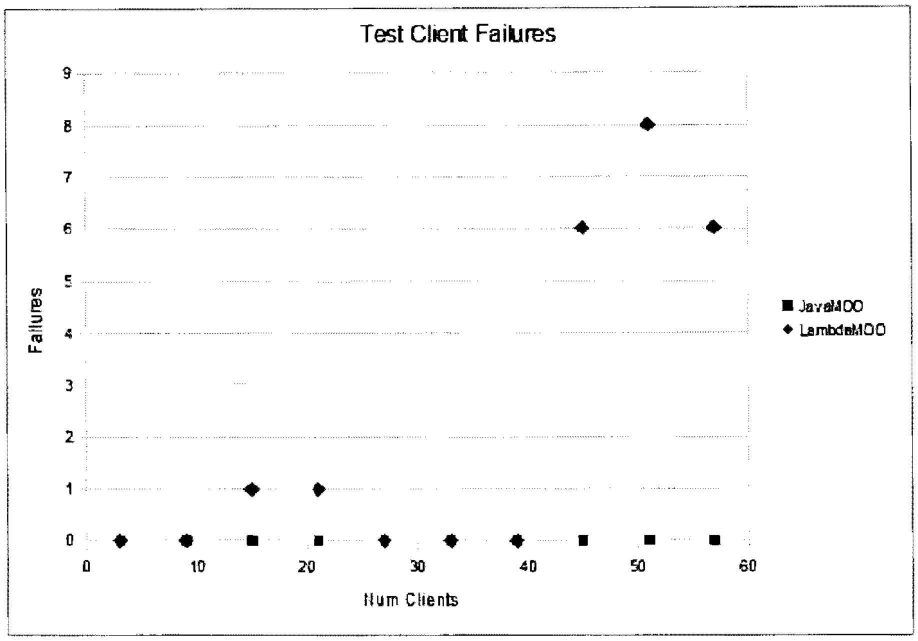


Figure 37: Performance Test Failures

The problem became more severe around 45 clients, where some tests froze waiting for a response from the recycle command. After waiting for several minutes the frozen tests were killed. JavaMOO did not experience errors or CPU load problems during testing.

Development Process

The JavaMOO development process is a change from the real time prototyping environment of LambdaMOO, but brings more powerful tools to the hands of developers. Although it is no longer possible to modify the logic of a running environment, this does not pose a barrier to the design of these environments.

The bulk of environment programming does not require real time modification, and the ability to easily run unit tests and modern debugging tools fills the gap. The usage of an integrated development environment such as Eclipse, along with well-defined programming interfaces, help to simplify the the development process. The usage of inversion of control configuration allows the developer to specify dependencies and objects without the need to recompile. The end result boils environment design down to defining domain objects, events, and handlers.

A major benefit for developers is the unification of the client and server programming languages. The server code now has access to many programming libraries not previously available, and can communicate with the clients using a native object serialization protocol that does not require a difficult to program text-to-object translation parser. Also, the Java programming language uses strong data typing that helps catch programming mistakes.

Another benefit of JavaMOO is the ability to choose the configuration, including the implementation of system facilities such as databases or network communication. These changes are not easily done in LambdaMOO.

Lastly, the debugging support available is very useful to developers. This includes real time debugging on a server instance running from within an IDE supporting breakpoints, and profiling tools to determine performance bottlenecks. The logging support has been greatly improved, with the ability to choose different logging levels between debug and fatal errors.

Distribution

Environment distribution using JavaMOO is simplified through a unified approach with external dependencies eliminated. The server can be installed through a single installation process containing an integrated environment, with a web and database server that is platform independent.

Legacy environments rely on LambdaMOO, which is possible to run on a Windows or Macintosh based machine but require extra software to be installed such as a Unix shell and compiler. Most schools using these environments may not readily have access to a Unix machine. Often a lab of windows based personal computers is used, one of which selected as the server. Since JavaMOO is able to be run on any platform supporting Java, a standard lab machine would work well as server for a class of 30 students.

Environments may require supporting material that is delivered using web pages and scripts including registration, supplemental materials, or help pages. This content requires some form of web server in order to be distributed. The legacy model was to host these materials on a centralized web server that would be accessed by all users of an environment across the world. For types of content where it is beneficial for data be gathered centrally, this model works well. Otherwise, being able to access material from the environment server will scale as the number of installations increases.

FUTURE WORK

Although we have provided a complete framework to develop and deliver virtual environments, there is still much room for future research and improvement. A few interesting projects for future JavaMOO development are presented.

Transactional Cache with Deadlock Detection

A current limitation of the JavaMOO persistence model is the lack of a object transaction interface. An object transaction allows a set of objects to be added and modified atomically with respect to the environment. The set of object modifications can be either entirely committed or rolled back to the central object store.

With the current persistence manager, if two events simultaneously modify the same object, the programmer must use object synchronization in order to ensure thread safety. This does not ensure that concurrent modifications performed across a set of objects occur in a serial fashion so that the results of the modification are predicable. The programmer must consider thread safety as part of the environment design, and introduces complexity to the programming model.

Error recovery is also more complex without object transactions. Suppose an event modifies a set of objects so that the event handler successfully modifies ten objects, but on the next object an exception occurs. Without an object transaction there is no easy way for the event handler to rollback the modifications for the previous ten objects.

It is desirable to extend the persistence manager to allow the concept of object transactions. Figure 38 shows example usage of object transactions. The persistence manager is asked to create a new transaction which can be used to perform all of the persistence operations. All players named Sam are queried and told to go home and the transaction is either committed, resulting in all of the players named Sam going home, or rolled back so that no changes were made to any player named Sam.

```
public void handleEvent(Event e, Session s) {
	PersistManager mgr = s.getPersistManager();
	Transaction t = mgr.newTransaction();
	try {
		/* Tell all player's named Sam to go home */
		Query<Player> query = t.newQuery(Player.class)
			.clause("name", Query.EQ, "Sam");
		for (PersistRef<Player> ref : t.getReferences(query)) {
			Player p = ref.get();
			p.goHome();
		}
		t.commit();
	} catch (Exception e) {
		/* Rollback if any error */
		t.rollback();
	}
}
```

Figure 38: Object Transaction Example

In order for such an interface to function properly, changes made to any objects obtained as part of a transaction must be visible to only that transaction until committed. This means that a separate copy of an object must be maintained.

One strategy for implementing transactions would be to use a form of copy-on-write semantics. No copy is done until the object is modified, then a copy is made. This would help reduce the amount of memory used in the transaction system, and improve performance by eliminating costly copy operations.

Performance optimizations for the underlying persistence storage could result from a transaction interface. In the case of an SQL database, all of the queries needed to execute for a transaction could be batched together to increase performance. Objects of similar types could also be grouped together to reduce the number of round trips to the database.

The benefits of a transaction system include a simplified multi-threaded programming model and the ability to atomically commit or rollback changes to a set of objects. This allows event handlers to be written more robustly in the face of errors.

Drawbacks include potential poor performance resulting from reduced concurrency and the possibility of object deadlocks. The same object must not be modified concurrently by two threads, specifically until one of the transactions is finished by committing or rolling back. The result could be a deadlock if two threads each require access to the same object.

The deadlock situation must be accounted for in any implementation of transactions. In the case of an object deadlock, a PersistException could be thrown, and victim thread could rollback the transaction and retry the operations. For an event handler, this would simply be the re-execution of the handleEvent method.

Scalability Research

Currently the JavaMOO server resides on a single machine. This means that there is no way to scale JavaMOO horizontally if an environment grows larger than what the hardware of a single machine can achieve. Research must be done to apply the scaling concepts from the MMOG community discussed in the literature review.

One possible scaling strategy is to distribute rooms or geographies across multiple JavaMOO servers. The client would be programmed to connect to each server either simultaneously, or in an as-needed basis. As the environment grew larger, more servers could be added for the new rooms and players.

Since JavaMOO is entirely embedded, both the server and the client code can be ran from within the same VM. This means a client could also act as a server, enabling a peer-to-peer type of environment. A second-life like environment could be developed where users connect to each other's servers to explore the world-wide distributed environment. This would allow the environment to scale as the number of clients grew.

Security Improvements

JavaMOO does not currently use an encrypted channel of communication between the client and server. A custom SSL socket factory could be developed for the RMI connection to improve security.

No authorization check occurs during event processing to ensure that the event actually came from the expected client. This means that connections could potentially be hijacked by a malicious third party to execute a man in the middle attack. Finally, the password mechanism should use some form of encryption such as SHA or MD5 to help ensure that user privacy is not violated.

LamdaMOO Emulator

The first attempted approach for JavaMOO was an automated translation of LambdaMOO code to Java. This proved problematic because of inherent incompatibilities between the two languages. As part of writing the current implementation of JavaMOO, another potential approach to migrate LambdaMOO environments was realized: execute LambdaMOO environments using an emulator running inside of JavaMOO.

The idea is to create a language interpretor as a script engine able to understand MOO verb code. As the code is interpreted, Java objects are looked up and modified.

A new persistent Java class called Lambda would contain properties and verbs in dynamic maps. An instance of this class would be created for each LambdaMOO object, and persisted to the database. Figure 39 shows an possible sequence of events for event processing in the LambdaMOO emulator.

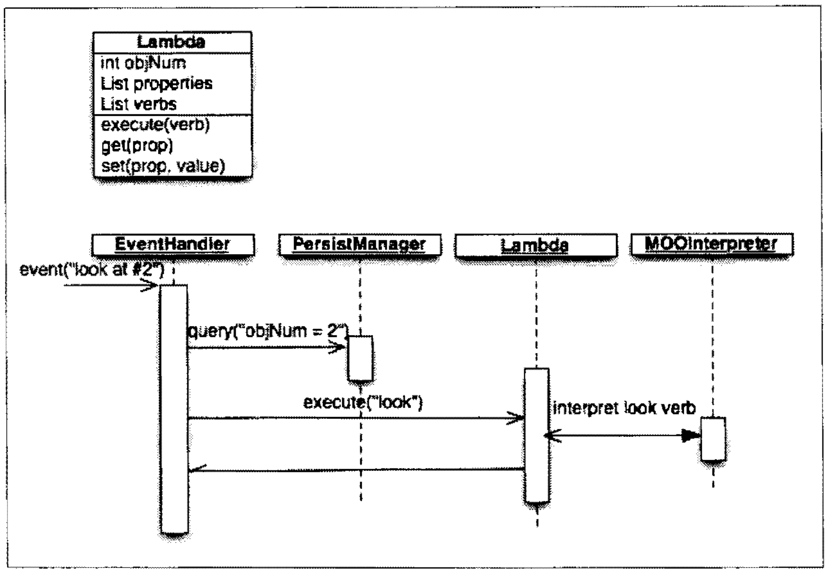


Figure 39: LambdaMOO Emulator Event Sequence

The LambdaMOO database is a text file which contains all of the objects and verbs that exist for the environment. The installation process for the emulator would process this text file to create the corresponding Lambda objects, populating them with the appropriate properties and verbs. Legacy environments could be installed using Java with minimal effort, providing many benefits such as real-time persistence, event-based client communication, multi-threading, and improved performance.

CONCLUSION

This thesis has presented an architecture named JavaMOO to help developers create multiuser virtual environments focusing on domain-specific design and rapid development. Best practices for developing environments were shared and an extensible design for system configuration, client-server communication, event handling, object persistence, content delivery, and agent control was presented. JavaMOO is a stand-alone application which can be easily deployed to remote sites without external dependencies and improves the current state of virtual environments.

It has been shown that existing environments can be implemented with JavaMOO using a client-server communication model customizable to each environment. A flexible persistence mechanism delivers per-object granularity to avoid complete memory residency of objects. Evaluation has shown acceptable performance of JavaMOO using a multi-threaded architecture. For the above reasons, the goals of this thesis have been met.

REFERENCES

Ambler, S. The Realities of Mapping Objects To Relational Databases. Software Development. 5, 10 October 1997, 71-74.

Apache Ant, Installing Ant, Retrieved March 2010, from http://ant.apache.org/manual/install.html.

Apache Launcher, Documentation, Retrieved March 2009, from http://commons.apache.org/launcher/.

Assiotis, M., & V. Tzanov, A Distributed Architecture for MMORPG, Netgames'06 ACM, October 2006.

Atkinson, M., Persistence and Java - a Balancing Act, Objects and Databases Lecture Notes in Computer Science, June 2000, 1-31, Springer-Verlag.

Avram, A. Domain-Driven Design Quickly. 2007. Lulu.com.

Bartle, R., Early MUD History, rec.games.mud newsgroup post, November 1990. Retrieved March 31, 2010, from http://www.mud.co.uk/richard/mudhist.htm.

Bartle, R., Interactive Multi-User Computer Games, Technical report, BT Martlesham Research Laboratories, December 1990. 28. Retrieved March 31, 2010, from http://www.mud.co.uk/richard/imucg.htm.

Beaumont, O., A.M. Kermarrec & E. Rivire, Peer to peer multidimensional overlays: Approximating complex structures. *In OPODIS, 11th International conference on principles of distributed systems*, 2007.

Beck, K., Test Driven Development: By Example, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 2002.

Big World, Press Release – Big World Education, Retrieved March 2008, from http://www.bigworldtech.com/news/press_080724.php.

BigIP, White Paper Load Balancing 101, Retrieved March 2008, from http://www.f5.com/pdf/white-papers/load-balancing101-wp.pdf.

Blizzard, World of Warcraft Realm Status. Retrieved October 2008, from http://www.worldofwarcraft.com/realmstatus/.

Bonk, C. J. & V.P. Dennen, Massive Multiplayer Online Gaming: A Research Framework for Military Training and Education. Technical Report 2005-1, U.S. Department of Defense, 2005.

Brandt, L., O. Borchert, K. Addicott, B. Cosmano, J. Hawley, G. Hokanson, D. Reetz, B. Saini-Eidukat, D.P. Schwert, B.M. Slator, & S. Tomac, Roles, Culture, and Computer Supported Collaborative Work on Planet Oit., Journal of Advanced Technology for Learning, 2006, 3(2), 89-98.

Brunskill, E. 2001. Building Peer-to-Peer Systems with Chord, a Distributed Lookup Service. *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems* (May 20 - 22, 2001). IEEE Computer Society, Washington, DC, 81.

Chan, L., J. Yong, J. Bai, B. Leong, & R. Tan, 2007. Hydra: a massively-multiplayer peer-to-peer architecture for the game developer. In *Proceedings of the 6th ACM SIGCOMM Workshop on Network and System Support For Games*(Melbourne, Australia, September 19 - 20, 2007). NetGames '07. ACM, New York, NY, 37-42.

Curtis, P., Mudding: Social Phenomena in Text-Based Virtual Realities, Proceedings of the 1992 Conference on the Directions and Implications of Advanced Computing, 1992, Berkeley, CA.

Curtis, P., & D. A. Nichols, MUDs Grow Up: Social Virtual Reality in the Real World, COMPCON, 1994, 193-200.

Curtis, P., LambdaMOO Programmer's Manual, 1997, Retrieved March 31, 2010, from http://mirrors.ccs.neu.edu/MOO/ProgrammersManual.txt.

Curtis, P., Not Just a Game: How LambdaMOO Came to Exist and What It Did to Get Back at Me, High Wired: On the Design, Use, and Theory of Educational MOOs, (Haynes & J. R. Holmevik Eds.), 1998, Ann Arbor, MI: Univ. of Michigan Press.

Franklin, S. & A. Graesser, Is it an Agent, or Just a Program?: A Taxonomy for Autonomous Agents, Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages, Springer-Verlag, 1996.

Frey, D., J. Royan, R. Piegay, A.M. Kermarrec, E. Anceaume, & F. Le Fessant , "Solipsis: A Decentralized Architecture for Virtual Environments". In Proc. of International Workshop on Massively Multiuser Virtual Environments (MMVE), 29-33, March 2008.

Hampel, T., T. Bopp, & R. Hinn, 2006. A peer-to-peer architecture for massive multiplayer online games. *Proceedings of 5th ACM SIGCOMM Workshop on Network and System Support For Games*. NetGames '06. ACM, New York, NY, 48.

Kent, S. "Alternate Reality: The History of Massively Multiplayer Online Games" GameSpy Magazine. Septemter 23, 2003.

Knutsson, B. et. al. Peer-to-peer support for massively multiplayer games. In Proceedings of INFOCOM, 2004.

Lu, F., S. Parkin, G. Morgan, Load Balancing for Massively Multiplayer Online Games, Proceedings of 5th ACM SIGCOMM workshop on Network and System Support for Games, October 30-31, 2006, Singapore

Gamma, E., R. Helm, R. Johnson, & J. Vlissides, 1995 *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Longman Publishing Co., Inc.

Mack, J., B. Slator, K. Boulais, V. Doan, & students of CS345, Learning by earning: The Dollar Bay Project., *Proceedings of the 36th Midwest Instructional Computing Symposium* (MICS-03), April 2003, 27.

Mortensen, T., "WoW Is the New MUD: Social Gaming From Text to Video" Games and Culture 1 (4), 397-413, 2006

NDSU Archeology Technologies Laboratory, Virtual Archaeologist Website, Retrieved March, 2008, from http://fishhook.cs.ndsu.nodak.edu/home/.

Open Simulator, Documentation, Retrieved February 2009, from http://opensimulator.org/wiki/OpenSim:Introduction_and_Definitions.

Ratnasamy, S., P. Francis, M. Handley, R. Karp, & S. Schenker, 2001. A scalable content-addressable network. *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols For Computer Communications* (San Diego, California, United States). SIGCOMM '01. ACM, New York, NY, 161-172.

Regan, P. & B. Slator, Case-based Tutoring in Virtual Education Environments, ACM Collaborative Virtual Environments, October 2002, Bonn, Germany.

Rowstron, A. I. & P. Druschel, 2001. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. *Proceedings of the IFIP/ACM* *international Conference on Distributed Systems Platforms Heidelberg* (November 12 - 16, 2001).

Second Life, Documentation, Retrieved March 2008, from http://wiki.secondlife.com/wiki/.

Slator, B. & R. Hooker, A Model of Consumer Decision Making for a Mud Based Game, Proceedings of the Simulation-Based Learning Technology, Workshop at the Third International Conference on Intelligent Tutoring Systems (ITS'96), Montral, 1996

Slator, B. & H. Chaput, Learning by Learning Roles: A Virtual Role-Playing Environment for Tutoring, Proceedings of the Third International Conference on Intelligent Tutoring Systems (ITS'96), Springer-Verlag, June 1996, pp. 668-676 (Lecture Notes in Computer Science, edited by Frasson, C., Gauthier, G., Lesgold, A.).

Slator, B. M., O. Borchert, A. Bergstrom, J. Hockemeyer, J. Clark, P. Juell, P. McClean, B. Saini-Eidukat, D. P. Schwert, A. R. White, C. Hill, J. Bauer, F. Larson, B. Vender, B. Bandli, B. Chen, M. Dean, R. Frovarp, G. Hokanson, C. Johnson, J. Kittleson, N. Kruger, J. Landrum, M. Li, B. Nichols, J. Opgrande, R. Potter, P. Regan, L. Ong Teo, A. Tokhi, S. Tomac, J. Turnbull, J. Willenbring, Q. Xioo, X. Ye, and M. Zuro, Recent Advances in Immersive Virtual Worlds For Education, Proceedings of the 34th Annual Midwest Instruction and Computing Symposium (MICS-01), April 2001

Slator B.M., J. Alt, J. Aus, D. Balliet, D. Balliet, C. Bergstrom, R. Blaha, K. Bopp, B.Carlson, S. Carlson, G. Collins III, P. Crary, J. Cusey, M. Deck, A. Dewald, S. Dieken, A. Elezovic, D. Ely, G. Engels, M. Ernst, K. Fimreite, E. Finke, C.

Fredrickson, N. Fredrickson, M. Guerard, T. Hall, M. Hanson, K. Hartman, W. Hawkinson, K. Hessinger, H. Ho, J. Hoert, C. Ho, B. Hokanson, M. Holzer, M. Hoque, S. Hossain, M. Hurlburt, B. Johnson, S. Kawamura, J. Levasseur, N. Lindvall, B. Lorentz, J. Louwagie, D. Mafua, R. Martens, J. Matthews, B. Miller, S. Moorhouse, D. Olson, K. Parisien, J. Reiser, C. Resler, J. Richardson, C. Romberg, S. Schilke, J. Schmidt, D. Schott, S. Seira, R. Sell, B. Seymour, L. Sjoblom, J. Tarnowski, S. Ternes, B. Thompson, T. Wells, M. Wolters, A. Wong, Rushing Headlong into the Past: the Blackwood Simulation, Proceedings of the Fifth IASTED International Conference on Internet and Multimedia Systems and Applications(IMSA 2001), August 2001, pp. 318-323

Slator, B.M., H. Chaput, R. Cosmano, B. Dischinger, C. Imdieke & B. Vender, 2006, A Multi-User Desktop Virtual Environment for Teaching Shop-Keeping to Children, Virtual Reality Journal, 9, pp. 49-56. Springer-Verlag.

Slator, B., et. al, From Dungeons to Classrooms: The Evolution of MUDs as Learning Environments, Studies in Computational Intelligence, Volume 62, Springer-Verlag Berlin, Heidelberg, 2007

Squire, K., Video Games in Education, International Journal of Intelligent Simulations and Gaming, 2003, 2(1).

Sun Microsystems, Press Release, Leap into Second Life, Retrieved March 2009, from http://www.sun.com/smi/Press/sunflash/2006-10/sunflash.20061010.2.xml

Spring Framework, Version 3.0 Reference Documentation, Retrieved March 31, 2010, from http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/pdf/spring-framework-reference.pdf

Taivalsaari, A., On the notion of inheritence, ACM Computing Surveys (CSUR), September 1996, 28 (3), pp. 438-497

Tanenbaum, A. S., & M. Van Steen, Distributed Systems, Prentice Hall, Upper Saddle River, NJ, 2002.

Vivendi Games, About Vivendi Games, Investor Guide, February 2007, Retrieved February 2008, from http://www.vivendi.com/ir/download/ pdf/Vivendi_Games_Overview_2_1_07.

Wikipedia, MOO Wikipedia Webpage, Retrieved February 2008, from http://en.wikipedia.org/wiki/MOO.

Wiley, B. Distributed Hash Tables Part 1, Linux Journal, October 2007, Retrieved March 2009, from http://www.linuxjournal.com/article/6797.

Woodcock, B. S. "An Analysis of MMOG Subscription Growth" MMOGCHART.COM 12.0. 29 . April 2008. Retrieved October 2008, from http://www.mmogchart.com/Chart4.html

Yamamoto, S., Y. Murata, K. Yasumoto, & M. Ito, 2005. A distributed event delivery method with load balancing for MMORPG. In *Proceedings of 4th ACM SIGCOMM Workshop on Network and System Support For Games* (Hawthorne, NY, October 10-11, 2005). NetGames '05. ACM, New York, NY, 1-8.

Yu, A. & S.T. Vuong, 2005. MOPAR: a mobile peer-to-peer overlay architecture for interest management of massively multiplayer online games. *Proceedings of the International Workshop on Network and Operating Systems Support For Digital*

Audio and Video (Stevenson, Washington, USA, June 13 - 14, 2005). NOSSDAV '05. ACM, New York, NY, 99-104.

Zhao, B. Y., J. D. Kubiatowicz, & A. D. Joseph, 2002. Tapestry: a fault-tolerant wide-area application infrastructure. *SIGCOMM Rev.* 32, 1, 81.