Autonomous Armada v1.2

Client Readme

August 6, 2012

David Esposito and Kevin Johnstone

Table of Contents

Introduction

Autonomous Armada is built on top of the Battle Challenge AI Framework. This game is aimed at begging to expert level programmers seeking a fun way to learn about programming or AI , or for experts to show off their skills. The game is designed to encourage bots to aggressively pursue actions to earn points. This means "campers" are welcome, but will not do well. Be on the offensive, but be smart since you know your enemies will be doing the same.

Strategy

A bot will receive points for sinking an enemy ship and occupying an island. Ships have a starting health amount which will be decremented each time the ship is hit. The ship is considered sunk when its health is less than or equal to 0. There are multiple islands on a map and an island corresponds to designated coordinate on the map. If a bot's ship occupies an island then it will be able to mine minerals from the island as well as receive points for each turn it is occupied. Minerals will be saved up and used to build new ships. Once enough minerals are saved up, the server will automatically place a ship on or near the players base.

Different From Other Challenges

Autonomous Armada aims to be different than other AI challenges in the fact that we provide ultimate freedom to the programmer. We allow parallelization for maximum optimization, file I/O for machine learning and use of your own machine so you know exactly how your code will perform. This may create some advantages to some programmers familiar with these technologies, as well as users with faster network connections or hardware but the 1 second delay and game strategy will provide enough challenge to keep players from having too much of an advantage from hardware.

We encourage creative coding; do whatever it takes to create a bot that can contend with the best!

Software Overview

Autonomous Armada uses Java sockets to support communication between client and server. The server will run in a remote location with a socket connection to each client.

The Turn

The server will provide each player with the current game board then allow the client some time (usually 1 second) to reply with moves for that turn. This is all handled over socket connections to ensure fast reliable communication. Each bot will return two lists each turn: a list of where each of their ships should move next turn (NORTH, SOUTH, EAST, WEST) and a list of where each ship should shoot. If the server does not receive a turn from a client in the time allotted then the client is ignored (provided no moves or shots).

Once the server has received all of the players' turns, the server will process each move for validity then apply them to the game board. Moves will be processed first (each ship receives its new location) then shots are processed and finally the game income is allocated.

Income

Income is allocated at the end of each turn by the server. The server will check each island to see if a ship is occupying it. If such a ship exists that player is awarded points to the cumulative score in addition to minerals for each island. At time of writing (v1.2), each island awards 1 mineral per turn. At the start of the next turn, the server will check to see if the player has enough minerals to build a new ship and automatically place the new ship(s) on or near the player's base.

End of Game

The game can end in two ways:

1. There is only one (or zero) player(s) remaining with ships or
2. The game has been played to the maximum number of turns (usually limited to 5 minutes or 300 turns).

Once the game is over, the results will be announced to each player across the socket. The server will then create a video of the game. On official servers this video will be automatically uploaded to YouTube.com and each client will be provided a link to the video. At this point the game is complete and the network will close all open sockets.

Building Your Bot

Getting started should not take more than 30 minutes and coding a somewhat competent bot should take around 5-10 hours (Warning: mileage may vary). How much do you want to invest into your bot?

Getting Started

When building your bot, you will need to complete these 4 easy steps:

Step 1: Create your new bot class that extends ClientPlayer. For this example we will be creating the class "MyNewBot", however you can name the class whatever you like.

```
package battlechallenge.bot;
	public class MyNewBot extends ClientPlayer {
		// variables here
		public MyNewBot(String playerName, int boardWidth, int
	boardHeight, int networkID) {
			super(playerName, boardWidth, boardHeight, networkID);
			// initializations here
		}
		// helper methods here
	}
```

Step 2: In BattleShipClient.java edit the botToPlay(String botName) method.

```
public static ClientPlayer botToPlay(String botName, String networkBotName) {
	if (botName.equals("StarterBot")) {
		return new StarterBot(networkBotName, 0, 0, 0);
	}
	if (botName.equals("EasyBot")) {
		return new EasyBot(networkBotName, 0, 0, 0);
	}
	if (botName.equals("MediumBot")) {
		return new MediumBot(networkBotName, 0, 0, 0);
	}
	else if (botName.equals("HardBot")) {
		return new HardBot(networkBotName, 0, 0, 0);
	}
	else if (botName.equals("MyNewBot")) {
		return new MyNewBot(networkBotName, 0, 0, 0);
	}
	else {
		return new MyNewBot(networkBotName, 0, 0, 0);
	}
}
```

Step 3: Override the doTurn() method. Ignore the parameters that the constructor takes in, and do not change them. Any parameters that you would like to set should be read in from a file.

```
package battlechallenge.bot;
public class MyNewBot extends ClientPlayer {
	// variables here
	public MyNewBot(String playerName, int boardWidth, int boardHeight, int
networkID) {
		super(playerName, boardWidth, boardHeight, networkID);
		// initializations here
	}
	@Override
	public List<ShipAction> doTurn() {
		return null;
	}
	// helper methods here
}
```

Step 4: Code!!! Finish the do turn method.

Finding Information

Check out the API for ClientGame.java. We will list a few helpful hints below.

Getting Ships:

The method ClientGame.getMyShips() will return a List<Ship>. NOTE: This method provides a deep copy of the ships list.

```
List<Ship> myShips = ClientGame.getMyShips();
for (Ship s : myShips) {
	System.out.println("Location: " + s.getLocation() + "\nRange: " +
s.getRange());
	// do something with my ships
}
List<Ship> opponentShips = ClientGame.getOpponentShips();
for (Ship s : opponentShips) {
	System.out.println("Location: " + s.getLocation() + "\nHealth: " +
s.getHealth());
	// do something with my opponents ships
}
```

Getting Islands:

The method ClientGame.getMyIslands() will return a List<Islands>. NOTE: This method does NOT provide a deep copy of the Islands list.

```
List<Island> myIslands = ClientGame.getMyIslands();
for (Island is : myIslands) {
	System.out.println("Location: " + is.getLocation());
	// do something with my islands
}
List<Island> islands = ClientGame.getAllIslands();
for (Island is : islands) {
	if (is.getOwnerId() == ClientGame.getNetworkId()) {
		// my island
	} else if (is.getOwnerId() > 0) {
		// opponent island
	} else {
		// unoccupied island
	}
}
```

Returning Moves

You will be returning a List<ShipAction>.

```
List<ShipAction> actions = new LinkedList<ShipAction>();
int randInt = (int)(Math.random() * 4);
Direction moveDirection = Direction.NORTH;
if (randInt == 1) {
      moveDirection = Direction.SOUTH;
} else if (randInt == 2) {
      moveDirection = Direction.EAST;
} else if (randInt == 3) {
      moveDirection = Direction.WEST;
}
Coordinate shotLocation = new Coordinate((int)(Math.random() * 10),
(int)(Math.random() * 10));
for (Ship s : myShips) {
      actions.add(new ShipAction(s.getIdentifier(), shotLocation,
moveDirection));
}
return actions;
```

Connecting to a Server

This should be the easist aspect of Autonomous Armada once you get it setup. The class BattleShipClient takes in a few command line args:

Long Command	Short Command	Description
--PlayerName	--n	The name of the player as it will appear on the network. Default: "Anonymous#"
--BotName	--b	The name of the bot when selecting which bot class to play. Default: "StarterBot"
--Port	--p	The port number for the server. Default: 3000
--IP	--i	The ip address of the server. Default: 127.0.0.1 (loopback)

NOTE: Remember both dashes, this will not work with a single dash in front of the argument.

Command Line

The following example uses the default port and IP while setting the PlayerName to "David" and the bot to "DavidBot".

java -cp ../libs/jewelcli-0.7.6.jar:. battlechallenge.client.BattleshipClient --n=David --b=DavidBot

Eclipse

Make sure that jewelcli-0.7.6.jar is in the build path (if you imported the project you should be fine).

Choose "Run->Run Configurations". Right click "Java Application" and select "New".

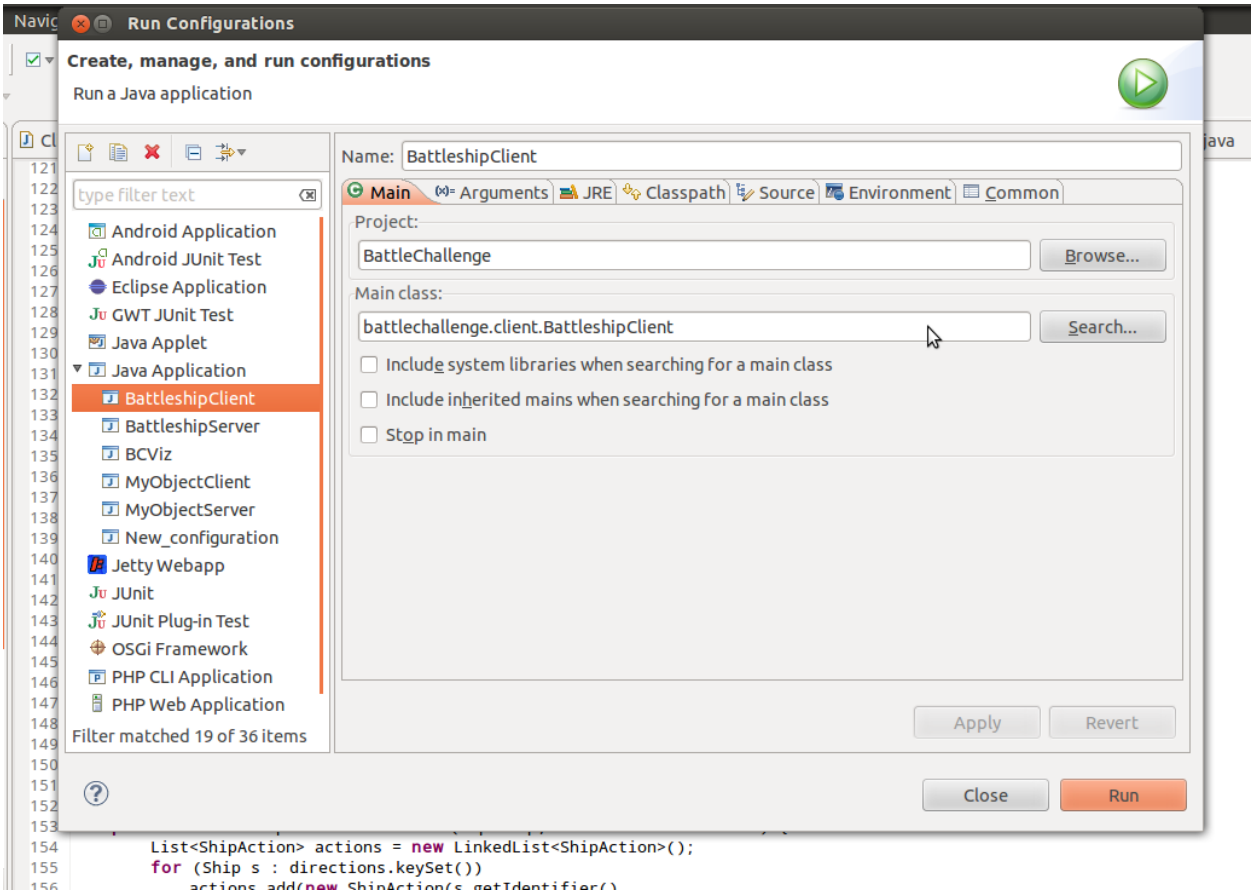
Main Tab:

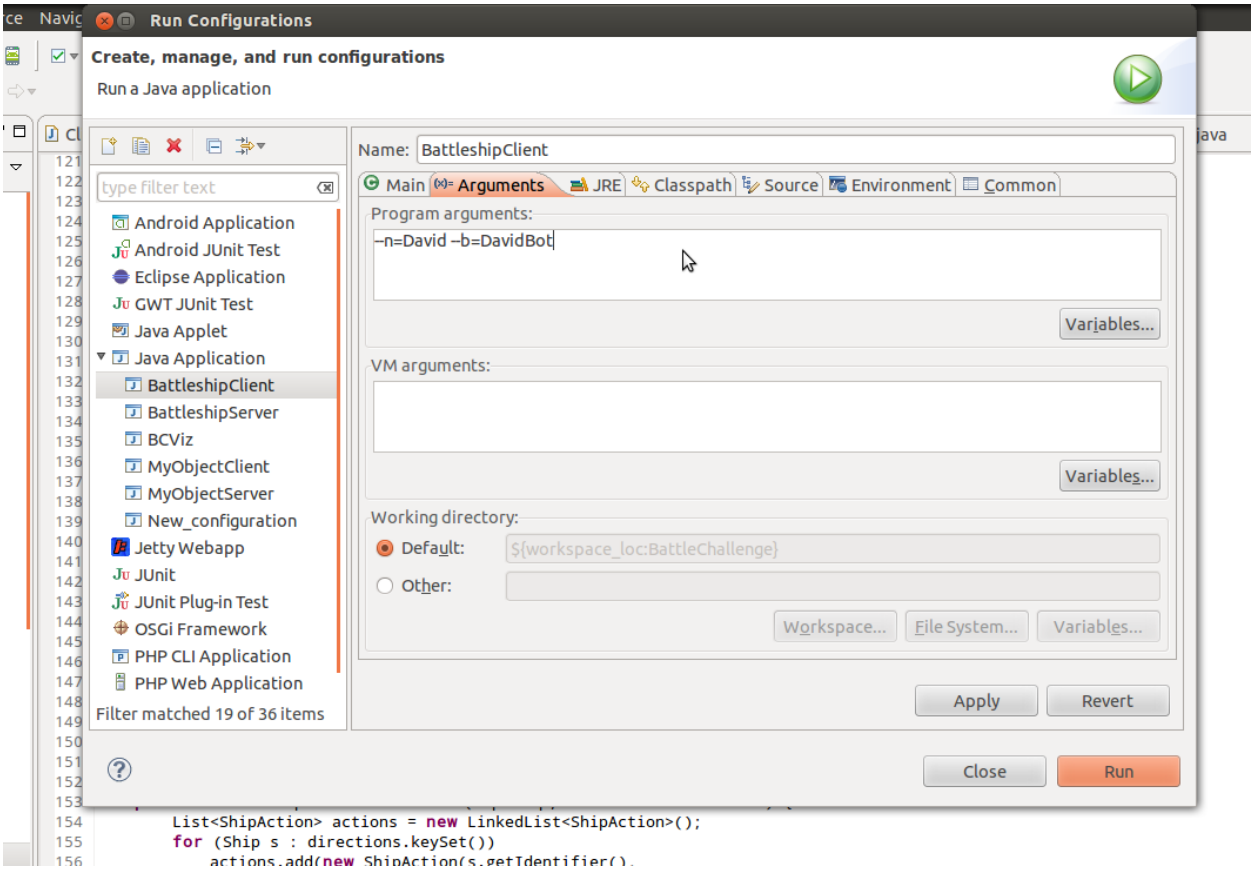
Main Class: "battlechallenge.client.BattleshipClient"

Argumants Tab:

Program Arguments: "--n=David --b=DavidBot"

NOTE: Do not use quotes when entering text fields.




Run Configurations
Create, manage, and run configurations
Run a Java application
type filter text
Android Application
Android JUnit Test
Eclipse Application
GWT JUnit Test
Java Applet
Java Application
BattleshipClient
BattleshipServer
BCViz
MyObjectClient
MyObjectServer
New_configuration
Jetty Webapp
JUnit
JUnit Plug-in Test
OSGi Framework
PHP CLI Application
PHP Web Application
Filter matched 19 of 36 items
Name: BattleshipClient
Main
Arguments
JRE
Classpath
Source
Environment
Common
Program arguments:
--n=David --b=DavidBot
Variables...
VM arguments:
Variables...
Working directory:
Default:
${workspace_loc:BattleChallenge}
Other:
Workspace...
File System...
Variables...
Apply
Revert
Close
Run
List<ShipAction> actions = new LinkedList<ShipAction>();
for (Ship s : directions.keySet())