5-11-2006

Simulator for Undergraduate Multi-Agent Systems

Charles Smith
Trinity University

Distributed Simulator for Multiagent Systems

Charles Keller Smith

A departmental honors thesis submitted to the Department of Computer Science at Trinity University in partial fulfillment of the requirements for Graduation with departmental honors.

April, 18 2006

Thesis Advisor

Department Chair

Associate Vice President for Academic Affairs,
Curriculum and Student Issues

Simulator for Undergraduate Multi-Agent Systems

Charles Keller Smith

Abstract

In recent years, Multi-Agent Systems (MAS) have for the first time begun to be accepted in mainstream computing. Software companies have been founded focusing purely on MAS software[1], telecommunications companies now use agent-based technologies in cell phones[2], and there have even been two successful DARPA funded, military-grade defense projects in the past ten years[3]. The growth in demand for MAS has spurred a dramatic increase in the number of different MAS development tools available. The various development platforms focus on mobile devices, large-scale distributed systems, and specific research applications; however, these tools leave an important facet of MAS development unsatisfied – undergraduate research and teaching. Each of the solutions available is either too complex, too specific, or in some way infeasible to be used by students in what is possibly their first introduction to MAS.

This research concentrates on creating a distributed, graphical MAS simulator in Java and an associated Application Program Interface (API) for developing agent-based systems at the undergraduate level. Whether in research or in the classroom, the well designed, easily extensible API allows students to create and immediately display their agents' interactions in the simulation environment with minimal programming. The API provides agents with the capacity for perception, communication, memory, and action.

Future undergraduate research and learning in the field of MAS will be greatly facilitated by this intuitive simulation platform. Students can learn MAS by observing agents visually, and student researchers can focus purely on programming and analyzing agent behavior.

ACKNOWLEDGMENTS

I would like to express my gratitude to Dr. Mark Lewis for his valuable advice, direction, and encouragement. I would also like to thank Andrew Krausnick and Dr. Zhang for their important contributions to the design and use of this simulator.

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

1.1. MOTIVATION

In recent years, Multi-Agent Systems (MAS) have for the first time begun to be accepted in mainstream computing. Software companies have been founded focusing purely on MAS software, telecommunications companies now use agent-based technologies in cell phones, and there have even been two successful DARPA funded, military grade defense projects in the past ten years. The growth in demand for MAS has spurred a dramatic increase in the number of different MAS development tools available. Some tools focus on mobile devices, some on large-scale distributed systems, and still others focus on complex features for specific research applications. However, these tools leave an important aspect of MAS development unsatisfied – undergraduate research and teaching. Each of the solutions available is either too complex, too specific, or in some way infeasible to be used by students in what is possibly their first introduction to MAS.

This research concentrates on creating an MAS simulator and associated API for developing agent-based systems at the undergraduate level. Whether in research or in the classroom, undergraduate students should be able to quickly understand this platform and easily use it to develop MAS simulations.

1.2. BACKGROUND

This section will give a broad overview of the field of MAS, beginning with an explanation of its place in larger world of Artificial Intelligence and narrowing down to the subfield of MAS simulation.

1.2.1. Artificial Intelligence

Artificial Intelligence (AI) can be seen as having roughly two schools of thought: Conventional AI and Computational Intelligence. Conventional AI typically uses explicit logical rules to draw conclusions from available data; examples include expert systems, case based reasoning, Bayesian networks, and behavior based AI. Computational Intelligence aims to use learning, adaptive, or evolutionary computation to solve problems intelligently. Unlike Conventional AI, computational systems do not use symbolic logic but instead involve iterative developmental learning. Often Computational Intelligence based systems will have to cope with problems where the environment is not entirely known, for example, a robot finding its way through an unfamiliar maze. Such systems include Neural Networks, Fuzzy systems, Evolutionary computation, Multi-Agent Systems, and Swarm Intelligence.[4] This paper focuses on a simulator for the subfields of Multi-Agent Systems and Swarm Intelligence.

1.2.2. Multi-Agent Systems and Swarm Intelligence

A Multi-Agent System (MAS) can be defined as a system composed of several agents, each with goals that might be difficult for an individual system to achieve.[5] The term "agent" has been used with increasing frequency in the past few years, both within AI and other disciplines. One might be as likely to hear the term "agent" in a discussion of economics or law, as in computer science. With the diverse use of the term a distinctive meaning of agenthood becomes lost. It is therefore useful to examine what makes up an agent.

The original sense of the word refers to the participant of a situation that carries out an action; this meaning, however, is not often used when discussing AI. Within the context of a MAS and throughout the course of this paper, an agent is defined as "an autonomous 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." [6] Autonomy represents the agent's ability to proceed without outside interaction, and an environment refers to the agent's surroundings, be they virtual or physical. An example of a physical MAS is Carnegie Mellon University's (CMU) Robotic Soccer Team. At CMU researchers use Sony® AIBO robot dogs as soccer players that coordinate to achieve an objective of scoring goals (see Figure 1). Each robot would represent an agent with complicated rules to determine which actions to perform under what circumstances in the game. The environment in this example would be the robots' physical limitations of the soccer field.

Figure 1. CMU's Sony AIBO robots coordinating to score a goal.[7]

The study of MAS has given rise to many sub-disciplines of agent research, such as Swarm Intelligence. Swarm Intelligence (SI) uses many of the concepts from MAS but typically focuses more on large numbers of simple agents acting together as a group. A common MAS that focuses on SI is a virtual ant colony that contains many ant "agents" who follow simple programmed rules causing them to behave similar to real ants. These virtual ants are autonomous in that, once started, they would proceed in executing their rules without intervention. In this system, the environment represents the virtual space available for the ant agents to explore. Such a virtual ant colony has been used in a technique called Ant Colony Optimization (ACO) to find approximate solutions to many difficult combinatorial optimization problems. SI systems can sometimes compose thousands or even hundreds of thousands of agents whose simple behaviors emerge into more complex, goal-oriented behavior.

These definitions of MAS and SI will allow a better understanding of the current research regarding agent simulation.

1.2.3. AOP

A further understanding of the origins of MAS comes from Agent Oriented Programming (AOP). AOP is a computational framework developed by Stanford professor Dr. Yoav Shoham in a 1993 journal publication.[8] Although somewhat dated now, AOP made a notable step toward the MAS tools that are developed today and is frequently cited in MAS publications. AOP can be viewed as a specialization of object-oriented programming. Many of the components that define agents and their environment today can be traced back to the agent *beliefs*, *decisions*, *capabilities*, and *obligations* described in this article. Dr. Shoham explained the state of an agent as its *mental state* and created primitives for inter-agent communication. These important steps gave language and direction to the mounting agent enthusiasm of the time.

1.2.4. MAS Simulation

When MAS researchers wish to investigate a multi-agent problem, such as ACO or soccer robots, one of the first requirements is to develop or acquire a simulation tool with which to run tests on the agents' rules and observe their interactions. The specific simulation needs of each researcher can vary widely

depending on factors such as the number of agents to be simulated, the complexity of rules, the level of cooperation between agents, and the amount of information that needs to be recorded for each experiment. For example, research that uses fewer than a hundred simple agents could likely be performed on a single computer; whereas research that uses thousands of medium to heavy-weight agents will almost certainly need a simulation system capable of distributing processing to multiple computers. A choice must be made early on in MAS research to develop a new simulation engine tailored to the researcher's individual needs or to use preexisting MAS development tools.

There are numerous MAS development tools available, some of which include: Aglets, BRAHMS, Cougaar, Jack, JADE, JAS, Jason, MACE, Repost, SeSAm, Spyse, Swarm, and VisualBots.[9] All of these tools provide agent development systems, but they vary widely in focus and scale. VisualBots, for instance, is an agent based simulator using Excel and Visual Basic, while Cougaar is a DARPA funded defense project offering Java agent development for large scale distributed applications. In addition to these tools, there are countless custom applications that have been developed by researchers who were not aware of these tools or felt they did not satisfy their needs. The CMU Robotic Soccer Team chose to implement a custom simulator, illustrated in Figure 2. Developing a custom system takes time, though, and the more time that must be spent developing a sufficient simulator, the less time there is available for actual MAS research.

Figure 2. CMU robot soccer simulator[10]

On the whole, the existing MAS tools either prove to be insufficient in functionality or far too complex and extensive for use at the undergraduate level. As a result, this research focuses on developing a graphical MAS simulator for undergraduate research and teaching with a balance between functionality and usability. Rapid development of robust MAS simulations is achieved through a graphical application and an extensive Java Application Program Interface (API).

2. SIMULATOR FEATURES

Since this research was focused on creating a simulator primarily for undergraduates, the first task was to determine what features and functionality would be important for students and professors at the undergraduate level. Trinity University professors Dr. Lewis and Dr. Zhang both had experience working with simulators, AI, and MAS. Their instrumental feedback combined with that of other Trinity students who had studied MAS focused the research on key functionality and provided vital feedback on how to improve usability.

Three major aspects of functionality that arose in the course of the research were that the simulator needed to be graphical, distributed, and contain a thorough API. The simulator needed to be graphical to allow students to easily observe how agent rules are being executed and whether any behavioral patterns are emerging. For instance, imagine how difficult it would have been for the CMU team to correct game strategy had they not developed a graphical interface to their simulator (shown in Figure 2). Another important factor for the simulator was the need to be distributed, both in the sense of a client-server setup and in the sense of multi-threaded. Student researchers investigating long term persistent problems needed a way to start a simulation on a server and leave a simulation running indefinitely, so the design adopted a distributed model. Finally, and perhaps most importantly, the

simulator needed to provide an API to facilitate rapid development of custom agents.

2.1. The World Environment

The *environment* represents the abstract 2D world in which entities reside. All objects in the environment are considered *entities*; agents are a subset of entities. In addition to agents, there might be static entities, such as walls or trees, or dynamic (non-agent) entities such as a growing or diminishing food source. The precise hierarchy of entities and agents will be discussed further in section 3.2.2, but for now it is sufficient to understand that agents and all objects in the environment are types of entities.

Internally, the 2D world environment is represented by a floating point coordinate system. When agents are created, x and y coordinates are specified as well as an entity type. The coordinates specify the entity's location in the world and the type indicates which of the researcher's custom entities to use. Once created, an entity will be added to the *world server*. The world server maintains a list of all entities and executes any rules that may be defined for those entities. Each entity's rules are executed in an update method contained within the entity's custom code. The update method executes entity specific rules and updates for one time step. Time in the world environment is delineated by *time steps* which occur at regular intervals defined by the researcher. The world server can, therefore, update the entire world one time step by calling the update method on every entity in the world.

It may be helpful at this point to look at an example of how a simple simulation can be customized by adding entities, boundaries, and agent rules. Suppose a researcher wanted to create a simulation bounded within a 100 by 100 unit grid that contained two agent types – predators and prey. Both predators and prey would move randomly throughout the environment, but if a predator came within five units of a prey, it would eat the prey. To create this simulation the researcher could add walls (a predefined static entity type) surrounding the desired region then create two custom entity types for the two agents. In addition to moving and eating, the agent code would ensure agents would not move into a wall unit, effectively bounding the simulation. The researcher might decide that world updates should occur every 0.25 seconds and that ten predators should be located in the midst of a group of prey at the start of the simulation. This straightforward simulation would take minutes to create and might look similar to Figure 3 when running. Every agent moves or eats four times a second, eventually killing off all of the prey agents. Although overly simplistic for a practical application, this simulation shows how a researcher can use custom entities and the update based rule system to tailor a simulation to individual needs.

2.2. Graphical Interface

The graphical interface provides one of the most important features for undergraduate students. Whether in research or teaching, students at an introductory level to MAS find it quite helpful to visually see agents interacting. This

visualization allows students to observe and alter agent rules if the agents are not achieving the desired goal.

The simulator uses a panel interface to display information about the state of the environment and the properties of the agents. Figure 3 enumerates the panels of the graphical interface. Agents within the simulation are shown in the display panel, and the header provides control buttons to add new agents to the environment, add custom agent types, and pause the world simulation. Status messages for the environment are related through the status bar and modal buttons such as zoom in and zoom out are located in the mode panel. Users may click on individual agents to see specific agent properties. By default, agent properties display the coordinate based location of the agent and the draw order (the order overlapping agents will be drawn on the screen). These properties are customizable, however, when researchers create custom agents.

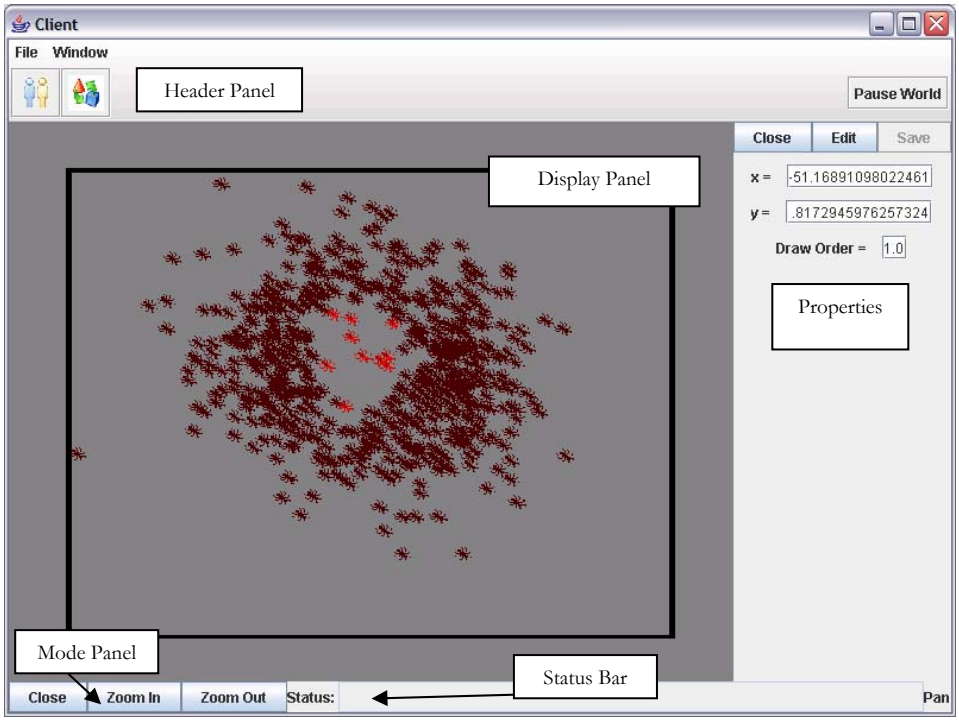

Figure 3. Graphical Interface; Red 'Predator' Ants eat Black 'Prey' Ants

2.3. AGENT ENVIRONMENT API

This section will look at the functionality provided by the API. As shown in the Simulator Class Diagram in Figure 4, the *world* and *client* packages provide the internal infrastructure for the simulator, and the *environment* package contains the agent API that researchers use to create custom agents. The *environment* package provides the Entity interface hierarchy, Entities sub-package, as well as sub-packages for Action, Communication, Memory, and Perception. The Entity interface outlines the required methods for all entities, and the abstract class EntityAdaptor provides

default implementations for most of these methods. AgentEntity and StaticEntity are abstract classes provided to distinguish between these functionally different entity types. For more information on design aspects of the agent hierarchy, refer to section 3.2.2.

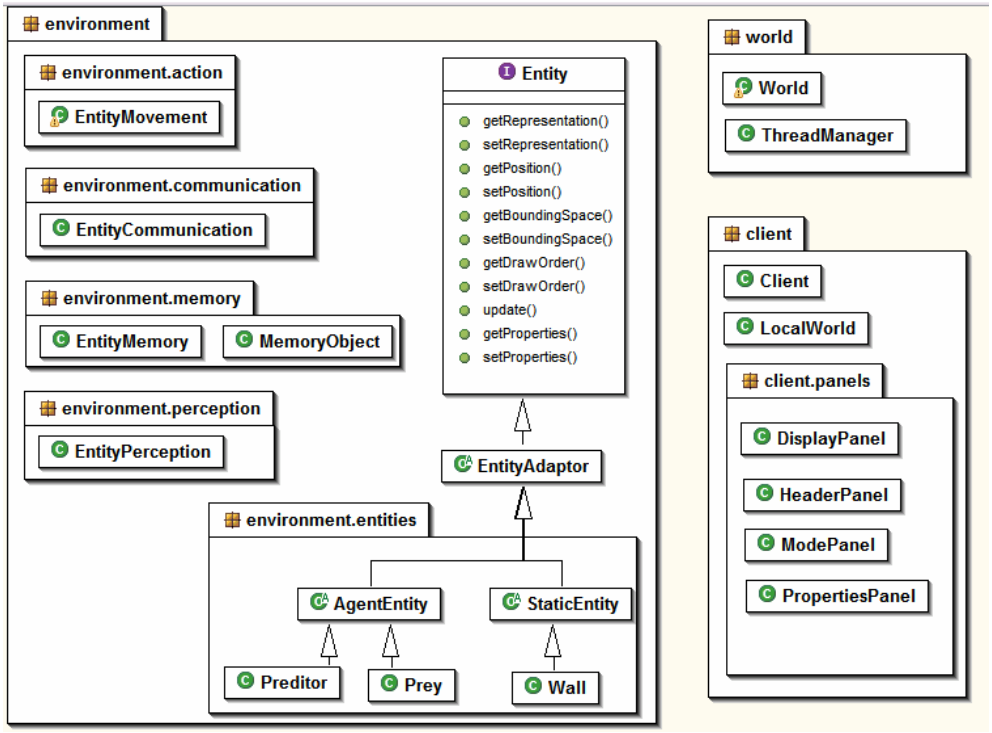

Figure 4. Simulator Class Diagram

2.3.1. Action

The *action* package contains all classes that allow agents to conduct physical actions in the environment. Physical action is defined by movement or other externally explicit interaction. Currently, the package provides the EntityMovement

class which offers a host of useful movement methods such as *move*, *moveRandom*, and *moveRandomAvoidWalls*.

2.3.2. Communication

Through the *communication* package, agents may easily send data or messages back and forth to one another. Any Object data or String message is packaged into a CommunicationData object then sent to the receiving entities using methods in the EntityCommunication class. What an entity does with received data is left to the discretion of the researcher, but by default all data will simply be stored in the agent's memory.

2.3.3. Memory

The *memory* package is provided to give a foundation for agent specific memory. The current EntityMemory class offers only basic functionality using Java generic collections for flexible storage. A subject of future research may be to enhance the capabilities of agent memory, providing functionality specific to MAS.

2.3.4. Perception

Entities' awareness of the environment around them is achieved through the *perception* package. The EntityPerception class provides methods such as *entitiesInRegion* that allow agents to determine what entities are in their immediate surroundings.

2.3.5. Creating an Agent

To develop custom entities researchers may create a class that inherits from AgentEntity or StaticEntity (dependent upon the functional purpose of the entity) and override any of the methods provided from the Entity interface to customize the entity. The actual rules of the agent will be programmed into the update method. To facilitate rule programming the remaining sub-packages of the environment provide essential agent functionality.

2.4. DISTRIBUTED SYSTEM

2.4.1. Server-Client Relationship

As explained in Section 2.1, the simulation runs on a server called the "world server". This server has no direct graphical interface but instead runs from the command line, allowing it to be executed as a background process for long-term persistent simulations. Users may connect to the server using the graphical client interface described in Section 2.2. Any number of clients may view the world at a given time, enabling multiple researchers to observe a simulation. Network communication is achieved through Java's Remote Method Invocation (RMI) technology. Essentially all method calls on the client side are relayed on to the server for processing. RMI does require the RMI registry to be running on the host server. This should not, however, cause significant burden on researchers as it comes installed with Java.

2.4.2. Multi-threaded processing

On large scale simulations the host server can be configured to run in multiple threads. This option allows a simulation to take full advantage of a multi-processor or hyper-threaded system. To distribute processing, the world server assigns entity updates to the first available thread. Each thread is then assigned to a processor by the operating system. In tests on a four processor system, simulations of up to 10,000 simple agents updating every 0.25 seconds could run smoothly before any lag in processing was noticed.

A future progression of the distributed model for the simulator might be to divide entity update processing among multiple computers, rather than just multiple threads. A logical implementation of this using computational geometry would allow client computers connected to the server to process updates for the entities within the client's immediate surroundings. This solution would negate network traffic between the client and server related to graphically displaying agents in the client's viewable area and would diminish the processing burden on the server by a factor of the number of clients.

2.5. AGENT PROPERTIES

One of additional features provided by the simulation platform is the ability to view and edit entity properties at runtime from the client interface. The properties panel shown in Figure 3 will change to represent a specific entity's properties when the user clicks on an entity in the display panel. Once an entity's

properties are selected, they may be edited by clicking the edit button; this will enter an edit mode. When in edit mode, world updates are paused for all clients connected to the server. During this time, any property of the entity may be modified. When editing is completed, the user may choose to either save or cancel changes and world updates will resume.

Agent properties are defined through the *getProperties* and *setProperties* methods in the Entity interface. These methods alter the EntityProperties class, defined in the entity code, that contains all of the editable fields for that entity. For example, the EntityAdaptor class defines the default EntityProperties to be *x, y,* and *drawOrder* as is visible in Figure 3. A custom entity can have any of its fields displayed and edited through the properties panel so long as the researcher overrides these two methods.

2.6. Saving/Loading

A vital function of any simulation is the capability to save and load the state of the environment. This functionality is included in the simulator through the client interface's file menu. When saving, the world server will serialize every entity in the world to a file specified by the user. Upon loading, the world server reads in the entities and repopulates the environment, returning the world to the state prior to saving. It should be noted that due to limitations of Java serialization a file may not be loaded if there have been significant changes to the entity classes contained in the file since it was saved.

2.7. Import an Environment

To provide a way to quickly develop a simulation environment, a text file in a specific format may be designed and imported by the simulator. This functionality differs from saving/loading, because it focuses on the initialization of a new simulation and is not affected by changes to the entity classes. The text file format requires the researcher to initially give the dimensions of the environment and define each entity used with a class name and representative character on each line. Then the researcher can provide a multi-line map of entities that will be created in the environment upon importation. Once the file is created, a new simulation can be initialized with the imported file's entity map.

3. DESIGN

3.1. CONSIDERATIONS

This research was undertaken to provide an MAS simulation platform for research and teaching that could be easily and quickly implemented by undergraduate students. To truly facilitate usability one must first understand how a system will be used. This can sometimes be difficult to predict, as future users can often use a system in unexpected ways, especially if there is insufficient documentation to guide them through the process.

Consequently, the first design challenge was to identify the needs of the undergraduate student researcher and understand how the student would use a system that provided the needed functionality. The various uses and actors in the system were identified early in the design process. Figure 5 shows a use case diagram that reflects the initial design of the platform.

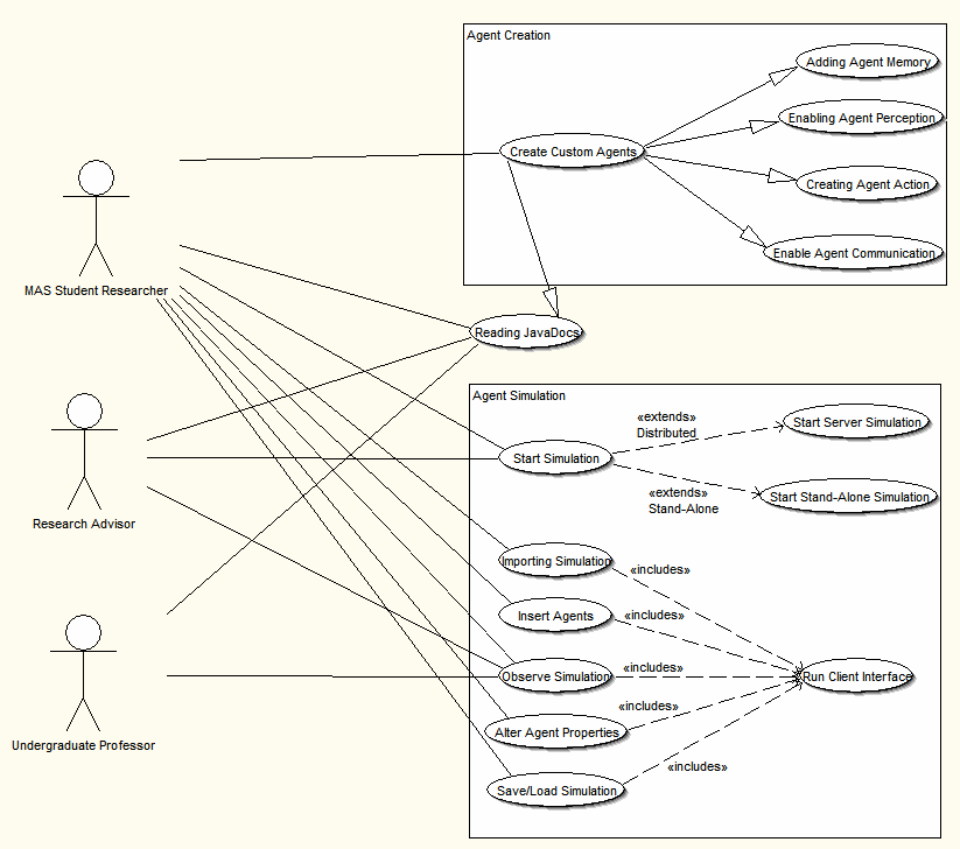

Figure 5 - Simulator Use Case

Once the functional uses had been identified, the design focused on implementing the various systems. The "agent creation" system was implemented as the *environment* package, and the "agent simulation" system became the internal architecture of the platform, principally the *world* and *client* packages. The following sections will investigate design decisions and challenges for these systems in further detail. Throughout this chapter, references will be made to the various packages,

classes, and methods that make up the platform. It may be helpful for the reader to refer to Appendix A – API and Javadocs.

3.2. Environment API

A cornerstone of this research focused on designing an *Application Program Interface* (API) for the environment to aid in the process of creating custom agents. Throughout the design process, Bloch's principles of effective API design,[11] his book *Effective Java,*[12] and *Design Patterns*[13] guided the creation of the API. Focus was placed on making the API easy to use, learn, and extend while providing sufficient power to satisfy the requirements of an MAS.

3.2.1. Environment Façade

When designing the structure of the environment API, the Façade design pattern was used to provide a unified interface to the internal subsystem functionality. The initial design lacked this structure and users instead had to reference the relevant functions from internal classes, primarily the World class. The same functionality was offered but through added complexity, communication, and dependencies. These difficulties were alleviated by creating a series of simplified interfaces containing the more general functionality of the internal subsystem. Figure 6 shows an illustration of how a façade can shield the subsystem components, making the subsystem easier to deal with for the client classes (agent classes). In

addition, the interfaces were named in a format more familiar to MAS researchers. For example, if a researcher wanted an agent to search for all entities in the surrounding area, in the old model the agent code would have to call one of the many search functions in the World class, causing a dependency on the internal subsystem. In the new model, however, the researcher can use the *entitiesInRange* method in Perception class that is part of the *environment* package. In this way, dependencies between the *environment* and *world* packages are compartmentalized into the four façade interfaces: *action, communication, memory,* and *perception*.

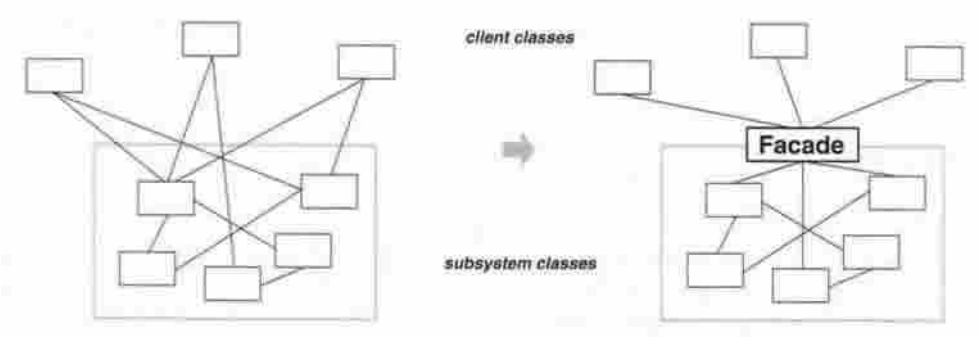

Figure 6 – Façade[14]

3.2.2. Entity Hierarchy

Another goal of the environment API was to facilitate agent creation through an easily extensible entity hierarchy. Figure 7 shows the Entity interface as the super-type to the hierarchy. All agents, walls, or other objects in the environment must implement the Entity interface and the methods therein. Most of the methods

in the Entity interface are relatively simple. For example, *getRepresentation* returns the image to be displayed on the screen, and *getPosition* returns the entity's position. However, to further facilitate rapid agent creation the EntityAdaptor was created. The EntityAdaptor is an abstract class that provides a default implementation to all of the methods in the Entity interface except *update*. The *update* method defines the rules of the agent and must be agent-specific. The StaticEntity and AgentEntity abstract classes define two functionally different entity types. There is currently no custom code in either of these classes, but it seems logical that after extensive agent coding, methods common to all agents or static entities would arise and could be placed in these classes. The classes also allow code to identify an Entity by type as either an AgentEntity or a StaticEntity.

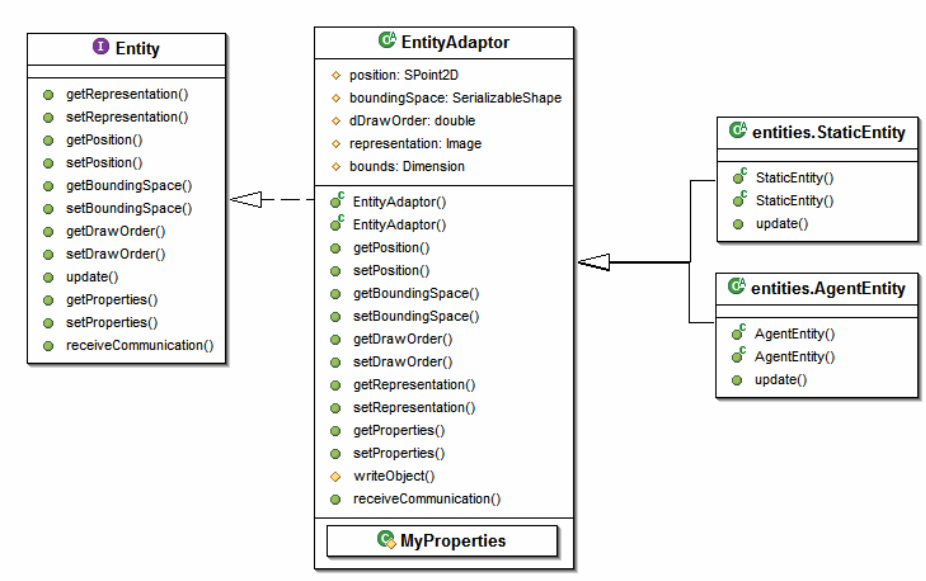

Figure 7 - Entity Hierarchy

This hierarchy is intended to make life easier for the API user, but other than implementing the Entity interface, adherence to this structure is not required. The typical way to quickly code an agent would be to make a class that implements AgentEntity, code an *update* method, and code any Entity methods for which the default method provided by the EntityAdaptor is unacceptable. The resulting agent could be added to a simulation and would follow the rules provided in the update method. This would be the typical implementation; however, the only requirement is that all entities implement the Entity interface.

3.3. Internal Architecture

The internal architecture consists primarily of the *client* and *world* packages. The *client* package contains extensive Java Swing code for the graphical interface as well as a local copy of the World[1]. The *world* package contains the World server class and all related classes. Although the previous section is entitled the "Environment API", the entire project is open source, and all classes can be extended and customized. It would not be common, but users could create custom Client or World classes for their simulations. Consequently, documentation and effective API design was treated with the same high regard that the entire project received.

---

[1] It should be noted that there are varying definitions of the term "world". The proper noun, World, refers to the singleton World server class. The *world* package is the container for all classes relating to the world server and is distinguished by italics. The lowercase world environment relates to the abstract space in which all entities reside. Finally the "local world" refers to the remote copy of the World class on the client machine.

The following sections will take a closer look at the hurdles encountered in creating the internal architecture and what design decisions were used to overcome these obstacles.

3.3.1. Distributed Client/Server Model

The most significant single design decision for the simulator was deciding to make the simulator distributed in a client/server model. This decision affected how almost every aspect of the simulation was designed, due to the inherent limitations of communicating over a network. The distributed model was undertaken to allow researchers to run persistent simulations remotely on a server for survivability and to assist in group simulations. Remote Method Invocation (RMI) was chosen as the best method of network communication because of its ease of use and support for simple object serialization.

One of the design consequences of using RMI was that any object communication between the server and client must be serializable and small in size to ensure a fluid transfer. As a result, serializable versions of many Java library objects, such as shapes, had to be created as well as a serializable Entity. Entities presented a particular challenge. All entities in a client's viewable area had to be transferred from the world server every update period (typically once every 0.25 seconds). Since there was no way to know how large a user's custom entities would

be, the entire entity could not be sent directly over the network. The solution was that a small subset of each entity would be transferred, containing only the information most vital to the client display such as location, bounding sphere, and image. This design improved efficiency for all simulations and made some simulations with larger entities possible.

3.3.2. Inserting an Agent, Abstract Factory

A form of the AbstractFactory pattern was used in overcoming a challenge with remote insertion of agents between the client and server side. The challenge was that the server code, having been compiled previously into a JAR file, was unaware of what custom agent classes the researcher may have created. So when the researcher goes to insert a new agent from the graphical client interface the world needs some way to recognize what agent types are available and which type is currently being inserted.

To solve this problem, each Agent type to be used in a simulation is added through a graphical panel in the client interface using its full class path. The client transmits the class path information to the server, which uses Java Reflection technology to determine whether the class is in fact an agent. If the class path is valid, the server sends an identifying number back to the client corresponding to that agent's class path. Once all agent types have been added to the simulation, a

researcher may select which agent type he/she wants to insert, select a world location in the client display panel, and select the number of agents to insert. This information is transmitted to the world server for insertion along with the identifying number of the desired agent type.

The server uses an abstract factory to create an instance of the specified agent type at the given location through a constructor using Java Reflection. The typical AbstractFactory pattern abstracts the creation of the object but the object type is known. This method differs as the type is not known except for the identifying number and must be looked up and created through reflection. In this way, an agent can be created from a custom entity type after the world server has been compiled.

These challenges exemplify a few of the obstacles encountered throughout the design of the simulator and the respective solutions.

4. COMPARISON WITH OTHER MAS

Multi-agent research has been an ongoing field for over a decade now, and in that time many tools and languages have been created to assist in the development of MAS. Due to the broad scope and varied applications of agent systems, each of these tools seeks to satisfy a niche in the needs of MAS developers. This research sought to create a graphical simulator and API to assist the development of undergraduate research and teaching in MAS. Some of the other MAS development tools include Aglets, BRAHMS, Cougaar, Jack, JADE, JAS, Jason, MACE, Repast, SeSAM, Spyse, Swarm, SPADES and VisualBots. A full list of tools with web links is available in Wikipedia's "Multi-agent system" article.

In this section, a closer look will be taken at the more common development tools, namely JADE, Cougaar, Repast, and Jess. For each platform, an overview of the features and an examination of the different design decisions that distinguish the platforms will be highlighted. Similarities, differences, and potential integration with this research will also be discussed.

4.1. JADE

One of the more popular development tool currently available is the Java Agent Development Framework, or JADE.[15] This framework is an open source platform for peer-to-peer agent based applications with a focus on the personal

communications sector. The software is distributed and trademarked by Telecom Italia (Tilab) and the board of directors includes Tilab, Motorola, Whitestein Technologies AG., Profactor GmbH, and France Telecom R&D.

The Jade platform takes a distinctly different approach to agent development than this research. Jade is less of a graphical simulator and more of a smaller scale, state-based simulation. There are graphical tools, as seen in Figure 8; however, the interface displays state based information about the agents, instead of the fully visible graphical environment present in this research's simulator. In addition, each agent in Jade has its own thread, allowing more complex agents but smaller overall simulations. A state-based approach provides benefits for applications where seeing agent interaction is not as important as knowing the precise details of agents' states. A correlation to the Jade state analysis in the current research is the graphical properties panel that appears whenever the user clicks on an entity. One improvement Jade presents to this model is a "Sniffer" agent that can be used to search for other agents given search criteria. The ability to search could improve the current research, because presently a user must visually identify and click on an agent to see its properties.

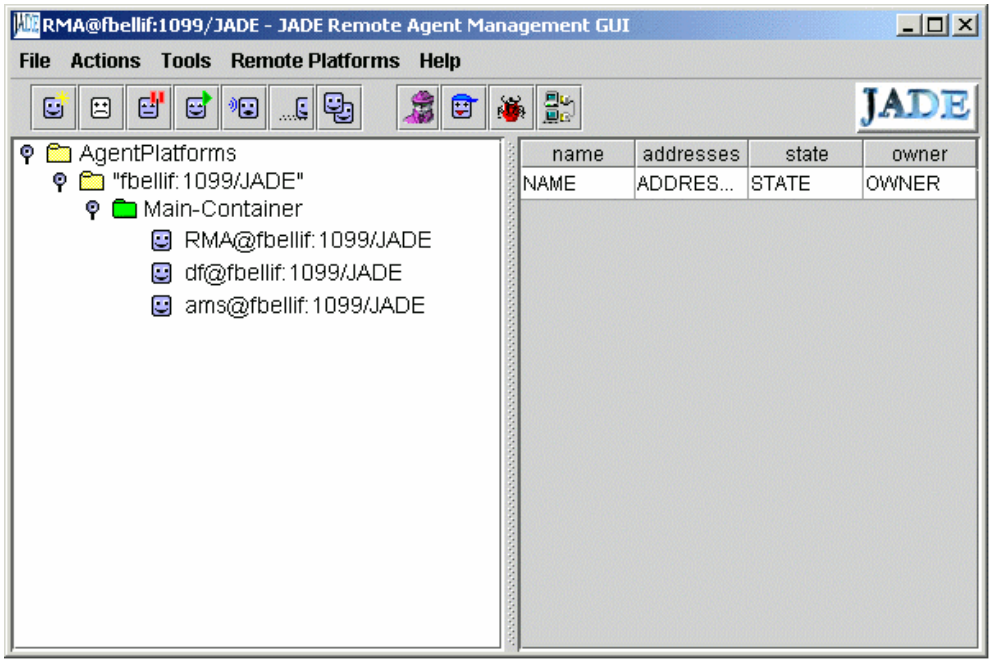

Figure 8 - Jade Graphical Interface[16]

In addition to state simulation, Jade goes to great measures to conform to the Foundation for Intelligent Physical Agents' (FIPA) specifications and to implement a robust messaging system. As of June 8th 2005, FIPA is the official standards organization for agents and multi-agent systems for the IEEE Computer Society. The specifications represent a collection of standards which are intended to promote the interoperation of heterogeneous agents and the services that they can represent.[17] Jade complies with these specifications by providing a naming and yellow-page service, message transport and parsing service, and a library of FIPA interaction protocols ready to be used. The interface for the Jade agent messaging system is shown in Figure 9.

Conformity to FIPA specifications was not included in this research, in part because of a lack of awareness of the existence of specifications until the recent incorporation of FIPA into the IEEE. As is discussed in Section 5, Acceptance and Future Direction, adopting these specifications might provide a productive avenue of future research, if it is determined that the specifications further the goals of undergraduate MAS research and teaching. Also discussed in Section 5 is a potential messaging and notification system, similar to that of the Jade platform, which would enhance the usefulness of the API.

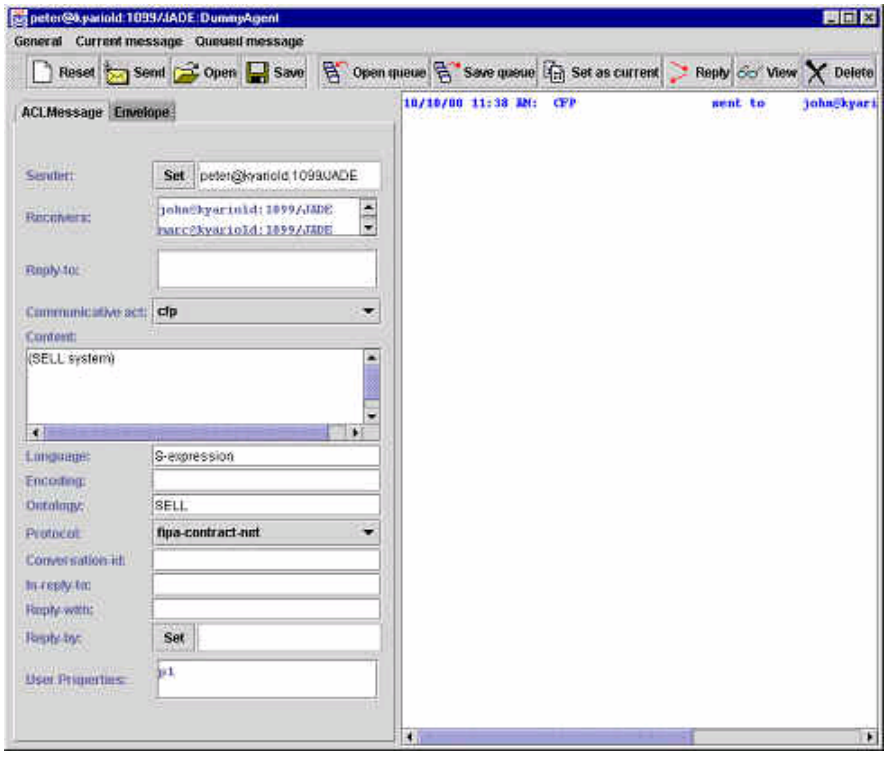

Figure 9 - Jade Messaging Interface[18]

Jade provides full integration with JESS through a package called *JessBehavior*. When using this package Jade provides the shell of the agent and JESS is the engine of the agent that performs all necessary reasoning. In addition, since the Jade platform as well as this research are both developed in Java, there may be potential for integration between these two systems. This integration is further investigated in Section 4.5.

4.2. COURGAAR

Cougaar represents the largest-scale distributed, agent-based architecture available today. This Java project has been the product of two consecutive, multi-year DARPA research programs into large agent systems spanning eight years. "The first program conclusively demonstrated the feasibility of using advanced agent-based technology to conduct rapid, large scale, distributed logistics planning and re-planning. The second program developed information technologies to enhance the survivability of these distributed agent-based systems operating in extremely chaotic environments."[19] The architecture that resulted from these programs provides a framework to develop distributed, agent based applications.

The Cougaar platform offers a long list of features. The framework itself is designed after JavaBeans and uses a Cougaar Component Model (CCM) to allow individual software units to interact with one another through abstract interfaces called services. The system uses an independent blackboard as a shared message

space across computers in the distributed network. Great effort has also been put into making the system persistent to failures. The state of every published object is saved regularly on non-volatile media. Two naming services, White Pages and Yellow Pages, are offered to map agent names to network addresses and allow agents to query for other agents of similar properties. A Community Service is provided to group agents of common functional purpose and facilitate communication with these homogenous groups. Cougaar provides a suite of services based on Java Servlets that allow agents to have an HTTP-based web presence. In addition to these and other services, Cougaar provides various software component plugins that contribute specific pieces of business logic to agents.

Clearly the Cougaar platform's scope far exceeds that of this research; however, Cougaar's vast scope serves as both its greatest benefit and its greatest limitation. For a problem of enormous complexity that demands 100 machines and military grade security and stability, Cougaar is the natural choice. On the other hand, most undergraduate research does not have nearly such stringent demands, and in fact, the complexity of Cougaar acts as a hindrance to the relatively simple research and teaching applications desired by undergraduate students and professors. From the onset, this research has focused on satisfying the need left unfulfilled by current MAS systems.

4.3. REPAST

Repast presents the most similar platform to this research currently available. Repast stands for Recursive Porous Agent Simulation Toolkit and is an open source agent modeling toolkit originally created at the University of Chicago. The platform is currently maintained by the non-profit volunteer Repast Organization for Architecture and Development (ROAD). The Repast software was based on the Swarm toolkit[20] and now offers many features multiple implementations in several different languages. According to the ROAD homepage:

> "Our goal with Repast is to move beyond the representation of agents as discrete, self-contained entities in favor of a view of social actors as permeable, interleaved, and mutually defining; with cascading and recombinant motives. We intend to support the modeling of belief systems, agents, organizations, and institutions as recursive social constructions."[21]

The features of Repast include a range of two-dimensional agent environments and visualizations; dynamic access and modification of agent properties, agent behavioral equations, and model properties at run time; a fully concurrent discrete event scheduler; a variety of agent templates and examples; simulation results logging and graphing tools; and libraries for genetic algorithms,

neural networks, random number generation, and specialized mathematics. The libraries are developed in an object oriented manner and offer a full API to developers. The platform is available on Windows, Mac OS, and Linux platforms as well as many languages including Java, C#, Managed C++, Visual Basic.Net, Managed Lisp, Managed Prolog, and Python scripting (although the primary and original language is Java). Some of these features are illustrated in the progression of screenshots in Figure 10. This example Repast simulation was taken from the platform website; the green dots are simple agents called Heat Bugs that absorb and expel heat and a heatspace which diffuses this heat into the area surrounding the bug. Heat bugs have an ideal temperature and will move about the space attempting to achieve this ideal temperature.

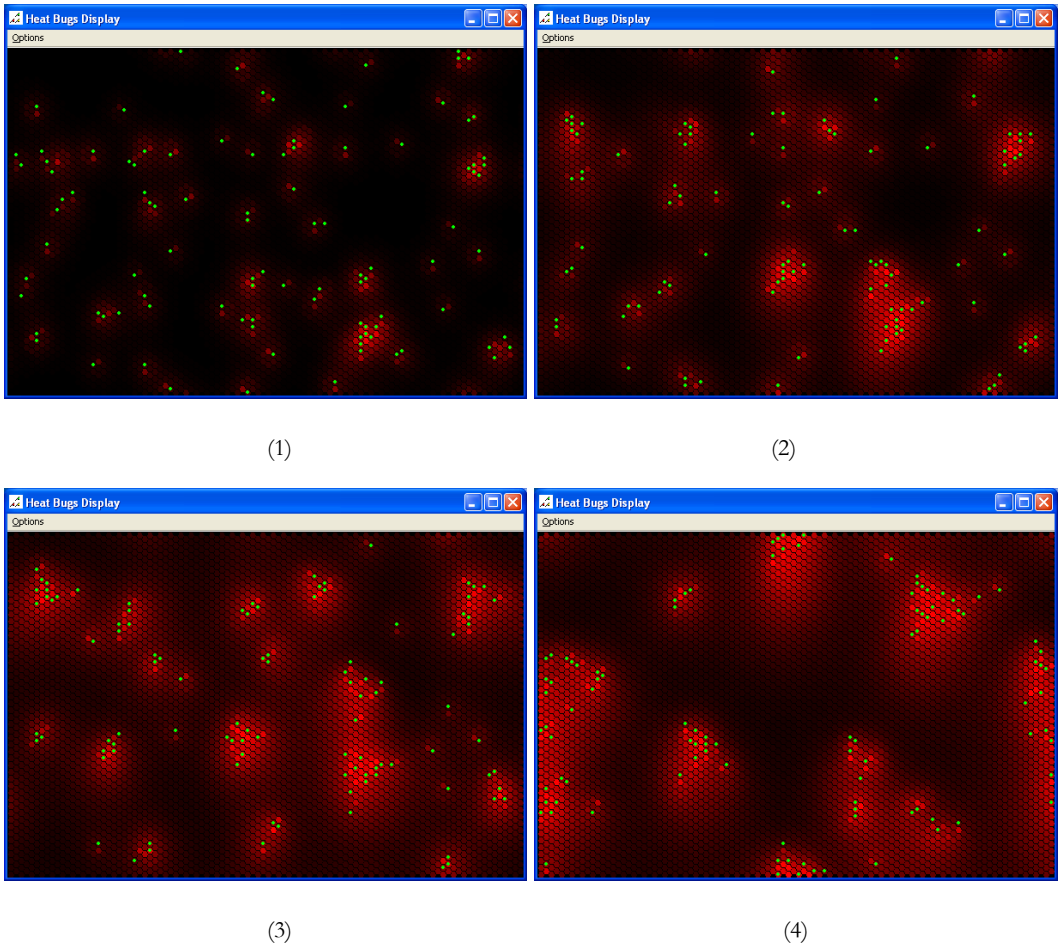

Figure 10 - Repast Heat Bug simulation[22]

When Repast is compared to the current research, there are many noticeable similarities that arise. Both simulators utilize a two-dimensional environment, provide agent libraries, and are written in Java. Most importantly, both platforms seem to represent a similar direction in scope and goals. There are some notable differences. Repast does not appear to offer any distributed (client/server) or multithreaded support. While diminutive in comparison to Cougaar, the scale of the

Repast platform is still quite large. For example, the total number of classes in the Java deployment of Repast is 580, whereas the current research totals only 41. Although these extra classes represent added functionality, they also signify increased complexity and may present obstacles for rapid development in an undergraduate situation.

Despite the differences, the similarities of the two projects one might question why this research was not integrated with the Repast platform from the onset. In short, the answer comes from the fact that Repast was not discovered until late in the course of this research; however, the definitive answer to this question would require a more thorough analysis of whether the Repast platform would satisfy the goals of this research. Section 4.5 discusses possible integration between this research and Repast, and section 5.2 lists this integration as a potential future enhancement to this research.

4.4. JESS

Jess is a rule engine and scripting environment written in Java by Ernest Friedman-Hill at Sandia National Laboratories. The Jess engine provides the capacity to reason using a knowledge base supplied from a series of declarative rules. Jess' fast, small, and light engine allows full access to outside Java code and the Java API. The language has become very popular among MAS researchers, and a new, much anticipated version will soon be released. The new Jess 7, codenamed

Charlemagne, uses the Eclipse open source environment to provide a graphical rule development environment, see Figure 11.

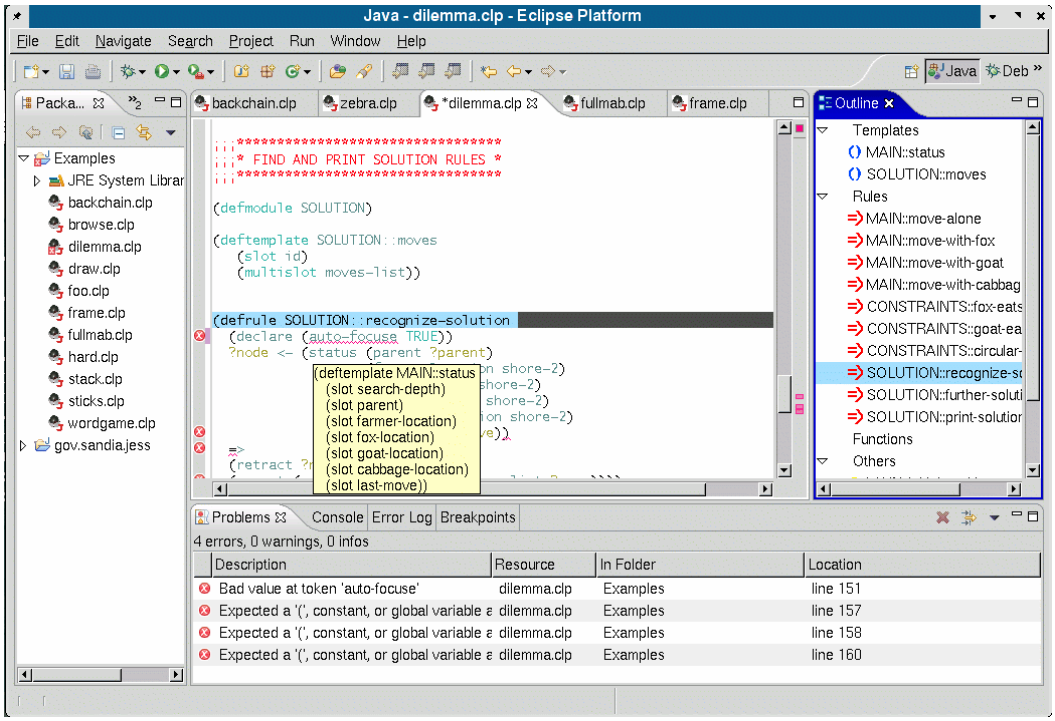

Figure 11 – Jess 7 (Charlemagne) Preview[23]

4.5. Potential Integration

JADE, Repast, and Jess all present promising integration opportunities as they are all based in the Java language and have comparable scope and direction. Cougaar is simply too large to attempt integration, and the Cougaar system would not benefit from such an effort. Naturally, all of these possibilities would depend on

further investigation of the respective platform internals, but this section will outline at a high level some of potential benefits of integration.

One of JADE's primary disadvantages is the lack of a 2D graphical environment in which to view agent interaction, which presents an opportunity for integration. Since the JADE state-based model is more extensive, it would likely be best to use JADE as the foundation and add certain features from this research to the JADE platform. With some modifications, the graphical environment display panel in the client interface could display JADE agents based on their locations. This likely represents the most effective integration possible for JADE.

The primary features not offered by the Repast platform that this research might be able to provide are the distributed client/server model and the multithreaded world server. The possibility of a direct integration of these features is tenuous at best, due to the fact that a distributed approach typically must be taken into account from the onset of the design. Rather than directly transitioning these features from one platform to another, the design for distribution in this research could provide a viable model for Repast. There were many hurdles that had to be overcome in order to achieve a distributed solution in this research. The methods and designs successfully used in this research could provide valuable direction when distributing the Repast platform.

Finally, the Jess system provides a very practical integration for this research. Similar to the way JADE created the *JessBehavior* package, this research could relatively easily implement similar integration. The JESS system would provide the reasoning engine and this research's platform would act as the shell. Since JESS allows Java code in its rules, JESS agents could extend the Entity interface and implement necessary methods, but instead of having world updates, the JESS engine would drive the environment. This integration could be especially advantageous if implemented with the new Jess 7 framework.

5. ACCEPTANCE AND FUTURE DIRECTION

5.1. ACADEMIC USES

Throughout the development of the research there have been many opportunities for the simulator and API to be utilized and tested. As part of this research, many test simulations have been performed, and the simulator has been extensively tested by two student researchers in disciplines outside of classic computer science areas. One of the many benefits of MAS and this simulator is that cross-discipline research becomes more possible than ever before. A student or research team with a symbiotic comprehension of computer science and another academic area can progress research in both fields by developing MAS simulations.

5.1.1. Research currently using the Simulator

Perhaps the most validating aspect to this research has been the ongoing testing and feedback received from the two undergraduate research students who have been using the simulator for MAS research. From the onset of the project, Andrew Krausnick and Tom Dietzel have been employing the API and simulator to create custom social and economic MAS simulations.

Krausnick's work has investigated the correlations between virtual agent societies and human populated virtual worlds. Virtual societies emerge as a result of the social interaction of large agent groups, and virtual worlds come from cumulative

human interaction in a virtual space, such as in a Massively Multiplayer Online Role Playing Game (MMORPG). Through the course of his research, Krausnick has extended the API to create a layer of classes directed specifically at the development of social and economic agents.

With this enhancement to the existing API, Krausnick's research has presented some interesting societal systems. In one simulation, he created wheat farmers, bread makers, miners, jewelers, and traders. The wheat farmers harvest wheat fields for the bread makers in exchange for bread and jewelry; the bread makers make bread to trade; miners mine gems for the jewelers in exchange for jewelry and bread; the jewelers create jewelry for trade; and the traders act as brokers for the exchange of goods. All agents are constrained by a need for sustenance satisfiable by bread and a desire for riches satisfiable by jewelry. The simulation of such social environments provides intriguing sociological results. A visualization of this simulation is represented in Figure 12.

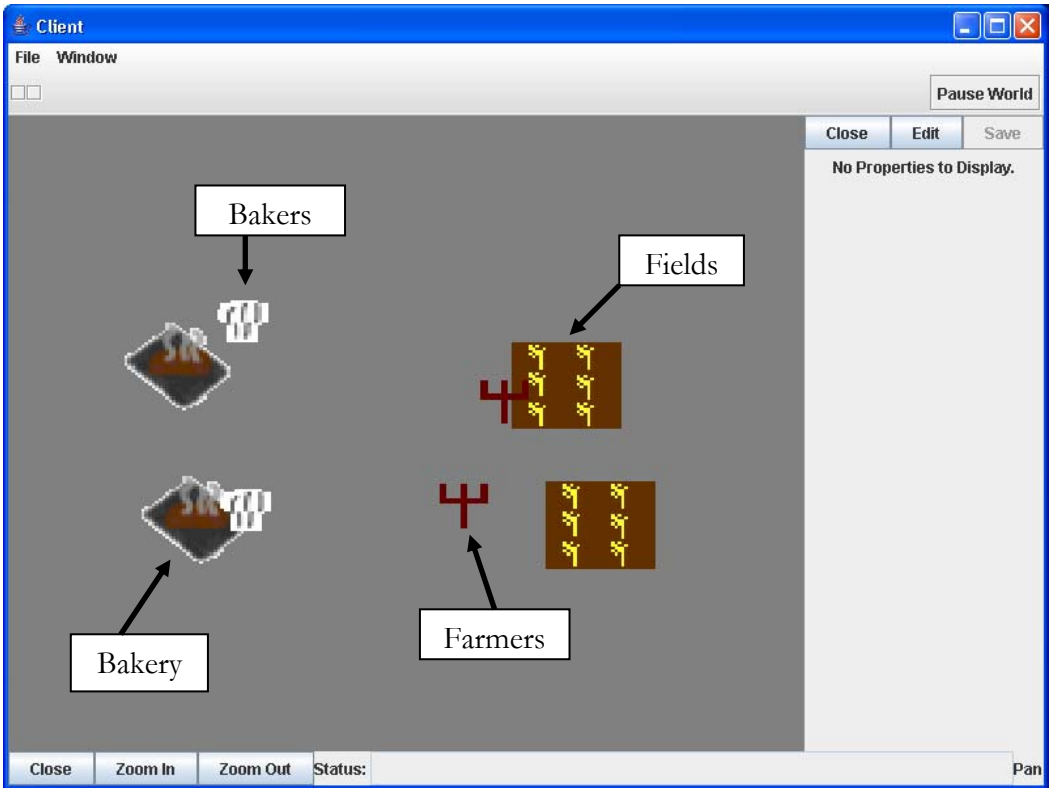

Figure 12 - Krausnick's Farm Simulation

The creation of such a modular, extensible economic society has revealed some intriguing correlations between virtual societies and virtual worlds. In his research, Krausnick outlines the foundations necessary for an integration of these two fascinating concepts.

Dietzel's research has also focused on representing social and economic system through agents. His research investigates how individual agents' needs and actions lead to an overall societal impact on every agent. Dietzel sought to study how his agents performed under varied situation and societal models. Once again,

the visual representation provided by the agent simulator allows direct observation this large system of interactions in a way not before possible.

5.2. Potential Future Enhancements

There are three main areas of this project that could be enhanced as a result of future work or research. First, as an ongoing part of the current research, this MAS development system will hopefully be accepted as an open source project. Open source acceptance would allow development on the project to continue and thrive even after the conclusion of this research. In addition, an open source web space would provide a single source for professors or students to download the simulator for academic use.

The second area that could benefit from future enhancement is the agent development API in the *environment* package. Currently, most of the basic functionality necessary to develop agents (action, perception, memory, and communication) is present, but there remains much room for improvement. Some of these developments include more advanced agent interactions for the *action* package; varied methods of viewing and observing the environment for the *perception* package; more sophisticated, MAS specific, memory; and a complete event based messaging system for agent communication. These enhancements would provide greater functionality to the agent library, freeing researchers to further focus on writing agent rules and studying agent behavior.

Finally, there are various improvements that could be made to the internal *client* and *world* simulator packages. Perhaps the most important addition would be to develop a central logging system on the world server. This logger would allow agents to report data or state information to a log file. The file could then be analyzed following a simulation to determine precisely what occurred with each agent throughout the simulation. Further developments to the internal system might include the distributed approach described in section 2.4 or a three dimensional interface for spatially demanding simulations.

Further enhancements to the internal architecture might include conformity to FIPA specifications, search functionality, and further integration. Conforming to specifications would provide many benefits in the process and might open doors for the simulator to new users and integration possibilities. A search function in the client graphical interface is also very needed, as currently a user must find an agent and manually click on it to view properties. If the user could search by certain criteria, finding and editing agents would be greatly facilitated. The integration possibilities discussed in section 4.5 offer solid potential for future research as well. The integration with Jess would be particularly advantageous given the upcoming release of the graphical Jess 7.0 version.

5.3. Future Use

5.3.1. Applications for Research

There was a time when agent research was an isolated subfield of computer science that might only be analyzed in graduate school. Today, however, MAS research has really taken off. There are many promising avenues of research at both the graduate and undergraduate level. As undergraduate students learn about agent-based systems in their introductory AI courses, they will want to apply this knowledge in research. Until now, though, there has not been a simulation platform directed at the entry level student who may not have extensive experience in MAS and AI. This simulator makes it possible for the undergraduate student to quickly begin experimenting with actual MAS research, rather than spending a few weeks or even months learning a complex system. A large investment of time is acceptable for a long term research project in graduate school, but the undergraduate is looking for something that can be learned quickly and provide the necessary functionality. This simulator satisfies that need for the undergraduate researcher.

5.3.2. Applications for Teaching

In addition to the promising applications this simulator holds for research, there are many possible ways in which it could aid learning and teaching of MAS. When students are first introduced to MAS, they typically have a general

understanding of computer science principles but may have difficulty with how behaviors can emerge from simple agent rules. The visualization of this process through a graphical simulator can greatly assist visual learning students. Moreover, due to the intuitiveness of the API, professors of an AI or MAS course could give students assignments to create specific simulations. Over the course of a week or two, undergraduate computer science students could easily learn how to extend the API and develop basic simulations. Whether used for research or teaching, this simulator offers solid potential to further academic knowledge in the field of MAS.

5.4. Conclusion

The ultimate hope of this research is that it will be used by students. In either a research or teaching venue, this simulator has proven to offer practical results for furthering knowledge. If this project is accepted as an open source project, the academic community may use or develop this research from one centralized location. The simulator offers a host of valuable features, but there are also many important enhancements that could be made. Until the project develops a reputation, however, the extent to which the platform is used will largely depend on the professors involved in the project. I recommend each of you to encourage your students to build upon this research and use it to discover great advances through MAS.

6. APPENDIX A – API AND JAVADOCS

This appendix provides the API documentation, or Javadocs, for selected classes of the platform. Reprinting all of the classes would have been far too cumbersome and would not be effective. These classes are the most important to the platform and the most referenced throughout this document. Figure 4 shows how these classes relate in a UML class diagram.

FIGURE

1. CLIENT

CLASS CLIENT

```
java.lang.Object
  └ client.Client
```

All Implemented Interfaces:

RemoteClient

```
public class Client
extends java.lang.Object
implements RemoteClient
```

Description: The Client class contains the GUI for viewing the world and a copy of the LocalWorld, as it is known by the client.

Since:

Sep 6, 2005

Author:

Keller

Constructor Summary

Client(LocalWorld localWorld) Description: Constructor that initializes the Client to the specified LocalWorld.
Client(LocalWorld localWorld, javax.swing.JFrame frame, int pixelWidth, int pixelHeight, double displayWidth, double displayHeight) Description: Constructor that initializes the Client to the specified LocalWorld, JFame, pixelWidth, pixelHeight, displayWidth, and dispalyHeight.

Method Summary

BuilderPanel	**getBuilderPanel**() Description: Accessor Method
DisplayPanel	**getDisplayPanel**() Description: Accessor Method
EventList	**getEventList**() Description: Accessor Method
javax.swing.JFrame	**getFrame**() Description: Accessor Method
HeaderPanel	**getHeaderPanel**() Description: Accessor Method
LocalWorld	**getLocalWorld**()
ModePanel	**getModePanel**() Description: Accessor Method
PropertiesPanel	**getPropertiesPanel**() Description: Accessor Method
java.lang.String	**getStatusMsg**()
int	**getUpdatePeriod**() Description: Accessor Method
static void	**main**(java.lang.String[] args)

	Description: Main method to run a Client.
void	refresh() Description: Refreshes the display panel with current data from the LocalWorld.
void	setStatusMsg(java.lang.String statusMsg)

Constructor Detail

Client

```
public Client(LocalWorld localWorld)
```

Description: Constructor that initializes the Client to the specified LocalWorld. By default, the client will be initialized to view a 200x200 space in the world at resolution 800x600.

Parameters:

`localWorld` - the world the client will be viewing

Since:

Sep 27, 2005

Client

```
public Client(LocalWorld localWorld,
              javax.swing.JFrame frame,
              int pixelWidth,
              int pixelHeight,
              double displayWidth,
              double displayHeight)
```

Description: Constructor that initializes the Client to the specified LocalWorld, JFame, pixelWidth, pixelHeight, displayWidth, and dispalyHeight.

Parameters:

`localWorld` - the world the client will be viewing
`frame` - frame in which the Client should build the GUI
`pixelWidth` - resolution width of the GUI
`pixelHeight` - resolution height of the GUI
`displayWidth` - width of the viewable area in the World
`displayHeight` - width of the viewable height in the World

Since:

Sep 27, 2005

Method Detail

main

```
public static void main(java.lang.String[] args)
```

Description: Main method to run a Client. Server location is specified by command line arguments.

Parameters:

`args` - [0]: World Sever URL; for example, "rmi://computername:Port/WorldServer"

Since:

Sep 27, 2005

---

refresh

```
public void refresh()
```

Description: Refreshes the display panel with current data from the LocalWorld.

Since:

Sep 27, 2005

---

getPropertiesPanel

`public PropertiesPanel getPropertiesPanel()`

Description: Accessor Method
Returns:
Returns the propertiesPanel.
Since:
Sep 13, 2005

---

getDisplayPanel

`public DisplayPanel getDisplayPanel()`

Description: Accessor Method
Returns:
Returns the displayPanel.
Since:
Oct 6, 2005

---

getFrame

`public javax.swing.JFrame getFrame()`

Description: Accessor Method
Returns:
Returns the frame.
Since:
Oct 6, 2005

---

getModePanel

`public ModePanel getModePanel()`

Description: Accessor Method
Returns:
Returns the modePanel.
Since:
Oct 6, 2005

---

getEventList

`public EventList getEventList()`

Description: Accessor Method
Returns:
Returns the actionList.
Since:
Oct 11, 2005

---

getHeaderPanel

`public HeaderPanel getHeaderPanel()`

Description: Accessor Method
Returns:
Returns the headerPanel.
Since:
Oct 17, 2005

---

getLocalWorld

`public LocalWorld getLocalWorld()`

---

getStatusMsg

`public java.lang.String getStatusMsg()`

---

setStatusMsg

```
public void setStatusMsg(java.lang.String statusMsg)
```

---

getUpdatePeriod

```
public int getUpdatePeriod()
```

Description: Accessor Method
Returns:
Returns the uPDATE_PERIOD.
Since:
Nov 1, 2005

---

getBuilderPanel

```
public BuilderPanel getBuilderPanel()
```

Description: Accessor Method
Returns:
Returns the builderPanel.
Since:
Dec 31, 2005

---

2. ENVIRONMENT

INTERFACE ENTITY

All Superinterfaces:
java.io.Serializable
All Known Implementing Classes:
AgentEntity, BlueAgent, EntityAdaptor, Preditor, Prey, StaticEntity, Wall, WhiteAgent

---

```
public interface Entity
extends java.io.Serializable
```

Description: The entity interface defines the methods standard to all entities in the world. Every entity in the world must extend Entity. Entity is made serializable in order for the entities to be saved to a file. Entities themselves are not intended to be sent accross the network, for that use serializable_objects.SBasicEntity or world.RemoteWorld.lookupPropertiesByID().

Since:
Sep 10, 2005
Author:
Keller

Method Summary

SerializableShape	**getBoundingSpace**() Description: Returns the bounding space of the entity.
double	**getDrawOrder**() Description: Returns the order in which the entity should be drawn on the screen.
SPoint2D	**getPosition**() Description: Returns the position of the *center* of the entity.
EntityProperties	**getProperties**() Description: Returns an instance of the EntityProperties interface specific to this entity.
java.awt.Image	**getRepresentation**()

	Description: Returns the representation for the entity.
void	**receiveCommunication**(CommunicationData commData) Description: This method provides a way for other entities to communicate with the current entity by sending CommunicationData.
void	**setBoundingSpace**(SerializableShape shape) Description: Sets the bounding space of the entity.
void	**setDrawOrder**(double drawOrder) Description: Sets the order in which the entity should be drawn on the screen.
void	**setPosition**(SPoint2D position) Description: Sets the *center* of the entity.
boolean	**setProperties**(EntityProperties properties) Description: Assigns the entity's properties to the properties paramater.
void	**setRepresentation**(java.awt.Image image) Description: Sets the representation for the entity.
void	**update**() Description: Method containing all updates to be processed on the entity each time unit.

Method Detail

getRepresentation

`java.awt.Image` **`getRepresentation`**`()`

Description: Returns the representation for the entity. This is the representation that will be displayed on the Client's display.

Returns:

The Image to represent the entity.

Since:

Sep 13, 2005

---

setRepresentation

`void` **`setRepresentation`**`(java.awt.Image image)`

Description: Sets the representation for the entity. This is the representation that will be displayed on the Client's display.

Parameters:

`image` - Image to represent the entity

Since:

Mar 12, 2006

---

getPosition

SPoint2D **`getPosition`**`()`

Description: Returns the position of the *center* of the entity. If there is no defined center to the entity, the center of the bounding rectangle of the bounding space should be used.

Returns:

An SPoint2D representing the center of the entity.

Since:

Sep 13, 2005

---

setPosition

`void` **`setPosition`**`(`SPoint2D `position)`

Description: Sets the *center* of the entity. If there is no defined center to the entity, the center of the bounding rectangle of the bounding space should be used.

Parameters:

`position` - An SPoint2D representing the center of the entity.

Since:
Mar 12, 2006

---

getBoundingSpace

`SerializableShape` **`getBoundingSpace`**`()`

Description: Returns the bounding space of the entity. The bounding space is used within to world to determine if the entity intersects with other entities in the world. Any shape may be used, however, there is a slight optimization to using a Rectangle2D. Be sure to update the bounding space whenever the entity is moved or resized.
Returns:
The bounding space of the entity, as a SerializableShape
Since:
Sep 13, 2005

---

setBoundingSpace

`void` **`setBoundingSpace`**`(SerializableShape shape)`

Description: Sets the bounding space of the entity. The bounding space is used within to world to determine if the entity intersects with other entities in the world. Any shape may be used, however, there is a slight optimization to using a Rectangle2D. Be sure to update the bounding space whenever the entity is moved or resized.
Parameters:
`shape` - The bounding space of the entity, as a SerializableShape
Since:
Mar 12, 2006

---

getDrawOrder

`double` **`getDrawOrder`**`()`

Description: Returns the order in which the entity should be drawn on the screen. Higher values will be drawn on top of lower values.
Returns:
A double representing the draw order
Since:
Sep 13, 2005

---

setDrawOrder

`void` **`setDrawOrder`**`(double drawOrder)`

Description: Sets the order in which the entity should be drawn on the screen. Higher values will be drawn on top of lower values.
Parameters:
`drawOrder` - A double representing the draw order
Since:
Mar 12, 2006

---

update

`void` **`update`**`()`

Description: Method containing all updates to be processed on the entity each time unit. The update method allows an entity to change state or interact with the world. The world will call the update method on every entity once every time unit. Any code to autonomously alter the entity at runtime should be put in the update method.
Since:
Sep 23, 2005

---

getProperties

`EntityProperties` **`getProperties()`**

Description: Returns an instance of the EntityProperties interface specific to this entity. The getPropertiesPanel method will be called on this instance by each client when the entity is clicked by the user on the client side. The properties panel will then display the controls and settings specific to that entity. This method may return null if the entity should not have properties.

Returns:

An instance of EntityProperties that defines getPropertiesPanel, or null if no properties are desired for the enity.

Since:

Sep 23, 2005

---

setProperties

`boolean` **`setProperties`**`(EntityProperties properties)`

Description: Assigns the entity's properties to the properties paramater. The setProperties method is called from a client where the user has changed a property on the GUI. The client will call this method with the new EntityProperties as a paramater. The entity must then attempt to change its properties to those contained in properties paramater. If this attempt is successful, the method returns true, otherwise false. If you entity does not support property alterations, it should return false.

Parameters:

`properties` -

Returns:

True on successful replacement of entity's properties with the properties paramater, otherwise false

Since:

Sep 23, 2005

---

receiveCommunication

`void` **`receiveCommunication`**`(CommunicationData commData)`

Description: This method provides a way for other entities to communicate with the current entity by sending CommunicationData.

Parameters:

`communicationData` -

Since:

Mar 22, 2006

---

3. ENVIRONMENT

INTERFACE ENTITYPROPERTIES

All Superinterfaces:

java.io.Serializable

All Known Implementing Classes:

EntityAdaptor.MyProperties

```
public interface EntityProperties
extends java.io.Serializable
```

Description: The properties interface is used to define properties for entities that should be sent over the network. Only necessary properties should be included as it is costly to send many properties over the network for each entity.
Important: this class should be made static, otherwise the associated entity will be sent accross the network as well - greatly reducing performace.

Since:
Sep 8, 2005
Author:
Keller
See Also:
`Entity`

Method Summary

`javax.swing.JPanel`	`getPropertiesPanel()` Description: Returns the JPanel that represents these properties.
`boolean`	`refreshValues(EntityProperties props)` Description: This method should update all properties with current values.

Method Detail

getPropertiesPanel

`javax.swing.JPanel` **`getPropertiesPanel()`**

Description: Returns the JPanel that represents these properties. May contain edit boxs, sliders, ect. to adjust properties contained in the EntityProperties instance. Any changes made to the properties by these tools will be returned to the entity by the Client using environment.Entity.setPropertiesPanel()
Returns:
The properties panel to be displayed on the client.
Since:
Sep 23, 2005
See Also:
`Entity`

refreshValues

`boolean` **`refreshValues(EntityProperties props)`**

Description: This method should update all properties with current values. The method is used to refresh the property values on the fly without having to remake the properties panel.
Parameters:
`entity` - The entity that corresponds with these properties.
Returns:
True on successful assingment, false on failure or if property refreshing is not supported.
Since:
Nov 1, 2005

4. CLIENT

CLASS LOCALWORLD

```
java.lang.Object
  └ client.LocalWorld
```

```
public class LocalWorld
extends java.lang.Object
```

Description: The LocalWorld handles all communication with the World Server. It serializes objects, as necessary, and may cache data between refresh requests that is known not to change.

Since:
Sep 10, 2005
Author:
Keller

Constructor Summary

LocalWorld() Description: Default constructor, used to initialize LocalWorld on a stand-alone system.
LocalWorld(java.lang.String worldServerURL) Description: Constructs a LocalWorld to connect to the specified World Server URL.

Method Summary

long	**addEntity**(int entityTypeIndex, SPoint2D center) Description: Adds an entity of the specified type at point center in the World.
int	**addEntityType**(java.lang.String entityClassName) Description: Adds a type of entity to the known entities in the World.
java.util.Vector <java.lang.String>	**getEntityTypes**() Description: Returns the fully qualified class names of all entities known to the world.
SPoint2D	**getInitialViewingPosition**() Description: Retuns the inital viewing position of the LocalWorld.
java.lang.String	**getWorldServerURL**() Description: Accessor Method
boolean	**loadWorld**(java.io.File file) Description: Loads the elements of the World back to a point saved in the past.
EntityProperties	**lookupPropertiesByID**(java.lang.Long id) Description: Returns the EntityProperties for an entity in the world, given its unique identifier.
boolean	**pauseWorldUpdates**() Description: Pauses all World updates.
boolean	**removeEntity**(long id) Description: Removes the specified entity from the World.
boolean	**removeEntityType**(java.lang.String entityClassName) Description: Removes a type of entity from the known entities in the World.
boolean	**resumeWorldUpdates**() Description: Resumes all World updates.
boolean	**saveWorld**(java.io.File file) Description: Saves elements of the world that will be necessary to restore the world later.
java.util.Vector <SBasicEntity>	**search**(java.awt.geom.Rectangle2D rectangle2D) Description: Searches the world within the specified rectangle and returns a Vector of all SerializableEntities intersecting that area.
java.util.Vector <SBasicEntity>	**search**(java.awt.geom.Rectangle2D rectangle2D, SEntityFunctor entityFunctor) Description: Searches the world within the specified rectangle for entities meeting the criteria of entityFucntor, and returns a Vector of matching SerializableEntities intersecting that area.
boolean	**setProperteisByID**(java.lang.Long id, EntityProperties properties) Description: Attempts to assign the given EntityProperties to the entity specified by the unique identifier.

Constructor Detail

LocalWorld

`public` **`LocalWorld`**`()`

Description: Default constructor, used to initialize LocalWorld on a stand-alone system. Only use this constructor if you do not wish to run a seperate Client and Server.

Since:

Sep 27, 2005

LocalWorld

`public` **`LocalWorld`**`(java.lang.String worldServerURL)`

Description: Constructs a LocalWorld to connect to the specified World Server URL.

Parameters:

`worldServerURL` - URL of the World Server

Since:

Sep 27, 2005

Method Detail

getInitialViewingPosition

`public` SPoint2D **`getInitialViewingPosition`**`()`

Description: Retuns the inital viewing position of the LocalWorld.

Returns:

initial default viewing position

Since:

Sep 27, 2005

---

search

`public java.util.Vector<`SBasicEntity`>`
`search``(java.awt.geom.Rectangle2D rectangle2D)`

Description: Searches the world within the specified rectangle and returns a Vector of all SerializableEntities intersecting that area.

Parameters:

`rectangle2D` - Area in world to be searched.

Returns:

Vector of SerializableEntities i specified area

Since:

Sep 27, 2005

---

search

`public java.util.Vector<`SBasicEntity`>`
`search``(java.awt.geom.Rectangle2D rectangle2D,`
SEntityFunctor `entityFunctor)`

Description: Searches the world within the specified rectangle for entities meeting the criteria of entityFucntor, and returns a Vector of matching SerializableEntities intersecting that area.

Parameters:

`rectangle2D` - Area in world to be searched.

`entityFunctor` - Instance of SEntityFunctor that defines the isAccepted method

Returns:

Vector of SerializableEntities intersecting specified area that match the criteria of the entityFunctor

Since:

Sep 27, 2005

See Also:

`EntityFunctor`

---

lookupPropertiesByID

`public EntityProperties lookupPropertiesByID(java.lang.Long id)`

Description: Returns the EntityProperties for an entity in the world, given its unique identifier.
Parameters:
`id` - The unique identifier for the desired entity.
Returns:
The properties for that entity.
Since:
Sep 27, 2005

---

setProperteisByID

`public boolean setProperteisByID(java.lang.Long id, EntityProperties properties)`

Description: Attempts to assign the given EntityProperties to the entity specified by the unique identifier. Returns true on a successful assignment, false on failure or if the entity does not support Property assignment.
Parameters:
`id` - The unique identifier for the desired entity.
`properties` - The properties to be assigned to the entity.
Returns:
True on a successful assignment, false on failure or if the entity does not support Property assignment.
Since:
Sep 27, 2005

---

getWorldServerURL

`public java.lang.String getWorldServerURL()`

Description: Accessor Method
Returns:
Returns the worldServerURL.
Since:
Sep 12, 2005

---

getEntityTypes

`public java.util.Vector<java.lang.String> getEntityTypes()`

Description: Returns the fully qualified class names of all entities known to the world. Entities may be added to the world using addEntityType() and removed using removeEntityType().
Returns:
A vector of Strings containing the fully qualified class names of all entities known to the world.
Since:
Oct 16, 2005

---

addEntityType

`public int addEntityType(java.lang.String entityClassName)`

Description: Adds a type of entity to the known entities in the World. In order to add an entity to the World, the entity's type must first be made known to the world through this mehtod. Once the entity type is known, an entity of that type may be added in the World using addEntity().
Parameters:
`entityClassName` - The *fully qualified class path* of the entity type to be added.
Returns:
The index of the entity in the entityTypes vector (as returned by getEntityTypes). Returns -1 if type was not added or if class name is incorrect. Calling this method will add the entity to the end of the vector (as per Vector.add(Object)).
Since:
Oct 16, 2005

---

removeEntityType

```
public boolean removeEntityType(java.lang.String entityClassName)
```

Description: Removes a type of entity from the known entities in the World. Use this method if an entity type will no longer be used, and no further entities of that type will be added in the World. A call to this method will alter the indexing of the entityType Vector. All entity types in the vector will be assigned new indicies.

Parameters:

`entityClassName` - The *fully qualified class path* of the entity type to be added.

Returns:

True if the entity type was successfully removed, otherwise false.

Since:

Oct 16, 2005

---

addEntity

```
public long addEntity(int entityTypeIndex,
                      SPoint2D center)
```

Description: Adds an entity of the specified type at point center in the World. The entity's type must first be added to the world using addEntityType(). To add an entity using this method the Entity type *must* have a constructor that accepts a single SPoint2D paramater. All other properties of the entity will be added with default values, as per the respective entity type's constructor.

Parameters:

`entityTypeIndex` - This is the index of the entity's type in the vector of entityTypes (as per getEntityTypes()).

`center` - The centermost point of the entity.

Returns:

The unique long ID created for the new entity. Returns -1 on error.

Since:

Oct 16, 2005

---

removeEntity

```
public boolean removeEntity(long id)
```

Description: Removes the specified entity from the World. If the entity cannot be found, this method will return false.

Parameters:

`entity` - The entity to be removed from the World.

Returns:

True if the entity was successfully removed from the World, otherwise false.

Since:

Oct 16, 2005

---

pauseWorldUpdates

```
public boolean pauseWorldUpdates()
```

Description: Pauses all World updates.

Returns:

True on success, false on failure/error.

Since:

Nov 3, 2005

---

resumeWorldUpdates

```
public boolean resumeWorldUpdates()
```

Description: Resumes all World updates.

Returns:

True on success, false on failure/error.

Since:

Nov 3, 2005

---

saveWorld

`public boolean saveWorld(java.io.File file)`

Description: Saves elements of the world that will be necessary to restore the world later. These elements will include the list of entities in the World.

Parameters:

`file` - The file to save the world properties to.

Throws:

`java.rmi.RemoteException`

Since:

Nov 15, 2005

---

loadWorld

`public boolean loadWorld(java.io.File file)`

Description: Loads the elements of the World back to a point saved in the past. For a load to be successful, the entities' serialized output stream must be the same. In other words, if any changes have been made since the file was saved to fields in the entity classes that are not declared transient, the file will not be able to be loaded.

Parameters:

`file` - The World file to load - a file creaded by saveWorld().

Throws:

`java.rmi.RemoteException`

Since:

Nov 15, 2005

---

5. CLIENT

INTERFACE REMOTECLIENT

All Known Implementing Classes:

Client

`public interface RemoteClient`

Description: This interface defines the paradigm for RemoteClients. All methods defined in this interface must be implemented to allow for network communication from the Server to the Client.

Since:

Sep 6, 2005

Author:

Keller

---

6. WORLD

INTERFACE REMOTEWORLD

All Superinterfaces:
java.rmi.Remote
All Known Implementing Classes:
World

```
public interface RemoteWorld
extends java.rmi.Remote
```

Description: The Remote World interface provides the methods that may be called on the World through RMI.

Since:
Sep 6, 2005
Author:
Keller

Method Summary

long	**addEntity**(int entityTypeIndex, SPoint2D center) Description: Adds an entity of the specified type at point center in the World.
int	**addEntityType**(java.lang.String entityClassName) Description: Adds a type of entity to the known entities in the World.
java.util.Vector<java.lang.String>	**getEntityTypes**() Description: Returns the fully qualified class names of all entities known to the world.
SPoint2D	**getInitialViewingPosition**() Description: Returns the default initial viewing position for a client.
boolean	**loadWorld**(java.io.File file) Description: Loads the elements of the World back to a point saved in the past.
EntityProperties	**lookupPropertiesByID**(long id) Description: Returns the EntityProperties for an entity in the world, given its unique identifier.
boolean	**pauseWorldUpdates**() Description: Pauses all World updates.
boolean	**removeEntity**(long id) Description: Removes the specified entity from the World.
boolean	**removeEntityType**(java.lang.String entityClassName) Description: Removes a type of entity from the known entities in the World.
boolean	**resumeWorldUpdates**() Description: Resumes all World updates.
boolean	**saveWorld**(java.io.File file) Description: Saves elements of the world that will be necessary to restore the world later.
java.util.Vector<SBasicEntity>	**search**(SRectangle2D rectangle2D) Description: Searches the world within the specified rectangle and returns a Vector of all SerializableEntities intersecting that area.
java.util.Vector<SBasicEntity>	**search**(SRectangle2D rectangle2D, SEntityFunctor entityFunctor) Description: Searches the world within the specified rectangle for entities meeting the criteria of entityFucntor, and returns a Vector of matching SerializableEntities intersecting that area.
boolean	**setProperteisByID**(long id, EntityProperties properties) Description: Attempts to assign the given EntityProperties to the entity specified by the unique identifier.

Method Detail

getInitialViewingPosition

`SPoint2D` **`getInitialViewingPosition`**`()`
`throws java.rmi.RemoteException`

Description: Returns the default initial viewing position for a client. This is the 2D point that the client's screen should be centered on in the world upon initialization.

Returns:
SPoint2D represent initial viewing position

Throws:
`java.rmi.RemoteException`

Since:
Sep 27, 2005

---

search

`java.util.Vector<SBasicEntity>` **`search`**`(SRectangle2D rectangle2D)`
`throws java.rmi.RemoteException`

Description: Searches the world within the specified rectangle and returns a Vector of all SerializableEntities intersecting that area.

Parameters:
`rectangle2D` - Area in world to be searched.

Returns:
Vector of SerializableEntities i specified area

Throws:
`java.rmi.RemoteException`

Since:
Sep 27, 2005

---

search

`java.util.Vector<SBasicEntity>` **`search`**`(SRectangle2D rectangle2D,`
`SEntityFunctor entityFunctor)`
`throws java.rmi.RemoteException`

Description: Searches the world within the specified rectangle for entities meeting the criteria of entityFucntor, and returns a Vector of matching SerializableEntities intersecting that area.

Parameters:
`rectangle2D` - Area in world to be searched.
`entityFunctor` - Instance of SEntityFunctor that defines the isAccepted method

Returns:
Vector of SerializableEntities intersecting specified area that match the criteria of the entityFunctor

Throws:
`java.rmi.RemoteException`

Since:
Sep 27, 2005

See Also:
`EntityFunctor`

---

lookupPropertiesByID

`EntityProperties` **`lookupPropertiesByID`**`(long id)`
`throws java.rmi.RemoteException`

Description: Returns the EntityProperties for an entity in the world, given its unique identifier.

Parameters:
`id` - The unique identifier for the desired entity.

Returns:
The properties for that entity.

Throws:
`java.rmi.RemoteException`

Since:
Sep 27, 2005

---

setProperteisByID

```
boolean setProperteisByID(long id,
                          EntityProperties properties)
                          throws java.rmi.RemoteException
```

Description: Attempts to assign the given EntityProperties to the entity specified by the unique identifier. Returns true on a successful assignment, false on failure or if the entity does not support Property assignment.

Parameters:
`id` - The unique identifier for the desired entity.
`properties` - The properties to be assigned to the entity.

Returns:
True on a successful assignment, false on failure or if the entity does not support Property assignment.

Throws:
`java.rmi.RemoteException`

Since:
Sep 27, 2005

---

getEntityTypes

```
java.util.Vector<java.lang.String> getEntityTypes()
                                                   throws java.rmi.RemoteException
```

Description: Returns the fully qualified class names of all entities known to the world. Entities may be added to the world using addEntityType() and removed using removeEntityType().

Returns:
A vector of Strings containing the fully qualified class names of all entities known to the world.

Throws:
`java.rmi.RemoteException`

Since:
Oct 16, 2005

---

addEntityType

```
int addEntityType(java.lang.String entityClassName)
                  throws java.rmi.RemoteException
```

Description: Adds a type of entity to the known entities in the World. In order to add an entity to the World, the entity's type must first be made known to the world through this mehtod. Once the entity type is known, an entity of that type may be added in the World using addEntity().

Parameters:
`entityClassName` - The *fully qualified class path* of the entity type to be added.

Returns:
The index of the entity in the entityTypes vector (as returned by getEntityTypes). Returns -1 if type was not added or if class name is incorrect. Calling this method will add the entity to the end of the vector (as per Vector.add(Object)).

Throws:
`java.rmi.RemoteException`

Since:
Oct 16, 2005

---

removeEntityType

```
boolean removeEntityType(java.lang.String entityClassName)
                         throws java.rmi.RemoteException
```

Description: Removes a type of entity from the known entities in the World. Use this method if an entity type will no longer be used, and no further entities of that type will be added in the World. A call to this method will alter the indexing of the entityType Vector. All entity types in the vector will be assigned new indicies.

Parameters:
`entityClassName` - The *fully qualified class path* of the entity type to be added.

Returns:
True if the entity type was successfully removed, otherwise false.

Throws:
`java.rmi.RemoteException`

Since:
Oct 16, 2005

---

addEntity

```
long addEntity(int entityTypeIndex,
               SPoint2D center)
               throws java.rmi.RemoteException
```

Description: Adds an entity of the specified type at point center in the World. The entity's type must first be added to the world using addEntityType(). To add an entity using this method the Entity type *must* have a constructor that accepts a single Point2D paramater. All other properties of the entity will be added with default values, as per the respective entity type's constructor.

Parameters:
`entityTypeIndex` - This is the index of the entity's type in the vector of entityTypes (as per getEntityTypes()).
`center` - The centermost point of the entity.

Returns:
The unique long ID created for the new entity. Returns -1 on error.

Throws:
`java.rmi.RemoteException`

Since:
Oct 16, 2005

---

removeEntity

```
boolean removeEntity(long id)
                     throws java.rmi.RemoteException
```

Description: Removes the specified entity from the World. If the entity cannot be found, this method will return false.

Parameters:
`entity` - The entity to be removed from the World.

Returns:
True if the entity was successfully removed from the World, otherwise false.

Throws:
`java.rmi.RemoteException`

Since:
Oct 16, 2005

---

pauseWorldUpdates

```
boolean pauseWorldUpdates()
                         throws java.rmi.RemoteException
```

Description: Pauses all World updates.

Returns:
True on success, false on failure/error.

Throws:
`java.rmi.RemoteException`

Since:
Nov 3, 2005

---

resumeWorldUpdates

```
boolean resumeWorldUpdates()
                          throws java.rmi.RemoteException
```

Description: Resumes all World updates.

Returns:
True on success, false on failure/error.

Throws:
`java.rmi.RemoteException`

Since:
Nov 3, 2005

---

saveWorld

```
boolean saveWorld(java.io.File file)
                  throws java.rmi.RemoteException
```

Description: Saves elements of the world that will be necessary to restore the world later. These elements will include the list of entities in the World.

Parameters:

`file` - The file to save the world properties to.

Returns:

True on successful save, otherwise false.

Throws:

`java.rmi.RemoteException`

Since:

Nov 15, 2005

---

loadWorld

```
boolean loadWorld(java.io.File file)
                  throws java.rmi.RemoteException
```

Description: Loads the elements of the World back to a point saved in the past. For a load to be successful, the entities' serialized output stream must be the same. In other words, if any changes have been made since the file was saved to fields in the entity classes that are not declared transient, the file will not be able to be loaded.

Parameters:

`file` - The World file to load - a file creaded by saveWorld().

Returns:

True on successful load, otherwise false.

Throws:

`java.rmi.RemoteException`

Since:

Nov 15, 2005

---

7. WORLD

CLASS WORLD

```
java.lang.Object
  └ java.rmi.server.RemoteObject
      └ java.rmi.server.RemoteServer
          └ java.rmi.server.UnicastRemoteObject
              └ world.World
```

All Implemented Interfaces:

java.io.Serializable, java.rmi.Remote, RemoteWorld

```
public class World
extends java.rmi.server.UnicastRemoteObject
implements RemoteWorld
```

Description: The World class is responsible for managing all entities and updates within the world. The class is a singleton that exists only on the server (or on the running machine in the stand-alone implementation).

Since:

Sep 10, 2005

Author:

Keller

See Also:

Serialized Form

Method Summary

long	**addEntity**(Entity entity) Description: Adds an entity to the pending list of entities that will be added to the world after the next update.
long	**addEntity**(int entityTypeIndex, SPoint2D center) Description: Adds an entity of the specified type at point center in the World.
int	**addEntityType**(java.lang.String entityClassName) Description: Adds a type of entity to the known entities in the World.
java.util.Vector <java.lang.String>	**getEntityTypes**() Description: Returns the fully qualified class names of all entities known to the world.
SPoint2D	**getInitialViewingPosition**() Description: Returns the default initial viewing position for a client.
static World	**getInstance**() Description: Returns the singleton instance of World.
int	**getUpdatePeriod**() Description: Accessor Method, returns the period between each World update.
boolean	**loadWorld**(java.io.File file) Description: Loads the elements of the World back to a point saved in the past.
EntityProperties	**lookupPropertiesByID**(long id) Description: Returns the EntityProperties for an entity in the world, given its unique identifier.
static void	**main**(java.lang.String[] args) Description: Main method used to run the server.
boolean	**pauseWorldUpdates**() Description: Pauses all World updates.
boolean	**removeEntity**(Entity entity) Description: Marks an entity for removal, causing it to be reomved form the World after the next update.
boolean	**removeEntity**(long id) Description: Removes the specified entity from the World.
boolean	**removeEntityType**(java.lang.String entityClassName) Description: Removes a type of entity from the known entities in the World.
boolean	**resumeWorldUpdates**() Description: Resumes all World updates.
boolean	**saveWorld**(java.io.File file) Description: Saves elements of the world that will be necessary to restore the world later.
java.util.Vector <Entity>	**search**(SerializableShape shape) Description: Returns all entities intersecting the specified shape.
java.util.Vector <Entity>	**search**(SerializableShape shape, EntityFunctor entityFunctor) Description: Returns entities intersecting the specified shape that are accepted by the entityFunctor.
java.util.Vector <Entity>	**search**(SPoint2D center, double radius) Description: Returns all entities intersecting the circle with the specified center and radius.
java.util.Vector <Entity>	**search**(SPoint2D center, double radius, EntityFunctor entityFunctor) Description: Returns all entities intersecting the circle with the specified center and radius.
java.util.Vector <SBasicEntity>	**search**(SRectangle2D rectangle2D) Description: Searches the world within the specified rectangle and returns a Vector of all SerializableEntities intersecting that area.

java.util.Vector <SBasicEntity>	**search**(SRectangle2D rectangle2D, SEntityFunctor entityFunctor) Description: Searches the world within the specified rectangle for entities meeting the criteria of entityFucntor, and returns a Vector of matching SerializableEntities intersecting that area.
boolean	**setProperteisByID**(long id, EntityProperties properties) Description: Attempts to assign the given EntityProperties to the entity specified by the unique identifier.
boolean	**startWorldUpdates**()

Method Detail

main

`public static void main(java.lang.String[] args)`

Description: Main method used to run the server. This method will intialize the World on the server specified in the command line arguments.

Parameters:

`args` - [0]: The string URL that the server is running on. For example, "rmi//computername:Port/WorldServer"

Since:

Sep 27, 2005

---

addEntity

`public long addEntity(Entity entity)`

Description: Adds an entity to the pending list of entities that will be added to the world after the next update. Creates a new unique long ID for the entity and returns it.

Parameters:

`entity` - The entity to add.

Returns:

The long ID of the new entity.

Since:

Sep 27, 2005

---

removeEntity

`public boolean removeEntity(Entity entity)`

Description: Marks an entity for removal, causing it to be reomved form the World after the next update.

Parameters:

`entity` - Entity to be removed.

Returns:

True if successeful, otherwise false.

Since:

Oct 16, 2005

---

search

```
public java.util.Vector<Entity> search(SPoint2D center,
                                       double radius)
```

Description: Returns all entities intersecting the circle with the specified center and radius.

Parameters:

`center` - The center of the circle to search.

`radius` - The radius of the circle to search.

Returns:

A vector of entites intersecting the specified circle.

Since:

Sep 27, 2005

---

search

```
public java.util.Vector<Entity> search(SPoint2D center,
                                       double radius,
                                       EntityFunctor entityFunctor)
```

Description: Returns all entities intersecting the circle with the specified center and radius.
Parameters:
`center` - The center of the circle to search.
`radius` - The radius of the circle to search.
`entityFunctor` - An instance of entityFunctor that defines the isAccepted method.
Returns:
A vector of entites intersecting the specified circle.
Since:
Sep 27, 2005

---

search

```
public java.util.Vector<Entity> search(SerializableShape shape)
```

Description: Returns all entities intersecting the specified shape. There is a slight performace benifit if the shape is a Rectangle2D.
Parameters:
`shape` - The shape to search.
Returns:
A vector of entites intersecting the specified shape.
Since:
Sep 27, 2005

---

search

```
public java.util.Vector<Entity> search(SerializableShape shape,
                                       EntityFunctor entityFunctor)
```

Description: Returns entities intersecting the specified shape that are accepted by the entityFunctor. There is a slight performace benifit if the shape is a Rectangle2D.
Parameters:
`shape` - The shape to search.
`entityFunctor` - An instance of entityFunctor that defines the isAccepted method.
Returns:
A vector of entites intersecting the specified shape that are accpted by the entityFunctor.
Since:
Sep 27, 2005
See Also:
`EntityFunctor`

---

search

```
public java.util.Vector<SBasicEntity> search(SRectangle2D rectangle2D)
                                     throws java.rmi.RemoteException
```

Description copied from interface: `RemoteWorld`
Description: Searches the world within the specified rectangle and returns a Vector of all SerializableEntities intersecting that area.
Specified by:
`search` in interface `RemoteWorld`
Parameters:
`rectangle2D` - Area in world to be searched.
Returns:
Vector of SerializableEntities i specified area
Throws:
`java.rmi.RemoteException`

---

search

```
public java.util.Vector<SBasicEntity> search(SRectangle2D rectangle2D,
                                             SEntityFunctor entityFunctor)
                                      throws java.rmi.RemoteException
```

Description copied from interface: RemoteWorld
Description: Searches the world within the specified rectangle for entities meeting the criteria of entityFucntor, and returns a Vector of matching SerializableEntities intersecting that area.
Specified by:
search in interface RemoteWorld
Parameters:
`rectangle2D` - Area in world to be searched.
`entityFunctor` - Instance of SEntityFunctor that defines the isAccepted method
Returns:
Vector of SerializableEntities intersecting specified area that match the criteria of the entityFunctor
Throws:
`java.rmi.RemoteException`
See Also:
`EntityFunctor`

---

getInitialViewingPosition

```
public SPoint2D getInitialViewingPosition()
                                 throws java.rmi.RemoteException
```

Description copied from interface: RemoteWorld
Description: Returns the default initial viewing position for a client. This is the 2D point that the client's screen should be centered on in the world upon initialization.
Specified by:
getInitialViewingPosition in interface RemoteWorld
Returns:
SPoint2D represent initial viewing position
Throws:
`java.rmi.RemoteException`

---

lookupPropertiesByID

```
public EntityProperties lookupPropertiesByID(long id)
                                      throws java.rmi.RemoteException
```

Description copied from interface: RemoteWorld
Description: Returns the EntityProperties for an entity in the world, given its unique identifier.
Specified by:
lookupPropertiesByID in interface RemoteWorld
Parameters:
`id` - The unique identifier for the desired entity.
Returns:
The properties for that entity.
Throws:
`java.rmi.RemoteException`

---

setProperteisByID

```
public boolean setProperteisByID(long id,
                                 EntityProperties properties)
                          throws java.rmi.RemoteException
```

Description copied from interface: RemoteWorld
Description: Attempts to assign the given EntityProperties to the entity specified by the unique identifier. Returns true on a successful assignment, false on failure or if the entity does not support Property assignment.
Specified by:
setProperteisByID in interface RemoteWorld
Parameters:
`id` - The unique identifier for the desired entity.
`properties` - The properties to be assigned to the entity.

Returns:
True on a successful assignment, false on failure or if the entity does not support Property assignment.
Throws:
`java.rmi.RemoteException`

---

getEntityTypes

```
public java.util.Vector<java.lang.String> getEntityTypes()
                                                    throws java.rmi.RemoteException
```

Description copied from interface: `RemoteWorld`
Description: Returns the fully qualified class names of all entities known to the world. Entities may be added to the world using addEntityType() and removed using removeEntityType().
Specified by:
`getEntityTypes` in interface `RemoteWorld`
Returns:
A vector of Strings containing the fully qualified class names of all entities known to the world.
Throws:
`java.rmi.RemoteException`

---

addEntityType

```
public int addEntityType(java.lang.String entityClassName)
                  throws java.rmi.RemoteException
```

Description copied from interface: `RemoteWorld`
Description: Adds a type of entity to the known entities in the World. In order to add an entity to the World, the entity's type must first be made known to the world through this mehtod. Once the entity type is known, an entity of that type may be added in the World using addEntity().
Specified by:
`addEntityType` in interface `RemoteWorld`
Parameters:
`entityClassName` - The *fully qualified class path* of the entity type to be added.
Returns:
The index of the entity in the entityTypes vector (as returned by getEntityTypes). Returns -1 if type was not added or if class name is incorrect. Calling this method will add the entity to the end of the vector (as per Vector.add(Object)).
Throws:
`java.rmi.RemoteException`

---

removeEntityType

```
public boolean removeEntityType(java.lang.String entityClassName)
                         throws java.rmi.RemoteException
```

Description copied from interface: `RemoteWorld`
Description: Removes a type of entity from the known entities in the World. Use this method if an entity type will no longer be used, and no further entities of that type will be added in the World. A call to this method will alter the indexing of the entityType Vector. All entity types in the vector will be assigned new indicies.
Specified by:
`removeEntityType` in interface `RemoteWorld`
Parameters:
`entityClassName` - The *fully qualified class path* of the entity type to be added.
Returns:
True if the entity type was successfully removed, otherwise false.
Throws:
`java.rmi.RemoteException`

---

addEntity

```
public long addEntity(int entityTypeIndex,
                      SPoint2D center)
               throws java.rmi.RemoteException
```

Description copied from interface: `RemoteWorld`
Description: Adds an entity of the specified type at point center in the World. The entity's type must first be added to the world using addEntityType(). To add an entity using this method the Entity type *must* have a constructor that

accepts a single Point2D paramater. All other properties of the entity will be added with default values, as per the respective entity type's constructor.
Specified by:
addEntity in interface RemoteWorld
Parameters:
`entityTypeIndex` - This is the index of the entity's type in the vector of entityTypes (as per getEntityTypes()).
`center` - The centermost point of the entity.
Returns:
The unique long ID created for the new entity. Returns -1 on error.
Throws:
`java.rmi.RemoteException`

---

removeEntity

```
public boolean removeEntity(long id)
                     throws java.rmi.RemoteException
```

Description copied from interface: RemoteWorld
Description: Removes the specified entity from the World. If the entity cannot be found, this method will return false.
Specified by:
removeEntity in interface RemoteWorld
Returns:
True if the entity was successfully removed from the World, otherwise false.
Throws:
`java.rmi.RemoteException`

---

pauseWorldUpdates

```
public boolean pauseWorldUpdates()
                           throws java.rmi.RemoteException
```

Description copied from interface: RemoteWorld
Description: Pauses all World updates.
Specified by:
pauseWorldUpdates in interface RemoteWorld
Returns:
True on success, false on failure/error.
Throws:
`java.rmi.RemoteException`

---

resumeWorldUpdates

```
public boolean resumeWorldUpdates()
                            throws java.rmi.RemoteException
```

Description copied from interface: RemoteWorld
Description: Resumes all World updates.
Specified by:
resumeWorldUpdates in interface RemoteWorld
Returns:
True on success, false on failure/error.
Throws:
`java.rmi.RemoteException`

---

startWorldUpdates

```
public boolean startWorldUpdates()
```

---

getInstance

```
public static World getInstance()
```

Description: Returns the singleton instance of World.
Returns:
The singleton instance of World.
Since:
Sep 2, 2005

---

getUpdatePeriod

```
public int getUpdatePeriod()
```

Description: Accessor Method, returns the period between each World update.
Returns:
Returns the Update_Period.
Since:
Sep 12, 2005

---

saveWorld

```
public boolean saveWorld(java.io.File file)
                  throws java.rmi.RemoteException
```

Description copied from interface: RemoteWorld
Description: Saves elements of the world that will be necessary to restore the world later. These elements will include the list of entities in the World.
Specified by:
saveWorld in interface RemoteWorld
Parameters:
`file` - The file to save the world properties to.
Returns:
True on successful save, otherwise false.
Throws:
`java.rmi.RemoteException`

---

loadWorld

```
public boolean loadWorld(java.io.File file)
                  throws java.rmi.RemoteException
```

Description copied from interface: RemoteWorld
Description: Loads the elements of the World back to a point saved in the past. For a load to be successful, the entities' serialized output stream must be the same. In other words, if any changes have been made since the file was saved to fields in the entity classes that are not declared transient, the file will not be able to be loaded.
Specified by:
loadWorld in interface RemoteWorld
Parameters:
`file` - The World file to load - a file creaded by saveWorld().
Returns:
True on successful load, otherwise false.
Throws:
`java.rmi.RemoteException`

BIBLIOGRAPHY

"Application Program Interface." Wikipedia. 26 Mar. 2006 <http://en.wikipedia.org/wiki/Application-programming_interface>.

"Artificial Intelligence." Wikipedia. 26 Mar. 2006 <http://en.wikipedia.org/wiki/Artificial_intelligence>.

Bloch, Joshua. Effective Java. 1st ed. Boston: Addison-Wesley, 2001.

Bloch, Joshua. How to Design a Good API and Why It Matters. Library-Centric Software Design (LCSD). 2005. 26 Mar. 2006 <http://lcsd05.cs.tamu.edu/slides/keynote.pdf>.

Buckland, Mat. Programming Game AI by Example. 1st ed. Plano, TX: Wordware, 2005. 85.

Carnegie Mellon Robot Soccer. Carnegie Mellon University. 26 Mar. 2006 <http://www.cs.cmu.edu/~robosoccer/main/>.

Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns. 1st ed. Boston: Addison-Wesley, 1995.

Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns. 1st ed. Boston: Addison-Wesley, 1995.

"Jade - Java Agent DEvelopment Framework." Jade. 26 Mar. 2006 <http://jade.cselt.it/>.

"Multi-Agent Systems." Wikipedia. 26 Mar. 2006

<http://en.wikipedia.org/wiki/Multi-agent_systems>.

REFERENCES

1. "JACK Intelligent Agents, Software Agent System." The Agent Oriented Software Group. 3 Apr. 2006 <http://www.agent-software.com/>.

2. "Jade - Java Agent DEvelopment Framework." Jade. 26 Mar. 2006 <http://jade.cselt.it/>.

3. Cognitive Agent Architecture Open Source Project Site. DARPA. 26 Mar. 2006 <http://www.cougaar.org/>.

4. "Artificial Intelligence." Wikipedia. 26 Mar. 2006 <http://en.wikipedia.org/wiki/Artificial_intelligence>.

5. "Multi-Agent Systems." Wikipedia. 26 Mar. 2006 <http://en.wikipedia.org/wiki/Multi-agent_systems>.

6. Buckland, Mat. Programming Game AI by Example. 1st ed. Plano, TX: Wordware, 2005. 85.

7. Carnegie Mellon Robot Soccer. Carnegie Mellon University. 26 Mar. 2006 <http://www.cs.cmu.edu/~robosoccer/main/>.

8. Shoham, Yoav. "Agent-Oriented Programming." Elsevier 60.1 (1993): 51-92.

9. "Multi-Agent Systems." Wikipedia. 26 Mar. 2006 <http://en.wikipedia.org/wiki/Multi-agent_systems>.

10. Carnegie Mellon Robot Soccer. Carnegie Mellon University. 26 Mar. 2006 <http://www.cs.cmu.edu/~robosoccer/main/>.

11. Bloch, Joshua. How to Design a Good API and Why It Matters. Library-Centric Software Design (LCSD). 2005. 26 Mar. 2006 <http://lcsd05.cs.tamu.edu/slides/keynote.pdf>.

12. Bloch, Joshua. Effective Java. 1st ed. Boston: Addison-Wesley, 2001.

13. Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns. 1st ed. Boston: Addison-Wesley, 1995.

14. Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns. 1st ed. Boston: Addison-Wesley, 1995. 185.

15. "Jade - Java Agent DEvelopment Framework." Jade. 26 Mar. 2006 <http://jade.cselt.it/>.

16. "Jade - Java Agent DEvelopment Framework." Jade. 26 Mar. 2006 <http://jade.cselt.it/>.

17. Foundation for Intelligent Physical Agents. IEEE Computer Society. 26 Mar. 2006 <http://www.fipa.org/>.

18. "Jade - Java Agent DEvelopment Framework." Jade. 26 Mar. 2006 <http://jade.cselt.it/>.

19. Cognitive Agent Architecture Open Source Project Site. DARPA. 26 Mar. 2006 <http://www.cougaar.org/>.

20. "Swarm Development Group." Swarm 2.2. 29 Mar. 2006 <http://wiki.swarm.org>.

21. Repast Agent Simulation Toolkit. 29 Mar. 2006 <http://repast.sourceforge.net>.

22. Repast Agent Simulation Toolkit. 29 Mar. 2006 <http://repast.sourceforge.net>.

23. Jess. 29 Mar. 2006 <http://www.jessrules.com/>.