



NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

**MAJIC: A JAVA APPLICATION FOR CONTROLLING
MULTIPLE, HETEROGENEOUS ROBOTIC AGENTS**

by

Gregory P. Ball

September 2007

Thesis Advisor:
Thesis Co-advisor:

Craig Martell
Kevin Squire

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			<i>Form Approved OMB No. 0704-0188</i>	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.				
1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE September 2007	3. REPORT TYPE AND DATES COVERED Master's Thesis	
4. TITLE AND SUBTITLE MAJIC: A Java Application for Controlling Multiple, Heterogeneous Robotic Agents			5. FUNDING NUMBERS	
6. AUTHOR(S) Gregory P. Ball				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			10. SPONSORING/MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.				
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited			12b. DISTRIBUTION CODE A	
13. ABSTRACT (maximum 200 words) Current capability to command and control a team of heterogeneous robotic agents is limited by proprietary command formats and operating systems. A specific challenge in this context is the specification, the programming, and the testing of software for such a wide variety of mobile robot teams. This work explores the applicability of an application program interface (API), called the Multi-Agent Java Interface Controller (MAJIC), that supports command, control, and coordination of heterogeneous robot teams. MAJIC encapsulates scripted commands, pre-programmed behaviors, and simultaneous, multi-agent control. By exploiting the powerful techniques of polymorphism and object-oriented programming, a generic MajicBot class will provide the necessary level of abstraction between the user and the proprietary architectures. Utilizing the technique of inheritance, future NPS students will be able to extend the generic class in order to easily add new robot-specific libraries. Students will also be able to utilize the existing libraries to program and test their own robot behaviors in real-world environments utilizing the MAJIC package. A final display of the versatility and power of programming behaviors within the MAJIC software architecture is demonstrated by a series of example programs conducted on a team of robots consisting of a Sony Aibo, a Mobile Robots Pioneer, and a K-Team Hemisson.				
14. SUBJECT TERMS Robotics, control architecture, heterogeneous control, abstraction, object-oriented programming, Java, UML			15. NUMBER OF PAGES 157	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UU	

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

**MAJIC: A JAVA APPLICATION FOR CONTROLLING MULTIPLE,
HETEROGENEOUS ROBOTIC AGENTS**

Gregory P. Ball
Lieutenant, United States Navy
B.S., Ferris State University, 1991

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

**NAVAL POSTGRADUATE SCHOOL
September 2007**

Author: Gregory P. Ball

Approved by: Craig Martell
Thesis Advisor

Kevin Squire
Thesis Co-advisor

Peter Denning
Chairman, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Current capability to command and control a team of heterogeneous robotic agents is limited by proprietary command formats and operating systems. A specific challenge in this context is the specification, the programming, and the testing of software for such a wide variety of mobile robot teams. This work explores the applicability of an application program interface (API), called Multi-Agent Java Interface Controller (MAJIC), that supports command, control, and coordination of heterogeneous robot teams. MAJIC encapsulates scripted commands, pre-programmed behaviors, and simultaneous, multi-agent control.

By exploiting the powerful techniques of polymorphism and object-oriented programming, a generic MajicBot class will provide the necessary level of abstraction between the user and the proprietary architectures. Utilizing the technique of inheritance, future NPS students will be able to extend the generic class to easily add new robot-specific libraries. Students will also be able to utilize the existing libraries to program and test their own robot behaviors in real-world environments utilizing the MAJIC package.

A final display of the versatility and power of programming behaviors within the MAJIC software architecture is demonstrated by a series of example programs conducted on a team of robots consisting of a Sony Aibo, a Mobile Robots Pioneer, and a Narrow Roads Hemisson.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

I.	INTRODUCTION.....	1
A.	OVERVIEW	1
B.	MOTIVATION	2
C.	OBJECTIVES	3
D.	SCOPE	4
E.	THESIS ORGANIZATION.....	4
II.	SOFTWARE VISION DOCUMENT.....	5
A.	INTRODUCTION.....	5
1.	Purpose of the Vision Document	5
2.	Application Overview	5
B.	USER DESCRIPTION	5
1.	User Demographics.....	5
2.	User Profiles	5
3.	Users Environment	6
4.	Key User Needs	6
a.	Communication.....	6
b.	Control.....	6
c.	Coordination.....	7
5.	Alternatives.....	7
C.	APPLICATION OVERVIEW	7
1.	Application Perspective	7
2.	Application Position Statement	8
3.	Application Capabilities Summary	8
4.	Assumptions and Dependencies.....	8
D.	APPLICATION FEATURES	9
1.	Button Toolbar	9
2.	Robot Configurability.....	9
3.	Scripted Language	9
4.	Command Line.....	9
5.	Behavioral Programming	9
6.	Informational Displays	10
7.	File I/O	10
E.	USE CASE DIAGRAMS.....	10
F.	APPLICATION ALTERNATIVES	24
III.	SYSTEM DESIGN.....	27
A.	INTRODUCTION.....	27
B.	SYSTEM ARCHITECTURE	27
1.	Overview	27
a.	Style of Architecture.....	27
b.	Goals of Architecture	27
c.	Style of Architecture.....	29

	d.	<i>Goals of Architecture</i>	29
	2.	Components	30
	a.	<i>Presentation/UI Components</i>	31
	b.	<i>Application Logic Components</i>	31
	3.	Integration	32
C.		GRAPHICAL USER INTERFACE	33
	1.	Overview	33
	2.	Content Model	34
D.		BEHAVIORAL DESIGN	35
	1.	Domain Model	35
	2.	Boundary Use Cases	37
	3.	Sequence Diagrams	39
	4.	Operational Contracts	47
E.		OBJECT DESIGN	50
	1.	Class Diagrams	50
	a.	<i>Startup Class Diagram</i>	52
	b.	<i>Add Class Diagram</i>	53
	c.	<i>Load Action Class Diagram</i>	54
	d.	<i>Parse Command Class Diagram</i>	55
	2.	Class Descriptions	56
	a.	<i>The MajicFrame Class</i>	56
	b.	<i>The MajicParser Class</i>	59
	c.	<i>The MajicBot Class</i>	61
	d.	<i>The MajicAct Class</i>	64
	3.	Class Extensions	66
	a.	<i>The MajicHemisson Class</i>	66
	b.	<i>The MajicAibo Class</i>	70
	c.	<i>The MajicPioneer Class</i>	74
F.		DESIGN ALTERNATIVES	78
IV.		IMPLEMENTATION	79
A.		OVERVIEW	79
B.		JAVA	79
C.		AIBO	81
	1.	R-Code SDK	81
	2.	Open-R SDK	82
	3.	Universal Realtime Behavior Interface	83
D.		ARIA	84
E.		INSTALLATION GUIDE	85
	1.	URBI Installation for Aibo	86
	2.	Javax.comm installation for Hemisson	86
	3.	ARIA installation for the Pioneer	87
F.		USER'S GUIDE	87
	1.	MAJIC Main Screen	88
	a.	<i>Button Panel</i>	89
	b.	<i>Robot Display Area</i>	89

	c.	<i>Command Line</i>	89
	d.	<i>Message Area</i>	90
2.		Adding a Robot to the Team	90
3.		Passing Commands to the Robot	92
	a.	<i>Robot ID</i>	94
	b.	<i>The MOVE Command</i>	94
	c.	<i>The TURN Command</i>	94
	d.	<i>The GET Command</i>	94
	e.	<i>The SET Command</i>	95
	f.	<i>Command Line Example</i>	96
4.		Invoking the Help Screen	96
5.		Saving a MAJIC Session	97
6.		Saving a Parameter Log	98
7.		Loading Actions on the Robot	98
8.		Removing a Robot from the Team	100
9.		Quitting the MAJIC Application	102
G.		PROGRAMMER'S GUIDE	103
	1.	Stand-alone Programming	103
	2.	Creating Robot Libraries	106
	3.	Performing a Majic Act	111
	a.	<i>The MajicAct Class</i>	111
	b.	<i>The AiboSquare Class Extension</i>	112
	c.	<i>Creating a Majic Act with a Session File</i>	114
	d.	<i>Serializing a MajicAct Object</i>	115
V.		RESULTS	117
	A.	OVERVIEW	117
	B.	INDIVIDUAL ROBOT PROGRAMMING	117
		1. MAJIC vs Proprietary Programming with Aibo	118
		2. MAJIC vs Proprietary Programming with Pioneer	124
	C.	PROGRAMMING HETEROGENEOUS ROBOT TEAMS	129
VI.		CONCLUSIONS AND RECOMMENDATIONS	135
	A.	RESEARCH CONCLUSIONS	135
	B.	RECOMMENDATIONS FOR FUTURE WORK	135
		LIST OF REFERENCES	137
		INITIAL DISTRIBUTION LIST	139

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

Figure 1.	U.S. Army soldiers arrive with a robot, left, to remove explosive devices from a street in the center of Baghdad, Iraq, Sunday, May 9, 2004. (From: AP Photo/Mohammed Uraibi).....	1
Figure 2.	MOCU (from SPAWAR San Diego).....	2
Figure 3.	Use Case Model.	11
Figure 4.	MAJIC Architecture.....	28
Figure 5.	UML Component Model.....	30
Figure 6.	Prototype of Main GUI Display.....	33
Figure 7.	Domain Model UML.	36
Figure 8.	UC-5 Application Startup Sequence Diagram.	39
Figure 9.	UC-6 Application Shutdown Sequence Diagram.	40
Figure 10.	UC-1 Add Bot Sequence Diagram.....	41
Figure 11.	UC-2 Remove Bot Sequence Diagram.	42
Figure 12.	UC-3 Load Action Sequence Diagram.	43
Figure 13.	UC-4a Move Bot Sequence Diagram.	44
Figure 14.	UC-4b Turn Bot Sequence Diagram.....	45
Figure 15.	UC-4c Set Bot Parameters Sequence Diagram.....	46
Figure 16.	UC-4d Get Bot Parameters Sequence Diagram.....	47
Figure 17.	Overall Class Diagram.....	51
Figure 18.	Startup Class Diagram.	52
Figure 19.	Add Class Diagram.....	53
Figure 20.	Load Action Class Diagram.....	54
Figure 21.	Parse Command Class Diagram.....	55
Figure 22.	MajicFrame Class Model.....	57
Figure 23.	MajicParser Class Model.	60
Figure 24.	MajicBot Class Model.	62
Figure 25.	MajicAct Class Model.	65
Figure 26.	MajicHemisson Class Model.	67
Figure 27.	MajicAiBo Class Model.	72
Figure 28.	MajicPioneer Class Model.....	76
Figure 29.	MAJIC Main Screen.	88
Figure 30.	Robot Selection Screen.....	90
Figure 31.	Connection Input Screen.....	91
Figure 32.	Adding an AIBO.....	92
Figure 33.	MAJIC Command Line Example.	95
Figure 34.	Aibo Help Screen.....	97
Figure 35.	Robot Selection for Load Action.	99
Figure 36.	Load Action Selection Screen.....	100
Figure 37.	Removing a Robot.	101
Figure 38.	Bot 1 is Dead.....	102
Figure 39.	The WanderDog Program.....	105
Figure 40.	The Generic MajicBot Template.	108

Figure 41.	The Abstract MajicBot Class	110
Figure 42.	The MajicAct Class.....	112
Figure 43.	MajicAct AiboSquare Example	113
Figure 44.	Aibo Session Sample.	115
Figure 45.	Majic Act Maker.	116
Figure 46.	Aibo's Layers of Abstraction.....	118
Figure 47.	WanderDog with Line Numbers.....	120
Figure 48.	URBI WanderDog.	122
Figure 49.	URBI WanderDog (cont.).....	123
Figure 50.	Pioneer Architecture.	125
Figure 51.	Pioneer Program using ARIA.	126
Figure 52.	Pioneer Program using ARIA (cont.).....	127
Figure 53.	Pioneer Program using MAJIC.....	128
Figure 54.	GreatRace Example.	130
Figure 55.	GreatRace Example (cont.).....	131
Figure 56.	GreatRacePioneerServer Example.....	132
Figure 57.	GreatRacePioneerServer (cont.).	133
Figure 58.	MajicRace Example.	134

LIST OF TABLES

Table 1.	MAJIC Capabilities Summary.....	8
Table 2.	Content Model.	34
Table 3.	MAJIC Script Commands.....	93

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would like to thank the staff and students at the Naval Postgraduate School for their support and assistance. I consider myself fortunate to have met the gentlemen who comprise the graduating class of September 2007, and to be able to count them among my friends. Specifically I would like to thank:

Craig Martell for his guidance, wisdom, and his refusal to let me give up when I was unsure if I would be able to develop my research into a viable thesis.

Kevin Squire for his technical expertise, insights, and ensuring I covered all the bases when developing this document.

James Robinson for being a friend, a PT partner, and a constant source of entertainment.

Pat Staub for the friendship, the physical training, and the after school band project: Off By One.

Eric Sjoberg for being a sounding board and a friend, and for cleverly helping me take my mind off school with grueling 20 mile hikes and half marathons.

My parents, Phil and Dana Ball, for the academic support over all the years, not to mention the love and support with everything else in between.

And finally, most importantly, my wife, Marie, and my children. Without their tireless understanding and support this thesis, and just about everything else I've managed to accomplish over the years would not be possible – and not nearly as enjoyable.

Thank you all.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. OVERVIEW

As teams of autonomous, mobile robots gain popularity in areas such as automated factories, education, and military applications, so does the necessity for a robust, scalable robot control architecture. The military has incorporated many robotic systems into the battlefield. Missions range from autonomous robotic surveillance systems to improvised explosive devices (IED) disposal robots (Figure 1).



Figure 1. U.S. Army soldiers arrive with a robot, left, to remove explosive devices from a street in the center of Baghdad, Iraq, Sunday, May 9, 2004.
(From: AP Photo/Mohammed Uraibi)

The military's multitude of mission scenarios requires a variety of robotic assets able to conduct land, air, sea, and even undersea operations. Unmanned aerial vehicles (UAVs), autonomous underwater vehicles (AUVs), and unmanned surface vehicles (USVs) are fast becoming a regular part of the military landscape. These assets, however, generally employ proprietary protocols requiring the creation of custom command and control systems capable of controlling only a single type of asset, or a limited subset of assets.

Military organizations, such as SPAWAR Systems Center San Diego, are currently working to solve these command and control issues. Projects like the Multi-Robot Operator Control Unit (MOCU) have begun to explore the development of systems that provide unmanned vehicle and sensor operator control interfaces capable of controlling and monitoring multiple sets of heterogeneous systems simultaneously (Figure 2) [4].



Figure 2. MOCU (from SPAWAR San Diego).

B. MOTIVATION

In order for solutions to the obstacles that currently face autonomous military applications to be realized on the battlefield, they must first be researched in the

classroom. With the formation of the Autonomous Coordination Systems Laboratory, the Naval Postgraduate School has taken the first step toward providing those solutions.

As in the real world setting, researchers quickly run into the same obstacles even in the simplified laboratory environment. Establishing communication and control of commercial-off-the-shelf (cots) robotic agents can quickly become a daunting task. Once several brands of robots are combined to conduct experiments such as swarming or simultaneous location and mapping (slam), command and control can become overwhelming.

Currently software packages, such as PYRO, OROCOS, and others provide control for various brands of robots individually, but few packages allow concurrent control of heterogeneous robotic teams. Such a system would allow end users the ability to command and control the teams without worrying about how those commands are translated to individual robots.

The functionality of this type of system would allow students to easily coordinate and conduct experiments on teams of heterogeneous robots without expending precious time and energy developing a command and control architecture specific to their needs.

In addition, the system's concise, intuitive scripted language offers students a means to conduct experiments and create behavioral programs that are easy to understand and require minimal lines of code when compared to proprietary robot languages.

C. OBJECTIVES

The objectives of this thesis are to utilize sound software engineering practices in the specification, design, and development of a multi-agent command and control system for the NPS Robotics Laboratory. These objectives shall be accomplished by:

- Thorough system specification and design using UML and other software engineering practices.
- Developing a modular, object-oriented Java package whose components can be used as a whole to coordinate a heterogeneous team of agents, or individually for robot specific applications.

- Designing a package that is easily upgradeable with regard to the addition of future robot-specific libraries.

D. SCOPE

The scope of this thesis is to first establish communication, command, and control of a variety of robots through the coordination of the Java programming language and each robot's onboard operating system. Once established for individual brands of robots, a master controlling JAVA architecture will be developed that will allow coordination and control of multiple brands of robots simultaneously. This system will be tested on several brands of robots in a real world environment.

E. THESIS ORGANIZATION

Chapter II establishes the system and user requirements necessary to develop a comprehensive, multi-agent control architecture.

Chapter III formalizes the requirement specifications into an architectural design by decomposing the system into a subset of systems and identifying software patterns common to this type of architecture.

Chapter IV discusses the necessary hardware components and software concerns regarding the implementation of the Multi-Agent Java Interface Controller (MAJIC). A brief user's guide and a programmer's guide are also included. The programmer's guide provides detailed instructions on the procedures required to extend MAJIC. Examples include creating robot-specific libraries, creating an object of scripted actions loadable at run-time, and using MAJIC's classes as stand-alone modules.

Chapter V provides information detailing the benefits gained by utilizing the MAJIC application. This chapter also examines the reduction in lines of code and the improvement of self-documenting code achieved by writing programs for robots using the MAJIC Script.

Chapter VI contains a summary and recommendations for future work.

Appendices provide a glossary and system source code.

II. SOFTWARE VISION DOCUMENT

A. INTRODUCTION

1. Purpose of the Vision Document

The purpose of this document is to provide the foundation and reference for all detailed requirements development. Here the high-level user needs are gathered, analyzed, and defined to identify the required application features for the Multi-Agent Java Interface Controller.

2. Application Overview

The intention of the MAJIC application is to provide a means of command and control for researchers conducting experiments on teams of heterogeneous robots. The system will enable users to control the robot's motion, set and retrieve parameter values, and run scripted behaviors on the agent's onboard operating system.

B. USER DESCRIPTION

1. User Demographics

There are several departments at the Naval Postgraduate School that support robotic research and whose students, at some point in their studies, could find this application to be a helpful tool. The initial focus of the MAJIC application, however, is the NPS students conducting robotic research in the Autonomous Coordination Laboratory. Students and robotic researchers at other universities may also find that the MAJIC application meets their individual needs.

2. User Profiles

The users of the MAJIC application will undoubtedly possess a strong familiarity with computers. Although not a requirement to use the application, students in the CS Lab will most likely be familiar with the Java programming language. The majority of these users will be experienced in computer science fields such as artificial intelligence and robotics.

3. Users Environment

The CS Lab houses computers with both Windows and Linux operating systems. The MAJIC application can run under either OS utilizing the Java Virtual Machine (JVM). This Lab also possesses wireless communications and wireless network capabilities. The robots employed by the CS Lab require either 802.11 or blue tooth communications.

4. Key User Needs

When conducting research with robotic systems, researchers typically run into common roadblocks. Tasks such as establishing communication with the robot and gaining control over its parameters and devices can require extensive man-hours that could otherwise be spent conducting experiments. These concerns are outlined below.

a. Communication

Commercial-off-the-shelf robots typically arrive with operating systems that possess their own proprietary protocols. A researcher must first establish communication with the robot's embedded system, and then must learn the proper format and protocols to transfer information to and from the robotic agent. The MAJIC application will move the communication implementation from the user's domain to the application domain. Instead of learning proprietary protocols for individual robots, the user will be able to utilize MAJIC's scripted language to pass common commands to any robot currently managed by the application.

b. Control

Gaining access to individual robot parameters is an absolute necessity in order for researchers to conduct any non-trivial experiments. In many cases, MAJIC can add a layer of abstraction to such tasks. This allows users the ability to intuitively obtain desired responses without extensive knowledge of robot-specific operating systems and parameter passing protocols.

c. Coordination

Establishing communication and control over an individual robot or type of robot often is not broad enough to provide the student with the assets they need to conduct their research. The MAJIC application provides a central location that allows the simultaneous control of any number of heterogeneous agents.

5. Alternatives

Every robot in the Computer Science Lab comes with its own COTS software. Furthermore, a student researcher can find open-source software that provides communication, command, and control of every robot that is currently in the CS Lab. In fact, some of that software is encapsulated in the MAJIC application. In many cases this software can even provide the programmer with levels of functionality that get abstracted away by MAJIC.

The trade-off, however, is the time required to acquire, install, and learn how to extract that functionality from these COTS and open-source systems. That time is compounded by the fact that the process must be repeated for each brand of robot in the lab. Furthermore, once completed, the researcher is faced with powerful, robot-specific systems that lack interoperability.

This lack of interoperability forces the student to spend valuable time developing another higher-level system that provides interoperability in order to conduct any heterogeneous, team-oriented experiments. Once completed, the student may find that the addition of a new robot to the lab or a change in the experiment's scope will force him to redesign his high-level system or start over from scratch. Percussionist wanted.

C. APPLICATION OVERVIEW

Section C provides a high-level view of application capabilities, application interfaces, and system configurations.

1. Application Perspective

The MAJIC application is a multi-robot control architecture designed to provide users a quick, convenient interface to coordinate and control groups of robotic agents. It

eliminates the time required to implement and learn software applications provided by COTS products and open-source options for multiple robots. In addition, it provides offers users a scripting language that standardizes commands to all supported robot types.

2. Application Position Statement

NPS Students and robotics researchers require an intuitive software application that provides command and control of a team of heterogeneous robotic agents. MAJIC provides centralized control, ease of use, run-time configurability, and interoperability. Unlike COTS and Open-source options, MAJIC provides a standard set of MAJIC Script commands that translate to any member of the robotic team.

3. Application Capabilities Summary

USER BENEFITS	SUPPORTING FEATURES
Robot Team Configurability at run time	Add Bot Button, Remove Bot Button, Robot Selection Screen
Command Passing Capability	Command Line, Keyboard arrow keys
Robot-specific Information	Help button, Robot Display Screen, Message Area
Robot-specific Behavior Capability	Load Action button, File Selection Screen
Parameter Log Files	Save log button
Session Log Files	Save Session button

Table 1. MAJIC Capabilities Summary.

4. Assumptions and Dependencies

MAJIC will be developed with the Java programming language and run on any platform that has the Java Runtime Environment installed. MAJIC requires access to a Wireless Area Network (WAN) to communicate with many of the supported robot types.

D. APPLICATION FEATURES

Section D provides a high-level description of the application features. The application capabilities necessary to deliver the user benefits are determined and defined in this section.

1. Button Toolbar

The button toolbar provides a convenient area for users to quickly gain access to many of the common tasks that they are expected to repeatedly perform when using the MAJIC application. Buttons to add/remove robots, load behaviors, save files, view help screens, and quit the application will be located here.

2. Robot Configurability

The capability to add and remove robots during a MAJIC session will allow the user to alter their robot team's configuration as dictated by situational or research requirements. GUI buttons and pop-up menus will provide the means necessary to conduct robot team management.

3. Scripted Language

MAJIC Script will present the user with a standardized set of commands for all robot types supported by MAJIC.

4. Command Line

The command line option will allow users to send a line of MAJIC Script to any robot currently managed by the MAJIC session. The application will provide basic command line parsing and syntax verification. Commands will be stored to allow cycling through previous commands via the keyboard arrow keys.

5. Behavioral Programming

Pre-programmed behaviors will be available for uploading to a robot via a load button and selection screen. These programs will consist of MAJIC Script and Java. Users who wish to utilize MAJIC programming can create their own behaviors and add them to the directory.

6. Informational Displays

Message areas will give users feedback for command line responses and robot parameter information. A display area will provide information for any robot library that supports robot-specific displays.

7. File I/O

The user will be allowed to load files in the form of MajicAct behaviors and save log files of MAJIC commands and robot parameters recorded during a MAJIC session.

E. USE CASE DIAGRAMS

Use cases provide an ordering mechanism for requirements. They are a critical tool in the analysis, design, and implementation processes by providing context for the requirements of the system. Not only can they offer an understanding of why a requirement is what it is, but they can help define how the system meets those objectives [10].

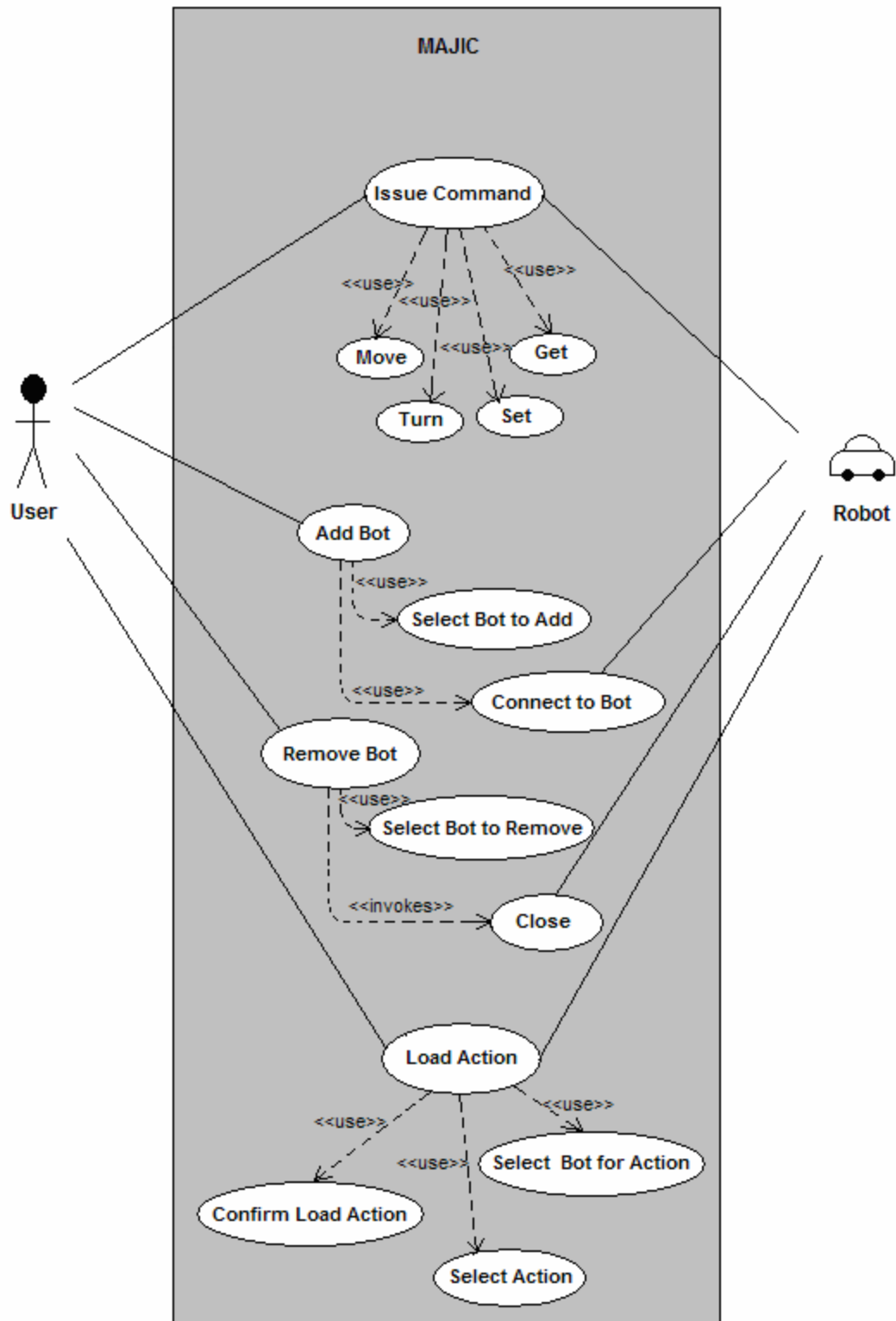


Figure 3. Use Case Model.

Use case: UC-1 Add Bot

Primary Actor: User

Other Actors: MAJIC Application, Robot users

Stakeholders and Interest:

- User wants a quick, error free connection to the selected robot.
- Robot wants error free communication and server connections established with application.

Entry conditions:

- Application is running.
- Communication Network is up and stable.
- Robot is running and available for communication connection.

Exit conditions:

- Application displays connection status message.
- Communication with robot is established.

Flow of events:

1. The User selects a robot.
 - a. Use UC-1a Select Bot to Add.
2. The application instantiates the appropriate MajicBot class.
3. The application establishes connection.
 - a. Use UC-1b Connect to Bot
4. The application displays GUI messages.

Alternate Flows:

- 1a. User cancels selection.
- 3a. Application unable to establish connection.
 - a. Connection times out.
 - b. Error message displayed in message area.

Special Requirements:

1. Maximum number of robots can not be exceeded.

Use case: UC-1a Select Bot to Add

Primary Actor: User

Description: Sub-use case of UC-1 Add Bot

Other Actors: MAJIC Application

Stakeholders and Interest:

- User wants a quick, intuitive means of selecting any robot supported by the application.

Entry conditions:

- Application is running.
- Communication Network is up and stable.
- Robot is running and available for communication connection.

Exit conditions:

- Application instantiates the appropriate MajicBot Class based on user input.

Flow of events:

1. User selects desired robot type from drop menu
2. User confirms via *OK* button
3. Application instantiates robot

Alternate Flows:

- 2a. User cancels selection via *CANCEL* button.

Special Requirements:

None

Use case: UC-1b Connect to Bot

Primary Actor: User

Description: Sub-use case of UC-1 Add Bot

Other Actors: MAJIC Application, Robot users

Stakeholders and Interest:

- User wants a quick, intuitive means of inputting communication address.
- Robot wants error free communication and server connections established with application.

Entry conditions:

- Application is running.
- Communication Network is up and stable.
- Robot is running and available for communication connection.

Exit conditions:

- Application instantiates the appropriate MajicBot Class based on user input.

Flow of events:

1. User inputs IP Address, Com Port, etc.
2. User confirms via *OK* button.
3. Application establishes connection with robot.
4. Application initializes robot-specific connection protocols.

Alternate Flows:

- 2a. User cancels selection via *CANCEL* button.

Special Requirements:

None

Use case: UC-2 Remove Bot

Primary Actor: User

Other Actors: MAJIC Application, Robot users

Stakeholders and Interest:

- User wants a quick, error free disconnection from the selected robot.
- Robot wants all application related server applications shutdown and an error-free communication disconnection from application.

Entry conditions:

- Application is running.
- Communication Network is up and stable.
- Robot is connected and communicating with application.

Exit conditions:

- Robot is disconnected from application.
- Application displays disconnection status message.

Flow of events:

1. The User selects a robot.
 - a. Use UC-2a Select Bot to Remove.
2. The application kills MajicAct thread associated with selected robot if thread is active.
3. The application closes the MajicBot connection.
4. The application removes selected robot from MajicBot array.

5. Remove GUI display associated with selected robot.
6. Displays GUI message.

Alternate Flows:

- 1a. User cancels selection.

Special Requirements:

1. Robot total must be greater than zero to remove a bot.

Use case: UC-2a Select Bot to Remove

Primary Actor: User

Description: Sub-use case of UC-2 Remove Bot

Other Actors: MAJIC Application

Stakeholders and Interest:

- User wants a quick, intuitive means of selecting any robot currently being managed by the application.

Entry conditions:

- Application is running.
- Communication Network is up and stable.
- Robot is connected and communicating with application.

Exit conditions:

- Desired robot has been selected by User.

Flow of events:

1. User selects desired robot type from drop menu.
2. User confirms via *OK* button.
3. Selection Dialog returns selected robot.

Alternate Flows:

- 2a. User cancels selection via *CANCEL* button.

Special Requirements:

None

Use case: UC-3 Load Action

Primary Actor: User

Other Actors: MAJIC Application, Robot users

Stakeholders and Interest:

- User wants a quick, intuitive interface to load a preprogrammed behavior and run it on the appropriate robot.
- Robot wants error free communications between embedded operating system and scripted behavior.

Entry conditions:

- Application is running.
- Communication Network is up and stable.
- Robot is connected and communicating with application.

Exit conditions:

- Application displays action status message.
- Communication between robot and MajicAction is established.
- MajicAction thread is started and passing commands to appropriate robot.

Flow of events:

1. The User selects a robot.
 - a. Use UC-3a Select Bot for Action.
2. The application confirms Loading Action for pre-existing actions.
 - a. Use UC-3b Confirm Load Action.
3. The application terminates pre-existing action thread.
4. The User selects an Action to load.
 - a. Use UC-3c Select Action.
5. The application loads the selected Action object.
6. The application sets the selected Action's robot to the selected bot.
7. The application starts Action thread.
8. The application displays GUI messages.

Alternate Flows:

- 1a. User cancels selection.
- 2a. User selects *NO* button during confirmation.
- 4a. User cancels load action.
- 6a. Application displays error message if robot and behavior are not compatible.

Special Requirements:

1. Robot total must be greater than zero to load an action.

Use case: UC-3a Select Bot for Action

Primary Actor: User

Description: Sub-use case of UC-3 Load Action

Other Actors: MAJIC Application

Stakeholders and Interest:

- User wants a quick, intuitive means of selecting any robot currently being managed by the application.

Entry conditions:

- Application is running.
- Communication Network is up and stable.
- Robot is connected and communicating with application.

Exit conditions:

- Desired robot has been selected by User.

Flow of events:

1. User selects desired robot type from drop menu
2. User confirms via *OK* button
3. Selection Dialog returns selected robot.

Alternate Flows:

- 2a. User cancels selection via *CANCEL* button.

Special Requirements:

None

Use case: UC-3b Confirm Load Action

Primary Actor: User

Description: Sub-use case of UC-3 Load Action

Other Actors: MAJIC Application, Robot User

Stakeholders and Interest:

- User wants a safeguard against accidentally loading a new behavior over a behavior that is currently active.
- Robot wants application to prevent conflicting behavior threads from passing commands to its operating system simultaneously.

Entry conditions:

- Application is running.
- Communication Network is up and stable.
- Robot is connected and communicating with application.
- A behavior is currently active for selected robot.

Exit conditions:

- User responds to confirmation dialog.

Flow of events:

1. User confirms via *YES* button
2. Selection Dialog returns confirmation.

Alternate Flows:

- 1a. User terminates load sequence via the *NO* button.

Special Requirements:

None

Use case: UC-3c Select Action

Primary Actor: User

Description: Sub-use case of UC-3 Load Action

Other Actors: MAJIC Application

Stakeholders and Interest:

- User wants a quick, intuitive interface to select a behavior object from a directory of preprogrammed behaviors to load onto the robot.

Entry conditions:

- Application is running.
- Communication Network is up and stable.
- Robot is connected and communicating with application.

Exit conditions:

- User has selected desired file to load.

Flow of events:

1. User selects file from a directory of behaviors.

2. User confirms selection via *LOAD ACTION* button.
3. Selection Dialog returns confirmation.

Alternate Flows:

- 2a. User terminates load sequence via the *CANCEL* button.

Special Requirements:

None

Use case: UC-4 Issue Command

Primary Actor: User

Other Actors: MAJIC Application, Robot users

Stakeholders and Interest:

- User wants a quick, intuitive interface to pass motion commands to the robot, set parameters on the robot, and retrieve parameter values from the robot.
- Robot wants error free communications between embedded operating system and scripted commands.

Entry conditions:

- Application is running.
- Communication Network is up and stable.
- Robot is connected and communicating with application.

Exit conditions:

- Robot performs desired action.
- Application displays status message.

Flow of events:

1. The User enters a *move()* command.
 - a. Use UC-4a Move Bot.
2. The User enters a *turn()* command.
 - a. Use UC-4b Turn Bot.
3. The User enters a *set()* command.
 - a. Use UC-4c Set Bot Parameters.
4. The User enters a *get()* command.
 - a. Use UC-4d Get Bot Parameters.
5. The application displays GUI messages.
- 6.

Alternate Flows:

none

Special Requirements:

none

Use case: UC-4a Move Bot

Primary Actor: User

Description: Sub-use case of UC-4 Issue Command

Other Actors: MAJIC Application, Robot users

Stakeholders and Interest:

- User wants a quick, intuitive interface to control robot's forward and backward motion.
- Robot wants error free communications between embedded operating system and scripted commands.

Entry conditions:

- Application is running.
- Communication Network is up and stable.
- Robot is connected and communicating with application.

Exit conditions:

- Robot moves distance specified by the User.
- Application displays status message.

Flow of events:

1. User enters *move()* command.
2. Command event is passed to MajicParser event handler.
3. Commands format is verified by MajicParser.
4. Command is passed to MajicBot for verification.
5. Command is passed to Robot.
6. Status message is displayed.

Alternate Flows:

3a. Parser detects improper command format and passes appropriate error message to step 6.

4a. Application detects improper command and passes appropriate error message to step 6.

Special Requirements:

None

Use case: UC-4b Turn Bot

Primary Actor: User

Description: Sub-use case of UC-4 Issue Command

Other Actors: MAJIC Application, Robot users

Stakeholders and Interest:

- User wants a quick, intuitive interface to adjust Robot's heading.
- Robot wants error free communications between embedded operating system and scripted commands.

Entry conditions:

- Application is running.
- Communication Network is up and stable.
- Robot is connected and communicating with application.

Exit conditions:

- Robot turns toward direction specified by the User.
- Application displays status message.

Flow of events:

1. User enters *turn()* command.
2. Command event is passed to MajicParser event handler.
3. Commands format is verified by MajicParser.
4. Command is passed to MajicBot for verification.
5. Command is passed to Robot.
6. Status message is displayed.

Alternate Flows:

3a. Parser detects improper command format and passes appropriate error message to step 6.

4a. Application detects improper command and passes appropriate error message to step 6.

Special Requirements:

None

Use case: UC-4c Set Bot Parameters

Primary Actor: User

Description: Sub-use case of UC-4 Issue Command

Other Actors: MAJIC Application, Robot users

Stakeholders and Interest:

- User wants a quick, intuitive interface to adjust Robot's parameters.
- Robot wants error free communications between embedded operating system and scripted commands.

Entry conditions:

- Application is running.
- Communication Network is up and stable.
- Robot is connected and communicating with application.

Exit conditions:

- Robot parameters equal values specified by the User.
- Application displays status message.

Flow of events:

1. User enters *set()* command.
2. Command event is passed to MajicParser event handler.
3. Commands format is verified by MajicParser.
4. Command is passed to MajicBot for verification.
5. Command is passed to Robot.
6. Status message is displayed.

Alternate Flows:

3a. Parser detects improper command format and passes appropriate error message to step 6.

4a. Application detects improper command and passes appropriate error message to step 6.

Special Requirements:

None

Use case: UC-4d Get Bot Parameters

Primary Actor: User

Description: Sub-use case of UC-4 Issue Command

Other Actors: MAJIC Application, Robot users

Stakeholders and Interest:

- User wants a quick, intuitive interface to retrieve Robot's parameters.
- Robot wants error free communications between embedded operating system and scripted commands.

Entry conditions:

- Application is running.
- Communication Network is up and stable.
- Robot is connected and communicating with application.

Exit conditions:

- Robot turns toward direction specified by the User.
- Application displays status message.

Flow of events:

1. User enters *set()* command.
2. Command event is passed to MajicParser event handler.
3. Commands format is verified by MajicParser.
4. Command is passed to MajicBot for verification.
5. Command is passed to Robot.
6. Robot waits for update to occur.
7. Robot returns updated value.
8. Status message is displayed.

Alternate Flows:

3a. Parser detects improper command format and passes appropriate error message to step 6.

4a. Application detects improper command and passes appropriate error message to step 6.

Special Requirements:

None

F. APPLICATION ALTERNATIVES

Every robot in the Computer Science Lab comes with its own COTS software. Furthermore, a student researcher can find open-source software that provides communication, command, and control of every robot that is currently in the CS Lab.

In fact, some of that software is encapsulated in the MAJIC application. In many cases this software can even provide the programmer with levels of functionality that get abstracted away by MAJIC. The abstractions that give MAJIC its interchangeability for multi-agent operations inherently generalize the system.

Some proprietary software like MobileRobots MobileEyes [1] and generic software like URBI [5], provide powerful GUI applications for remote robot control and monitoring. Several of these COTS packages even incorporate sensor data to provide SLAM capabilities. These packages, however, provide no abstractions that allow separating the “controller” from the rest of the system. For example, a control system based on occupancy grids might be intimately tied to a particular type of robot and laser scanner [3].

Other systems, such as OROCOS [18] and CARMEN [16] provide modular architectures, similar to MAJIC’s, that are capable of accomplishing many predefined tasks on a single robot. For example CARMEN, a robot navigation toolkit developed at Carnegie Mellon, is an open-source collection of software for mobile robot control. CARMEN is modular software designed to provide basic navigation primitives including: base and sensor control, logging, obstacle avoidance, localization, path planning, and mapping [16].

Another single-robot control system that has generated much interest in the robot community is Pyro. Pyro, which stands for Python Robotics, is a robotics programming environment written in the python programming language.

Programming robot behaviors in Pyro is akin to programming in a high-level, general-purpose programming language in that Pyro provides abstractions for low-level robot specific features much like the abstractions provided in high-level languages [3].

The abstractions provided by Pyro allow robot control programs written for small robots to be used to control much larger robots without any modifications to the “controller”. This represents an advance over previous robot programming methodologies in which robot programs were written for specific motor controllers, sensors, communications protocols and other low-level features [3].

While Pyro supports many of the popular robot brands individually, it provides no means of controlling a heterogeneous team of robots simultaneously. With the initiation of each Pyro session, the user must select the specific robot library to load.

In cases where the research lab or student is only using a single brand of robot with a specific sensor, a COTS or proprietary system such as those described above could provide a greater granularity of command and control over that provided by MAJIC.

The trade-off, however, is the time required to acquire, install, and learn how to extract that functionality from these COTS and open-source systems. That time is compounded by the fact that the process must be repeated for each brand of robot in the lab. Furthermore, once completed, the researcher is faced with powerful, robot-specific systems that lack interoperability.

This lack of interoperability forces the student to spend valuable time developing another higher-level system that provides interoperability in order to conduct any heterogeneous, team-oriented experiments. Once completed, the student may find that the addition of a new robot to the lab or a change in the experiment’s scope will force him to redesign his high-level system or start over from scratch.

THIS PAGE INTENTIONALLY LEFT BLANK

III. SYSTEM DESIGN

A. INTRODUCTION

The purpose of this chapter is to transform the analysis model developed in chapter two into a system design model. This transformation will occur through the detailed decomposition of the system into smaller subsystems. Details such as the design goals and the strategies to achieve those goals are explored and identified. The style of the system's architecture and its goals of extensibility and modularity are addressed. The systems components are categorized into presentation and logical levels. An explanation of the system's integration and a detailed description of its user interface are provided as well.

B. SYSTEM ARCHITECTURE

1. Overview

a. Style of Architecture

The Multi-Agent Java Interface controller is a desktop Java application designed for platform independence. The system establishes communications with embedded robot servers via wireless connections to issue commands and receive feedback from the agents (see above figure). The architecture maintains a strict client/server relationship. The MAJIC application client pulls information from the embedded servers upon the user's request. No data is pushed from the server to the client.

b. Goals of Architecture

The primary architectural goal is extensibility. As future brands of mobile, robotic agents are added to the NPS Robotics Laboratory, the MAJIC architecture must be able to easily incorporate the new Java Libraries specific to those agents. Abstract classes should be utilized to allow future libraries to easily *plug in* to the existing architecture.

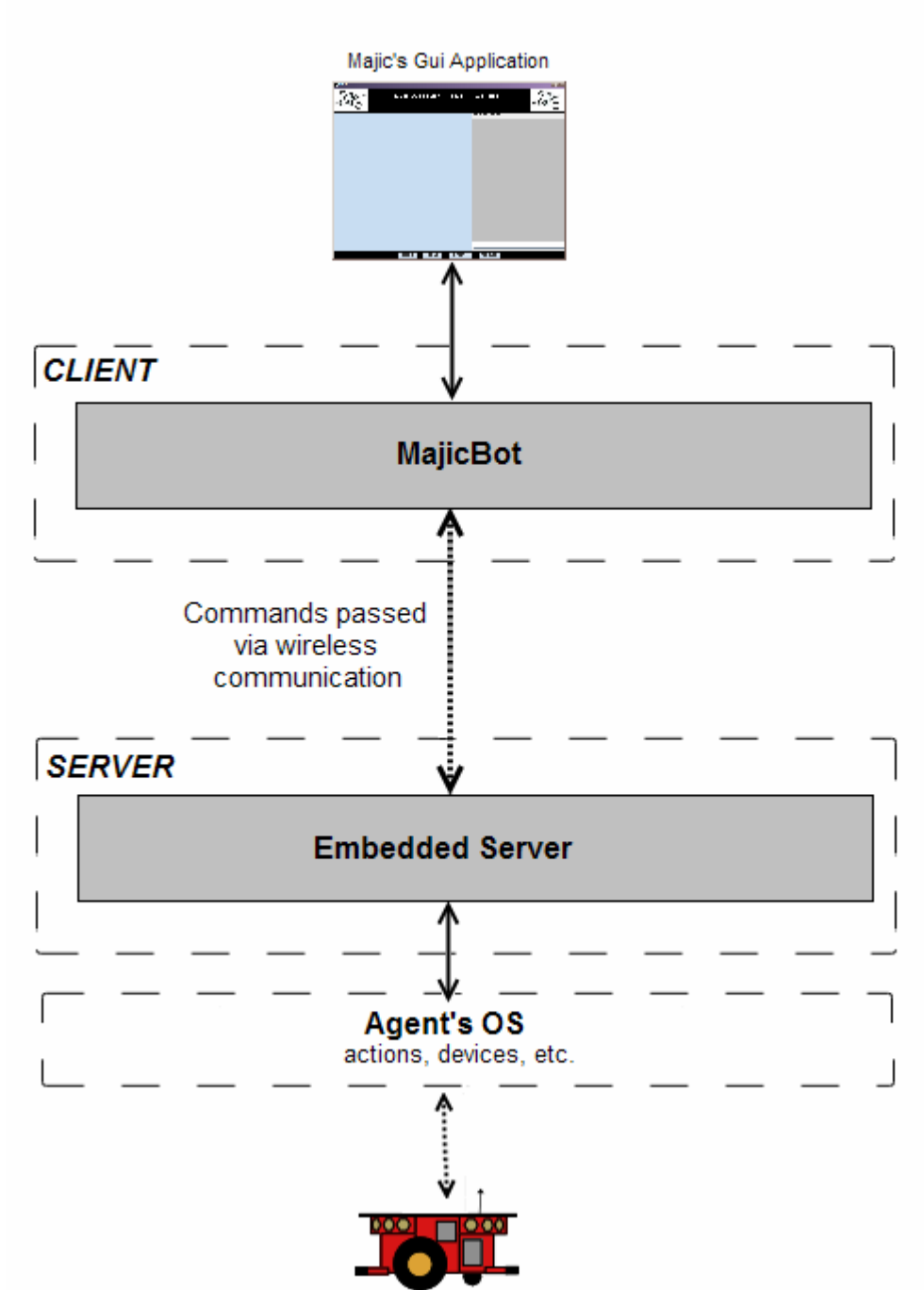


Figure 4. MAJIC Architecture.

c. *Style of Architecture*

The Multi-Agent Java Interface controller is a desktop Java application designed for platform independence. The system establishes communications with embedded robot servers via wireless connections to issue commands and receive feedback from the agents (see Figure 4). The architecture maintains a strict client/server relationship. The MAJIC application client pulls information from the embedded servers upon the user's request. No data is pushed from the server to the client.

d. *Goals of Architecture*

The primary architectural goal is extensibility. As future brands of mobile, robotic agents are added to the NPS Robotics Laboratory, the MAJIC architecture must be able to easily incorporate the new Java Libraries specific to those agents. Abstract classes should be utilized to allow future libraries to easily *plug in* to the existing architecture.

Another important architectural goal is platform independence. Many robotic laboratories consist of multiple computers running a variety of operating systems. Java utilizes a platform independent JVM to execute its byte code, thus allowing the Majic package to operate on any system that utilizes the Java JRE.

Lastly, the components of the architecture should consist of weakly coupled, strongly cohesive modules. Modularity, especially with regard to the specific robot libraries, will allow programmers to use those libraries as stand-alone classes when conducting robot-specific programming.

2. Components

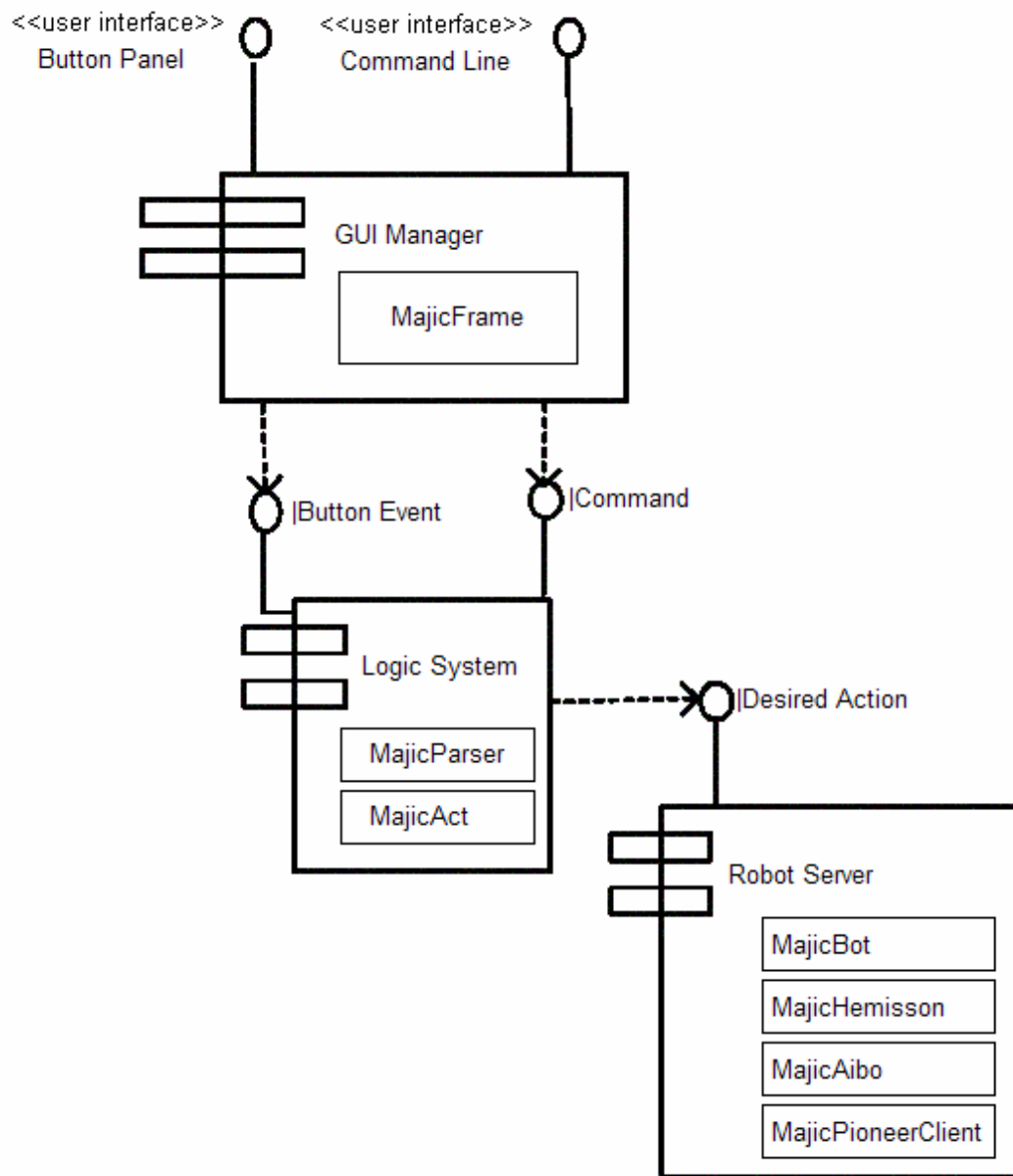


Figure 5. UML Component Model.

a. *Presentation/UI Components*

Component: C-00: GUI Manager

Description:

The presentation layer GUI Manager Component contains the MajicFrame Class. This class allows the User to interact with the logic layer via command line and button actions.

Environmental Constraints:

Requires Java JRE version 5.0 or greater.

Available Interfaces:

Java javac.swing GUI includes JButtons and JTextField.

b. *Application Logic Components*

Component: C-01: Logic System

Description:

The logic layer consists of the MajicParser and MajicAct Classes. This layer receives user input from the presentation layer and parses that input into valid commands that are passed to the appropriate MajicBot Class extention.

Environmental Constraints:

Requires Java JRE version 5.0 or greater.

Requires wireless connection to robot servers via 811, Bluetooth, etc.

Available Interfaces:

none.

Component: C-02: Robot Server

Description:

The Robot Server Component receives commands from the logic layer and communicates those commands to the appropriate robot via wireless connections specific to the robot's server.

Environmental Constraints:

Requires Java JRE version 5.0 or greater.

Requires wireless connection to robot servers via 811, Bluetooth, etc.

Available Interfaces:

none.

3. Integration

Components will communicate using direct procedure calls. The GUI components will use standard Java events. Majic Actions loaded by logic components will run in separate threads in order to control a robot independent of the GUI application. Communication between the logic components and the robot servers are specific to callbacks and messages based on the server's required format.

Extensions of the MajicBot Class can be added to the package without recompilation. MajicAct objects can be created with the software provided in the Majic package utilities folder and loaded at run-time, requiring no system downtime.

C. GRAPHICAL USER INTERFACE

1. Overview

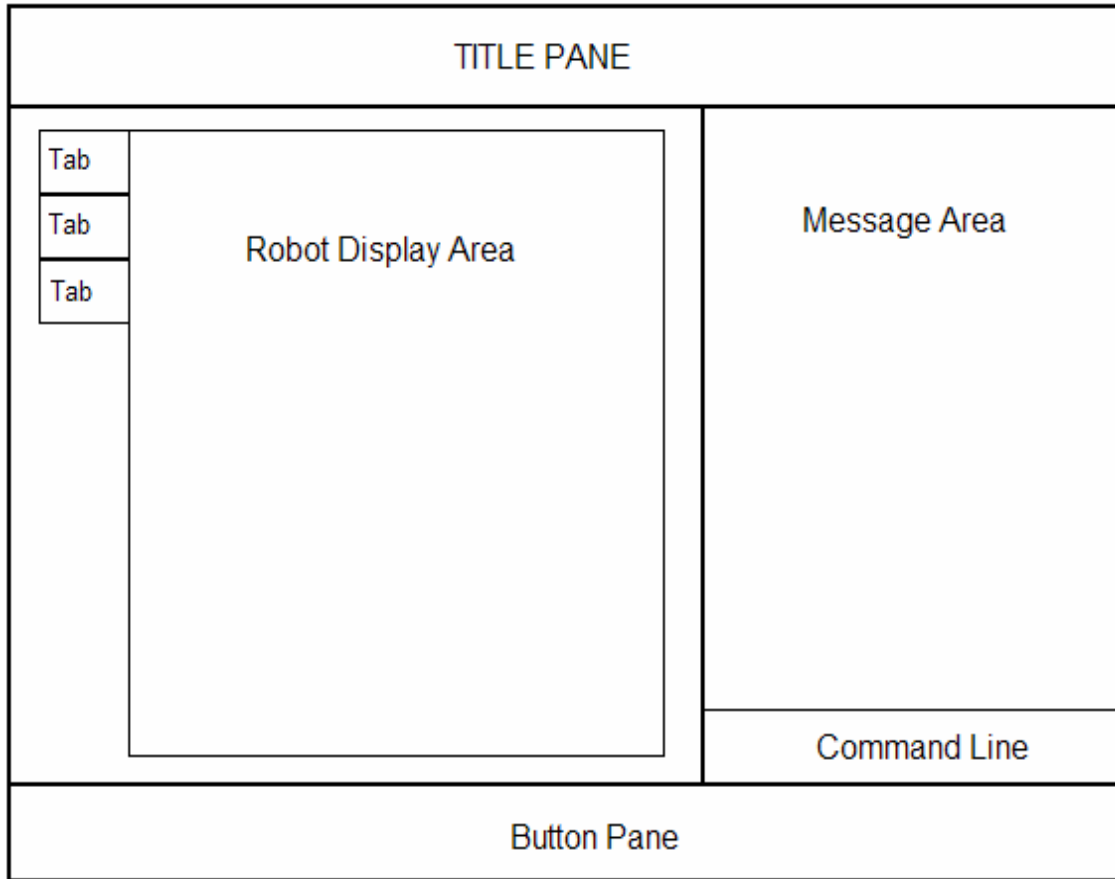


Figure 6. Prototype of Main GUI Display.

The primary goal of the user interface is that of task support and efficiency. The layout and functionality provided by the interface are designed to be well matched to the user's tasks, and these tasks can be completed with a reasonable amount of keystrokes and clicks.

The secondary goal is that of usability and learnability. The system is intended to provide users with easy access to robot command and control. Passing commands to any robot being managed by the system should be uncomplicated and intuitive. Setting or querying a specific robot's parameters should be equally understandable.

2. Content Model

Each entry in the content model is an area where users see information, initiate commands, or select options.

UI Component	Purpose	Behavior
Command Pane	Allows user to send scripted commands to a particular robot.	
Message Area	Provides feedback to user.	Area will display valid commands, robot responses, and error messages.
Command Line	Allows user to type commands to robots.	Initially blank. Will automatically clear if valid command is entered. If invalid command, field will not clear and error message will post. Keyboard arrow keys will allow cycling through previous commands.
Robot Pane	Provides user with information for all active robots.	
Display Area	Displays robot-specific information.	The display pane is optional. Default pane states “no display available”. If utilized by the specific robot class, pane can display general information and provide user feedback.
Tabs	Allows any active robot to be selected.	Clicking tabs will activate that robots display. Hovering over a tab will display the brand of robot to which the display pertains.
Button Pane	Provides user with JButtons for additional functionality.	Buttons that allow user to perform tasks such as adding and deleting robots will reside here.
Dialog Boxes	Provide user pop-up messages and query user input.	JDialogBoxes will perform simple message display and input gathering.

Table 2. Content Model.

D. BEHAVIORAL DESIGN

1. Domain Model

The purpose of the domain model is to capture key concepts in the problem domain. The domain model clarifies the meaning of these concepts and determines the relationships among them [9]. The model below provides a structural view of the various entities involved that will later be enhanced by the dynamic views provided by the use case models.

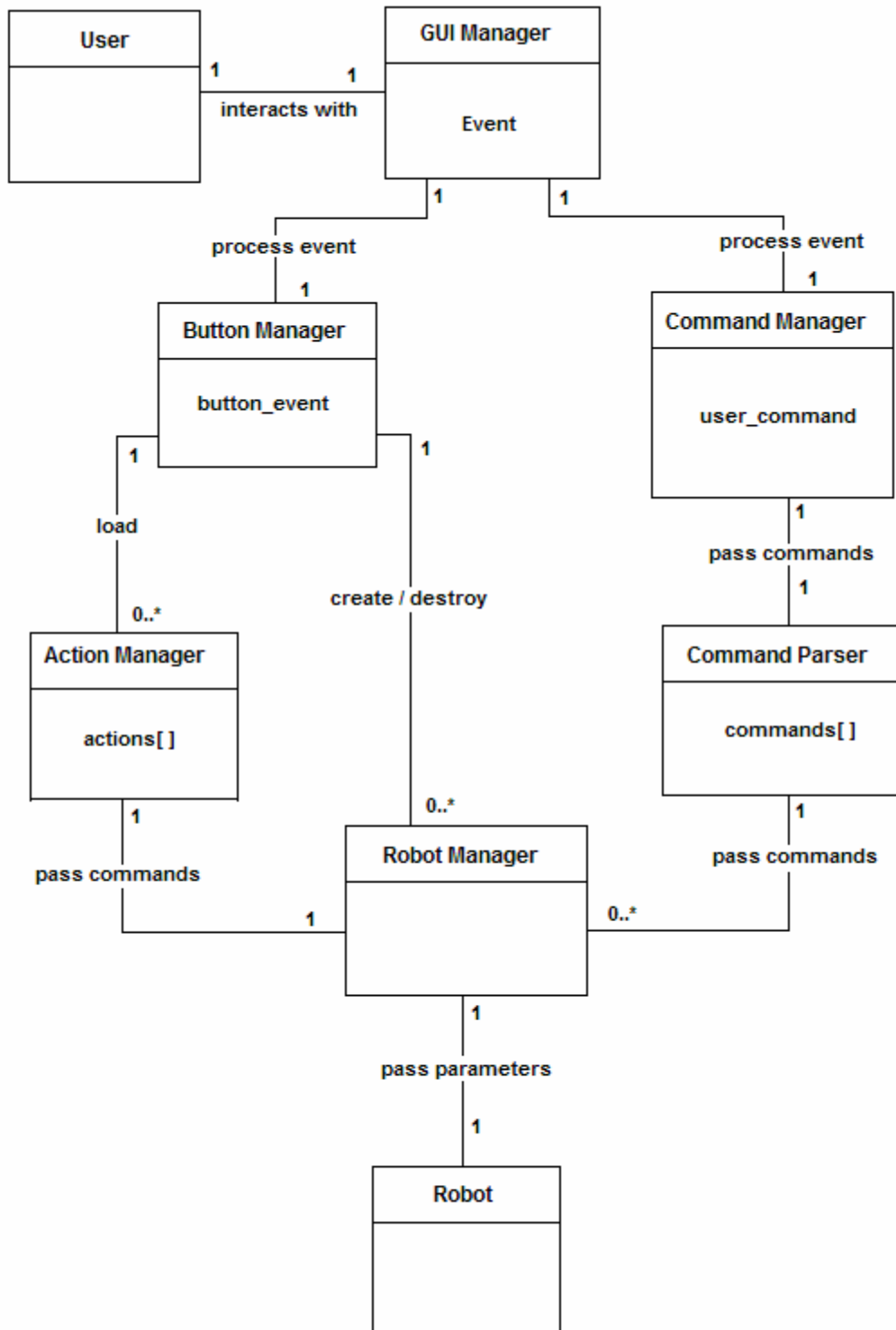


Figure 7. Domain Model UML.

2. Boundary Use Cases

Now that the concepts and vocabulary of the domain model are established, the functional requirements of the system can be determined by revisiting the use cases from Chapter II and identifying the boundary conditions of the system. Cases dealing with such conditions as system startup and shutdown are described in the following boundary use cases.

Use case: UC-5 Application Startup

Primary Actor: User

Other Actors: MAJIC Application

Stakeholders and Interest:

- User wants the application to initialize quickly and without error.

Entry conditions:

- Application is installed on the hardware.

Exit conditions:

- Application GUI is displayed.
- Application is listening for user input.

Flow of events:

1. The application initializes.
2. The application initializes the MajicParser class.
3. The application initializes the button event listeners.
4. The application displays the main screen.

Alternate Flows:

- 1a. Application fails to initialize.
 1. Error message displayed.
 2. Application terminates.

Special Requirements:

None

Use case: UC-6 Application Shutdown

Primary Actor: User

Other Actors: MAJIC Application, Robot users

Stakeholders and Interest:

- User wants the application to terminate quickly and without error.
- Robot wants application to close all communication ports and terminate all software running on robot's onboard operating system.

Entry conditions:

- Application is running.
- Network is up and stable.
- User has pressed the *quit* button.

Exit conditions:

- All applications running on robots operating systems are terminated.
- All connections to robots are closed.
- Application has terminated.

Flow of events:

1. The application passes the *close()* command to all robots with whom it is currently connected.
2. The robot servers terminate all local processes.
3. All communication connections between the application and the robots are closed.
4. The application terminates.

Alternate Flows:

None

Special Requirements:

None

3. Sequence Diagrams

The purpose of sequence diagrams is to formalize the dynamic behavior of the system by tying use cases to objects. Visualizing the communication among objects can help determine additional objects required to formalize the use cases [9]. In this regard, sequence diagrams offer another perspective on the behavioral model and are instrumental in discovering missing objects and grey areas in the requirements specification. The following sequence diagrams depict the boundary use cases identified in section 2 above, and the interaction among the software objects described in the overall class diagram to support the use cases from Chapter II.

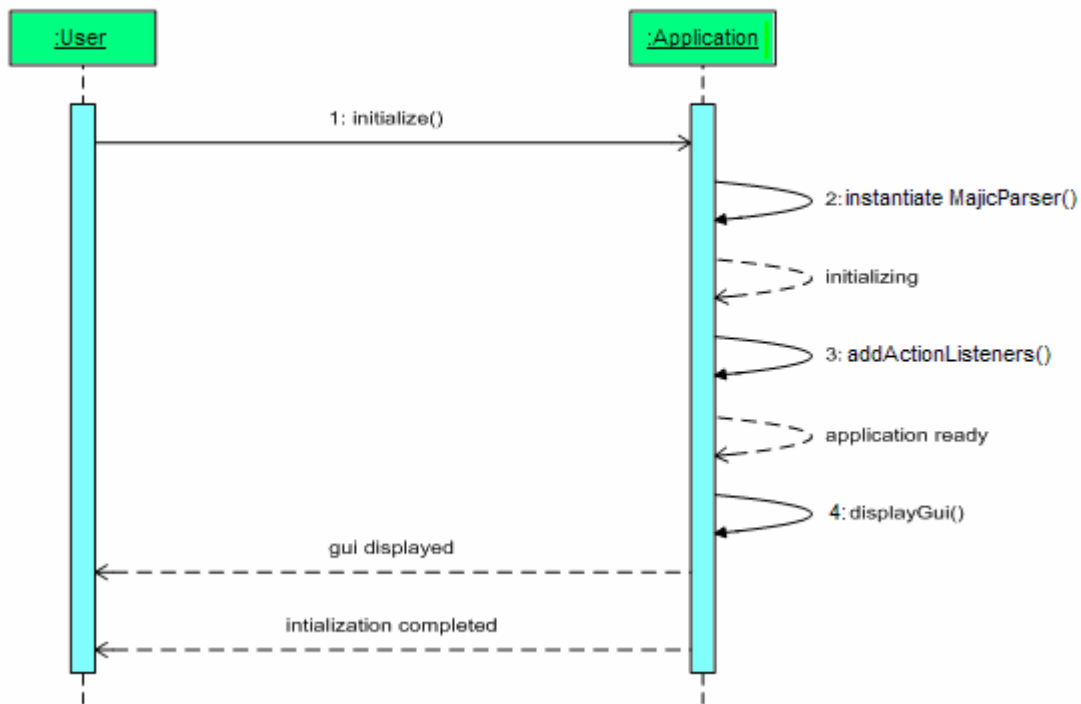


Figure 8. UC-5 Application Startup Sequence Diagram.

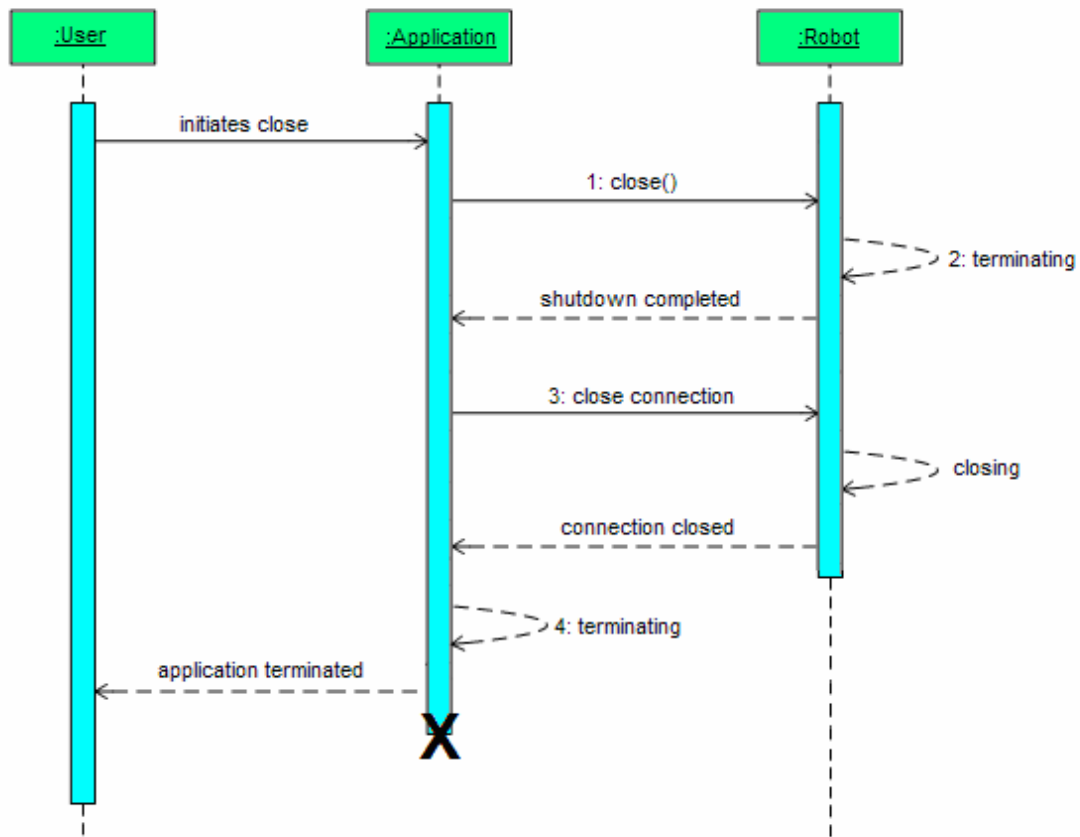


Figure 9. UC-6 Application Shutdown Sequence Diagram.

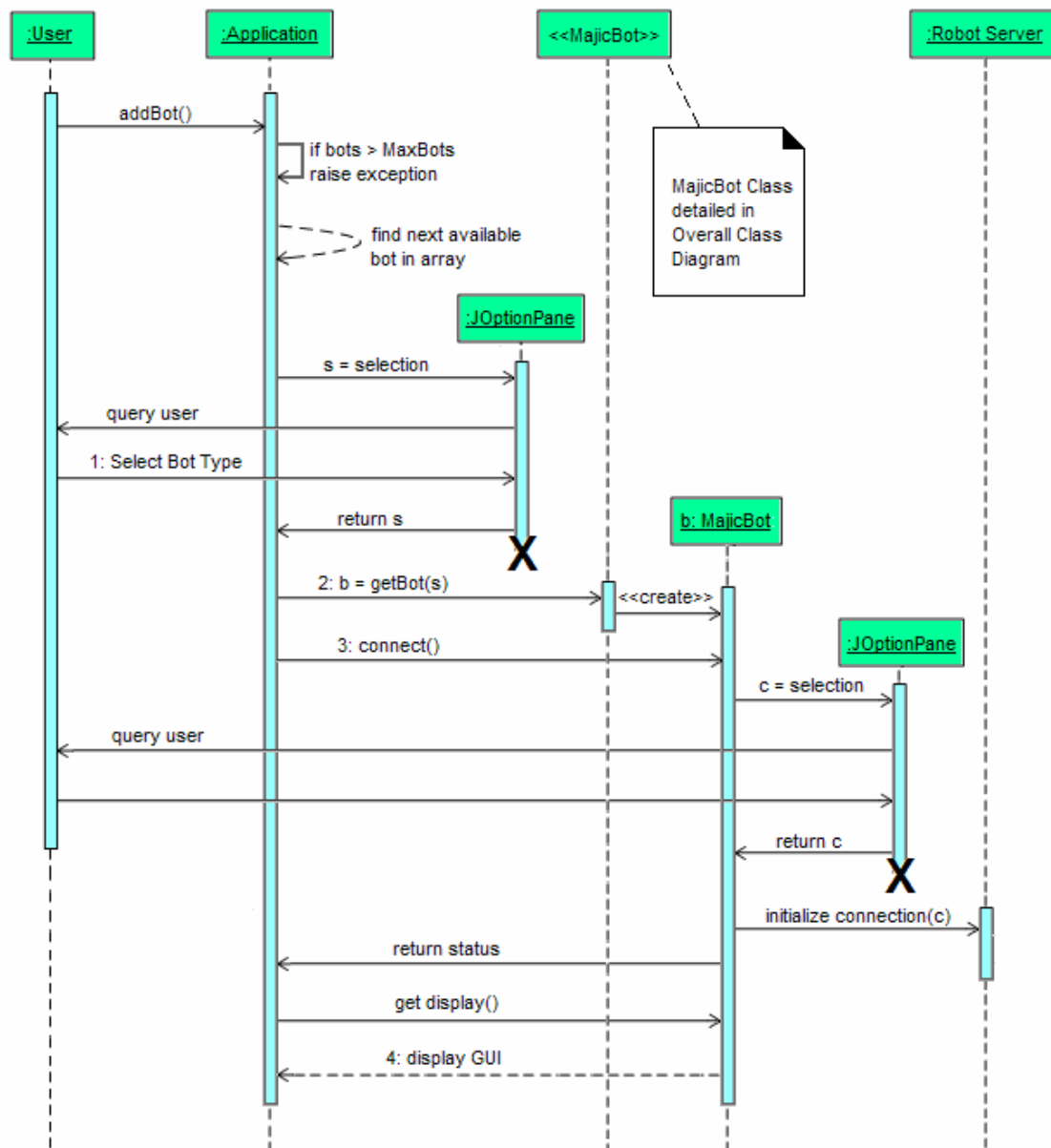


Figure 10. UC-1 Add Bot Sequence Diagram.

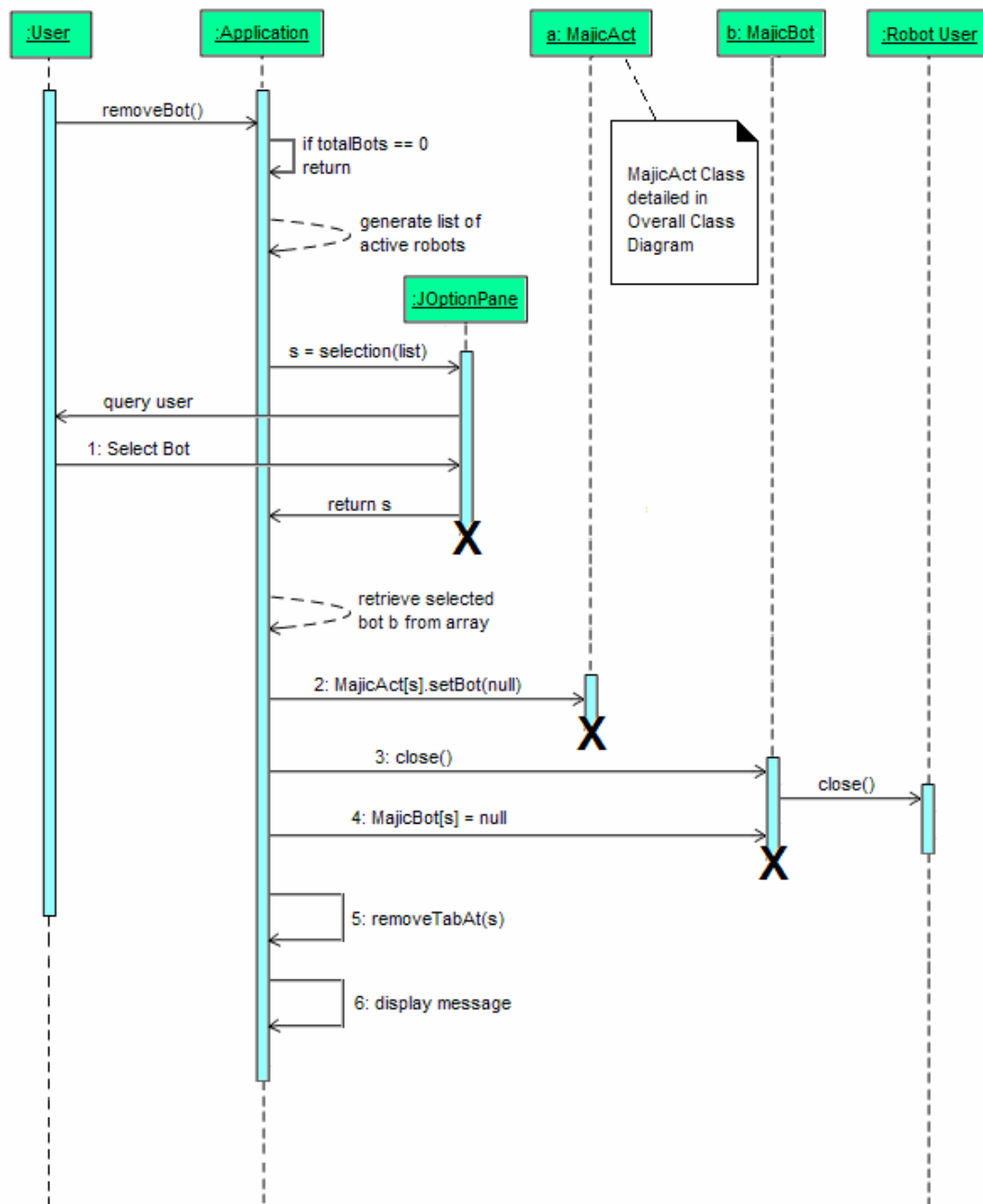


Figure 11. UC-2 Remove Bot Sequence Diagram.

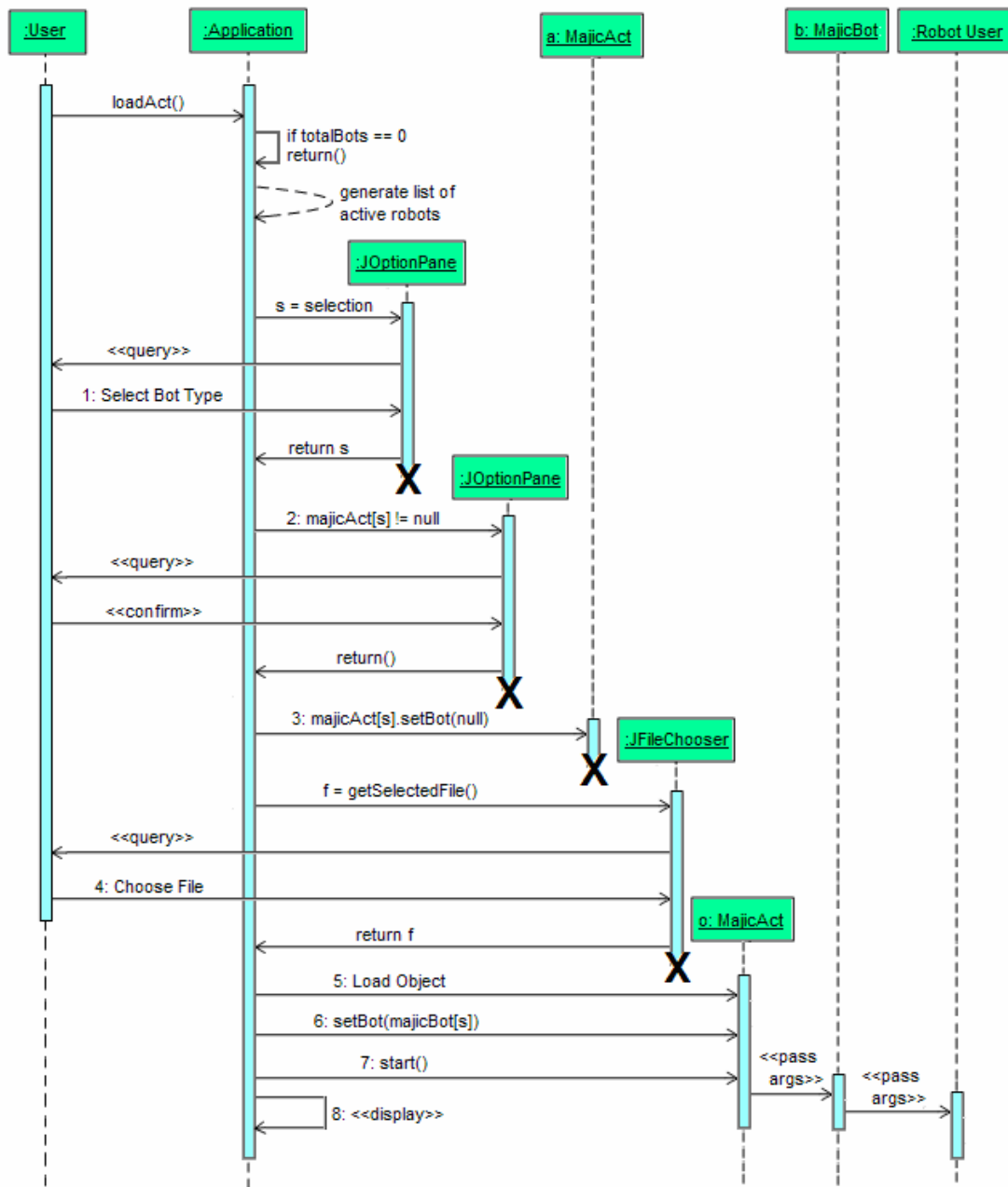


Figure 12. UC-3 Load Action Sequence Diagram.

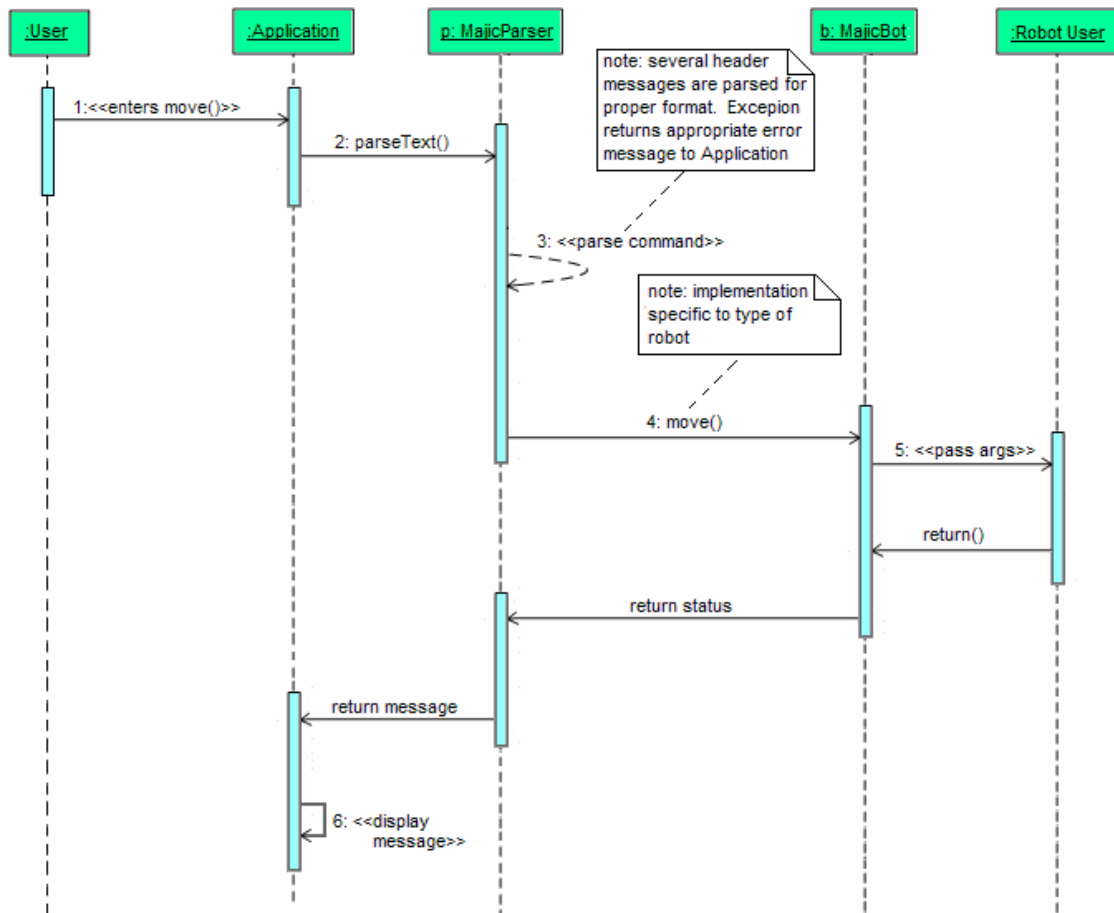


Figure 13. UC-4a Move Bot Sequence Diagram.

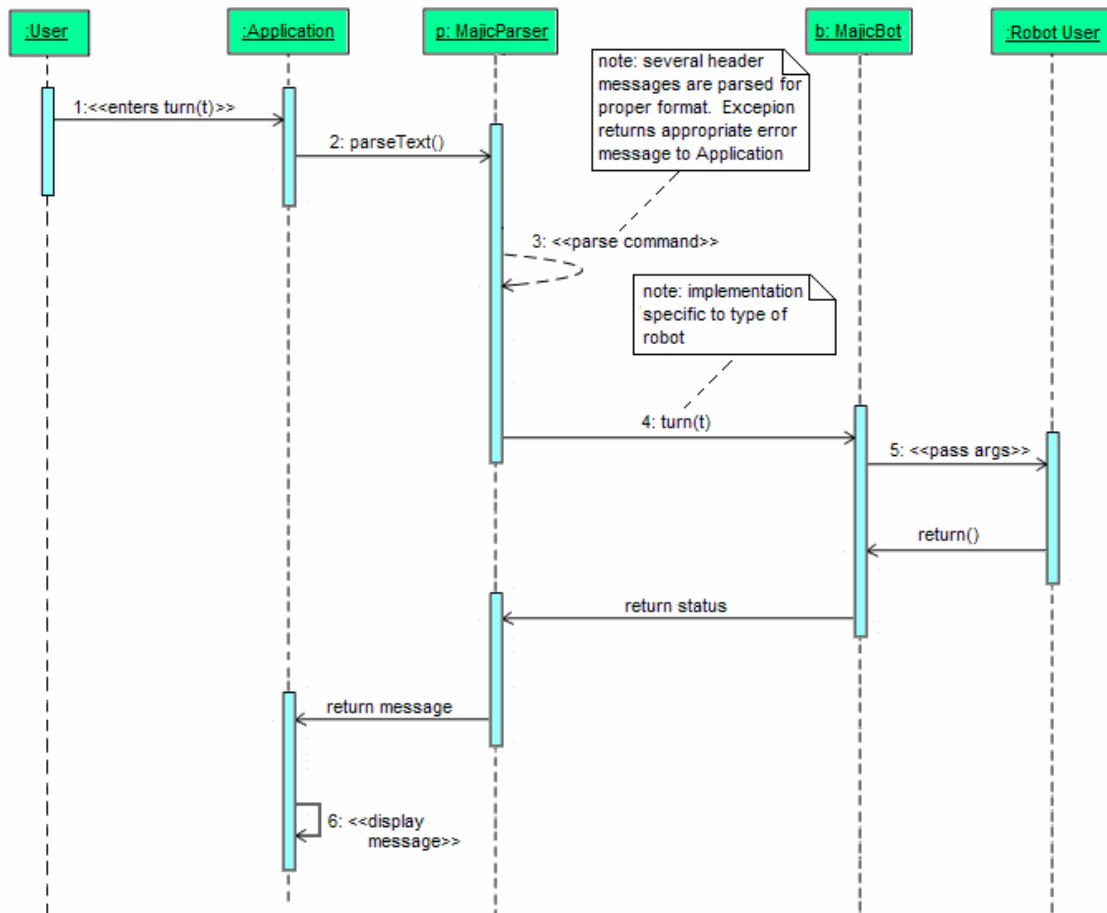


Figure 14. UC-4b Turn Bot Sequence Diagram.

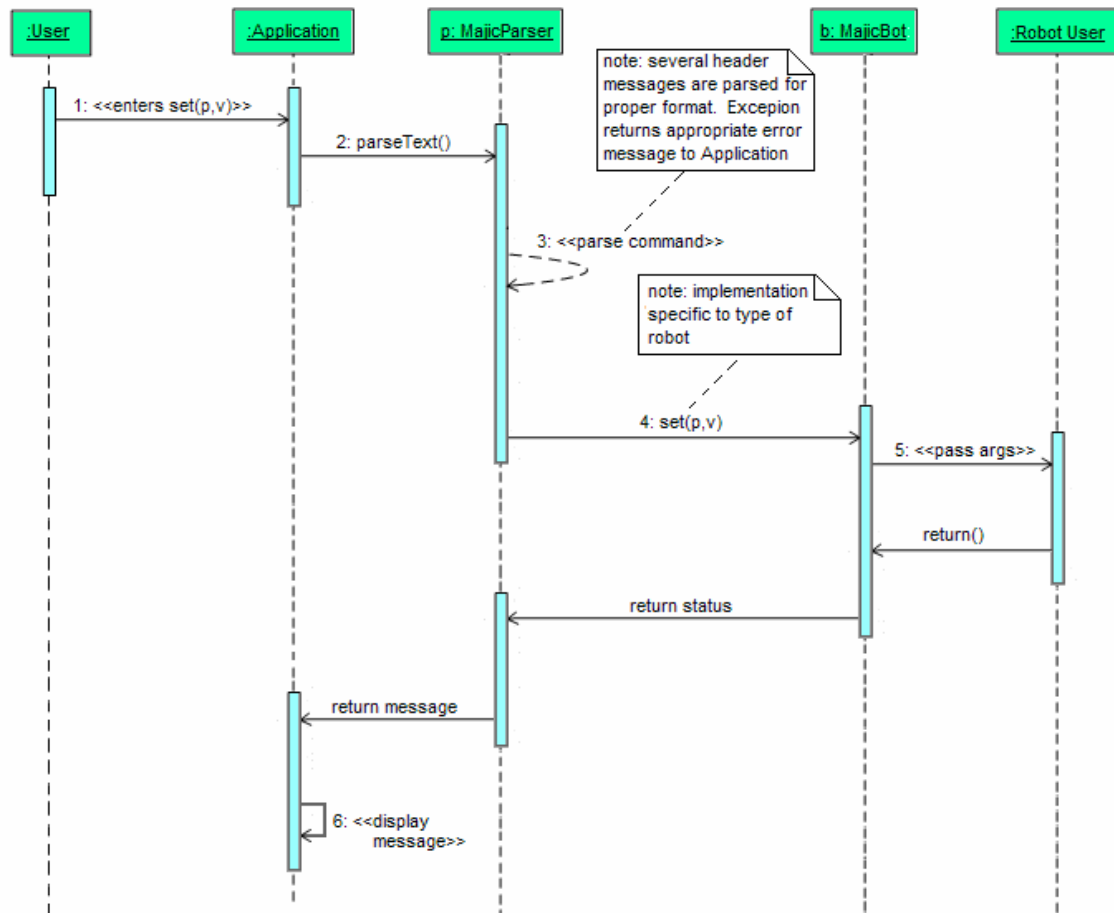


Figure 15. UC-4c Set Bot Parameters Sequence Diagram.

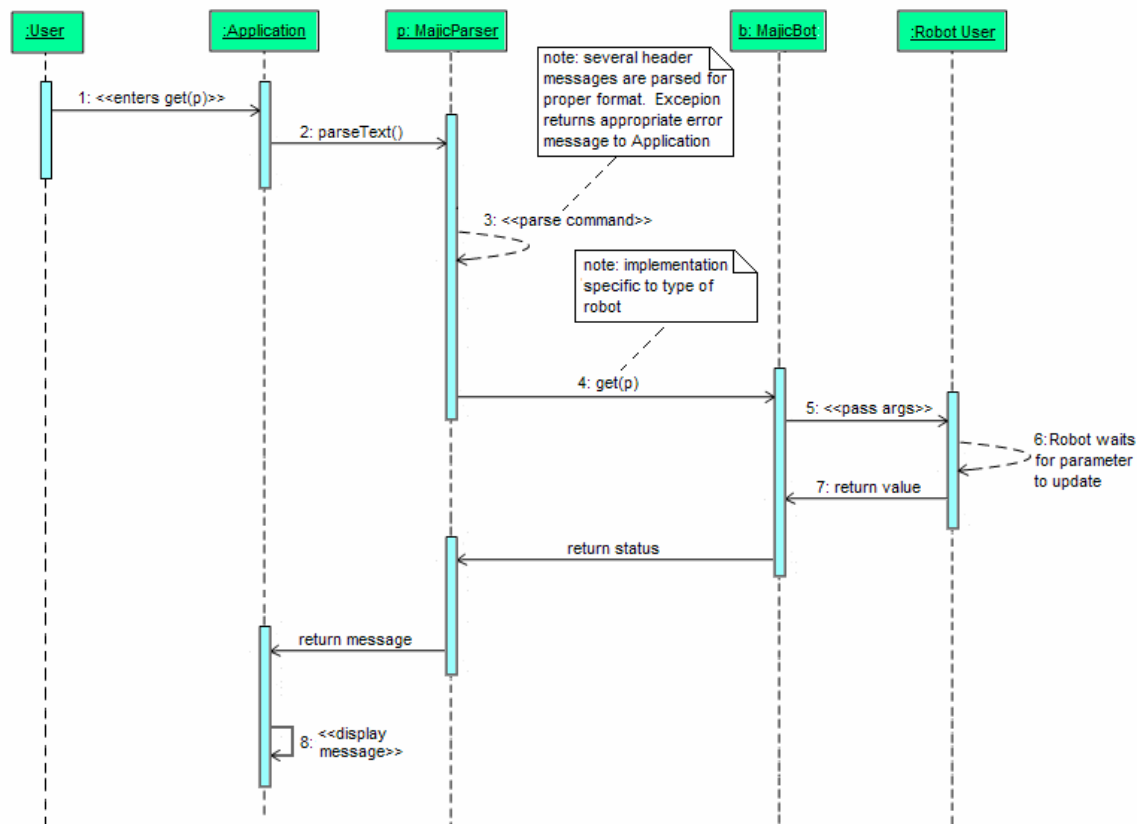


Figure 16. UC-4d Get Bot Parameters Sequence Diagram.

4. Operational Contracts

Operational Contracts represent the final phase of the behavioral model design. These contracts build upon the foundations established during the use case specifications and the domain model and sequence diagram designs.

Expounding upon the events and responses delineated in the sequence diagrams, the operational contracts assign the vague method proposals concrete attributes such as function names, parameters, return values, and a brief definition of purpose.

Utilizing the concepts and relations identified in the domain model, the operational contracts determine the precise definition of the pre-conditions and post-conditions required for the proposed methods.

The following operational contracts extract and formalize the major methods proposed in the domain model and sequence diagrams.

Contract: C1: Startup

Cross Reference: UC-5: Application Startup

Preconditions:

1. This instance of MajicFrame is initializing

Postconditions:

1. A MajicBot array majicBot[] was created and initialized.
2. A MajicAct array majicAct[] was created and initialized.
3. A MajicParser instance commandLine was created.
4. This instance of MajicFrame is associated with commandLine.
5. JButtons were created and their actionListeners were associated with this instance of MajicFrame.
6. The application's main screen was displayed.

Contract: C2: Shutdown

Cross Reference: UC-6: Application Shutdown

Preconditions:

1. The JButton(ActionEvent) corresponding to the *Kill Majic* button was fired.

Postconditions:

1. The close() command was issued to all active objects in the majicBot[] array.
2. The System.exit(0) command was issued and the application terminated.

Contract: C3: Add MajicBot

Method: addBot()

Cross Reference: UC-1: Add Bot

Preconditions:

1. The JButton(ActionEvent) corresponding to the *Add* button was fired.

Postconditions:

1. A MajicBot instance was created and added to the majicBot[] array.
2. Robot counter totalBots was incremented by one.

3. This instance of MajicFrame was associated with the MajicBot instance as its parent frame.
4. A status message was displayed on the main GUI display.

Contract: C4: Remove MajicBot

Method: removeBot()

Cross Reference: UC-2: Remove Bot

Preconditions:

1. The JButton ActionEvent corresponding to the *Kill Bot* button was fired.

Postconditions:

1. The MajicAct instance associated with the selected bot was set to *null*.
2. The *close()* method of the selected majicBot[] was called.
3. The robot counter totalBots was decremented by one.
4. A status message was displayed on the main GUI display.

Contract: C5: Load MajicAct

Method: loadAct()

Cross Reference: UC-3: Load Action

Preconditions:

2. The JButton ActionEvent corresponding to the *Load Act* button was fired.

Postconditions:

3. The MajicAct instance selected by the User was instantiated and added to the majicAct[] array.
4. The MajicAct instance was associated with the MajicBot instance with which it corresponds to in the majicBot[] array.
5. The *start()* method of this MajicAct instance was called.
6. A status message was displayed on the main GUI display.

Contract: C6: Parse Command

Method: parseText()

Arguments:

1. String text: the JTextField text that is contained in the commandLine field of main display GUI.

Return Type:

1. String: method status messages are returned to be displayed in the main display message area.

Cross Reference: UC-4: Issue Command**Preconditions:**

1. The JTextField ActionEvent corresponding to the *commandLine* JTextField was fired.

Postconditions:

1. A local MajicBot instance mBot was associated with the MajicBot object of the majicBot[] array that corresponds to the robot specified in the commandLine text.
2. The MajicBot method specified by the commandLine text was called on the mBot object.
3. The commandLine text was added to the cmdList array.
4. The command total totalCmds was incremented by one.
5. The commandLine text was cleared.
6. A status message was displayed on the main GUI display.

E. OBJECT DESIGN

The system analysis conducted for the Multi-Agent Java Interface Controller in the previous sections is instrumental in identifying the necessary application objects. The manner in which these objects interface with one another is precisely detailed using a combination of the conceptual models, the class model, and the method contracts.

Software reuse is addressed as the necessary off-the-shelf components and design patterns are identified to help provide existing solutions to some of the common software challenges facing the MAJIC application.

1. Class Diagrams

The class diagrams below range from general diagrams depicting the major classes and their associations to task-specific diagrams depicting their interactions, dependencies, and visibilities.

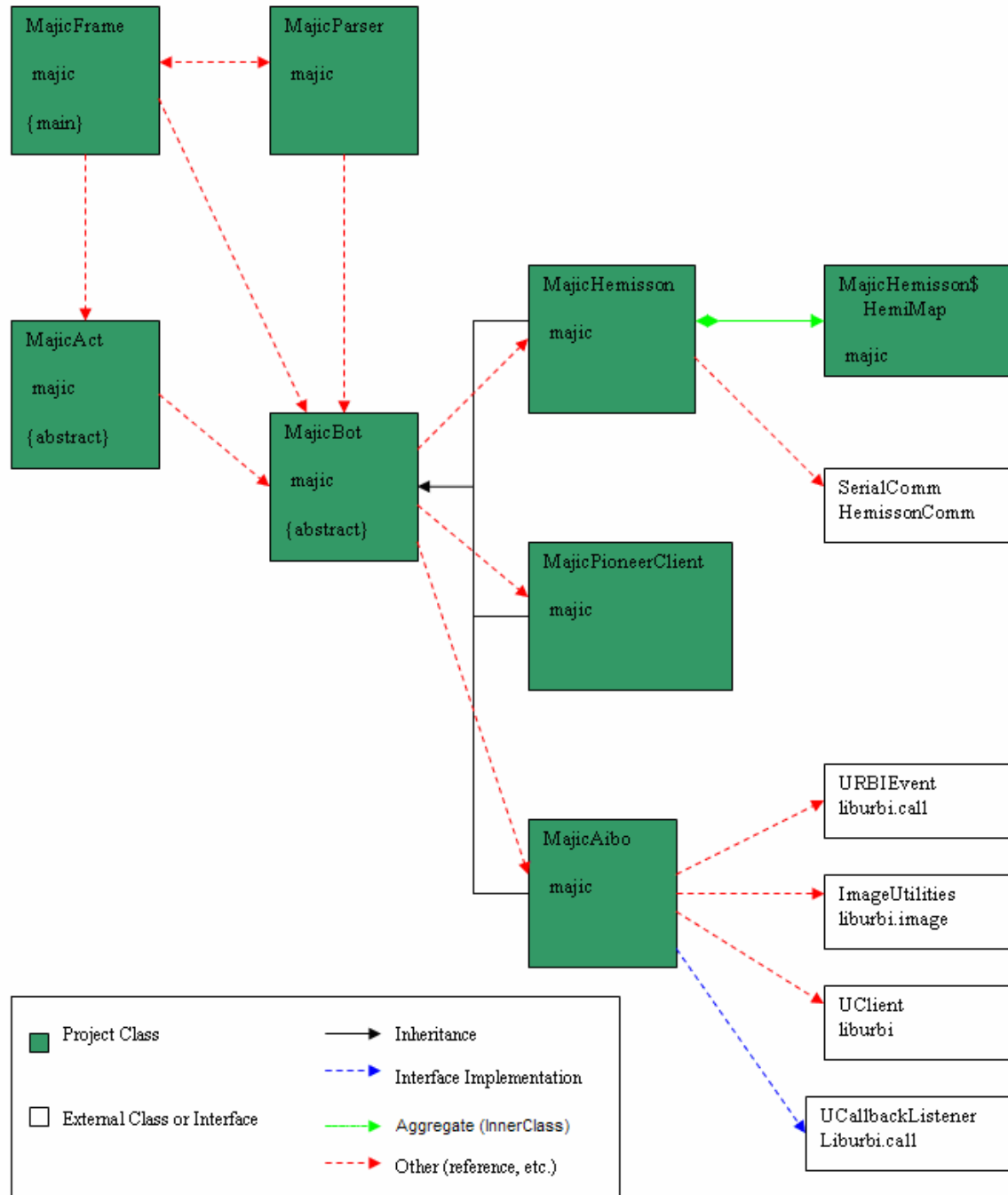


Figure 17. Overall Class Diagram.

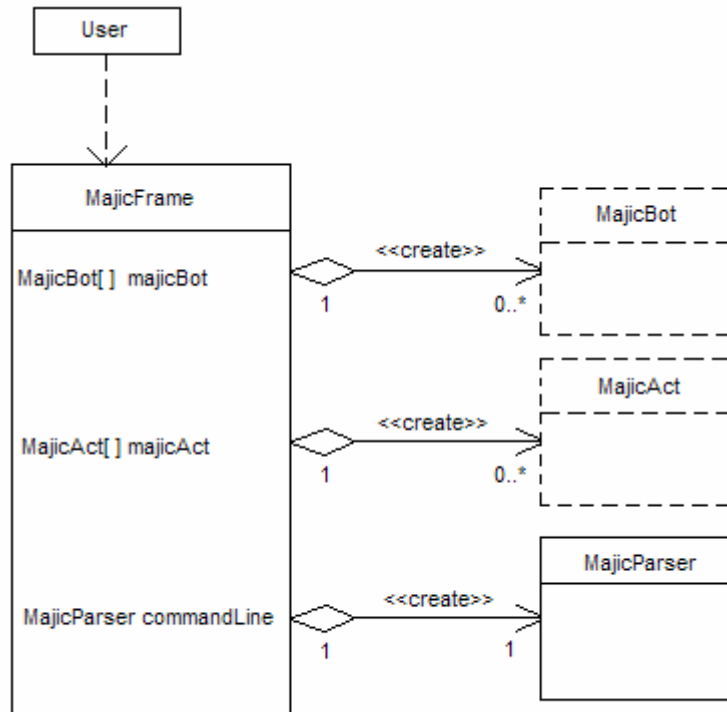


Figure 18. Startup Class Diagram.

a. Startup Class Diagram

There are several classes that are aggregates of the MajicFrame class. In order to assign instances of these classes to the instance of the MajicFrame class during start up, the creator design pattern is utilized. The Startup Class Diagram below combines the requirements of contract C-1 with the concepts of the MAJIC domain.

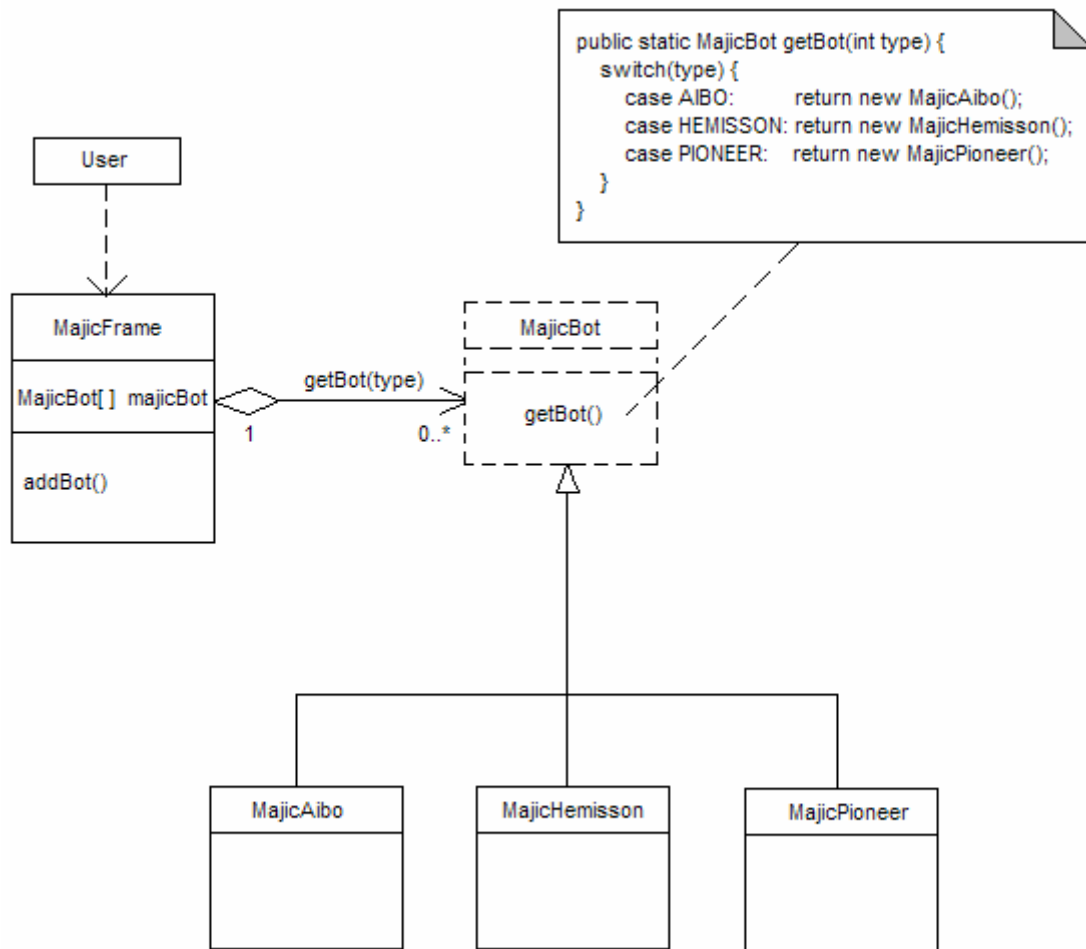


Figure 19. Add Class Diagram.

b. Add Class Diagram

In order for the design of the MAJIC application to remain susceptible to future robot additions, the prototype design pattern is utilized. The prototype pattern allows for the creation of a set of nearly identical objects whose type will be determined by the user at runtime. This flexibility is ideal for creating the `majicBot[]` array necessary to contain a mixture of robot types.

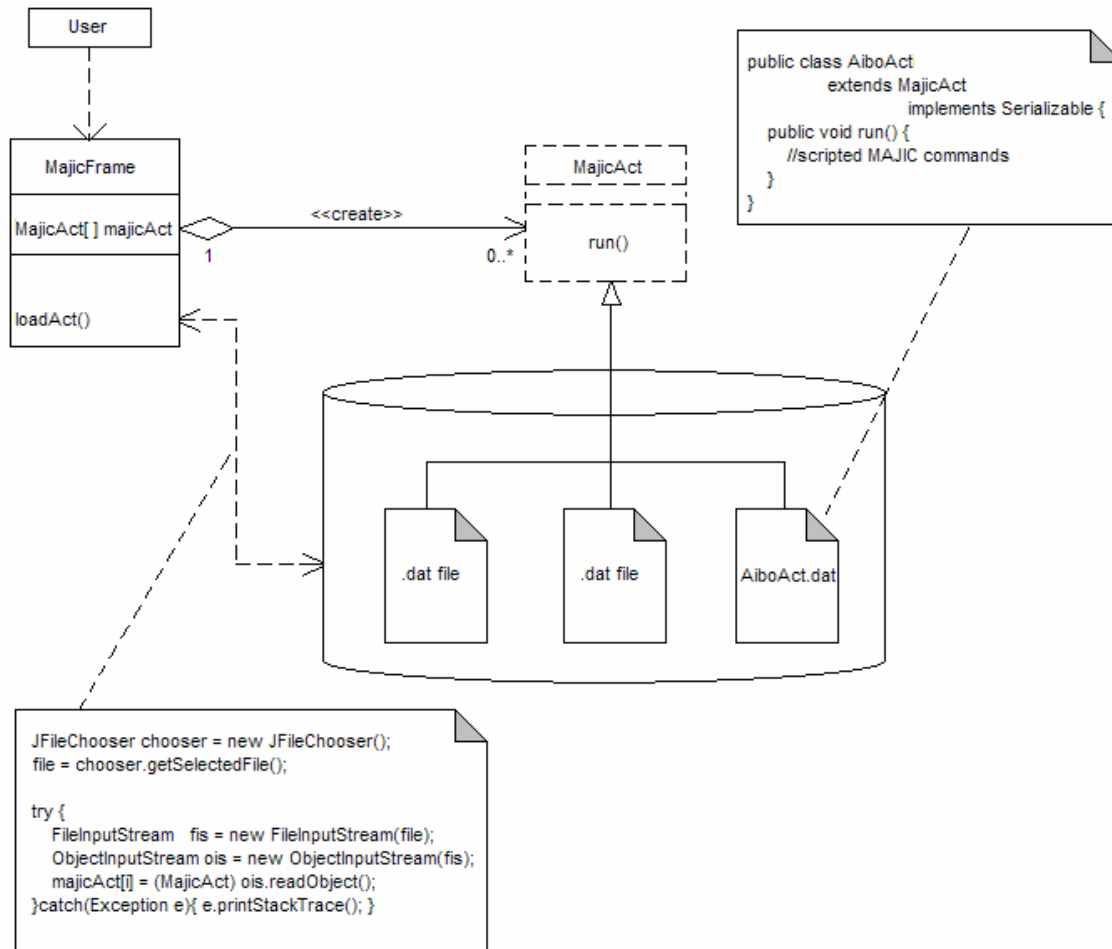


Figure 20. Load Action Class Diagram.

c. Load Action Class Diagram

Again, the prototype design pattern is selected to accomplish the task of loading a MajicAct object from a database directory. The MajicAct abstract class is an extension of the Thread class that implements Serializable. This implementation allows Java to store the class as an object. An object-oriented database design automatically preserves relationships among objects and prevents the requirement for assembling the object data upon retrieval. This flexibility combined with the flexibility of the prototype design allows the user to select and load the desired action onto the desired robot at run-time.

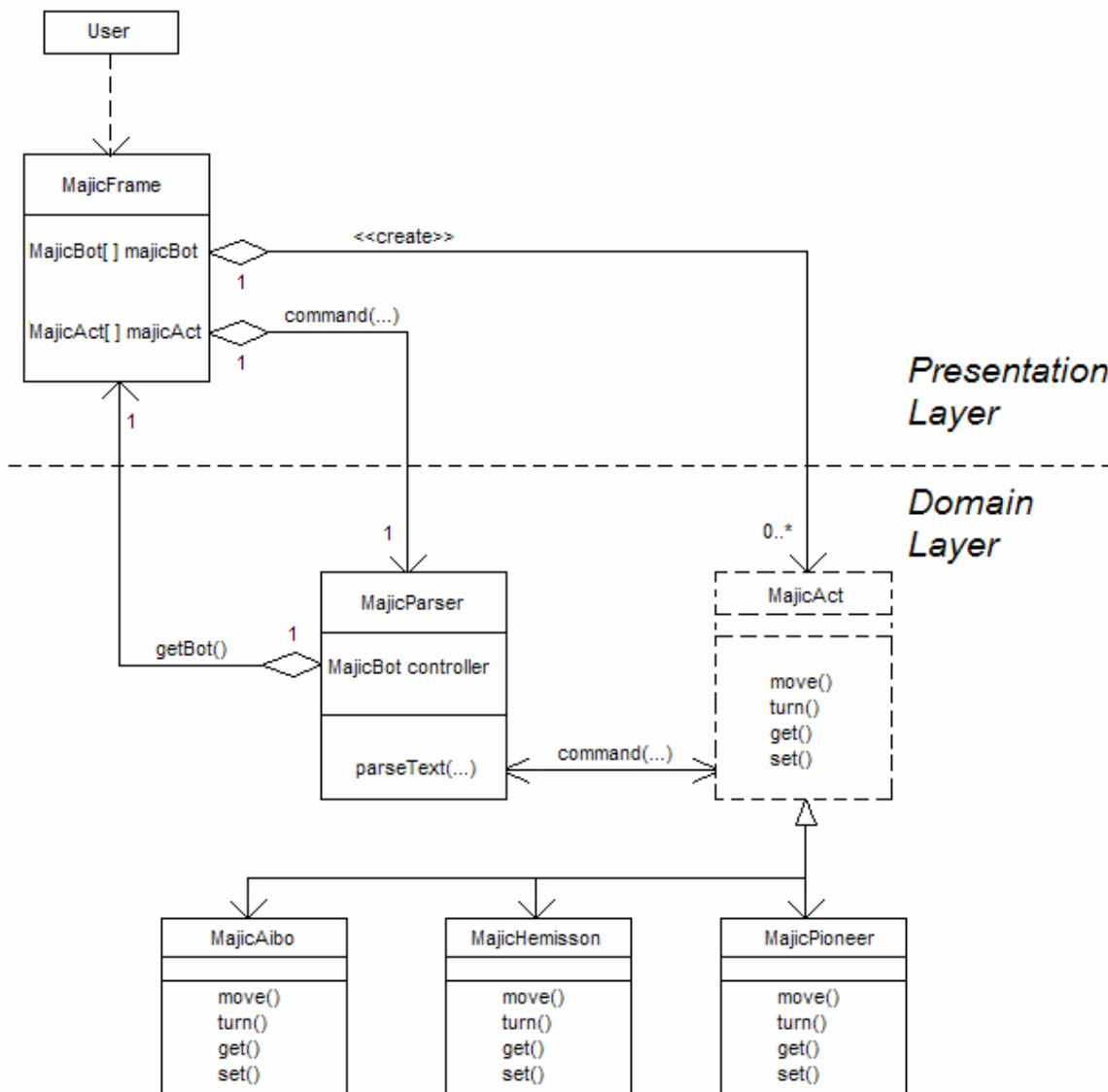


Figure 21. Parse Command Class Diagram.

d. Parse Command Class Diagram

The Parse Command task utilizes the Use Case Controller Pattern. This pattern allows the application to assign the responsibility of parsing a `commandLine` event to the controller class, **MajicParser**. The controller class is only responsible for coordinating the activities specified in the `commandLine` text, and then delegating the

work to the appropriate MajicBot extension. The commands are entered in the presentation layer GUI of the MajicFrame and forwarded to the domain layer controller class to be handled. The MajicParser is responsible for the delegation of all the system events delineated in the UC-4 Issue Command use case.

2. Class Descriptions

All the classes required for the MAJIC application to perform its specified requirements are derived from the diagrams and models previously discussed. In the following section, these classes are specified in further detail. Attributes are identified and assigned types and visibilities. Methods and their interface relationships are determined. A brief description of the overall purpose of each class, as well as a class model are also provided below.

a. The MajicFrame Class

The MajicFrame Class is the front-end, graphical user interface for MAJIC. As detailed in the UC-5 Startup use case and the C1 Startup contract, MajicFrame instantiates the other classes and gathers user input. A handle to MajicFrame is passed to MajicParser to allow the parser to update the protected variable msgArea when the user sends commands to a bot.

MajicFrame
— static int MAX_BOTS = 20 — MajicBot[] majicBot — MajicAct[] majicAct — Updater updater — Timer timer — boolean update — MajicParser commandLine — JTabbedPane mainPane — JPanel[] botPanel — JButton killButton, helpButton, logButton, saveButton, addButton, removeButton, actionButton # JTextArea msgArea # int totalBots
— void addBot() — void removeBot() — void loadAct() — void displayHelp() — void saveLog() + MajicBot getBot(int)

Figure 22. MajicFrame Class Model.

(1) Attributes

int MAX_BOTS: This static integer constant represents the number of elements with which the `majicBot` and `majicAct` arrays will be instantiated. Although currently set to 20 for testing, the number is arbitrary and can be adjusted depending upon the maximum expected number of agents.

MajicBot[] majicBot[MAX_BOTS]: This private variable is an array of `MajicBot` objects set to the size of `MAX_BOTS`. Using polymorphism, various subclasses of `MajicBot` are added to this array by the `addBot()` method. In this manner, the MAJIC controlling classes need not know specifically which extension of `MajicBot` they are currently manipulating. For example, the implementation of the `MajicBot move()` command may differ significantly between the various extended classes, but neither the MAJIC application nor the user need be concerned with how those commands get implemented “under the hood” because the `majicBot` array casts them all as `MajicBot` objects.

MajicAct[] majicAct[MAX_BOTS]: A `MajicBot` can be controlled from the command line or by a special class of Actions called `MajicActs`. Like the `majicBot` array, this array will accept any Action Object specified by the user at run time as long as it meets the requirements prescribed by the `MajicAct` Class.

MajicParser commandLine: MAJIC implements its own limited scripted language using an extension of the `JTextField` class called `MajicParser` (see class description below). `MajicFrame` instantiates this class and passes it a pointer to allow the main GUI frame to be updated once the command has been parsed. Although the `commandLine` field is created and displayed by the `MajicFrame` Class, its listener is contained in the `MajicParser` class, therefore it is this class that houses the logic required to parse user input.

JTabbedPane mainPane: Every robot library has the option of containing its own display panel. Every time a robot is added to the `majicBot` array of currently active robots, its display panel will be added to the `mainPane`. A tab will also appear displaying the number of the bot added. Hovering the cursor over the tab will display a pop-up screen signifying the type of robot that that tab references. Removing a robot from the array with the remove option will also cause the tab and display panel to be removed from the `mainPane`.

JPanel[] botPanel: As discussed above, every `Majicbot` incorporates its own display panel. If not overridden, that panel will be initialized with a default message informing the user that no display is available for the selected robot type.

JButton killButton, addButton, removeButton, actionButton: The last of `MajicFrame`’s private attributes are the button attributes. These correspond to the following actions:

- `killButton`: closes all active robots, kills all active action threads, and terminates the `main()` function of the MAJIC application.

- `addButton`: invokes the `addBot()` method to add a user-specified robot to the `majicBot` array.
- `removeButton`: invokes the `removeBot()` method to remove a user-specified robot from the `majicBot` array.
- `actionButton`: invokes the `loadAct()` method to load a user-specified action object onto the specified robot.

`JTextArea msgArea`: This area of the main screen GUI is the location where users receive feedback concerning their selections and commands. This attribute is protected to allow the `MajicParser` Class to send updates regarding the status of `commandLine` parsing.

`int totalBots`: `totalBots` is a protected integer variable that keeps track of the current number of bots currently being controlled by MAJIC.

(2) Methods

`void addBot()`: `addBot` allows users to select a robot from the current library of `MajicBot` extensions and attempt to establish a connection to that robot. If the connection is successful, the new bot is added to the `majicBot` array and the `mainPane` area.

`void removeBot()`: this method allows users to remove a robot from the array of currently active agents. Once a robot is selected, the method performs some clean-up on the agent by killing any active `majicAct` thread, invoking the `close()` method of the robot, removing the bot's display pane, and removing the bot from the `majicBot` array.

`void loadAct()`: the load action method prompts the user to select a `MajicAct` object from a directory of action objects. Once selected that object thread is activated by first deactivating any thread already active for the robot, then invoking the action objects `start()` method.

`MajicBot getBot( int )`: When instantiated, all bots are assigned a number as part of their name. For example: `bot0`, `bot1`, etc. `getBot` returns a `MajicBot` object based on the number passed to the method. Therefore, to gain access to `bot0`, simply call `getBot(0)`. This method is assigned a public visibility in order to grant other classes access to the array of currently active bots.

b. The MajicParser Class

The `MajicParser` Class listens for `commandLine` events and verifies the syntax of user commands. This allows MAJIC to implement a scripted language that provides access to a variety of `MajicBot` methods and actual sensor and variable values

from the current array of active robots. Commands such as *move()* and *turn()* allow robot manipulation, while commands such as *get()* and *set()* can retrieve and assign values to robot-specific variables.

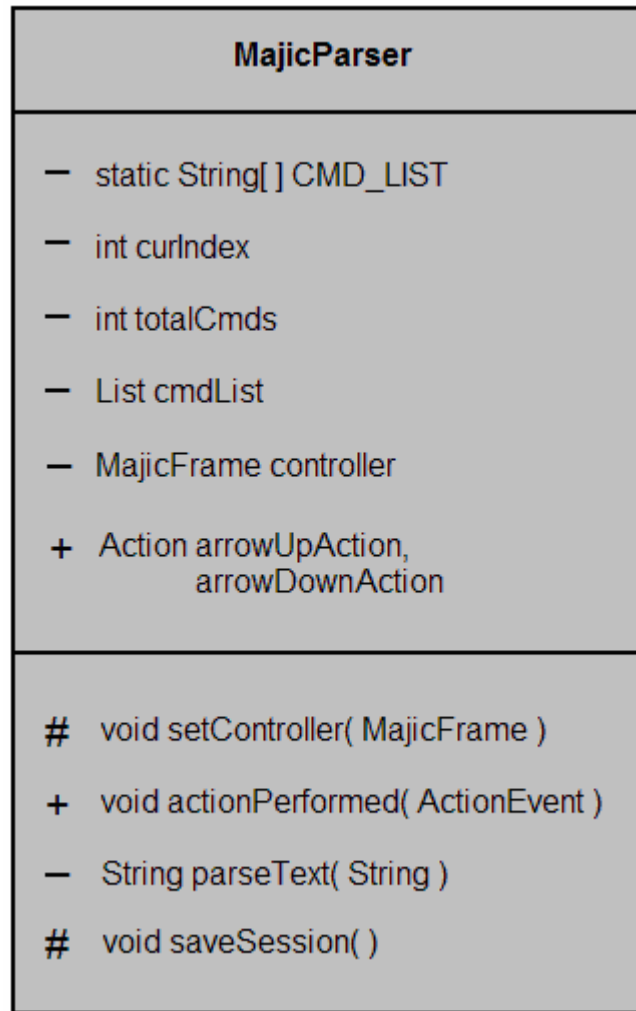


Figure 23. MajicParser Class Model.

(1) Attributes

String[] CMD_LIST: This constant array of string variables is a list of all the commands the parser currently recognizes. This could be increased if future MAJIC developers determine the need for more commands. Increasing commands would also require changing the `parseText( )` method and ensuring that all extensions of `MajicBot` could recognize the command.

int curIndex: this integer is an index that tracks the current location of the command list to be displayed in the commandLine field.

int totalCmds: a counter to keep track of how many valid commands were entered during the current MAJIC session.

List cmdList: This java.util.List is an ArrayList that records all valid commands during a MAJIC session. Presented as a tool to enhance the interface, this function allows the user to scroll through previous commands using the up/down arrow keys to prevent continually typing the same commands.

MajicFrame controller: As described in MajicFrame, the controller is a pointer to the MajicFrame class that allows the parser to update MajicFrame's protected msgArea once commands are parsed.

Action actionUpArrow, actionDownArrow: these actions fire when the arrow keys are pressed and work in conjunction with the command list to cycle previous commands on the command line.

(2) Methods

void setController(MajicFrame): this method is called from the constructor of MajicFrame in order to provide the necessary relationship between the MajicFrame and the MajicParser instances.

void actionPerformed((ActionEvent)): once the MajicParser instance is established as the listener for the commandLine field, any action triggered on the commandLine is sent to this method. Here the commandLine text is passed to the main msgArea for display as well as passed to the *parseText()* method for verification. The response from *parseText()* is displayed in the msgArea as well.

String parseText(String): the parseText method accepts a string argument from the command line and parses it into the local string array, parser[], using a *space* as the delimiter. The elements of the parser string are then checked for the proper format. An error message is returned if the method encounters an invalid command. Valid commands are processed and stored in the cmdList and a status message is returned.

c. *The MajicBot Class*

MajicBot is the prototype for any robot library that is intended to be incorporated into the MAJIC package. MajicBot is defined as an abstract class, which prevents it from being instantiated. It also contains abstract methods that must be overridden by any future extension of the class. These methods are required to allow for

the seamless assimilation of the robot-specific extension into the MAJIC package. The class also contains a static method that allows for the creation of specific robot types, which is necessary for the prototype design pattern.

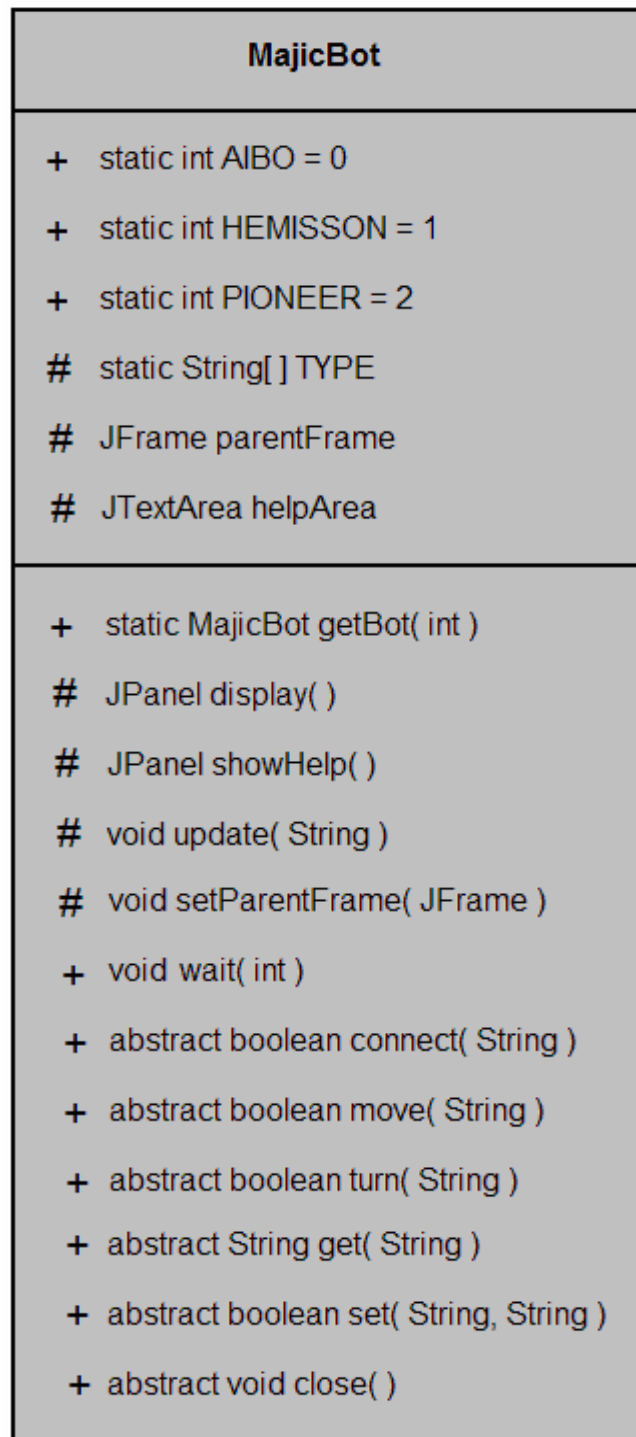


Figure 24. MajicBot Class Model.

(1) Attributes

int Aibo, Hemisson, Pioneer: The static integer attributes of MajicBot are used to index and track all types of robot libraries currently supported by MAJIC. In order to add a new robot type to the MAJIC package, an integer constant must be added to the MajicBot Class. As seen in the MajicBot class model, currently only Aibo, Hemisson, and Pioneer are supported. Instructions on adding an additional robot type are provided in the chapter on implementation.

String[] TYPE: The static constant TYPE is an array of strings that contain all the names of the currently supported robot types. The names of all future robot types must be added to this array of constants when a new library is added.

JFrame parentFrame: The MajicFrame instance establishes itself as the parent of every instance of MajicBot. This allows popup windows generated by an individual MajicBot instance to be properly located on the main screen display.

(2) Methods

MajicBot getBot(int): The static method *getBot()* returns an instance of the appropriate bot to the calling function. The bot that is instantiated depends on the integer argument that is passed to the method, but the bot that is returned is always of type MajicBot. This provides for the polymorphism necessary to allow MAJIC to treat all bots as instances of the MajicBot Class. If an invalid integer is passed to the method (an integer that does not match one of the integer constants) then a null value is returned and an error message displayed.

JPanel display(): This protected method provided a default display in the event that future robot extensions do not provide a GUI interface. The method is not abstract because a specific GUI interface is not an absolute requirement for MajicBot extensions and is provided merely as a courtesy to future programmers. Like the abstract methods, however, the display method can be overridden if future programmers wish to include additional GUI features with their libraries as a companion to scripted commands. If this method is not overridden, the default message “No display currently available” will appear in the display panel of the main screen.

void setParentFrame(JFrame): a method that sets the parentFrame to the JFrame that is passed in as an argument. This will typically be the MajicFrame instance of the current MAJIC session.

void wait(int): Pausing between commands is a common requirement for many robotic evolutions. This method is included in MajicBot to allow all robot extensions a common method to utilize for pausing evolutions.

boolean connect(String): *connect()* is an abstract method that must be overridden by the robot-specific extension of the MajicBot Class. The intention of this method is to return true if a valid connection was established or false otherwise. The argument passed to the method is a string that identifies the type of connection. This generally contains an IP address, COM Port, etc.

boolean move(String): *move()* is an abstract method that must be overridden by the robot-specific extension of the MajicBot Class. The string argument for most robot applications represents a specific time or distance for the robot to move. The method returns a boolean verification of the robot's success or failure to move.

boolean turn(String): *turn()* is an abstract method that must be overridden by the robot-specific extension of the MajicBot Class. The string argument for most robot applications represents a specific time, distance, or direction for the robot to turn. The method returns a boolean verification of the robot's success or failure to turn.

String get(String): *get()* is an abstract method that must be overridden by the robot-specific extension of the MajicBot Class. The string argument for most robot applications represents a robot-specific parameter whose value is returned as a String.

boolean set(String, String): *set()* is an abstract method that must be overridden by the robot-specific extension of the MajicBot Class. The first String argument represents the robot-specific parameter to be adjusted. The second String argument represents the value to assign that parameter. The method returns a boolean verification of the robot's success or failure to perform the action.

void close(): *close()* is an abstract method that must be overridden by the robot-specific extension of the MajicBot Class. The method is called when a robot is being removed from the active array, or when the MAJIC application is shutting down. This method is designed to allow the robot's onboard operating system the opportunity to perform any clean-up functions necessary to exit MAJIC. These functions might include closing any open communications, exiting any MAJIC-related software, etc.

d. *The MajicAct Class*

MajicAct, like MajicBot, is an abstract template for all predefined Action classes that will be loaded into the MAJIC package. MajicAct is defined as an abstract class which extends the Thread class and implements Serializable. The class is implemented as its own thread to allow the MAJIC application to control other robots and respond to user commands while the scripted commands of a particular Action Object are in progress. The class is serialized to allow storing MajicAct extensions as objects in a directory that the user will access at run time.

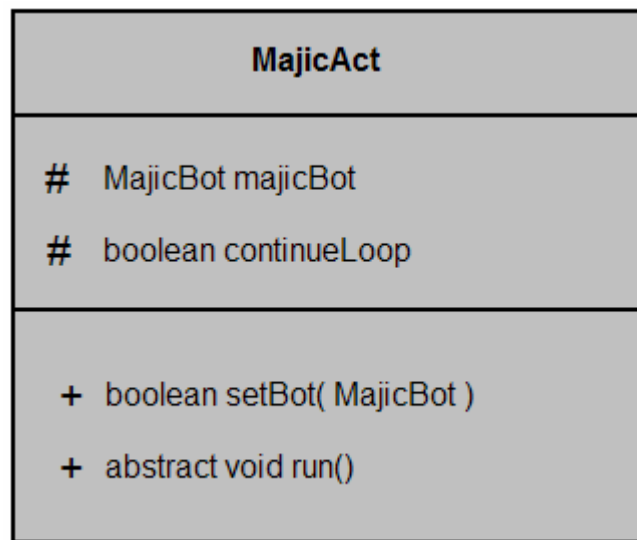


Figure 25. MajicAct Class Model.

(1) Attributes

MajicBot majicBot: the MajicBot instance on which this set of actions will be performed.

boolean continueLoop: a flag that determines if this thread should remain active. This boolean is initialized to *true* in the MajicAct constructor.

(2) Methods

Boolean setBot(MajicBot): Once a MajicAct object is loaded, the MajicBot that it will be controlling must be passed to the class. The method that receives that MajicBot is the *setBot()* method. This method serves several functions. Other than setting the majicBot attribute, it also checks to see if that attribute is *null*. If so, the boolean continueLoop is set to *false*. This boolean value can be used by a MajicAct class to determine when the thread needs to exit its loop and die. The method also returns a boolean which can be used to ensure that this action is only loaded for specific bots.

Void run(): the inclusion of the *run()* method is necessary when extending the Thread class. This method is abstract to ensure that all MajicAct extensions include a *run()* method.

3. Class Extensions

In order for the MAJIC application to assimilate heterogeneous robot Java libraries into the MAJIC architecture, the MajicBot class must be extended. These extended classes must include all the abstract classes defined in the MajicBot class. Although the interfaces to these abstract methods must match exactly, the implementation and source code within them will vary widely from one robot extension to the next. MajicBot extensions are designed to not only integrate into the MAJIC application, but also to serve as stand-alone classes for programmers who wish to do robot-specific programming without using the entire MAJIC framework. Three MajicBot extensions are included with the initial version of the MAJIC application.

a. The MajicHemisson Class

The Hemisson is a small two-wheeled robot designed and manufactured by K-Team for educational purposes. Using its eight infrared sensors, the basic Hemisson robot is able to detect and avoid obstacles and determine the intensity of ambient light. Other equipment on the basic Hemisson robot includes a programmable 8bit MCU, programmable LEDs, a buzzer, two DC motors, and a 9-volt battery. MAJIC also supports the use of K-Team's ultrasonic sensor.

MajicHemisson is an extension of the MajicBot class and is one of three robot specific libraries included in the MAJIC prototype. The class allows full access to all of the Hemisson's standard sensors as well as its sonar capabilities. Although it is included with the MAJIC package, the MajicHemisson class can also be used as a stand-alone class for anyone interested in conducting Hemisson specific programming.

MajicHemisson	
-	SerialComm serialComm
	JPanel hemiMap
	String[] parameter
	String[] paramVal
	double[] ir
	double hiBit
	double loBit
+	JPanel display()
	boolean connect(String)
	boolean move(String)
	boolean turn(String)
	String get(String)
	boolean set(String, String)
	void close()
	String sendCom(String)
	double get(int)
	class HemiMap extends JPanel

Figure 26. MajicHemisson Class Model.

(1) Attributes

SerialComm serialComm: SerialComm is a Java class provided in the download section of the K-Team Hemisson support page. This class uses the Java Communications API and Hemisson's Wireless BlueTooth module to establish a virtual RS232 serial connection with the Hemisson robot.

Serial communication: The Hemisson has a serial command interface to externally control Hemisson's functions and to retrieve Hemisson sensor data. The same serial interface can be used to control and retrieve data from the robot's external modules due to the design of its extension bus. The versatility of this serial interface control allows for un-tethered external command, control, and data acquisition of the Hemisson robot.

Java Communications API: The Java communications API can be used to write platform-independent communications applications. This version of the Java communications API contains support for RS232 serial ports and IEEE 1284 parallel ports. Configuring a system for the Java Communications API requires some extra work and is fully described in the API's documentation and in the implementation chapter of this document. Once installed, the virtual serial port can be utilized to communicate with other robots that use serial communications similar to that of the Hemisson.

Hemisson's Wireless BlueTooth:

Hemisson's BlueTooth specs:

- Speed 115,200 bps (38,400 bps is also supported)
- Frequency 2.4 GHz
- Range 30m
- Operation Maximum of 7 robots simultaneously
- On-Board Processor Microchip PIC18F
- Dimensions 35W x 26D

The BlueTooth Radio Module enables the wireless communication between the Hemisson robot and a PC at 115,200 bps. The link can reach up to 20 meters using the 2.4GHz band. Up to seven robots, equipped with radio modules, can communicate with the same computer at the same time. Direct communication between two robots is not possible but, messages can be transferred through the computer from one robot to another.

JPanel hemiMap: MajicHemisson *display()* method overrides the default MajicBot display to provide a visual display of the Hemisson's current sensor and odometry information. The graphics for this display are updated using the inner class, HemiMap, which is an extension of the javax.swing JPanel class. Any graphical updates to the bot can be invoked using the *repaint()* method of hemiMap.

String[] parameter: The parameter array is a constant array of strings that contains all the valid parameter names for the Hemisson. Currently this includes the wheels, the IR sensors, and the hi and low sonar bits.

String[] paramVal: The values of the parameter array are stored as strings in the paramVal array in order to be passed back to MAJIC in the proper format.

double[] ir: Programmers who use the MajicHemisson Class as a stand-alone class may find parameter values cast as doubles easier to implement into their specific applications vice String values. As a convenience for such programmers, the infra-red sensor values are also stored in an array of doubles. The *get()* method is overloaded to allow access to those double values by passing an index into the double array.

double hiBit, loBit: For the reasons specified above, the hi and low bits of the sonar return are also stored as doubles.

(2) Methods

Jpanel display(): Returns an instance of the hemiMap to be displayed on the main screen tabbed pane.

boolean connect(String): This method overrides the abstract *connect()* method of MajicBot. The method accepts a communication port number as a String argument. If *null* is passed to the method, a pop-up dialog will prompt the user to enter a port. The method will attempt to make a SerialComm connection to that port. A boolean is returned to signify the success or failure of the connection attempt.

boolean move(String): This method overrides the abstract *move()* method of MajicBot. The method accepts a String argument representing the desired speed to assign to the port and starboard wheels of the Hemisson Robot. If the value of the argument parses to an integer and falls between negative and positive nine, the *sendCom()* method is invoked. A boolean is returned to signify the success or failure of the connection attempt. Finally, if the hemiMap exists it is repainted to reflect the updates to the robot.

boolean turn(String): This method overrides the abstract *turn()* method of MajicBot. The method accepts a String argument representing the desired amount of seconds to turn the Hemisson Robot. If the value of the argument parses to an integer, the *sendCom()* method is invoked. A positive value signifies a right turn and a negative signifies left. After waiting the specified number of seconds, a boolean is returned to signify the success or failure of the operation. Finally, if the hemiMap exists it is repainted to reflect the updates to the robot.

String get(String): This method overrides the abstract *get()* method of MajicBot. The method accepts a String argument representing a specific parameter of the Hemisson Robot to be inspected. If the value of the argument matches one of the parameters in the parameter array, the *sendCom()* method is invoked. The response from the robot is

returned as a String. Invalid commands return a *null*. Finally, if the hemiMap exists it is repainted to reflect the updates to the robot.

boolean set(String, String): This method overrides the abstract *set()* method of MajicBot. The method accepts a String argument representing the specific parameter of the Hemisson Robot to be adjusted, and a second String argument that represents the value of that adjustment. If the values of the arguments are valid, the *sendCom()* method is invoked. A boolean is returned to signify the success or failure of the operation. Finally, if the hemiMap exists it is repainted to reflect the updates to the robot.

void close(): The communication port between the MAJIC application and the Hemisson Robot is closed via the *serialComm.close()* method.

String sendComm(String): This is a synchronized method that accepts a String command to be passed to the Hemisson Robot as an argument. This command is sent to K-Team's SerialComm class, which forwards the command to the robot and receives a String reply. That reply is passed back to the *sendComm()* method and is returned to the caller as a String.

double get(int): this method overloads the previous get method in order to return Hemisson parameter values as doubles instead of Strings. Programmers implementing Hemisson-specific programs may find it easier to incorporate double values into the logic portions of their software.

Class HemiMap extends JPanel: this class generates and maintains the graphical updates to the Hemisson display panel by overriding the *paintComponent()* method of JPanel.

b. The MajicAibo Class

The Artificial Intelligence Robot (AIBO) is a four legged robotic pet designed and manufactured by the Sony Corporation. Although intended to be sold commercially as a toy, AIBO has gained popularity within many research laboratories. AIBO incorporates an embedded CPU running at a frequency of 576 MHz and a capacity of 64 MB of main memory. The operating system, known as Aperios, is a Sony proprietary real-time OS kernel. The integration of the onboard computer with a vision system and articulators in a package vastly cheaper than conventional research robots has made AIBO a popular artificial intelligence research tool.

MajicAibo is an extension of the MajicBot class and utilizes the Universal Realtime Behavior Interface (URBI) software package developed by the Gostai Company

and described in this document's implementation chapter. The class allows full access to all of the Aibo's appendages, camera, ir sensors, and LEDs. The MajicAibo class can also be used as a stand-alone class for anyone interested in conducting AIBO-specific programming.

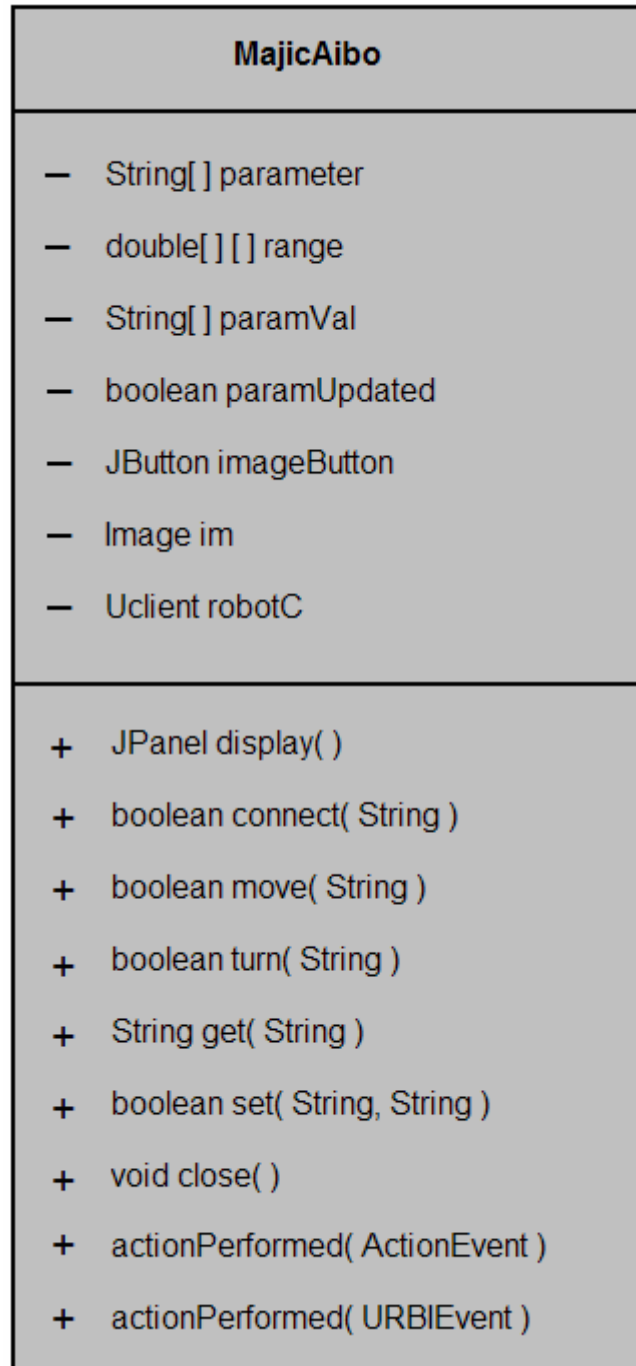


Figure 27. MajicAibo Class Model.

(1) Attributes

String[] parameter: this array of strings contains all the valid parameter names that can be passed to MajicAibo's *get()* and *set()* methods.

double[][] range: this double array contains the minimum and maximum allowable values for each of the parameters in the parameter array. These ranges are used to verify valid arguments in the *set()* method prior to passing the command to the AIBO Robot.

String[] paramVal: this array stores the value of a parameter whenever that parameter is updated by the MAJIC application. Currently, if the robot changes a value independently of the MAJIC application (changing articulator values during a movement for example) the change will not be automatically reflected in the paramVal array. Only explicit calls from the application will update the array. This results in a "last known value" effect for paramVal elements, but the technique is used for ease of implementation.

boolean paramUpdated: parameters are updated via callback methods from the URBI server operating onboard AIBO. paramUpdated is a boolean flag that notifies the program when a parameter value has been returned from the server.

JButton imageButton: the MajicAibo Class overrides the MajicBot *display()* method to provide a custom display for AIBO. Part of the custom GUI contains image captures of AIBO's camera. The imageButton allows the user to take a "snapshot" of AIBO's current camera image.

Image im: AIBO's current camera image is stored in the im attribute.

Uclient robotC: robotC is the URBI client necessary to pass commands from the MajicAibo instance to the URBI server running onboard the AIBO Robot.

(2) Methods

JPanel display(): this method overrides the *display()* method of MajicBot to provide camera images and attribute descriptions as a convenient AIBO reference for the user.

boolean connect(String): This method overrides the abstract *connect()* method of MajicBot. The method accepts an IP address as a String argument. If *null* is passed to the method, a pop-up dialog will prompt the user to enter an address. The method will attempt to connect a UClient to the URBI server. If successful, AIBO will be initialized. A boolean is returned to signify the success or failure of the connection attempt.

boolean move(String): This method overrides the abstract *move()* method of MajicBot. The method accepts a String argument representing the desired duration to assign to the *walk()* method of the URBI server. If positive, the robot walks forward for the desired

number of seconds. If the value of the argument is negative the robot walks backward. A boolean is returned to signify the success or failure of the connection attempt.

boolean turn(String): This method overrides the abstract *turn()* method of MajicBot. The method accepts a String argument representing the desired duration to assign to the *turn()* method of the URBI server. If positive, the robot conducts a right turn for the desired number of seconds. If the value of the argument is negative the robot turns left. A boolean is returned to signify the success or failure of the connection attempt.

String get(String): This method overrides the abstract *get()* method of MajicBot. The method accepts a String argument representing a specific parameter of the AIBO Robot to be inspected. If the value of the argument matches one of the parameters in the parameter array, the *send()* method of UClient is invoked. Once the paramUpdated flag signals an update, the response from the robot is returned as a String. Invalid commands return a *null*.

boolean set(String, String): This method overrides the abstract *set()* method of MajicBot. The method accepts a String argument representing the specific parameter of the AIBO Robot to be adjusted, and a second String argument that represents the value of that adjustment. If the values of the arguments are valid, the *send()* method of UClient is invoked. A boolean is returned to signify the success or failure of the operation.

void close(): The communication socket between the MAJIC application and the AIBO Robot is closed via the *robotC.disconnect()* method.

void actionPerformed((ActionEvent)): This method is called when the button event associated with the imageButton is fired. Once triggered, this method sends a camera request to the URBI camera callback.

void actionPerformed(URBIEvent): This method is called when an URBI event is fired. If the event corresponds to a parameter request, the appropriate paramVal element will be updated and the paramUpdated flag will be set to *true*. If the event corresponds to a camera event, the im attribute will be updated and passed to the display.

c. The MajicPioneer Class

The PIONEER 3-DX8 is a versatile intelligent mobile robotic platform. Utilizing a client-server model, the P3-DX8 offers an embedded computer option, allowing for onboard vision processing, Ethernet-based communications, laser, DGPS, and other autonomous functions. The Pioneer's powerful motors and 19cm wheels can reach speeds of 1.6 meters per second and carry a payload of up to 23 kg. The P3-DX

houses a ring of eight forward sonar sensors with an optional ring of eight rear sensors. Other options include laser-based navigation, bumpers, gripper, vision, stereo rangefinders, compass, and more.

MajicPioneer is an extension of the MajicBot class and utilizes The Advanced Robotics Interface for Applications (ARIA) software package developed by MobileRobots and ActivMedia and described in this document's implementation chapter. ARIA provides dynamic control of the Pioneer's velocity, heading, relative heading, and many other navigation settings as well as managing odometry, sensor readings, and other operating data.

The MajicPioneer class can also be used as a stand-alone class for anyone interested in conducting Pioneer-specific programming as long as they have the MajicPioneerServer Class running on the Pioneer's operating system.

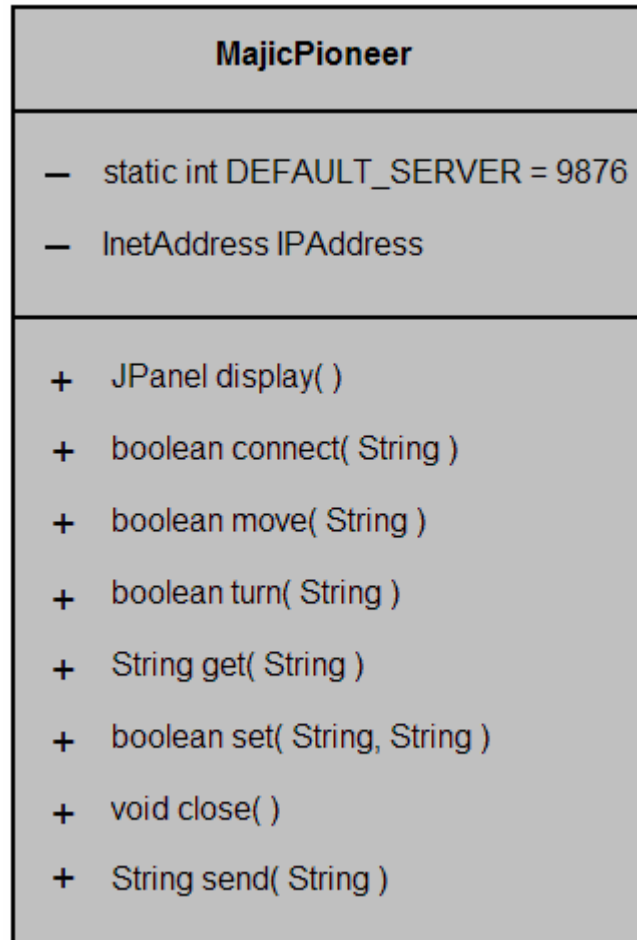


Figure 28. MajicPioneer Class Model.

(1) Attributes

int DEFAULT_SERVER: this integer constant is set to port 9876 which represents the port that the MAJIC server running onboard the Pioneer is monitoring. If custom servers are developed for the Pioneer, this number might need to be adjusted by the programmer.

InetAddress IPAddress: this attribute stores the IP address that currently belongs to the Pioneer Robot being connected to this instance of the MajicPioneer Class.

(2) Methods

JPanel display(): this method overrides the *display()* method of MajicBot to provide a Pioneer-specific display for the user.

boolean connect(String): This method overrides the abstract *connect()* method of MajicBot. The method accepts an IP address as a String argument. If *null* is passed to the method, a pop-up dialog will prompt the user to enter an address. The method will attempt to convert the String address to a valid IP address. A boolean is returned to signify the success or failure of the connection attempt.

boolean move(String): This method overrides the abstract *move()* method of MajicBot. The method accepts a String argument representing the desired amount of millimeters to assign to the *move()* method of the ARIA server. If positive, the robot moves forward the desired number of millimeters. If the value of the argument is negative the robot moves backward. A boolean is returned to signify the success or failure of the connection attempt.

boolean turn(String): This method overrides the abstract *turn()* method of MajicBot. The method accepts a String argument representing the desired degrees to assign to the *setDeltaHeading()* method of the ARIA server. If positive, the robot conducts a right turn for the desired number of degrees. If the value of the argument is negative the robot turns left. A boolean is returned to signify the success or failure of the connection attempt.

String get(String): This method overrides the abstract *get()* method of MajicBot. The method accepts a String argument representing a specific parameter of the Pioneer Robot to be inspected. The response from the robot is returned as a String. Invalid commands return a *null*.

boolean set(String, String): This method overrides the abstract *set()* method of MajicBot. The method accepts a String argument representing the specific parameter of the AIBO Robot to be adjusted, and a second String argument that represents the value of that adjustment. A boolean is returned to signify the success or failure of the operation.

void close(): Several ARIA methods are invoked to clean-up when the *close()* method is called. Those methods include the *stop()*, *disconnect()*, *disableSonar()*, and *shutdown()* commands.

String send(String): the *send()* method passes Pioneer commands from the MAJIC client to the robot's server via a DatagramSocket established using the IPAddress and the DEFAULT_SERVER. The String argument is converted to bytes, placed in a DatagramPacket, and sent to the server. Responses from the server are converted to Strings and passed back to the calling method.

F. DESIGN ALTERNATIVES

An alternative to the common interface design of MAJIC that has garnered much interest by the US military is an attempt to solve the problem of interoperability and information sharing at the protocol level. Projects such as Cursor on Target(CoT) and the Joint Architecture for Unmanned Ground Systems(JAUS) offer component based message passing architectures that define a data format and methods of communication between computing nodes.

In some applications, systems like JAUS have been utilized to automatically generate software wrappers to simplify the development of robotic systems and provide a rapid prototyping environment for use in sensor integration, Operator Control Unit (OCU) development and autonomous vehicle control [20].

Like all architectures, JAUS does not come free of implementation overhead. JAUS comprises three compliance levels. Level 1 compliance requires all communications from subsystem to subsystem (e.g. from controller to robot) to be JAUS messages. Level 2 compliance requires communications between nodes (i.e. processors) to be JAUS messages. Level 3 compliance entails all components (i.e. processes) communicate via JAUS messages [21].

The two core elements to JAUS integration are message packing/unpacking and message delivery. An organization can either develop this code base in-house, subcontract parts or the entire development effort, or purchase a Software Development Kit (SDK) [21].

Even after compliance is attained, interoperability is not guaranteed. As Pedersen points out in his white paper, JAUS currently *promotes* interoperability but does not *provide* interoperability. For example, vendor A may be sending JAUS messages as TCP/IP packets and vendor B may be sending JAUS messages as UDP/IP packets. Although both vendors may be JAUS-compliant, since they are using different transport mechanisms, they will never communicate [21].

IV. IMPLEMENTATION

A. OVERVIEW

Before the MAJIC application could be designed and implemented, extensive research was conducted on each brand of robot in the NPS Autonomous Coordination Laboratory. Establishing communications with and control over individual robots was an arduous task, but a task that revealed many important requirements for the MAJIC project. The beginning of this section details some of the software options explored during the robot-specific research, which of those options were incorporated into the MAJIC package, and why those decisions were made. The remainder of this section provides guides to the installation, general use, and programming of MAJIC.

B. JAVA

Java is an object-oriented programming language developed by Sun Microsystems in the early 1990s under the management of James Gosling. Gosling had several specific goals in mind when creating the language. Chief among those goals was the development of a language that exhibited platform independence and an object-oriented programming methodology.

Enabling the execution of the same program on multiple operating systems is achieved by compiling the program into Java bytecode. This bytecode is interpreted by a Java Virtual Machine (JVM), which is a program written in the native code, and running on the host hardware. This technique allows such features as threading and networking to be conducted in a unified manner despite the proprietary requirements of the host [12]. This platform independence is crucial when developing a system such as MAJIC. The MAJIC application must interface with a multitude of heterogeneous robots, each with their own variety of proprietary protocols and operating systems. The popularity and portability of Java provides essential flexibility to the MAJIC architecture.

The use of an object-oriented language is also an essential aspect of the MAJIC design. Object orientation allows MAJIC to use abstraction to create a generic prototype

of a robot class, called MajicBot. This abstract class can then be extended to allow the utilization of powerful object-oriented features such as inheritance and polymorphism.

Inheritance is a property of object-oriented programming that allows all extensions of a class the ability to utilize the attributes and methods of the superclass as if those functions were their own. Thus, all the attributes and concrete methods of MajicBot are inherited by the subclasses that extend it. This technique provides a certain level of uniformity of implementation for all different types of robot libraries.

Another aspect of extending a Java class that helps to further propagate the standardization of libraries is the requirement to override all abstract methods of the superclass. Conforming to this requirement forces creators of future MAJIC libraries to address all the necessary methods of the MajicBot class when implementing their robot-specific code. This ensures that future classes will obey the proper protocol when interacting with the MAJIC application.

Still another Java property that assures a seamless transition of future Majic libraries is that of polymorphism. Object-oriented polymorphism allows the methods of derived class members to be accessed as if they were members of the superclass as long as the names and parameters match. In the case of the MAJIC application, all robot class instances are instantiated and stored in an array as MajicBots. No matter what robot-specific class is specified by the user at runtime, MAJIC will treat that object instance as if it is a MajicBot instance. Any new robot library can be implemented by MAJIC as long as the MajicBot template has been properly extended.

The final boon that Java brings to the MAJIC package is its current popularity at NPS. Due to its minimalist approach and clean design, Java is an ideal language for teaching object-oriented methodology to students who are new to object-oriented programming [11]. As such, Java is currently the indoctrinating language taught to Computer Science students at NPS. Because of their familiarity with the language, future CS students will be able to easily add new libraries to the MAJIC framework as the NPS robotics lab acquires additional varieties of robots.

As with any language, Java does not come without its limitations and drawbacks. Some researchers worry that Java is too slow to support real time robot control. Yet, the

TeamBots group argues that the real bottleneck to runtime efficiency is sensor and control I/O. In their experience, the benefits of Java (correctness, ease of use, rapid development) far outweighed the negligible runtime overhead [17].

Another drawback of Java is its lack of a standardized serial interface. Many robots, like K-Team's Hemisson, require a serial interface for robot control. However, due to the platform-independence of Java, serial programming in Java requires a standardized API with platform-specific implementations. Two popular options are RxTx and the JavaComm API. MAJIC utilizes and includes the latter. Unfortunately, the JavaComm API does not come with the standard Java packages and must be downloaded separately, and is not without some installation quirks.

Finally, while Java has gained in popularity and is supported by most of the popular robot brands, that support is not always the cleanest and easiest to implement. In the case of MobileRobot's Aria, for example, a Simplified Wrapper and Interface Generator is utilized to allow Java to communicate with the C/C++ encoded operating system. This form of "patchwork" communication can be difficult to implement and can cause the Java Native Interface to throw obscure errors on occasion

C. AIBO

Establishing a Java interface with the Sony AIBO required several iterations prior to discovering and selecting the Universal Realtime Behavior Interface (URBI) [7]. Early experiments were conducted with Sony's R-Code and Open-R Software development kits (SDK) and API's such as PYRO [3] and TEKKOTSU [15].

1. R-Code SDK

Sony's R-Code SDK offers an easy-to-use environment to execute a simplified interpreted scripting language that can be used to program the AIBO ERS-7.

Characteristics: Some of its characteristics include the use of sensor data, variables, R-Code's built-in commands, and its compatibility with Linux, MAC OSX, and Windows.

Advantages: The primary advantage of R-Code is the relative ease with which a programmer can get the SDK set-up and running. The R-Code scripted program can be executed by simply copying it to an AIBO Programming Memory Stick that contains the R-Code system files, or uploading the program while the stick is onboard AIBO using TELNET and a wireless LAN. Java can also send the scripted R-Code commands over a wireless LAN by establishing a socket with the AIBO server.

Disadvantages: Although suitable for hobby users, the ease with which an R-Code script can be written reduces the power that the language can deliver. Because the language is interpreted instead of compiled, it is unable to support complex calculations or large data structures. To fully realize the capability of the AIBO robot, MAJIC requires more flexibility than R-Code's built-in motions and commands offer.

2. Open-R SDK

Sony's Open-R SDK is a cross development environment based on gcc (C++) that allows the development of software for the AIBO ERS-7.

Characteristics: Open-R is a highly modularized development tool that can be run on Linux, MAC OSX, and Windows. The modularized hardware capabilities include changing the robot's form through module exchange, auto detection of the robot's hardware configuration, and module connectivity by a high-speed serial bus.

The modularized software consists of software modules called objects. The programming model allows concurrently running objects to communicate with one another. These connections are defined in a connection description file. Each object is loaded from the Memory Stick making it very easy to replace objects.

Advantages: The power that R-Code lacks is definitely available with Open-R. Programmers can access the Open-R system layer by using the Open-R API in order to completely control all the basic functions of the AIBO ERS-7. This provides access to such features as image processing, optimization of custom walk motions, feedback from sensory information to AIBO's behavior, and more. R-Code also supports wireless LAN and TCP/IP network protocols to provide remote access to AIBO.

Disadvantages: The trade-off for R-Code's powerful access to AIBO is the lack of its simplicity to implement and master. Preliminary knowledge requirements for an Open-R programmer include aptitudes in C++ programming, the use of GNU and Cygwin development tools, the use of shells and UNIX-like commands, and an understanding of how Windows-style paths are mapped to UNIX-style paths.

Another disadvantage of Open-R is that features found in commercial AIBO applications are not freely-available in the Open-R SDK. Interfaces such as AIBO's gait, voice recognition, object recognition, and MIDI sounds are controlled by the proprietary Sony Open-R middleware layer. To implement these features, programmers must develop custom programs that provide this functionality from scratch.

3. Universal Realtime Behavior Interface

URBI is a scripted interface language built on top of the Open-R architecture and designed to work over a client/server architecture in order to remotely control a robot. URBI is more than a simple driver for the robot, it is a universal way to control it, add functionalities by plugging in components, and develop a fully interactive and complex robotic application in a portable way [6]. URBI combines the best qualities of R-Code and Open-R because it is both easy to use and powerful.

One of URBI's major advantages is its ability to maintain a simplistic implementation while providing a high level of capability. With no programming philosophy or complex architecture to become familiar with, URBI is understandable in minutes and immediately useable.

The primary advantage URBI brings to MAJIC is its flexibility. URBI operates independent of the robot, operating system, and platform. URBI interfaces with Java seamlessly to provide modularity, parallel processing of commands, concurrent variable access, and event based programming.

URBI provides MAJIC all the access the application requires to manipulate and utilize AIBO's parameters, sensors, and auxiliary systems. Through URBI, the MAJIC application has all the power it requires to command and control the AIBO ERS-7.

Despite URBI's power and flexibility, there are several limitations to using it as the sole source of command and control of a robot team as opposed to embedding it within an API such as MAJIC.

The first is URBI's lack of a proprietary API designed for heterogeneous robot control. For example, URBI uses several Java programs to control Sony's Aibo and display its sensor information, but has no viable interface to co-ordinate information from more than one Aibo, much less a team of heterogeneous robots. Basically, users must develop their own API designed for their specific applications or a general API for many applications as in the case of MAJIC.

Furthermore, not all robot brands are supported by URBI. In fact, due to its client/server architecture, some popular robots are unable to utilize URBI at all. K-Teams Hemisson, for example, does not have the capacity to run an URBI server on board.

Finally, although the URBI language is relatively easy to learn, making fundamental changes to its architecture would require students to have an extensive knowledge of not only C/C++, but of URBI's "under the hood" design. Attempting to create libraries for future robot additions to the Autonomous Lab could cause students significant time and effort modifying URBI's architecture or simply not be possible at all.

D. ARIA

The Advanced Robotics Interface for Applications (ARIA) is an object-oriented, robot control applications-programming interface developed by MobileRobots and ActivMedia for their inventory of intelligent mobile robots [1]. ARIA provides easy, high-performance robot access and management, including access to a host of accessory sensors and effectors that are available for the pioneer series of mobile robots. ARIA also includes many useful utilities for general robot programming and cross-platform (Linux and Windows) programming as well. Single- and Multi-threaded operations are possible utilizing ARIA's own wrapper around Linux pthreads or Win32 threads.

Most important to the integration of ARIA into MAJIC's Java framework is the fact that each ARIA library has a Java wrapper. This provides programmers the

capability to write Java programs for the pioneer almost as if Java was ARIA's native language. This wrapper is generated by the Simplified Wrapper and Interface Generator (SWIG), which is a development tool that enables the use of C/C++ functions with high-level languages such as Java. SWIG works by taking the declarations found in C/C++ header files and using them to generate the wrapper code that high-level languages like Java need to access the underlying C/C++ code. This "wrapper" library provides a Java API, which simply makes calls using the Java Native Interface (JNI) into the regular ARIA "native" library. Details on implementing Java and ARIA are included in the installation section of this chapter.

E. INSTALLATION GUIDE

The execution of the basic MAJIC application merely requires a system that supports the Java SDK. Sun J2SE JDK 1.5.0 was used to develop the MAJIC software package. This JDK can be downloaded from:

http://java.sun.com/javase/downloads/index_jdk5.jsp

Once downloaded and installed on the system, put the "bin" directory into the systems PATH environment variable. (Environment variables can be set on Windows in the System control panel.)

The JDK also includes the runtime environment (JRE) that allows you to run Java programs. If you only need the JRE, it can also be obtained from the link above.

For Windows, download the Windows installer. For RedHat GNU/Linux, download the Linux RPM. For Debian GNU/Linux, either download the generic Linux self-extracting package, or read the Debian Java FAQ for a more complex procedure for creating an installable .deb package. (first install "java-common" which installs the FAQ at /usr/share/doc/java-common/debian-java-faq/index.html; Chapter 11 is the key information).

Once Java is installed on the system, the core of the MAJIC application will execute by launching the Majic.jar file. On its own, however, the utility of the MAJIC application cannot be achieved. Only through the addition of robot-specific Java libraries can the potential of MAJIC be realized. As previously stated, three such libraries are

included with this package. The Aibo, Hemisson, and Pioneer libraries each requires supporting software to adapt them to the Java environment. The installation of that software is detailed in the following sections.

1. URBI Installation for Aibo

In order to utilize the client/server architecture of URBI, both the client and the server require supporting software. Both requirements can be downloaded from the following website:

<http://www.urbiforge.com>

To establish MajicAibo as the URBI client, the latest liburbi-java must be downloaded from urbiforge. Liburbi-java 0.9.1 was used in the development of the Java client for this version of MAJIC. After the download is complete, installing the urbi-java library is simply a matter of unzipping the file. Programmers who wish to write urbi-java code will have to configure their IDE accordingly.

To establish the URBI server on Aibo, download the precompiled memorystick for the ERS-7. Copy the content of the directory in the root of a blank programmable pink memorystick (a "PMS"). Update the WLANCONF.TXT file with your specific network configuration. Once the memorystick is ready, insert it in the robot and start the ERS-7. The URBI server will be listening to port 54000.

Once these steps are complete the MajicAibo client will be able to establish a connection to the Aibo URBI server via the UClient Class of liburbi-java. For MAJIC users, this connection is made automatically once the pop-up dialogs have been completed.

2. Javax.comm installation for Hemisson

In order for Java to establish virtual serial port communications with Hemisson's Bluetooth Module, two software packages must be acquired. Both come inside the hem.S.JavaEclipse.3.1.zip file that can be downloaded from the Hemisson Download Page located at:

<http://www.k-team.com>

Inside that file is the commapi folder that contains the javax.comm package. Installing the Java Communications API requires platform-specific instructions that are detailed explicitly in the Readme.html found inside the commapi folder.

Also inside the zip file is the HemissonComm folder. This folder contains the SerialComm Class that utilizes the Java Communications API to create a java class that performs the functions necessary to communicate with the Hemisson. This class, developed by k-teams, is used by the MAJIC application to send and receive all the data required to command and control the Hemisson.

3. ARIA installation for the Pioneer

The Advanced Robotics Interface for Applications (ARIA) software package is available from MobileRobots at:

<http://robots.mobilerobots.com>

Versions for Windows, RedHat Linux, and Debian Linux are all available for download at the MobileRobots site. Also available on the same page are the platform-specific installation instructions.

ARIA is utilized on the server side of the Majic-Pioneer interface. The Java class, MajicPioneerServer, uses the ARIA library with Java via a SWIG wrapper and is included with the MAJIC package. This server must be loaded onto the Pioneer's internal operating system and executed before the MAJIC application can communicate with the robot.

F. USER'S GUIDE

The primary goal of the MAJIC application is to provide users an intuitive, high-level interface that grants powerful, low-level control over a collection of robots. The intent of the MAJIC design is to allow students and robotics researches an easy-to-use platform to test their theories in a real time environment. To maintain an uncomplicated interface, MAJIC utilizes a single, main screen for displays, messages, and commands (Figure 29).

The MAJIC philosophy is ease of use without loss of power. As Einstein once stated, “keep it as simple as it needs to be, but no simpler”. In that vain, there is no mind-bending lexicon to master or toolbar-laden interface to slog through in order to use MAJIC. Students can establish a connection with and begin passing commands to their robotic team in a matter of minutes.

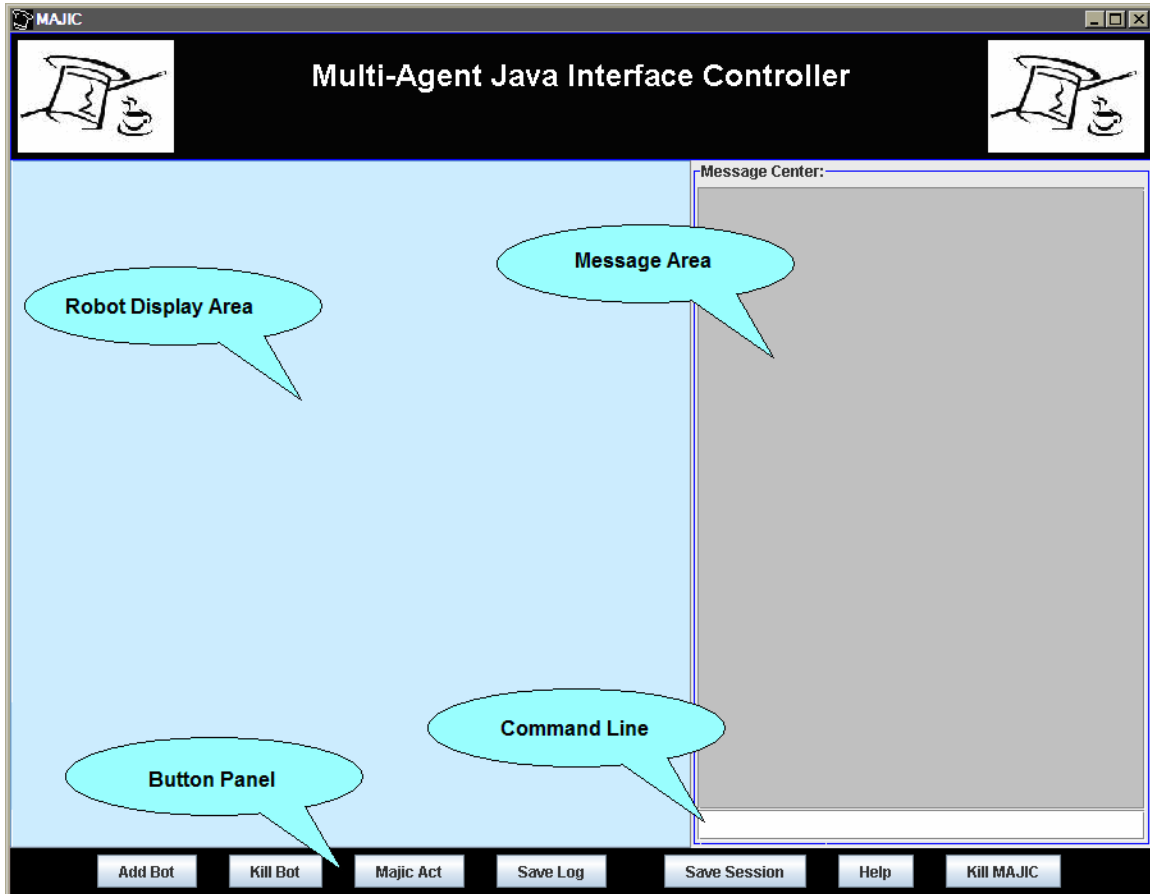


Figure 29. MAJIC Main Screen.

1. MAJIC Main Screen

The main screen of MAJIC is divided into four primary areas as seen above. Each area provides the user a place to perform actions, receive feedback, or do a combination of both. These areas are described below.

a. *Button Panel*

The button panel allows the user to perform several of the commands necessary for the general management of the robot team and the main screen's applications. Operations such as adding and removing robots, loading actions onto robots, saving parameter logs and session, getting help, and quitting MAJIC are performed in this area.

b. *Robot Display Area*

The robot display area is an optional area. Robot Libraries that utilize this area can enhance it with as much or as little complexity as desired. With the use of GUI buttons and fields, this area can provide users with a means to interact with their robots and receive feedback. If the area is not utilized, the default message "No Display Currently Available" will be displayed.

Each display area also generates a tab containing the robot's ID tag. This ID is used to pass commands to the robot. The tab is used to switch to that robot's display area when selected. In this manner, many robot displays can be maintained simultaneously on the main screen without cluttering the display.

c. *Command Line*

The command line provides an area for users to pass commands to or request information from any of the robots currently managed by the MAJIC system.

The format of the commands follows a simple, intuitive MAJIC script. Commands are parsed by the application and translated to the appropriate, proprietary format of the intended robot.

Despite its minimalist approach, the script allows controlling the robot's odometry, heading, peripheral devices, and internal parameters.

As a convenience for the user, the up and down arrow keys on the keyboard allow cycling through previous commands to reduce repetitive typing.

d. *Message Area*

The message area provides user feedback and status updates for all user actions and commands. Parameters requested from the robot are displayed in this area as well as error messages regarding button events.

This area contains a scroll bar that allows the user to scroll through a message history of their current session. Maintaining a historical record of session messages is helpful during trouble-shooting and trial-and-error research.

2. **Adding a Robot to the Team**

Adding a robot is the first action a user will perform after launching the MAJIC application. The only other button that will function without a robot loaded into the system is the *KILL MAJIC* Button, which exits the MAJIC application. Furthermore, no commands can be passed before a robot is connected.

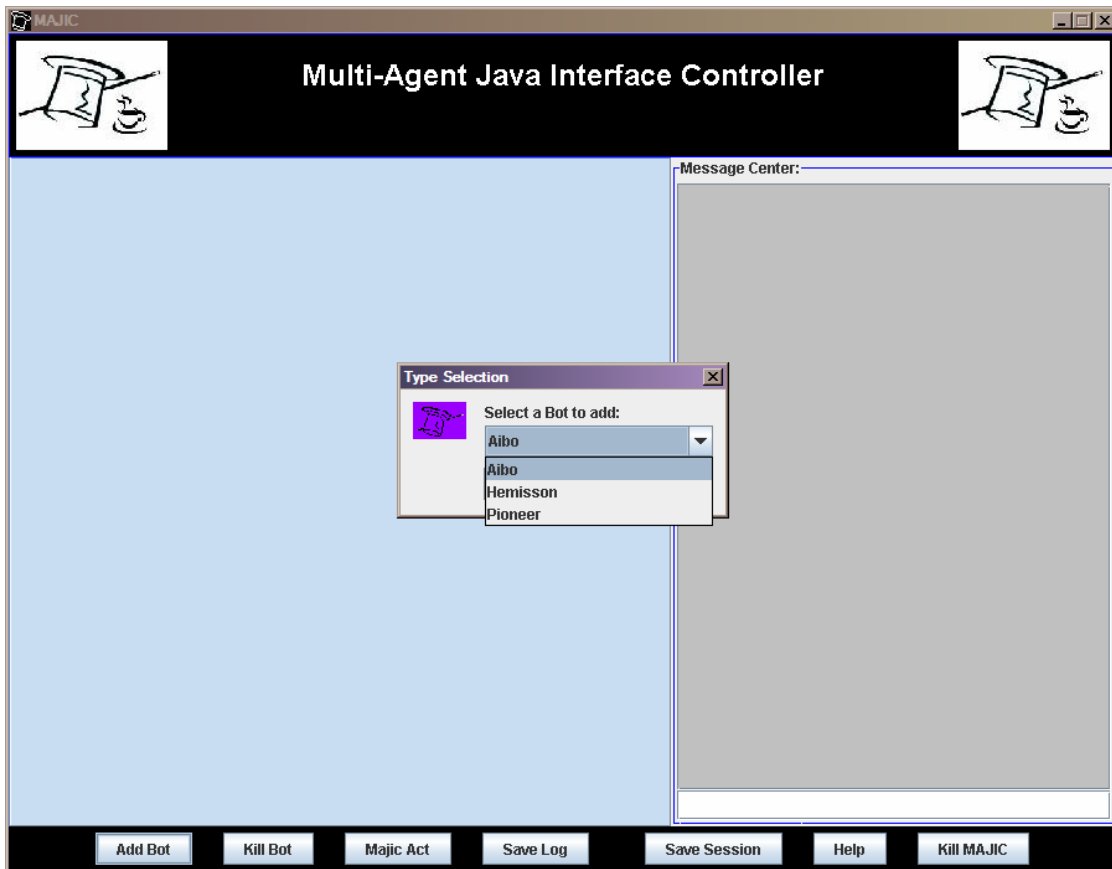


Figure 30. Robot Selection Screen.

The type of connection a robot requires depends on the type of robot and its method of communication. Because this data can vary, two pop-up screens allow users to dynamically specify the type and connection information for the robot to be added. The results of selecting the *ADD BOT* Button are shown in Figures 30 and 31.

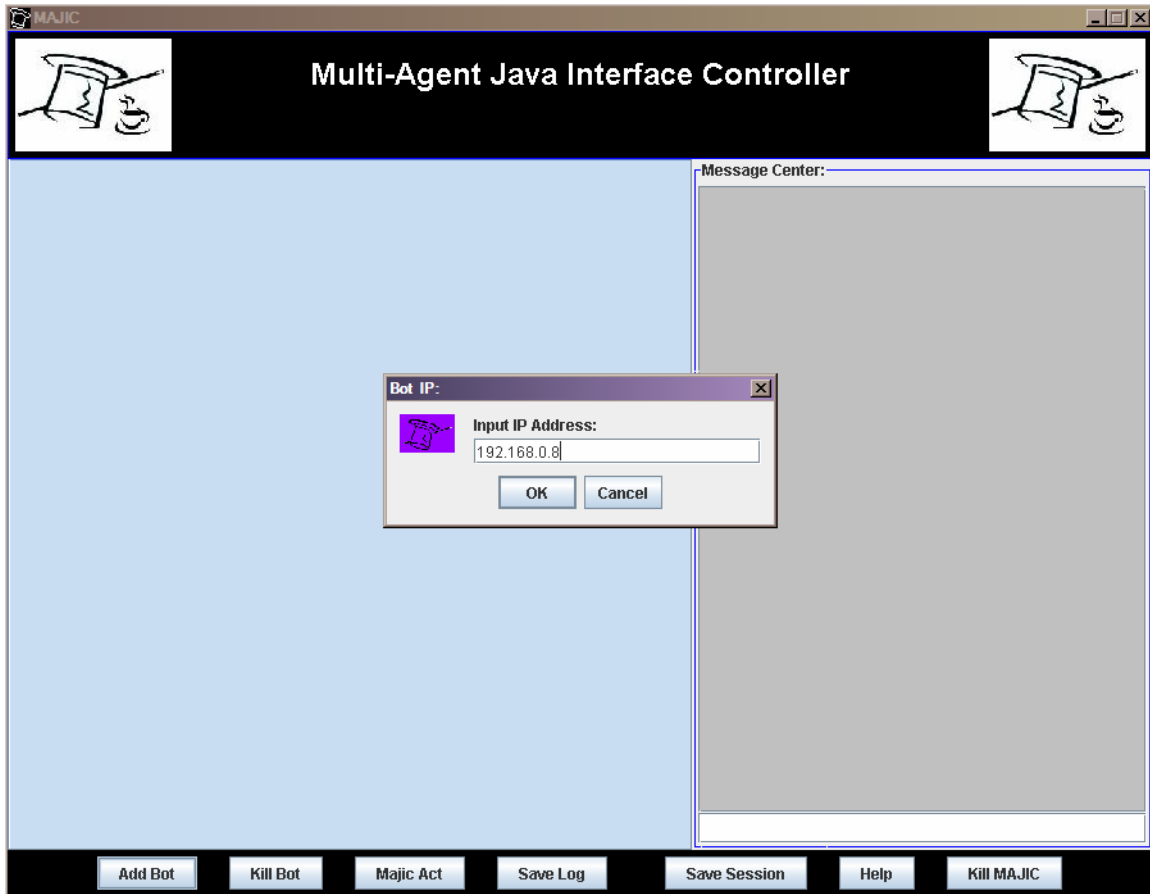


Figure 31. Connection Input Screen.

The drop down menu on the Type Selection screen contains all the brands of robots that this version of the MAJIC application supports. Once selected, a robot-specific connection screen will appear. Once the user has entered the required data and selected the *OK* option, MAJIC will attempt to establish a connection with the specified robot.

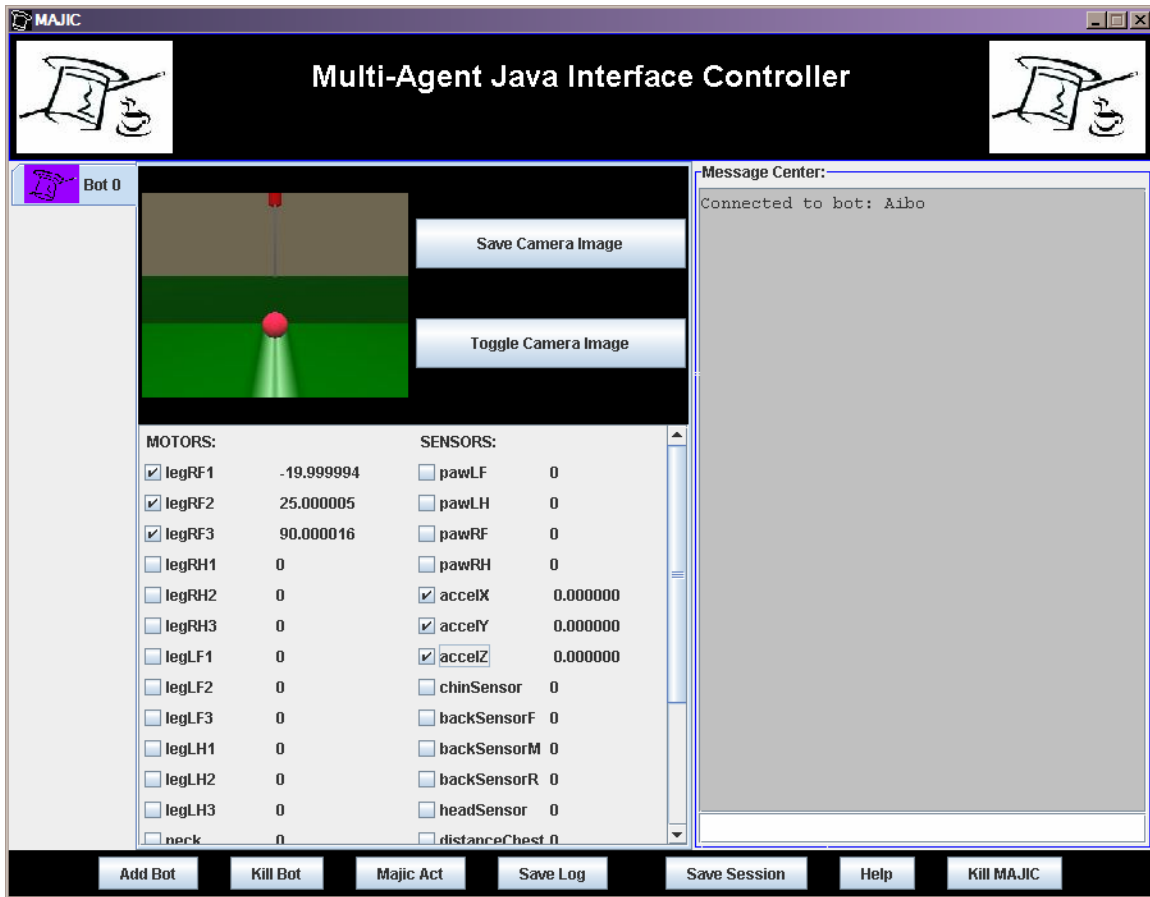


Figure 32. Adding an AIBO.

If a robot is successfully added, its display panel will appear in the tabbed pane of the Robot Display Area. A tab denoting the robot's ID will also appear to the left of the Robot Display Area. This ID is used at the beginning of each command to identify the command's recipient. Finally, a connection verification will appear in the Message Area.

If MAJIC is unable to establish a connection with the specified robot, the display area will remain unchanged and an error message will appear in the Message Area.

3. Passing Commands to the Robot

Once a robot is connected to MAJIC, the user can immediately establish communication with and control over it from the Command Line. User commands utilize the MAJIC Script format, a simple, intuitive script that is detailed and depicted in the table below.

All commands begin with a robot ID and are delineated by a single space. Robot-specific libraries are provided the freedom to implement the commands in a manner most appropriate for their robot type. Due to this flexibility, command arguments may vary slightly among various brands. Students and researches who use MAJIC are assumed to have a basic knowledge of their robot and its parameter requirements.

Commands can be sent to any robot currently active regardless of which robot tab is selected on the Robot Display Area.

Command	Arguments	Description
move	amount	Command directs the robot's odometry to conduct either forward or backward motion. The amount argument specifies a distance, duration, or other robot-specific parameter. Command returns a status message or error report.
turn	amount	Command directs the robot's odometry to change its heading based on indicated amount. The amount argument specifies a direction, offset, duration, or other parameter. Command returns a status message or error report.
get	parameter	Command queries the robot for the value of the specified parameter. The argument can be any valid parameter name recognized by the robot. The message returned contains either the requested value or an error report.
set	parameter value	Command sets the parameter indicated to the specified value. The parameter can be any valid parameter name recognized by the robot. The value may be checked for validity depending on the robot-specific library implementation. Command returns a status message or error report.

Table 3. MAJIC Script Commands.

a. *Robot ID*

All commands start with the following tag:

bot#

- an integer value based on the order in which the robot was added to the MAJIC team. The ID is assigned the lowest available integer greater than zero. The number is used as an ID to specify which robot is to receive the present command. Robot ID's are displayed on each robot's tab in the Robot Display Area.

b. *The MOVE Command*

The format for the move command:

bot# move a

a – an amount that represents the desired movement of the robot. Robot odometry can consist of wheels, legs, wegs, or various other forms of motion. To accommodate all robot types, the move command argument will vary slightly in implementation.

c. *The TURN Command*

The format for the turn command:

bot# turn a

a – this amount can represent a relative heading, absolute heading, degree angle, radian angle, duration, or other value depending on robot implementation.

d. *The GET Command*

The format for the get command:

bot# get p

p – this argument represents the name of the robot's parameter to be polled. Any parameter that is recognized by the robot can be queried by this command including images, infrared readings, sonar readings, etc.

e. The SET Command

The format for the turn command:

bot# set p v

p – this argument represents the name of the robot's parameter to be adjusted. Any parameter that is recognized by the robot can be queried by this command.

v – this argument represents the value with which the specified command is to be adjusted. Depending on the library implementation, this value may or may not be verified prior to being passed to the robot.

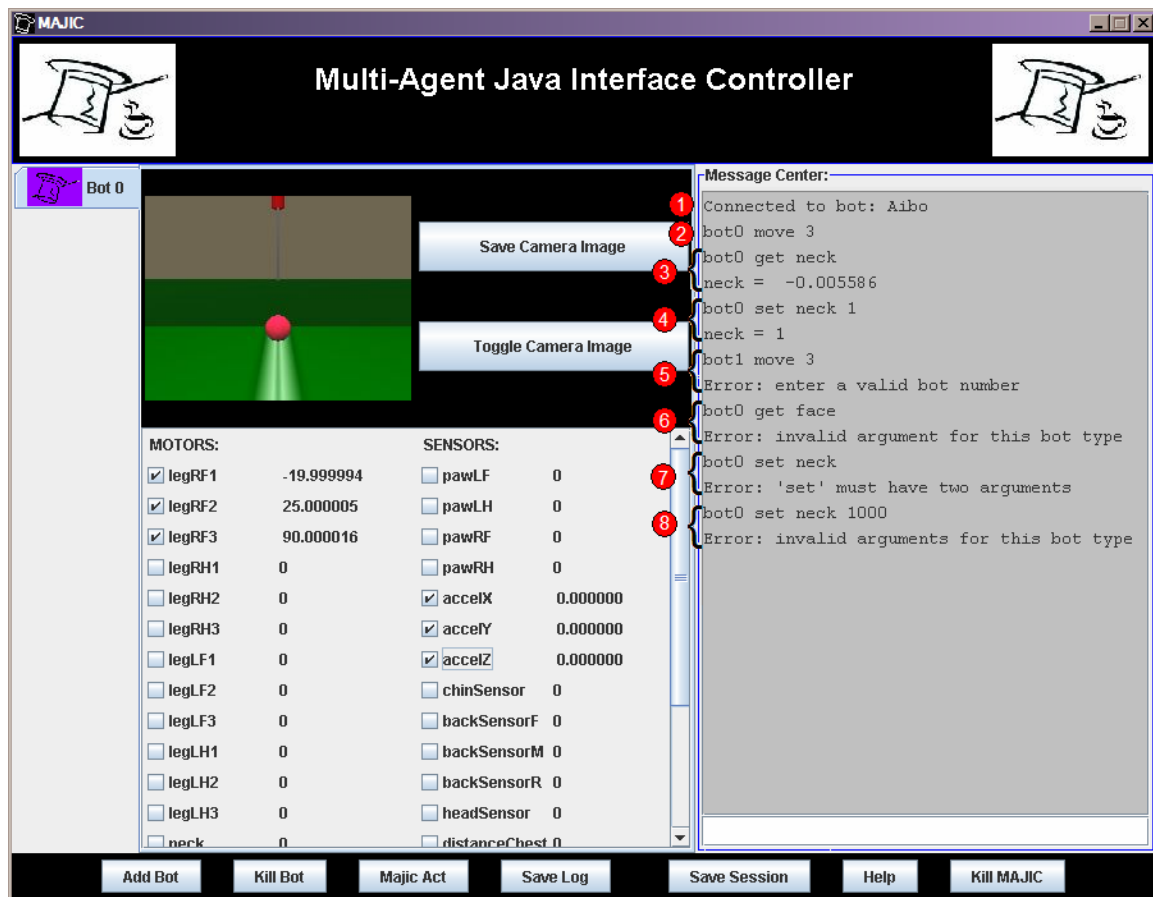


Figure 33. MAJIC Command Line Example.

f. Command Line Example

The figure above depicts the MAJIC message area and command line after several iterations of commands were passed to the AIBO robot, Bot 0.

1. This message indicates that AIBO was connected.
2. This message reflects a move command.
3. These messages reflect a get command and the status report message returned by the command.
4. These messages reflect a set command and the status message returned.
5. These messages reflect an invalid robot identification and the associated error message.
6. These messages reflect a get command with an invalid parameter and the associated error message.
7. These messages reflect a set command with bad syntax and the associated error message.
8. These messages reflect a set command with a value outside the allowable range for the specified parameter and the associated error message.

As seen above, a valid command is displayed in the Message Area, added to the command list, and removed from the command line. If the command is invalid, it will remain on the command line until corrected or overwritten.

4. Invoking the Help Screen

As can be seen above, many parameters can be set and retrieved from the command line. Each robot library has the option to include a help screen to clarify and specify the various parameters that can be manipulated for a given robot type. After the user presses the *Help* button and selects a robot's help screen to view, the screen will appear as an inset to the Message Area.

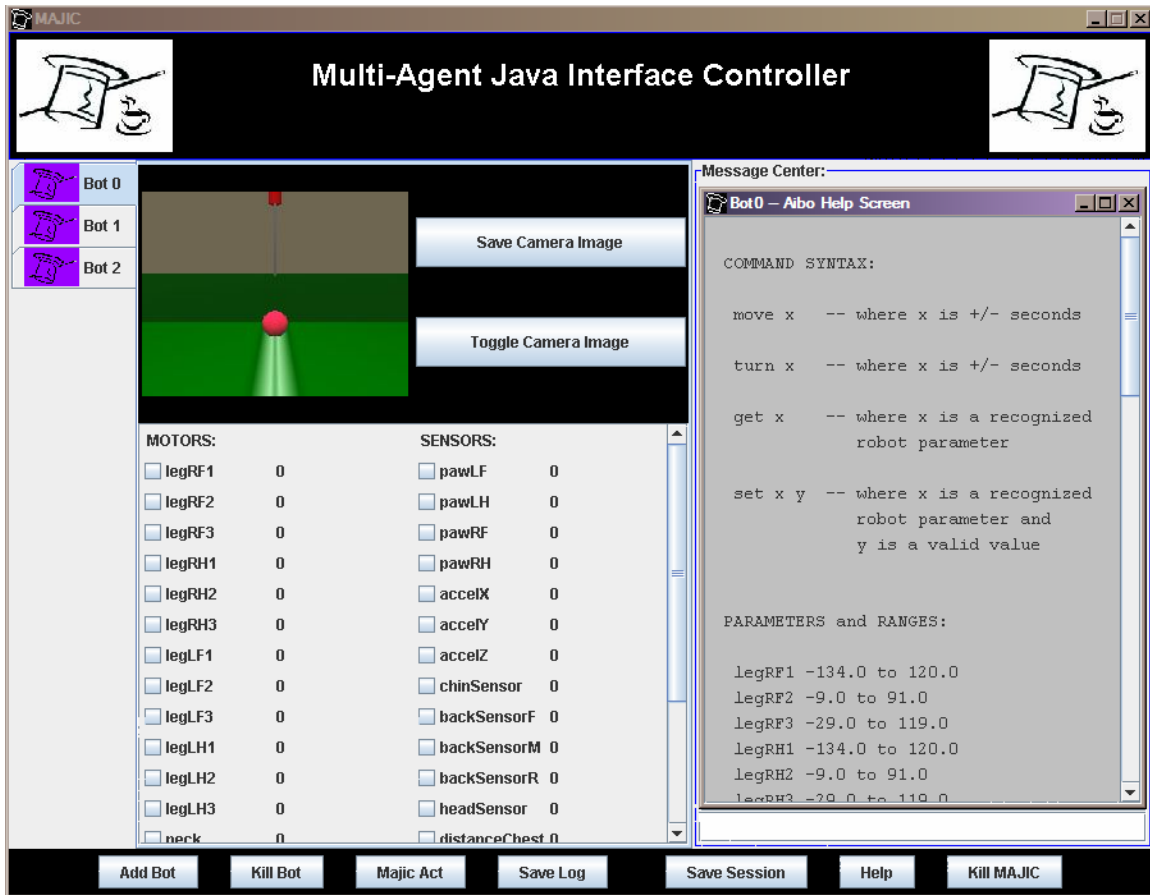


Figure 34. Aibo Help Screen.

As seen above, the Aibo Help Screen specifies the command syntax, parameters, and ranges specific to the Aibo robot. This screen will remain visible until closed, and can be relocated on the desktop to provide a useful reference when passing commands to the robot.

5. Saving a MAJIC Session

Another useful feature to capture command line data for the user is the *Save Session* button. Activating this button allows the user to specify a file in which to record all the commands that were sent to the robot in a given session. This file is stored as a text file in the LogFiles/sessions folder found in the MAJIC root directory. The file can be used to create behaviors called MajicActs that can be loaded on the robot at run time (see Loading Actions on the robot).

6. Saving a Parameter Log

Besides commands, the user will often desire a data log of various robot parameters during a particular event. After activating the Save Log button, selecting an active robot, and entering a file name, a text file will be stored in the LogFiles/parameters folder of the root directory. These parameter log files are very useful for researching, analyzing, and trouble shooting a particular event or behavior.

7. Loading Actions on the Robot

Besides testing single commands on the command line, MAJIC also offers researchers the capability to test pre-programmed *actions* on their robots and robot teams. These actions are Java classes containing a set of MAJIC Scripts that have been saved as data objects. The MajicActs folder of MAJIC's root directory contains a sample set of Java .dat files. These files can be loaded onto the robot by the user at run-time. The steps to perform that operation are detailed below.

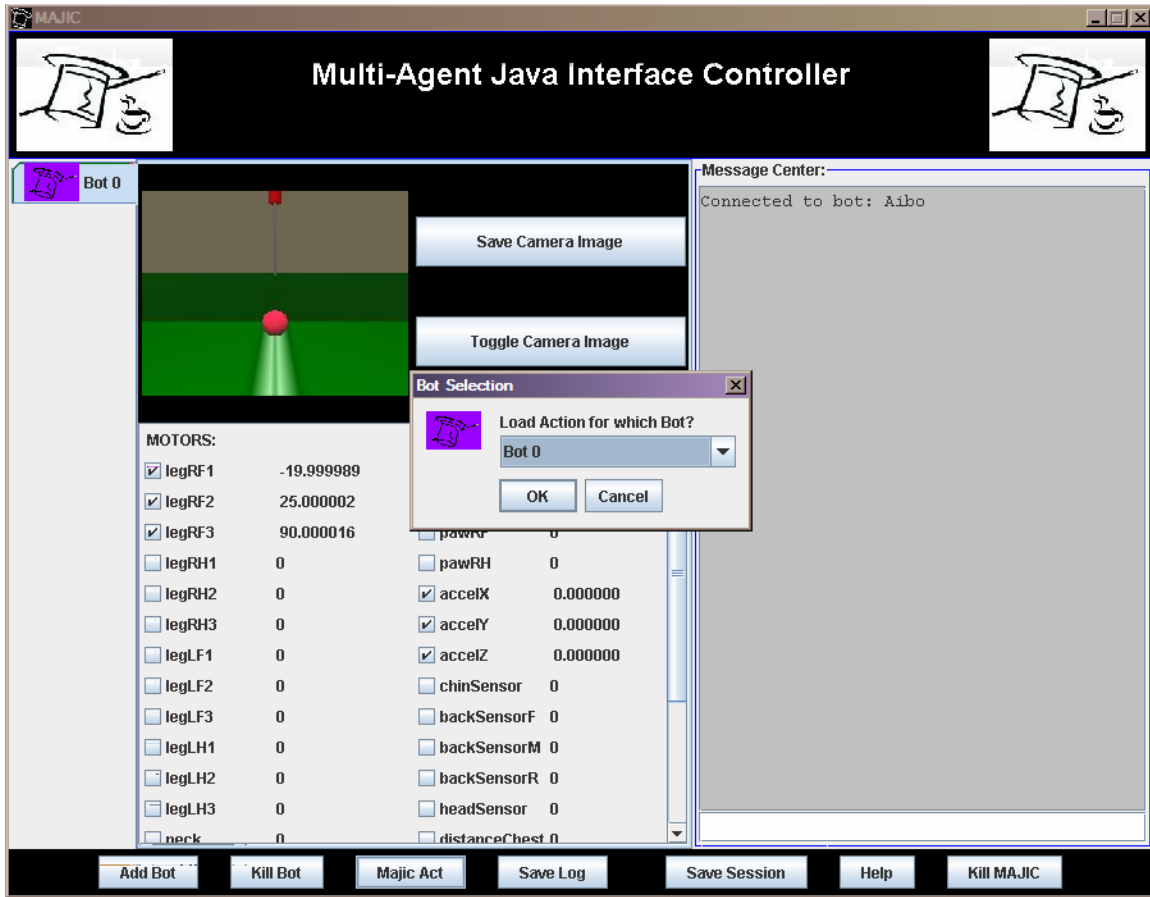


Figure 35. Robot Selection for Load Action.

After selecting the Majic Act button, the user is asked which robot is to receive the action behavior. The dropdown list is populated with all currently active robot IDs. The tab that is selected on the Robot Display Area has no bearing on which robot can be selected from the pop-up menu.

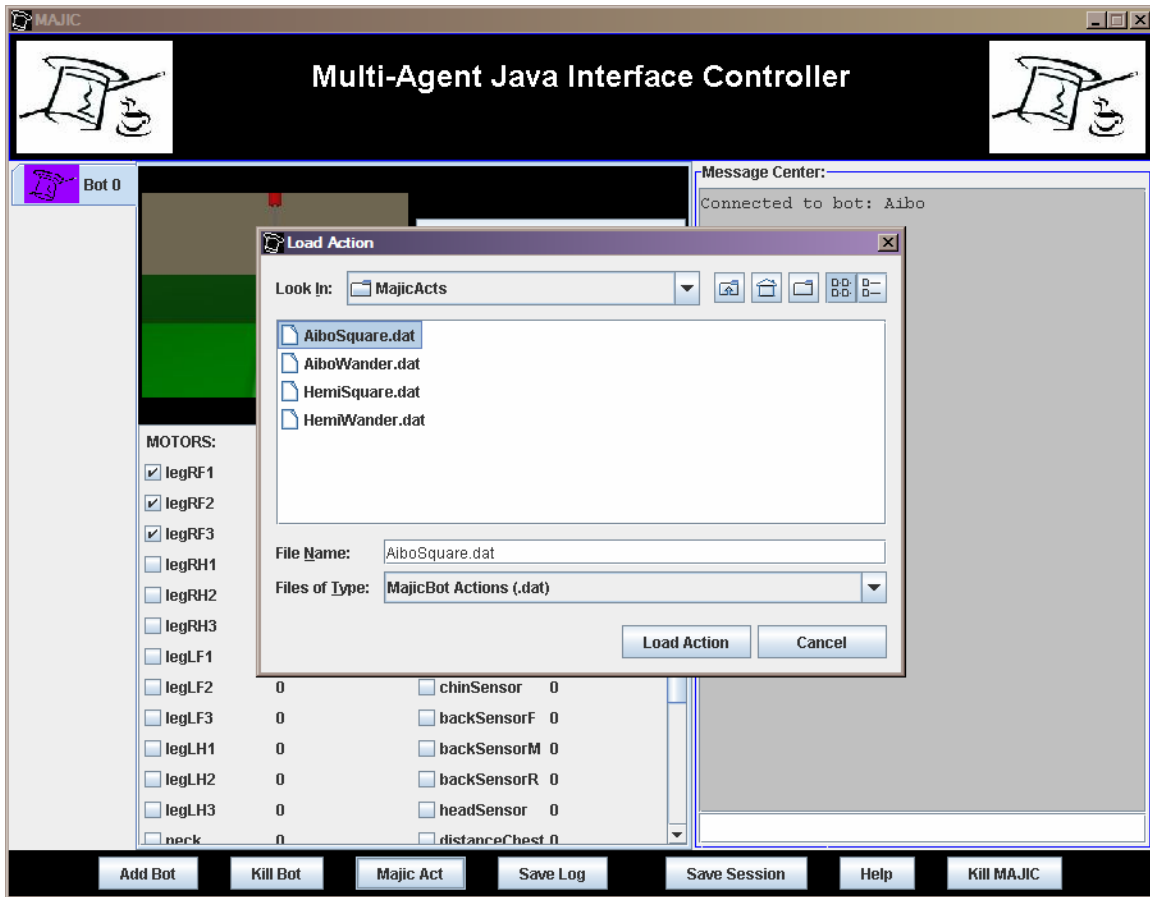


Figure 36. Load Action Selection Screen.

After selecting a robot, the Load Action screen prompts the user to select a pre-programmed MAJIC Action to load onto the robot. Once the file is selected and the Load Action button is pressed, the MajicAct behavior will begin to execute its scripted commands on the robot.

Some Majic Acts are generic and can run on any robot. Others are written for a particular robot type. In the case of the latter, the selected file must match the robot type or the action will be canceled and an error message will post in the Message Area.

8. Removing a Robot from the Team

Reorganizing a team is a useful tool once a robot has completed its obligations. The Kill Bot button is available for users who wish to retire a robot from service. Removing a robot from MAJIC is detailed below.

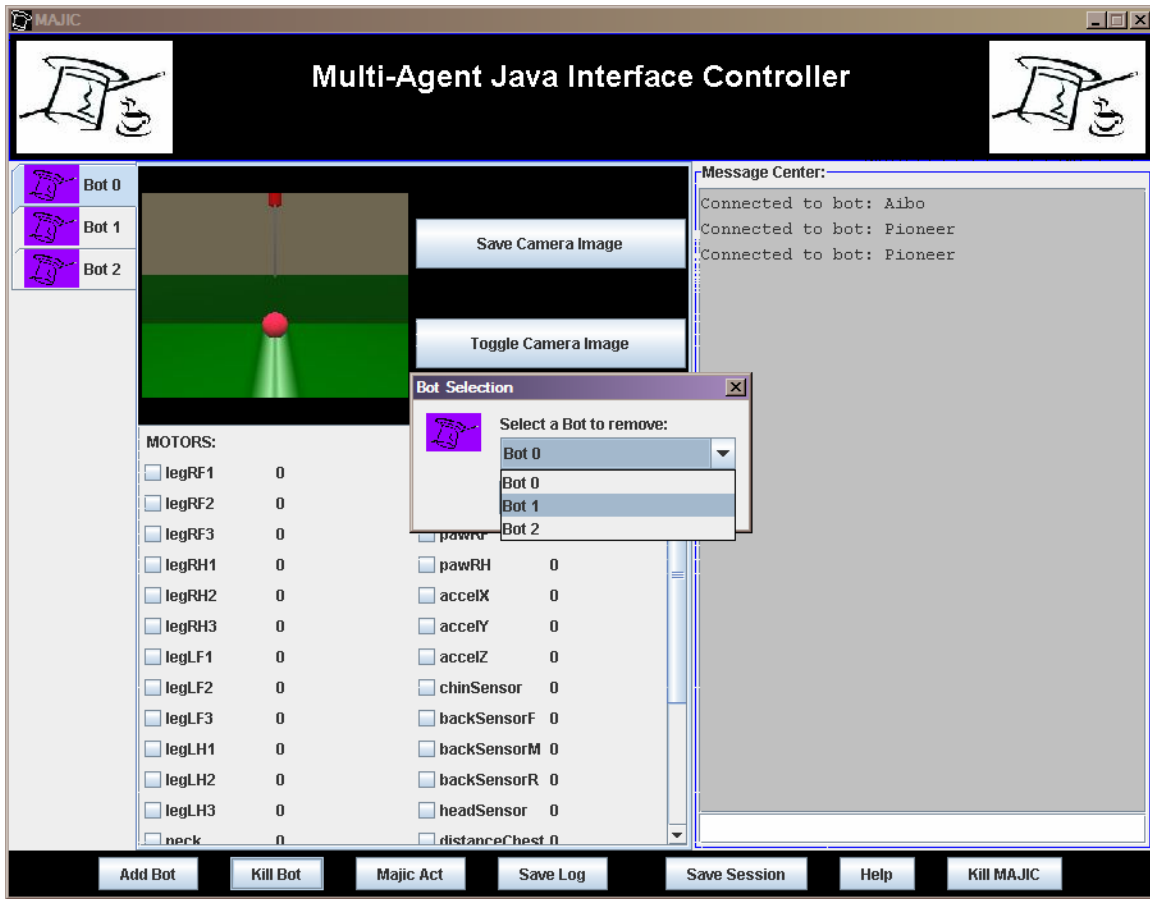


Figure 37. Removing a Robot.

Once the Kill Bot button has been selected, the Bot Selection screen will prompt the user to select a robot to remove. The drop-down list will be automatically populated with all currently active MajicBots. Any robot on the list can be selected for removal regardless of the Bot currently selected in the Robot Display Area.

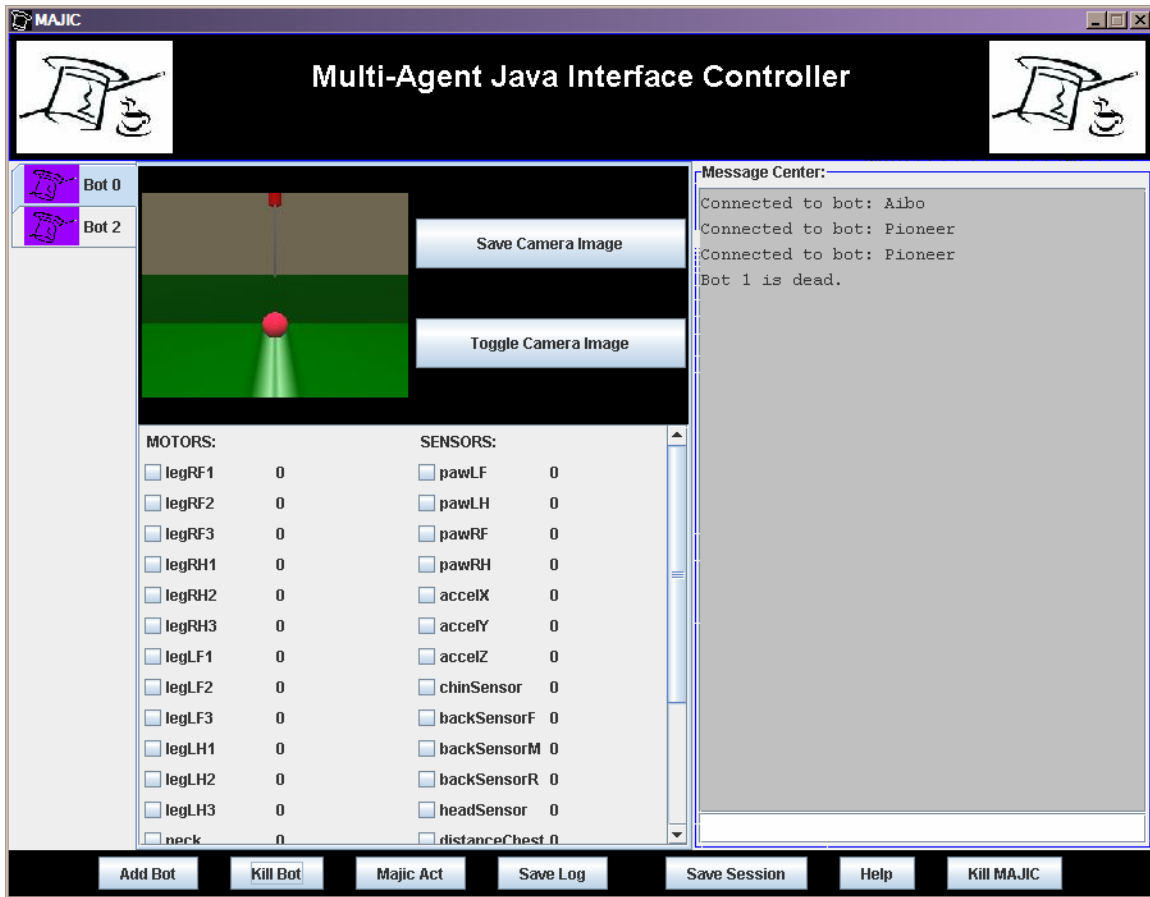


Figure 38. Bot 1 is Dead.

After selecting a robot, the associated tab and display panel will be removed from the Robot Display Area and a status report will appear in the Message Area. The robot ID numbers will not re-adjust to a consecutive ordering, but the next robot added to MAJIC will consume the lowest available ID number.

9. Quitting the MAJIC Application

The final section of the users guide discusses the Kill Majic button. When the user is ready to quit the application, selecting the Kill Majic button will close all connections to any active robots and then terminate the MAJIC application.

G. PROGRAMMER’S GUIDE

All students, researchers, and hobbyists with a desire to conduct experiments in artificial intelligence and robotics are eventually faced with the task of programming their robot. Learning all the proprietary specifications and syntax of an unfamiliar robot requires valuable time and effort that could be better utilized conducting the research itself. Repeating this process for a team of heterogeneous robots can quickly become an extremely involved and overwhelming process. Furthermore, even after command and control of individual robots is established, the programmer is still faced with the arduous task of coordinating them to interact and function as a team.

Programmer’s can use the Java libraries within the MAJIC package to address the above concerns. MAJIC’s flexible and modular design provides programmers with a variety of software development options. Each extension of the MajicBot class can be used as a stand-alone class for robot-specific programming. Alternatively, the classes can be used piece-meal to develop software for a team of specific robots or a specific team goal. Finally, programmers can create and test their own unique robot behaviors by creating extensions of the MajicAct class and saving them as .dat files in the MajicAct folder of MAJIC’s root directory.

Templates for MajicBot and MajicAct extensions can be located in the Utilities folder of the root directory. Also in this folder is the MajicActMaker program that converts MajicAct extensions to .dat files.

1. Stand-alone Programming

MAJIC provides an extra layer of abstraction for Java programmers. In essence, it bridges the gap between a Java program and a robot’s operating system. Since all MajicBot extensions override common methods, many of the time-consuming tasks facing a robot programmer can be abstracted to a higher-level.

Utilizing this higher level of abstraction, novice programmers can gain immediate access to their robot’s parameters and devices. Connecting to the robot, often an involved evolution, is reduced to a simple matter of invoking the connect() method. The

get() and set() methods can be used in conjunction with custom-made logic methods to produce sophisticated decision-making processes and robot-specific operations.

Below is an example of a stand-alone program using the AIBO ERS-7 and the MajicAibo Class. This simple example combines the majic package with programming logic to create a wander behavior for the AIBO.

The WanderDog Class, along with the others referenced by this guide, can be located in the Examples folder of the root directory.

```

1 import majic.*;

public class WanderDog {
    static MajicBot aiboBot;

    public static void main(String[] args){
        double distFwd, distRight, distLeft;

2 aiboBot = (MajicAibo) MajicBot.getBot( MajicBot.AIBO );
3 aiboBot.connect("192.168.0.6");

        while(true){
            distFwd = getDist();

4 if(distFwd > 100){
5     aiboBot.move("5");
        }else{
            aiboBot.set("headPan", "-90");
            distLeft = getDist();

6 aiboBot.set("headPan", "90");
            distRight = getDist();

            if(distRight > 100)
2 aiboBot.turn("3");
            else if(distLeft > 100)
                aiboBot.turn("-3");
            else
                aiboBot.turn("5");
        } //end if
    } //end while
} //end main

private static double getDist(){
    double d = 0;

8 String reply = aiboBot.get("distance");

    try{
        d = Double.parseDouble(reply);
    } catch(Exception e){}

    return d;
}
}

```

Figure 39. The WanderDog Program.

While it is not completely necessary to understand every line of code in the WanderDog program, it is important to understand what is required to write future stand-alone software.

Circle 1 denotes Java's import statement. The majic package must be imported into the program before utilizing MAJIC classes.

Circle 2 shows how an instance of MajicAibo can be instantiated by typecasting the object returned from the static method getBot() of MajicBot. An alternate method of instantiating an object for AIBO specific programming would be the following:

```
MajicAibo aiboBot = new MajicAibo();
```

With this method, the MajicBot typecasting can be avoided. In fact, with this method only the *majic.MajicAibo* class would need to be imported at Circle 1.

Circle 3 invokes the connect() method of the MajicAibo Class passing it an IP address. This makes connecting to the robot a very simple process.

Circle 8 gets a reading from the robot and Circle 4 uses it determine a course of action. Circles 4 and 8 begin to show how convenient yet powerful MAJIC programming can be when combined with standard conditional logic statements.

Circles 5, 6, and 7 demonstrate other uses of familiar MajicAibo methods to control the robot and assist in the decision-making process.

WanderDog is a toy example that was not designed to be particularly robust or fascinating. Even so, despite its apparent simplicity, it will actually cause an AIBO robot to wander around the room and make an attempt at basic obstacle avoidance. This in itself shows the potential for developing powerful robot-specific programs using a single, stand-alone class from the MAJIC libraries using very few lines of code.

2. Creating Robot Libraries

As demonstrated in the WanderDog example, using a robot library can provide novice programmers a simple and powerful interface to their robots. Undoubtedly, additional libraries will be necessary in order for MAJIC to maintain its relevance and utility to future programmers.

Generating a library is no simple task. First and fore most, a library programmer must discover or develop a means of communicating with and controlling the robot from a Java environment. In most cases, this process will involve the establishment of a client/server relationship in which a Java extension of MajicBot will act as a client to the robot's operating system. This client class usually exchanges proprietary data with the robot server over a wireless or virtual serial port connection.

Once this communication and control capability is established, incorporating the library client into the MAJIC architecture is merely a matter of some simple housekeeping updates in the abstract MajicBot Class.

The Generic Template for the MajicBot Class is pictured below and can also be located in the Utilities folder of the root directory.

```

// This class must be placed in the magic directory to be included in the package
package magic;

// only necessary if you override MajicBot's display() or showHelp() methods.
import javax.swing.*;

public class GenericMajicBot extends MajicBot {
    /*
        // Optional Overriden Methods

        // generate robot-specific display
        public JPanel display() {
            JPanel panel = new JPanel();
            return panel;
        }

        // populate robot-specific help file
        protected JPanel showHelp(){
            super.moveUnits = "my_robots_units ";
            super.turnUnits = "my_robots_units ";

            JPanel helper = super.showHelp();

            super.helpArea.append("my_robots_additional_information");

            return helper;
        }
    /**/

    public GenericMajicBot(){ }

    ////////////////////////////////////////
    // Required Overriden Methods
    ////////////////////////////////////////
    public boolean connect(String port){
        return true;
    }

    public boolean move(String arg){
        return true;
    }

    public boolean turn(String arg){
        return true;
    }

    public String get(String arg){
        String response = null;
        return response;
    }

    public boolean set(String arg, String value){
        return true;
    }

    public void update(String filename){}

    public void close(){ }
}

```

Figure 40. The Generic MajicBot Template.

The requirements to implement the template are documented by the Java comments and briefly described in the following paragraphs. One area of note is the optional *display()* and *showHelp()* methods that are commented out.

The *display()* method can be overridden if there is a necessity or desire to incorporate a Graphical User Interface for the libraries specific robot. A default message stating that there is currently no available display for this type of robot is provided in the abstract MajicBot Class. This display panel can provide significant user interaction and feedback is implemented by the library programmer. When combined with the *update()* method of MajicBot, the display can even be dynamically updated every second with current robot parameter information.

The *showHelp()* method can also provides the programmer with an option to provide the user with useful information about the specific robot that is referenced by the MajicBot extended library. Again, the MajicBot Class provides a default help file, but odometry units are generalized, and robot-specific parameters are not addressed.

If neither of these optional methods are overridden, the javax.swing import can be disregarded and the default display message and help file will appear in the Robot Display Area of the MAJIC application.

Once the MajicBot extension is completed, it must still be incorporated into the MajicBot Class. Several simple steps are required to add the library to the static attributes and methods of MajicBot. Those steps are labeled in the MajicBot excerpt in the figure below.

```

package majic;

import javax.swing.*.*;

public abstract class MajicBot {
    public static final int AIBO = 0;
    ❶ public static final int HEMISSON = 1;
    public static final int PIONEER = 2;

    ❷ protected static final String[] TYPE = {"Aibo", "Hemisson", "Pioneer"};

    protected JFrame parentFrame;

    protected MajicBot() {}

    public static MajicBot getBot(int type){
        switch (type){
            case AIBO:      return new MajicAibo();

            ❸ case HEMISSON: return new MajicHemisson();

            case PIONEER:   return new MajicPioneer();

            default:        System.out.println("Error creating MajicBot!");
                           Thread.dumpStack();
                           return null;
        }
    }
}

```

Figure 41. The Abstract MajicBot Class

Static constants are included in the MajicClass to improve code readability and clarity.

To avoid vague integers in the getBot() method and switch statement, the name of the new robot should be added to the list of constants (Circle 1).

The name should also be added to the TYPE array (Circle 2). This array is used to populate the drop-down menus of the MAJIC application.

Finally, a case statement must be added in order for MajicBot to create an instance of the new robot library when requested.

Once these three steps are completed and the MajicBot Class is recompiled, the MAJIC application will be able to fully employ the new robot library.

3. Performing a Majic Act

Although the command line of the MAJIC application allows robot researches to test their robot's reaction to single commands in a real time environment, a researcher may at times find this too limited. In some instances, the student will have an entire behavior or set of actions that need to be tested on the robot. While it would be possible to code in all of these scripted actions from the command line, that process would undoubtedly prove tedious with repeated trials. Furthermore, capturing immediate responses to environmental stimuli would not be possible from the command-line approach when dealing with complex robot behaviors.

a. The MajicAct Class

To offer a convenient, research-friendly approach to behavioral testing, MAJIC provides programmers another abstract class, the MajicAct Class.

```

package majic;

// required to serialize the object
import java.io.*;

// threaded to run in the background of the MAJIC application
public abstract class MajicAct extends Thread implements Serializable
{
    // the robot that receives this behavior
    protected MajicBot majicBot;

    // a flag to kill the thread
    protected boolean  continueLoop;

    protected MajicAct(){
        continueLoop = true;
    }

    // passing a null argument kills the thread
    public boolean setBot(MajicBot mBot){
        majicBot = mBot;

        if(mBot == null)
            continueLoop = false;

        return true;
    }

    public abstract void run();
}

```

Figure 42. The MajicAct Class

The MajicAct Class utilizes the Prototype Design of the Creator Design Pattern. As such, this abstract class is designed as a template that must be extended by classes that provide behavior-specific implementations of the class. These implementations can either be specific to a particular robot type or generically applicable to all MAJIC robots.

b. The AiboSquare Class Extension

A robot-specific example, the AiboSquare Class, is provided and explained in detail below.

```

import majic.*;

import java.io.*;

❶ public class AiboSquare extends MajicAct implements Serializable {

    ❷ MajicAibo aiboBot;

    ❸ public boolean setBot(MajicBot mBot){
        ❹ super.setBot(mBot);

        ❺ if( !(mBot instanceof MajicAibo) )
            return false;

        ❻ aiboBot = (MajicAibo) mBot;
        return true;
    }

    ❼ public void run(){
        ❽ aiboBot.wait(4000);
        aiboBot.move("5");
        aiboBot.turn("3");
        aiboBot.move("5");
        aiboBot.turn("3");
        aiboBot.move("5");
        aiboBot.turn("3");
        aiboBot.move("5");
        aiboBot.turn("3");
        aiboBot.wait(6000);
    }
}

```

Figure 43. MajicAct AiboSquare Example

Circle 1 marks the class descriptor that is required by all extensions of the MajicAct Class. To allow the concurrent execution of behaviors for multiple MAJIC robots, Majic Acts are implemented as extensions of Java's Thread Class.

Majic Acts must also implement Java's Serializable interface in order to be stored in the MajicActs directory as *.dat* object files. This process is required in order to allow the user to select and load MajicAct objects from this directory at run-time. A MajicActMaker Class is provided in the MajicActs folder of the root directory and its use is explained later in this section.

Circle 2 marks the AiboSquare attribute, aiboBot. This allows for the robot-specific instantiation of a MajicAibo attribute. If this behavior was intended for generic use, this attribute could be omitted.

At Circle 3, the *setBot()* method of the MajicAct super class is overridden to provide it with additional functionality. The original method is called at Circle 4 to establish the robot that this behavior is intended to control.

Because this class is intended for a specific robot, the mBot attribute is checked for the proper type at Circle 5. If the robot argument is a valid Aibo instance, the aiboBot attribute will be type cast to the robot argument (Circle 6).

The *run()* method at Circle 7 is necessary for all MajicAct extensions regardless of what robot type the extension is designed for because they are extensions of the Thread Class. The *run()* method is the heart of the MajicAct Class. This method contains the scripted commands that comprise the intended behavior. In cases where a behavior requires continuous looping, the MajicAct *continueLoop* attribute can be utilized to monitor and kill the thread.

c. Creating a Majic Act with a Session File

When creating a behavior for a robot, the user may find it useful to conduct several experiments from the command line prior to populating the *run()* method of a MajicAct. Once the right combinations of MAJIC Script commands have been tried and tested, the user can utilize the file created during save session as a programming reference.

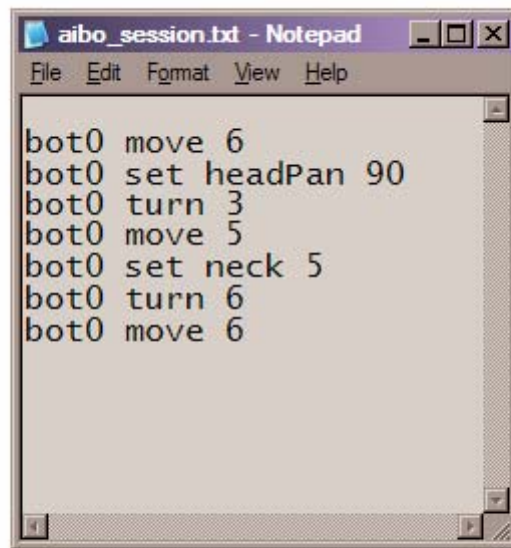


Figure 44. Aibo Session Sample.

The aibo session sample shown above is an example of a session file that could be used to create an aibo behavior for a MajicAct.

d. Serializing a MajicAct Object

The simple program above can be found in the MajicActs directory of the MAJIC package. The user can search this directory at run-time to choose *.dat* object files to be loaded into MAJIC and run on selected robots.

```

import majic.*;

import java.io.*;

class MajicActMaker {

    public static void main(String[] args) throws IOException {
        ❶ String curFileName = "AiboSquare";
        ❷ MajicAct majicAct    = new AiboSquare();

        curFileName = System.getProperty("user.dir") + "/" +
                                curFileName + ".dat";

        File          outFile = new File(curFileName);
        FileOutputStream fos = new FileOutputStream(outFile);
        ObjectOutputStream oos = new ObjectOutputStream(fos);

        oos.writeObject(majicAct);

        oos.close();

    }

}

```

Figure 45. Majic Act Maker.

Once the programmer has developed a MajicAct extended class, that class must also be stored in the MajicActs directory.

Two adjustments must be made to the MajicActMaker Class prior to converting a behavior class to a *.dat* file. Circle 1 marks the name of the *.dat* file that will appear on the file selection screen of MAJIC. Circle 2 marks the name of the MajicAct extended class. The above example depicts the creation of the AiboSquare behavior from the previous section. Once these names are specified, the MajicActMaker class can be compiled and run. Executing this class will generate the *.dat* object and store it in the MajicActs folder.

V. RESULTS

A. OVERVIEW

As previously stated, the primary goal of MAJIC is to provide students and researchers affiliated with the NPS Coordination Laboratory an intuitive, convenient platform on which to test theories and conduct experiments. MAJIC protects students from the rigors of such arduous tasks as developing complicated connection and communication infrastructures, learning complicated proprietary APIs and programming protocols, and designing intricate software architectures and programs for multi-agent control.

The beginning of this section examines the economy of code that can be achieved when programming in MAJIC. This section demonstrates how the implementation and application of the practices used by MAJIC and MAJIC programmers allow students with minimal programming experience to develop and test intricate robotic behaviors with relatively few lines of MAJIC Scripted code.

The initial examples demonstrate MAJIC's code efficiency when utilized on a single robot. The following section demonstrates even greater benefits when creating software programs that coordinate and combine dissimilar agents. The remainder of this section provides examples of output files generated during test-runs of several MAJIC sessions with various configurations of robotic teams and individuals.

B. INDIVIDUAL ROBOT PROGRAMMING

Even when conducting simple operations on a single robot, MAJIC proves to be easier to understand and more economical to implement than writing a proprietary program.

To illustrate the benefits of MAJIC Programming, an Aibo and Pioneer example are provided below.

1. MAJIC vs Proprietary Programming with Aibo

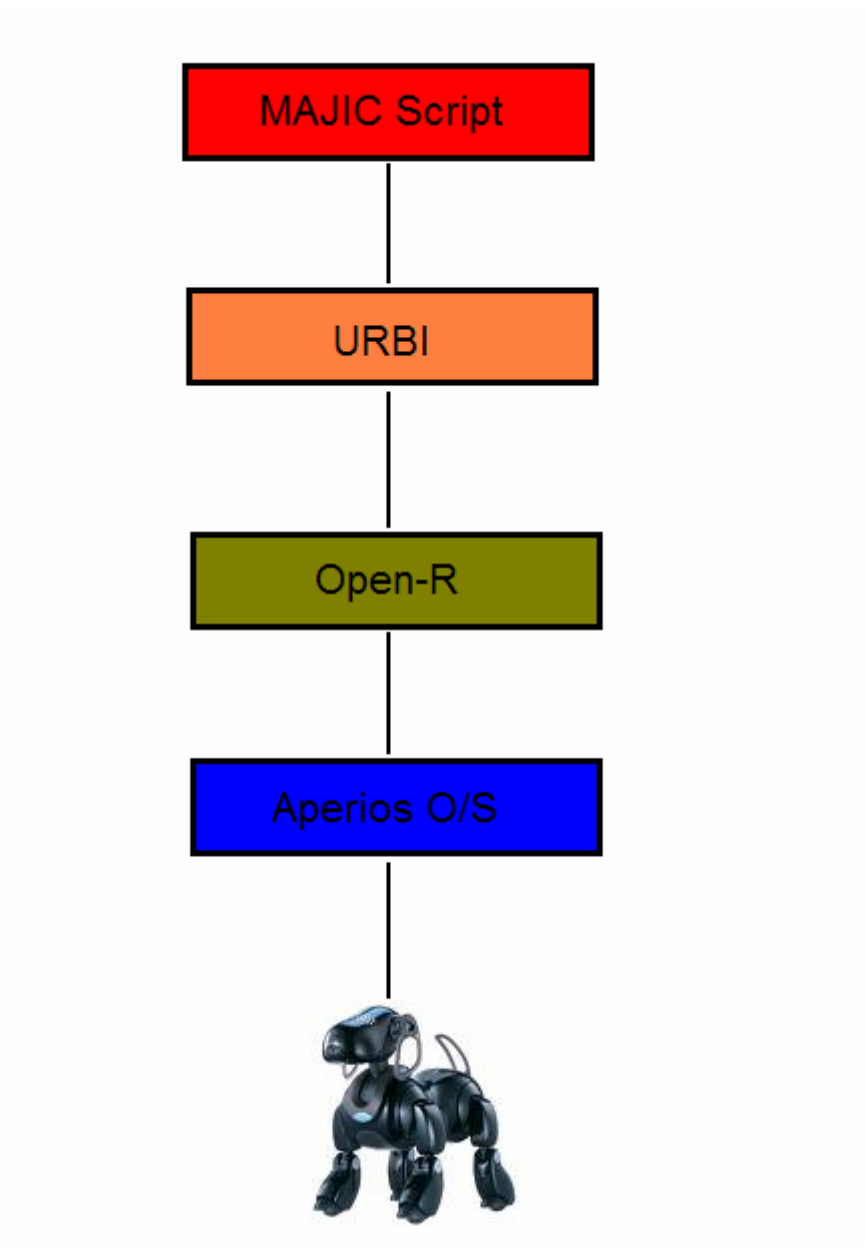


Figure 46. Aibo's Layers of Abstraction.

In some cases, there is a trade-off for the added level of abstraction provided by Majic Programming. Certainly, a program written in the native machine language of AIBO provides greater access and control than one written in Open-R. Furthermore, an

Open-R program generally provides greater power than an URBI program that runs on top of Open-R. Finally, a MAJIC program running on top of URBI can on occasion, experience limitations.

Yet, in general, MAJIC presents the researcher with more than enough capability to conduct proof-of-concept programming; and it does so while offering a scripted language that provides ease of understanding and economy of code.

```

1 import majic.*;
2
3 public class WanderDog {
4
5     static MajicBot aiboBot;
6
7     public static void main(String[] args){
8         double distFwd, distRight, distLeft;
9
10        aiboBot = (MajicAibo) MajicBot.getBot( MajicBot.AIBO );
11        aiboBot.connect("192.168.0.6");
12
13        while(true){
14            distFwd = getDist();
15
16            if(distFwd > 100){
17                aiboBot.move("5");
18            }else{
19                aiboBot.set("headPan", "-90");
20                distLeft = getDist();
21
22                aiboBot.set("headPan", "90");
23                distRight = getDist();
24
25                if(distRight > 100)
26                    aiboBot.turn("3");
27                else if(distLeft > 100)
28                    aiboBot.turn("-3");
29                else
30                    aiboBot.turn("5");
31            } //end if
32        } //end while
33    } //end main
34
35    private static double getDist(){
36        double d = 0;
37
38        String reply = aiboBot.get("distance");
39
40        try{
41            d = Double.parseDouble(reply);
42        } catch (Exception e) {}
43
44        return d;
45    }
46 }

```

Figure 47. WanderDog with Line Numbers.

To illustrate the economy of MAJIC programming, the WanderDog code is once again revisited. This program is written with a combination of Java and MAJIC Script. Used in conjunction, these languages can create a functional wandering and avoidance behavior for Aibo with a minimal amount of code. With the removal of the white space, the above program contains 34 lines of code.

Along with code economy, MAJIC Script also produces self-documenting code that provides easy readability. The common commands seen in WanderDog such as *move*, *turn*, *get*, and *set* are instantly recognizable and understandable.

This structure of common commands allows programmers of all levels to read a MAJIC program written for any supported robot and immediately get an understanding of how the program is intended to behave.

To further demonstrate these issues, the same wandering behavior written in a combination of URBI and Java is provided below.

```

1 import liburbi.call.URBIEvent;
2 import liburbi.UClient;
3 import liburbi.call.UCallbackListener;
4
5 class URBIWanderDog implements UCallbackListener {
6
7     private UClient  robotC = null;
8
9     private boolean  paramUpdated = false;
10
11     private double   d;
12
13     public static void main(String[] args) {
14         URBIWanderDog uwd = new URBIWanderDog();
15
16         uwd.begin();
17     }
18
19     public void begin(){
20         double distFwd, distRight, distLeft;
21
22         try {
23             robotC = new UClient("192.168.0.6");
24
25             robotC.send("motor on;");
26             robotC.send("robot.stand();");
27
28             robotC.setCallback(this, "dist");
29
30         } catch (Exception e){
31             System.err.println("Exception while sending "
32                               + e.getMessage());
33             System.exit(1);
34         }
35
36         while(true){
37             try {
38                 robotC.send("dist: distance.val;");
39                 distFwd = getDist();
40
41                 if(distFwd > 100){
42                     robotC.send("robot.swalk(5);");
43                 }else{
44                     robotC.send("headPan = -90;");
45

```

Figure 48. URBI WanderDog.

```

46         robotC.send("dist: distance.val;");
47         distLeft = getDist();
48
49         robotC.send("headPan = 90;");
50
51         robotC.send("dist: distance.val;");
52         distRight = getDist();
53
54         if(distRight > 100)
55             robotC.send("robot.sturn(3);");
56         else if(distLeft > 100)
57             robotC.send("robot.sturn(-3);");
58         else
59             robotC.send("robot.sturn(5);");
60
61         }//end if
62     }catch(Exception e){
63         e.printStackTrace();
64
65         System.exit(1);
66     }
67 }//end while
68 }//end main
69
70 public double getDist(){
71     while(!paramUpdated){}
72
73     paramUpdated = false;
74
75     return d;
76 }
77
78 // process Urbi callback events
79 public void actionPerformed(URBIEvent event){
80     if( event.getTag().equals("dist") ){
81         String distVal = event.getCmd();
82
83         try{
84             d = Double.parseDouble(distVal);
85         }catch(Exception e){}
86
87         paramUpdated = true;
88     }
89 }
90 }

```

Figure 49. URBI WanderDog (cont.).

The URBI implementation of WanderDog illustrates how much the code increases by just moving down one layer of abstraction in Aibo's software architecture. Discounting the white space in URBIWanderDog still leaves 65 lines of programming code. This increases the amount of code required to construct the basic wander behavior on Aibo by nearly a factor of two.

With the increase in code also comes a decrease in program readability. The `send("dist: distance.val;")` command does not lend the reader a clear idea of what the command's intentions involve. Other URBI specific commands such as `setCallback(this, "dist")`, `event.getTag()`, and `event.getCmd()` complicate and confuse the code readability. At the very least, these commands require the programmer to generally familiarize themselves with the URBI programming language enough to gain a working knowledge of its syntax and structure. Learning proprietary languages for each robot can lead to frustration and roadblocks for students who wish to simply test a simple behavior or AI algorithm.

2. MAJIC vs Proprietary Programming with Pioneer

To further illustrate the obstacles encountered by programmers faced with writing proprietary programs, consider the architecture of the Pioneer. Not all COTS products come with embedded servers conveniently preprogrammed like the Aibo. On platforms like the Pioneer, the programmer must develop and install programs directly on the robot in order to communicate with the agent's operating system or develop their own client and server to communicate with the pioneer as is done in MAJIC (see Figure 50).

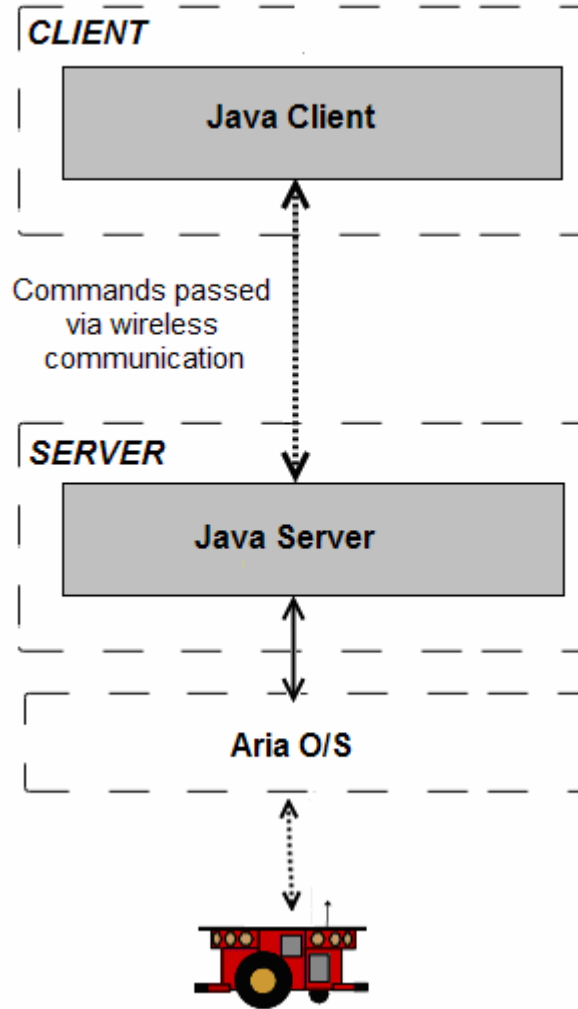


Figure 50. Pioneer Architecture.

Although Pioneer does come with its own programming API in the form of ARIA and ArNetworking, these interfaces are extremely specific and robust, and therefore require students to spend valuable time familiarizing themselves with this style of proprietary programming.

This section will demonstrate the difference between the development of a program designed to run directly on the Pioneer versus one written with MAJIC.

```

1 import java.io.*;
2 import javax.swing.*;
3 import java.net.*;
4
5 public class PioneerWander {
6     private static ArRobot robot;
7
8     static {
9         try {
10             System.loadLibrary("AriaJava");
11         } catch (UnsatisfiedLinkError e) {
12             System.err.println("Native code library libAriaJava failed to load.\n" + e);
13             System.exit(1);
14         }
15     }
16
17     public static void main(String[] args){
18         Aria.init();
19
20         robot = new ArRobot("robot1", true, true, true);
21
22         ArSimpleConnector conn = new ArSimpleConnector(args);
23
24         if (!conn.connectRobot(robot))
25         {
26             System.err.println("Could not connect to robot, exiting.\n");
27             System.exit(1);
28         }
29
30         int numSonar = robot.getNumSonar();
31         double[] sonar = new double[numSonar];
32
33         robot.runAsync(false);
34         robot.lock();
35         robot.enableMotors();
36         robot.unlock();
37
38         for(int i = 0; i < numSonar; i++)
39             sonar[i] = robot.getSonarRange(i);
40

```

Figure 51. Pioneer Program using ARIA.

```

41     robot.lock();
42     robot.move(5000);
43     robot.unlock();
44
45     wait(5000);
46
47     for(int i = 0; i < numSonar; i++)
48         sonar[i] = robot.getSonarRange(i);
49
50     robot.lock();
51     robot.setDeltaHeading(180);
52     robot.unlock();
53
54     wait(5000);
55
56     for(int i = 0; i < numSonar; i++)
57         sonar[i] = robot.getSonarRange(i);
58
59     robot.lock();
60     robot.move(5000);
61     robot.unlock();
62
63     wait(5000);
64
65     for(int i = 0; i < numSonar; i++)
66         sonar[i] = robot.getSonarRange(i);
67
68     robot.stop();
69     robot.disconnect();
70     robot.disableSonar();
71     Aria.shutdown();
72
73 }
74
75 public static void wait(int milSec){
76     milSec = Math.max(0, milSec);
77     try{
78         Thread.sleep(milSec);
79     }catch(Exception e){ e.printStackTrace(); }
80 }
81 }

```

Figure 52. Pioneer Program using ARIA (cont.).

```

1 import majic.*;
2
3 public class MajicPioneerWander {
4
5     static MajicBot pioneerBot;
6
7     public static void main(String[] args){
8         pioneerBot = (MajicPioneer) MajicBot.getBot( MajicBot.PIONEER );
9         pioneerBot.connect("192.168.0.6");
10
11         pioneerBot.move(5000);
12         pioneerBot.wait(5000);
13         pioneerBot.getSonar();
14
15         pioneerBot.turn(180);
16         pioneerBot.wait(5000);
17         pioneerBot.getSonar();
18
19         pioneerBot.move(5000);
20         pioneerBot.wait(5000);
21         pioneerBot.getSonar();
22
23         pioneerBot.close();
24
25     }
26 }

```

Figure 53. Pioneer Program using MAJIC.

Once again, the clear advantage of MAJIC programming is apparent. To develop and implement a native ARIA program on the Pioneer requires 60 lines of non-white space code (Figures 50 and 51). This program compared to the 18 line MAJIC program (Figure 52) demonstrates that MAJIC provides a 30% reduction in the amount of lines that a student would have to code in order to test a simple wander behavior on the Pioneer.

Furthermore, the readability is dramatically improved by the concise, intuitive structure of the MAJIC Script program. The MAJIC program makes it obvious that the wander behavior simply moves the pioneer forward to take a reading, turns it 180 degrees for a reading, and moves it back to its original position for a final reading. Yet, that simplified behavior is not so apparent from the 81 lines of code in the ARIA program.

C. PROGRAMMING HETEROGENEOUS ROBOT TEAMS

The benefits of MAJIC Script reach their full potential when utilized to create behaviors for teams of multiple, heterogeneous robots.

MAJIC Programming allows programmers to quickly develop a class that centralizes the control of various robotic agents. From this central class programmers can implement logic and behaviors based on the shared resources of the heterogeneous robot team.

Connections created via wireless configurations, virtual serial ports, and client/server architectures are encapsulated within the MAJIC libraries. Passing commands to the robots and receiving information from them is abstracted to a script that is intuitive and easy to understand.

The Great Race code below is a simple program that instructs a Hemisson, Pioneer, and Aibo to move forward five seconds as a toy example of a race. Although this simplistic example does not display the power of information sharing among the robots, it does demonstrate the sizeable code reduction and increased readability that can be gained using MAJIC Script even in the simplest of heterogeneous robot programs.

```

1 import java.io.*;
2 import java.net.*;
3 import javax.swing.*;
4 import java.awt.*;
5 import java.awt.event.*;
6
7 import HemiComm.SerialComm;
8 import liburbi.call.URBIEvent;
9 import liburbi.UClient;
10
11 class GreatRace {
12     private static final int DEFAULT_SERVER = 9876;
13
14     private SerialComm serialComm;
15     private UClient robotAibo = null;
16     private InetAddress pioneerIP;
17
18     public GreatRace(){
19         serialComm = new SerialComm("COM6");
20
21         try {
22             pioneerIP = InetAddress.getByName("192.168.0.5");
23
24             robotAibo = new UClient("192.168.0.6");
25
26             robotAibo.send("motor on;");
27             robotAibo.send("robot.stand();");
28
29         } catch (Exception e){
30             e.printStackTrace();
31
32             System.exit(1);
33         }
34     }
35
36     public void startRace(){
37         try {
38             sendHemisson("D,9,9");
39             sendPioneer("move 5000");
40             robotAibo.send("robot.swalk(5);");
41
42             Thread.sleep(5000);
43
44             sendHemisson("D,0,0");
45         } catch (Exception e){
46             e.printStackTrace();
47
48             System.exit(1);
49         }
50     }
51 }

```

Figure 54. GreatRace Example.

```

52     public synchronized String sendHemisson(String com){
53         String line = null;
54
55         try{
56             BufferedReader br = null;
57
58             br = serialComm.communique(com);
59             while (br.ready()) {
60                 line = br.readLine();
61             }
62
63         }catch (Exception e2) {
64             e2.printStackTrace();
65         }
66
67         return line;
68     }
69
70     public synchronized String sendPioneer(String command) {
71         String parsedStr = null;
72
73         try {
74             DatagramSocket clientSocket = new DatagramSocket();
75
76             byte[] sendData = new byte[1024];
77             byte[] receiveData = new byte[1024];
78
79             sendData = command.getBytes();
80
81             DatagramPacket sendPacket =
82                 new DatagramPacket(sendData, sendData.length, pioneerIP, DEFAULT_SERVER);
83
84             clientSocket.send(sendPacket);
85
86             DatagramPacket receivePacket =
87                 new DatagramPacket(receiveData, receiveData.length);
88
89             clientSocket.close();
90
91         }catch(Exception e){ e.printStackTrace(); }
92
93         return parsedStr;
94     }
95
96     public static void main(String args[]) {
97         GreatRace gr = new GreatRace();
98
99         gr.startRace();
100     }
101 }

```

Figure 55. GreatRace Example (cont.).

The 71 lines of code for the class above establish the communications necessary to command all three robots to move forward for the five-second race.

The Pioneer, however, requires a custom server running onboard to receive the incoming packets and translate them to ARIA instructions. A sample server is provided below.

```

1 import java.io.*;
2 import javax.swing.*;
3 import java.net.*;
4
5 public class GreatRacePioneerServer {
6     private ArRobot robot;
7     private double curDouble = 0;
8
9     static {
10         try {
11             System.loadLibrary("AriaJava");
12         } catch (UnsatisfiedLinkError e) {
13             System.err.println("Native code library libAriaJava failed to load.\n" + e);
14             System.exit(1);
15         }
16     }
17
18     private void initPioneer(String[] argv){
19         Aria.init();
20
21         robot = new ArRobot("robot1", true, true, true);
22
23         ArSimpleConnector conn = new ArSimpleConnector(argv);
24
25         if (!conn.connectRobot(robot))
26         {
27             System.err.println("Could not connect to robot, exiting.\n");
28             System.exit(1);
29         }
30
31         robot.runAsync(false);
32         robot.lock();
33         robot.enableMotors();
34         robot.unlock();
35
36         ServerListener servListener = new ServerListener();
37         servListener.start();
38     }
39
40
41     private boolean parseDouble(String str){
42         try{
43             curDouble = Double.parseDouble(str);
44         }catch(Exception e){
45             return false;
46         }
47
48         return true;
49     }
50
51     public boolean move(String arg){
52         if(parseDouble(arg)){
53             robot.lock();
54             robot.move(curDouble);
55             robot.unlock();
56         }else
57             return false;
58
59         return true;
60     }
61

```

Figure 56. GreatRacePioneerServer Example.

```

62     public static void main(String[] args){
63         GreatRacePioneerServer grps = new GreatRacePioneerServer();
64
65         grps.initPioneer(args);
66     }
67
68     public class ServerListener extends Thread {
69         DatagramSocket serverSocket;
70
71         public void run() {
72
73             try {
74                 serverSocket = new DatagramSocket(9876);
75             } catch (Exception e) {
76                 e.printStackTrace();
77                 System.exit(1);
78             }
79
80             while(true) {
81
82                 byte[] receiveData = new byte[1024];
83                 byte[] sendData = new byte[1024];
84
85                 DatagramPacket receivePacket =
86                     new DatagramPacket(receiveData, receiveData.length);
87
88                 try {
89                     serverSocket.receive(receivePacket);
90
91                     //get IPAddress and Port of incomming client////////
92                     InetAddress IPAddress = receivePacket.getAddress();
93                     int port = receivePacket.getPort();
94
95                     //parse packed string for user name and password////////
96                     String cmdLine = new String(receivePacket.getData());
97
98                     System.out.println(cmdLine);
99                     String[] parsedCmd = cmdLine.split(" ");
100                     String command = parsedCmd[0];
101                     String returnStr = null;
102
103                     if(command.equalsIgnoreCase("MOVE")){
104                         boolean valid = move(parsedCmd[1]);
105                         returnStr = "" + valid;
106                     }else
107                         returnStr = "false";
108
109                     //send packet back to client////////
110                     sendData = returnStr.getBytes();
111
112                     DatagramPacket sendPacket =
113                         new DatagramPacket(sendData, sendData.length, IPAddress, port);
114
115                     serverSocket.send(sendPacket);
116
117                 } catch (Exception e) { e.printStackTrace(); }
118             } //end while
119         } //end run()
120     } //end inner class
121
122 } //end server class

```

Figure 57. GreatRacePioneerServer (cont.).

While similar to the server developed for the MAJIC application, the 86-line server above is simplified specifically for the Great Race example. In this example, the server above will only process a “MOVE” command from the Great Race client.

As seen from the two programs above, to produce an architecture that performs a task as simple as moving three dissimilar robots forward can require 157 lines of code or more. Undoubtedly, a programmer who wishes to construct a program capable of non-trivial experiments with dissimilar robot teams would spend a great deal of time and energy developing a robust architecture of command and control.

Below is the Great Race program rewritten using the MAJIC package.

```
1 import majic.*;
2
3 class MajicRace {
4
5     public static void main(String[] args){
6         MajicBot aiboBot = MajicBot.getBot( MajicBot.AIBO );
7         MajicBot pioneerBot = MajicBot.getBot( MajicBot.PIONEER );
8         MajicBot hemissonBot = MajicBot.getBot( MajicBot.HEMISSON );
9
10        aiboBot.connect("192.168.0.6");
11        pioneerBot.connect("192.168.0.5");
12        hemissonBot.connect("COM6");
13
14        hemissonBot.move("9");
15        pioneerBot.move("5000");
16        aiboBot.move("5");
17
18        MajicBot.wait(5000);
19
20        hemissonBot.move("0");
21    }
22 }
```

Figure 58. MajicRace Example.

The MajicRace Class recreates the same 157 line Great Race experiment utilizing only 16 lines of Majic Script. Not only does the MAJIC Script provide a 90 percent reduction in code; it also dramatically improves the program clarity and readability.

VI. CONCLUSIONS AND RECOMMENDATIONS

A. RESEARCH CONCLUSIONS

The overarching goal of this thesis was the development of a software API offering students and researchers the capability of heterogeneous, multi-agent command and control by providing a layer of abstraction that is easily learnable and understandable for users with all levels of computer skills.

Utilizing sound engineering practices, those user requirements were specified and incorporated into the overall design of the MAJIC application. Through extensive use of the Unified Modeling Language, software engineering patterns, and object-oriented features such as inheritance and polymorphism, the MAJIC application achieved a modularity that allows for the addition of future updates with relative ease and minimal recompilation.

Furthermore, the MAJIC application allows programmers to create their own behaviors in the form of MAJIC ACTS and develop their own programs with the pseudo-scripted language, MAJIC Script. The simple, yet powerful set of commands encompassed in the MAJIC Script allow programmers to create programs for individual robots or multi-agent teams with a significant reduction in code compared to proprietary programs.

Finally, in the case of the robots studied for this thesis, MAJIC is fully capable of interfacing with virtual robotic packages such as WebBots, and the proprietary package that comes with the Pioneer. This adds even greater benefit to the researcher or student who wishes to test and troubleshoot outside the lab.

B. RECOMMENDATIONS FOR FUTURE WORK

Overall, this research successfully accomplished its objectives as defined in Chapter I. However, several areas could benefit from exploration, augmentation, and improvement.

An obvious area with any new software application is that process of extensive testing and debugging. Were MAJIC to receive greater exposure to students and researchers, its stability and functionality would undoubtedly benefit from their upgrades, patches, and feedback.

Along those lines, the implementation of additional libraries would obviously broaden MAJIC's capabilities and relevance to the robotic communities. Although the addition of these libraries requires minimal code alteration and recompilation, these requirements could be removed altogether with such techniques as dynamic class loading, or similar tactics.

Finally, MAJIC's modular architecture could be expanded to include modules that provide a finer grain to robot specification and interaction. For example, allowing the user to specify what types of sensors a robot will be equipped with or what type of motion model a robot's move command will utilize could allow the user to develop more intricate behaviors with greater detail.

LIST OF REFERENCES

- [1] MobileRobots INC., *Software, Documentation & Technical Support for MobileRobots Research Platforms*, 2006. Available at <http://robots.mobilerobots.com>. Accessed May 2007.
- [2] K-Team Corporation, *Hemisson Support Page*, 2006. Available at <http://www.k-team.com>. Accessed May 2007.
- [3] D.S. Blank, D. Kumar, L. Meeden, and H. Yanco, Pyro: A Python-based Versatile Programming Environment for Teaching Robotics. *Journal of Educational Resources in Computing (JERIC)*.pp. 1-8, 2004.
- [4] D. Powell, G. Gilbreath, M. Bruch, Multi-robot Operator Control Unit, Space and Naval Warfare Systems Center, San Diego. Available at <http://www.spawar.navy.mil/robots/resources/mocu/mocu.html>. Accessed June 2007.
- [5] J. Baillie, *The URBI Tutorial*, Gostai, May 2006, pp. 12-21.
- [6] J. Baillie, *URBI Language Specification*, Gostai, May 2006, pp. 21-25.
- [7] J. Baillie, *URBI Doc for Aibo ERS2xx ERS7 Devices Documentation*, Gostai, May 2006, pp. 3-6.
- [8] R. Murphy, *Introduction to AI Robotics*, Cambridge, The MIT Press, 2000, pp. 284-310.
- [9] B. Bruegge, A. Dutoit, *Object-Oriented Software Engineering Using UML, Patterns, and Java*, Second Edition, Boston, Pearson Prentice Hall, 2004, pp. 31-37.
- [10] D. Leffingwell, D. Widrig, *Managing Software Requirements A Use Case Approach*, Second Edition, Upper Saddle River, Pearson Education INC., 2003, pp. 148-163.
- [11] C. Wu, *An Introduction to Object-Oriented Programming with Java*, Fourth Edition, New York, McGraw Hill, 2006, pp. 12.
- [12] K. Sierra, B. Bates, *Head First Java*, Second Edition, Sebastopol, O'Reilly Media INC., 2005, pp. 18.
- [13] V. Raman, *Robosim – Pioneer Robot Interface*, University of Madras, India, 2003, pp. 5.
- [14] A. Bredenfeld, *Behavior Engineering for Robot Teams*, Fraunhofer Institute for Autonomous Intelligent Systems, 2003, pp. 8-12.

- [15] Carnegie Mellon University, Tekkotsu, Available at <http://www.cs.cmu.edu/~tekkotsu/index.html>. Accessed May 2007.
- [16] CARMEN, Carnegie Mellon University, Available at <http://carmen.sourceforge.net/home.html>. Accessed September 2007.
- [17] T. Balch, TeamBots 2004, Available at <http://www.cs.cmu.edu/~trb/TeamBots>. Accessed September 2007.
- [18] The Orocos Project, Smarter Control in Robotics & Automation, 2007. Available at <http://www.orocos.org>. Accessed August 2007.
- [19] Aibo, Sony, Available at <http://support.sony-europe.com>. Accessed September 2007.
- [20] B. Smuda, Software Wrappers for Rapid Prototyping JAUS-Based Systems, TACOM Research Development and Engineering Center, 2005, pp. 2-5.
- [21] J. Pedersen, A Practical View and Future Look at JAUS, resquared INC, 2006, pp. 3.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
Ft. Belvoir, Virginia
2. Dudley Knox Library
Naval Postgraduate School
Monterey, California