Design: Web-Based Strategy Game With Complex AIs

Josh Polkinghorn
BACS

Alana Weber
BSCS

Faculty Mentor: Laurie Murphy

CSCE 499, Fall 2012

Contents

List of Figures

1 Introduction

In this project, we aim to explore the field of artificial intelligence in depth by creating a web-based strategy game in which humans play against the computer. We will build an adaptation of the board game Risk in which players compete to gain control of the most planets by destroying the fleets of other players, then program several AIs to compete with human players. In order to effectively play against a human, the AI will need to make strategic decisions in response to the actions of other players, both human and AI. We intend to research and implement the most interesting and effective techniques and algorithms for AI strategizing. We will also implement a web-based client GUI and a server for handling player-to-AI interactions. Please refer to our Requirements Document [1] for an in-depth discussion of this project's purpose and high-level requirements.

2 Research Review

We have done research to explore our options in three main problem areas: the architecture of client-server communications, the interactivity of the client GUI, and the functionality of the AI's themselves.

First, we were faced with the challenge of designing an architecture that would allow for two AIs to play each other. We decided to abstract the idea of a player so that the game engine and server could not tell the difference between the human player and an AI player, which would allow for us to see which AI qualities were most effective. The largest issue was how to prompt the AI Clients to make a move when it was indeed their turn. To solve this, we adopted a Java-based server architecture that would allow us to synchronize requests between all players. See the Design Methodology section for a detailed explanation of this architecture.

Next, we encountered a problem in our initial planning of the client GUI. The original board game Risk involves attacking and capturing countries in the world, countries which are highly irregular shapes. Consequently, we researched options on how to create a clickable world map with which the user would interact. In the course of researching these possibilities, we explored the HTML canvas, scalable vector graphics (SVG), and color mapping. The HTML canvas seemed like a good option in terms of accessibility and interactivity, but it has no way of easily handling complex shapes, making it

less than ideal for drawing countries. Our next option was SVG, an XML-based file format that essentially contains a collection of points that would define the shape of a each individual country. This option was very attractive because of its scalability and versatility, as well as high graphical quality; unfortunately, we could not find an easy way to create different maps, which would have made thorough testing much harder. Additionally, SVG is not designed for user input, which was an essential feature we wanted. Next, color mapping would involve an underlying image with differently colored regions underneath the main game map. Clicks could be read by the underlying color map image to determine their location, and processed from there. This also presented a major problem in terms of flexibility and scalability because both the color map and the overlay would have to be built as static images. Ultimately, our research led us to choose the HTML canvas supported by Javascript as the main component of our GUI due to its flexibility and convenience.

For designing the AI's themselves, we consulted a number of sources, both book and websites, to get some idea of how to go about writing the AI's. From this research, we developed a basic list of requirements for the AI's – that is, what things they will need to understand and what kinds of decisions they will need to make. In short, each AI will need to process a game state and make a move based on that information. A move will consist of deployments (where to add more fleets), reinforcements (where to move fleets between its planets), and attacks, with attacks being the most complex computation. We also learned various techniques they might use to make those decisions. A more in-depth discussion on the results of our research and the proposed AI designs that followed can be found in the Design Methodology section of this document.

3 Design Methodology

3.1 Gameplay Description

The object of the game will be to defeat all other players by conquering all the planets on the game map. Players station fleets in their own planets and attack their neighbours to gain more territory. The planets are also organized into regions, and benefits come from controlling an entire region of planets.

To begin, each player gets 5 fleets per turn. At the beginning of the turn

the player may place these 5 fleets on any combination of planets he or she already owns. If the player controls an entire region of planets, he or she will be granted additional fleets per turn, based on the size of the region. There is no limit on the number of fleets than can be placed on a planet. This is the Deployment. Players may also move fleets from one planet they own to another. This is known as Reinforcement.

Next is the Attack stage of a player's turn. A player may choose to attack any planet adjacent to a planet he or she already owns. The player may choose however many fleets are stationed on his or her planet to use to attack the neighboring planet. Players may attack multiple planets per turn, even basing from the same planet, provided there are enough fleets stationed there.

Once the player has decided where to place fleets and which planets to attack, he or she submits the turn. The stages of the turn are processed in order. The Game Engine decides the winner of an attack on a fleet-by-fleet basis using the generation of a random number between 1 and 6 to simulate the roll of a die. The fleet with the higher number lives, and the fleet with the lower number is defeated. In the case of a tie, both fleets survive until the next round of dice rolling. When the number of defending fleets reaches 0, the attacker has won, and the planet changes ownership. The number of attacking fleets left over are stationed on the defeated planet. If a certain ratio (yet to be determined) of the attacker's fleets are defeated, the remaining attackers will retreat to the source planet. Attacks are processed in order; if a player chooses to attack a destination planet from multiple source planets, and the destination planet changes hands before all the attacks have finished, the remaining attacks are considered reinforcements.

3.2 Architecture Overview

The game implementation will consist of three major components, the client GUI, the server/game engine, and the AI clients. See Figure 1 for a general overview of these components and how they work together. See Use Cases and the corresponding sequence diagrams (Figure 4 and Figure 6) for step-by-step details of some example processes.

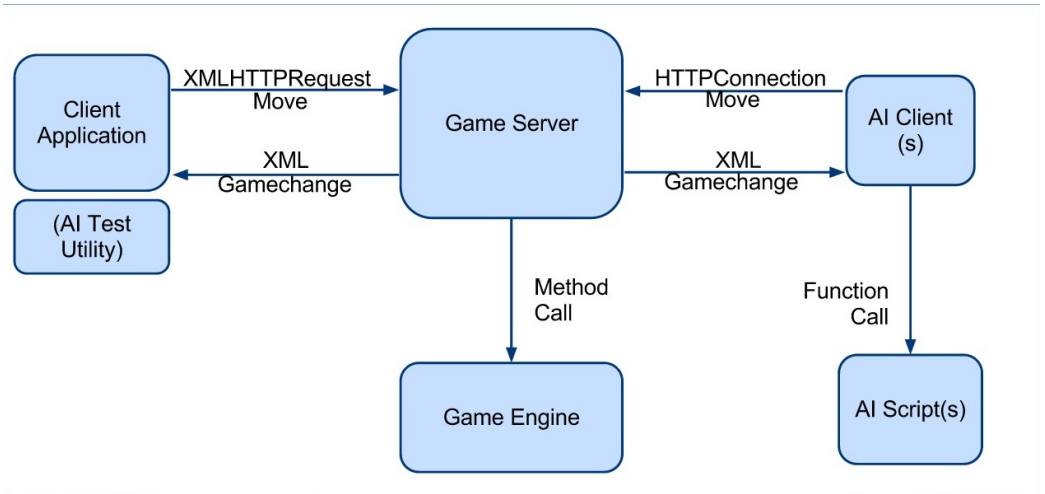


Figure 1: Architecture Overview

3.3 Client-Side GUI

The front-end user interface, written in HTML and Javascript, will display the state of the game to the user and gather input from the user. It will also connect to the server, submit moves to the server, and fetch game changes and use them to update the game display. The following two figures show screenshots of the Startup screen and the Game Screen. The Startup screen allows the user to choose game options before entering the game. These options include player color, the number of AI opponents, and the difficulty of the AI opponents. The Game screen is where the game itself will take place. The planets and their connections, as well as the number of fleets on each planet, and the color of the owner of the planet, are shown.

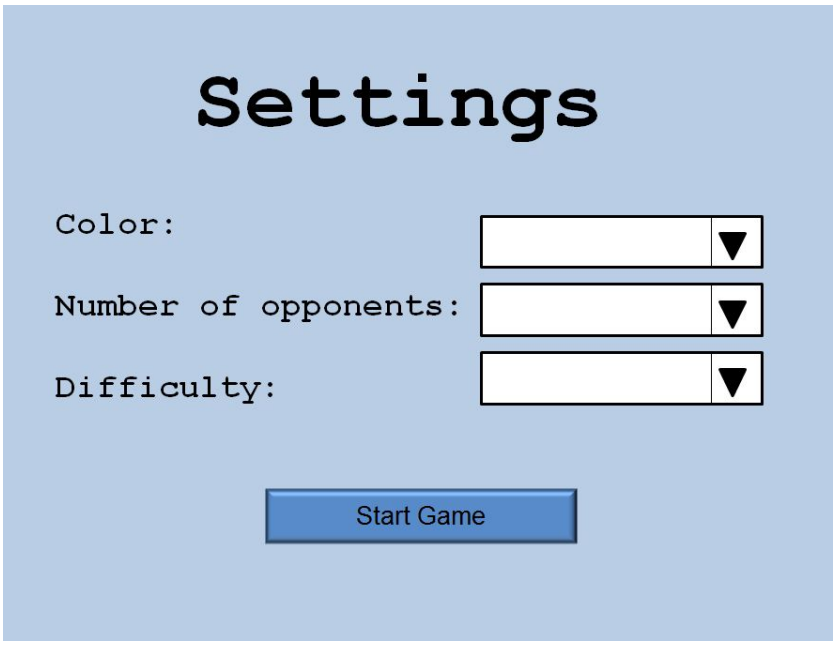


Figure 2: Startup Screen

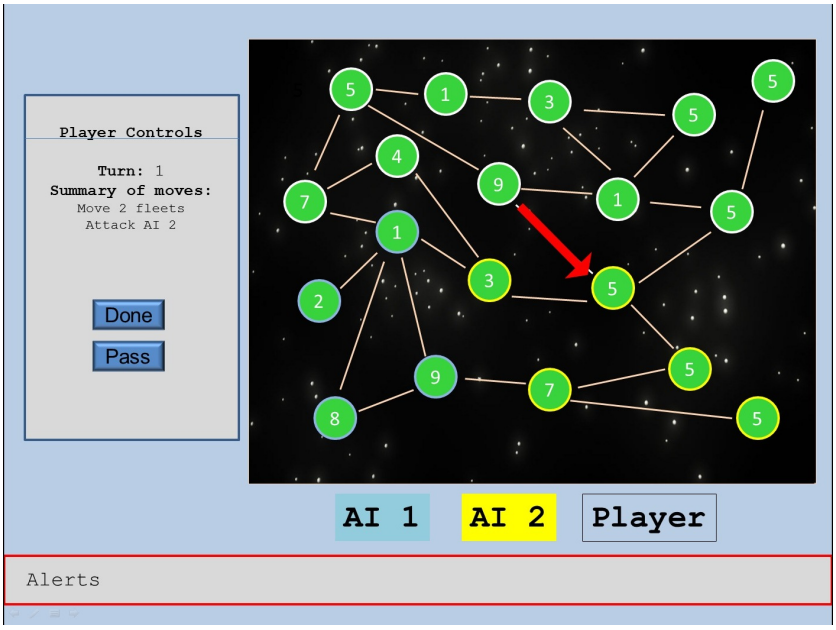


Figure 3: Game Screen

3.4 Communication Protocols

All components will communicate with each other using XML transferred via HTTP. The communications that will be passed back and forth are as follows: GameState (for beginning a game), Move, GameChange (for updating the game state), and Win/Elimination.

3.4.1 Gamestate

The GameState consists of a list of planets and their properties (x, y, radius, color, connections, owner). It also contains a list of regions and the planets contained in those regions. This communication is only sent to the players when the game first begins in order to set up.

3.4.2 Gamechange

A GameChange consists of a list of planets whose status changed with their IDs, updated owner, and updated numbers of fleets. This communication will be generated by the server and passed to all players after each turn. The players will then use this information to update their personal Gamestates.

3.4.3 Move

A move consists of a player ID, the source and destination planets of an attack, reinforcement, or deployment, as well as the number of fleets left on the source and destination planets. If the source and destination planets are the same, the portion of the move is a deployment. If the source and destination planets are owned by the same player, it is a reinforcement. If the owners are different, the numbers are handled as an attack.

3.4.4 Win/Elimination

In the case of a win or elimination, a special message will be included in the Gamechange that indicates a player has either won or been eliminated.

3.5 Game Engine and Game Server

The Game Engine and Game Server, both written in Java, are tightly coupled. They receive input from all players, process moves, constantly check

for a winner, and send out game changes to all players after every turn. See Figures 11 and 12 for class diagrams showing the design outline for the Game Engine and the Game Server.

3.6 AI Client

The AIs, written in Python, will act just like the human player by submitting moves to the server based on a provided gamestate. These moves will be determined by a separate decision script (AI script), also written in Python. As the AI client will be a script instead of a class, like the Engine and Server, we list its functions here instead of providing a UML diagram.

- Connect to the server and retrieve a Gamestate from the initial request.
- Use a read_XML() function to load the Gamestate, as well as subsequent Gamechanges, into memory.
- Call a separate script to generate a Move, then translate that move into XML to be sent to the server.

The AI client will terminate upon completion of or elimination from the game.

3.7 AI Functionality

The following are the different behaviors that the AI's may exhibit. The AI scripts will utilize combinations of these techniques in order to make a move.

- Random: The first, and easiest, type of AI will be one which randomly chooses where to place fleets and randomly chooses which planets to attack. It will be used primarily for testing purposes until we can develop more robust AIs.
- Prioritization: This type of AI will prioritize planets to defend and/or planets to attack based on certain criteria, including how many enemy fleets surround it, the number of connections, and the region value. This will possibly incorporate a genetic algorithm to determine the weights of the different criteria.
- Probabilistic: This type of AI will determine which planets to attack based on either the probability of success of the attack, or the highest amount of damage to the enemy fleets on the surrounding planets.

- Learning: This type of AI will keep track of the moves it has made, and rate their success based the number of planets won with the least number of fleets. Upon generating a move, it will check it against the log and pick the move whose success is known to be higher.
- Adaptive: This type of AI will have remember the moves the other players have made and determine how to respond.
- Predictive: This type of AI will attempt to predict other players' moves based on some sort of algorithm, then play accordingly.
- Aggressive: This type of AI will always attack as much as possible whenever possible.
- Defensive: This type of AI will attack as little as possible and build up defenses in its own planets instead in order to keep from being conquered.

3.8 Use Cases

Two main use cases will be detailed in this section, Startup and Make Move. The use case for Elimination and Win follow directly from Make Move with only slight modifications at the end.

3.8.1 Startup

- **Description**: User starts the game with specified options
- **Preconditions**: User has successfully opened the web page, and is viewing the Startup screen
- **Postconditions**: Upon clicking "Start Game," the user is taken to the Game Screen, where he or she views an initial game state with the specified options (number and difficulty of AI players)
- **Abnormal Postcondition**: Disconnection from the server displays an error message to the user

	User	Client	Server
1	User chooses number of players		
2	User chooses difficulty of computer players one by one		
3	User clicks "Start Game"		
4		Sends HTTP POST request to server with number of players and difficulty of each AI Player	
5			Creates a HandleDefineGame object with a GameEngine
6			Creates the appropriate number of Player objects
7			Begins the appropriate number of Python AI processes with the specified difficulties
8			Sends initial HTTP response with XML GameState to all players (client and AI)
9		Receives GameState from server	
10		Sends GameChange request as acknowledgement	
11			Waits for GameChange request from all Players. For each request that comes in, creates a HandleGameChange object.
12			Sends an HTTP response with XML GameChange to all players along with the ID of the first player to move.

Figure 4: Startup Use Case

The following diagram shows the sequence of events in the classes that carry out the Startup use case.

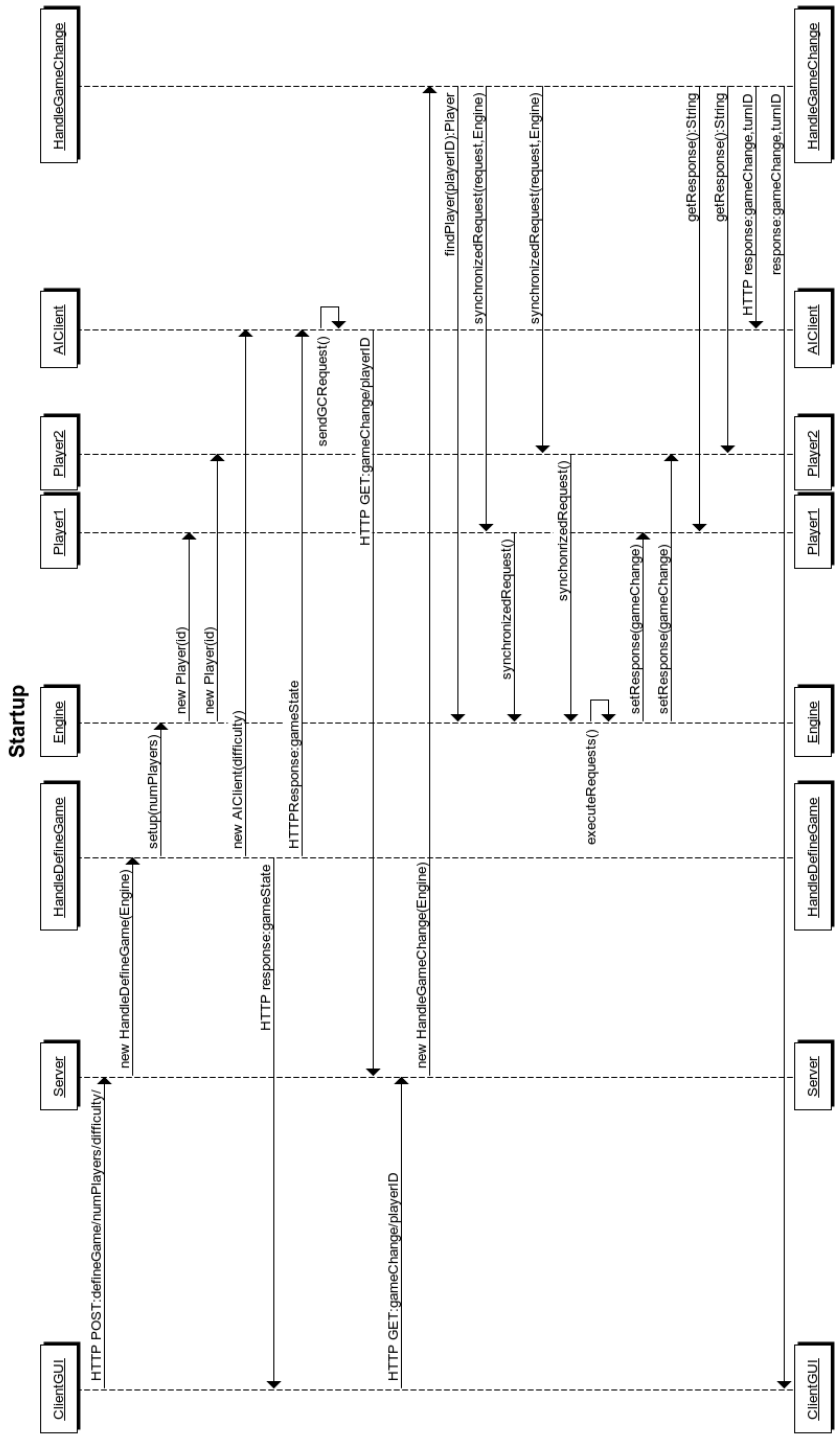


Figure 5: Startup Sequence Diagram

3.8.2 Make Move

- **Description**: User submits the move to the server
- **Preconditions**: User is viewing the Game Screen and has specified the deployments, reinforcements, and attacks he or she wishes to make
- **Postconditions**: User is viewing an updated game state containing the results from his or her moves as well as the results of the moves from all the AI players
- **Abnormal Postcondition**: Disconnection from the server displays an error message to the user and returns him or her to the Startup screen. Disconnection of another player shows that that player has been eliminated.

	User	Client	Server
1	Clicks "Done"		
2		Sends HTTP POST request to server with the player ID and a Move	
3			Creates a HandleGameChange object with a GameEngine
4			Finds the Player object representative of the client
5			Sets the request field of that Player object to the Move and begins synchronizedRequest in the Engine
6			Waits for a GameChange request from the rest of the players
7			Processes the move and creates a GameChange
8			Sends an HTTP response with XML GameChange to all players along with the ID of the next player to move.
9		Receives GameChange from server	
10		Sends GameChange request	
11			Receives the next move from an AI player
12			Sends an HTTP response with XML GameChange to all players along with the ID of the next player to move.
13		Receives GameChange from server	
14		Displays updated game state to the user	

Figure 6: Make Move Use Case

The following diagram shows the sequence of events in the classes that carry out the Startup use case.

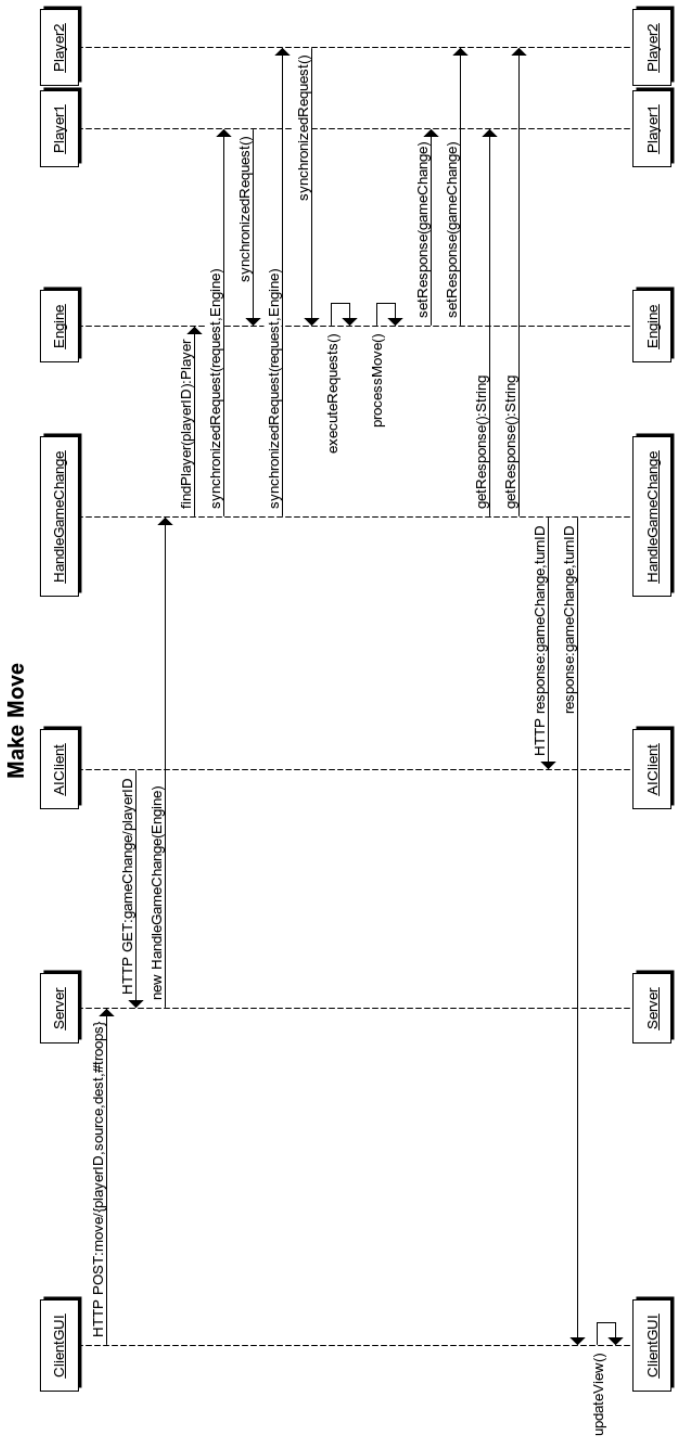


Figure 7: Make Move Sequence Diagram

3.9 Additional Utilities

3.9.1 Mapmaker

We will also create a map generator utility for development use. The inputs to the utility will be as follows: the number of planets desired, the connectivity of the map, the number of regions, and the size of the map. The program will run a series of algorithms to generate a semi-random game map based on those parameters. It will use the following steps to create the map:

1. Generate a random sampling of planets in a two-dimensional coordinate system.
2. Ensure that there is a minimum distance between planets by adjusting locations.
3. Determine a set of regions that collectively encompass all planets using an algorithm for finding the set of regions that have the minimal area.
4. Generate random connections between the planets with limited distance, correcting for overlap as necessary.

Once the map is complete, the program will write it to XML for use in the game. This is intended for development purposes only, but we may choose to modify it for integration into the server itself, allowing for spontaneous random map generation.

3.9.2 AI Test Utility

We will develop an AI test utility that will allow us to test multiple AI's against each other. Under normal circumstances, the AIs only begin execution when the human client starts the game. The Test Utility will be written in Javascript and will include much of the same client code that is used in the GUI. The utility will serve the following purposes:

- Begin the game with the specified number and difficulty of AI's to play.
- Keep a log of which moves each AI makes in the game for examination after the game.
- Run multiple games in sequence, recording the results of each one to a log.

3.10 Testing Plan

The game will be tested as much as possible after each step of the implementation process. In particular, the following list details which aspects of the game we will be tested and how we plan to test them.

- Client GUI: Use random clicking to ensure that each clickable location yields the correct results. This testing will also include collecting some user feedback on the aesthetic and intuitive nature of the GUI. We intend to ask four or five people to play with the GUI and approve or suggest changes.
- Game engine alone: Use jUnit tests to ensure that the XML game changes generated by the engine after certain moves are processed are correct.
- Human/AI Client - Server interactions: Test the responses of the server to bad requests, and the reaction of the client to bad server responses. Test results of network disconnection.
- Integration Testing: Ensure that all parts function together properly by examining the responses from the server to client requests.
- AI vs. AI Interactions: Use AI TestUtility to run two AI's against each other. This testing will be more to examine the behavior of the AI than to troubleshoot.

We will be testing the reaction of the game as a whole to the following abnormal conditions:

- Loss of network connectivity in the middle of the game. This always takes the user back to the Startup screen.
- Disconnection of another player in the middle of the game. This eliminates that player from the game.
- Invalid move from any player. This displays an error message and prompts the user to try again.
- Manipulation of the game by means other than the GUI (i.e., the possibility of hacking): Ensure that the engine validates the move based on the current gamestate, validates the player ID of the move, and only allows each player to make one request per round. Upon discovery of foul play, the user is returned to the Startup screen with an error message.

In the final stages of the project, we intend to gather a small number of people (around 10) to play against various AI's and calculate the percentage of wins and losses of various AI types, indicating to us which as the most successful.

4 Work Completed

4.1 GUI Prototype

We have created a prototype of the functionality of the map that will be included in the GUI. The two main functionalities that the map will need are the ability to click and drag to see large maps entirely, and the ability to select a planet and determine its ownership and other properties. Both of these functionalities have been implemented in this prototype. It creates 100 planets in random locations and sizes on an HTML canvas. It creates a large map, and allows the user to click and drag in order to see the entire map. It also allows the user to select a certain planet, and highlights the planet selected in red. The following figure shows a screenshot of this prototype.

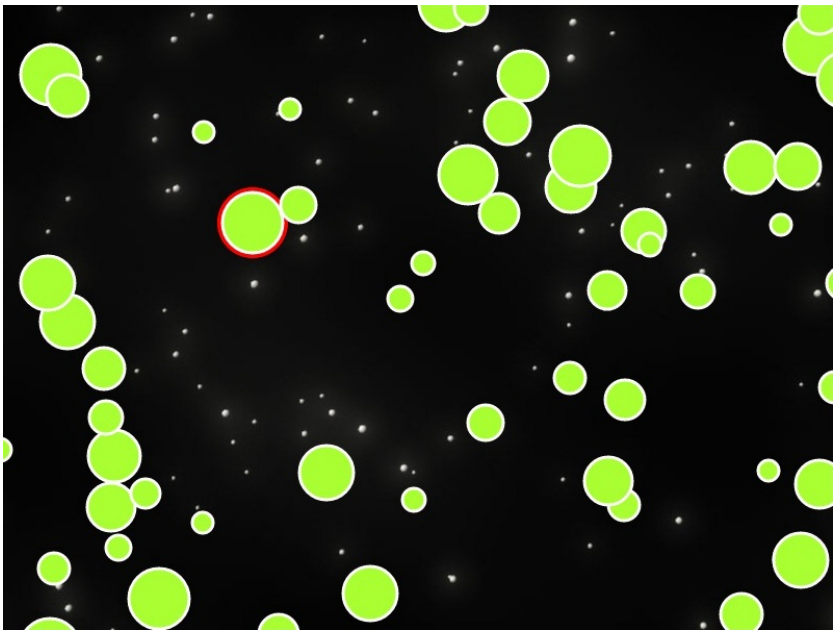


Figure 8: ClientGUI Prototype Screenshot

4.2 Server Prototype

We have created a prototype of the functionality of the turn-based architecture. This program includes a simple version of the Game Server and the Game Engine. Upon clicking "Start" on the prototype page, the server spins off two Python processes on separate threads which communicate with it based on turn ID's. If it is its turn, the Python or Javascript client sends the message "move". If not, the client sends the message "ready." The engine waits until all players have sent a communication before incrementing the turn ID and moving to the next round. The Javascript client GUI shows the results of each round. This is how turns will work in the game. During the creation of this prototype, we worked through challenges and learned quite a bit about Javascript headers and threading with Python. The following figure shows a screenshot of the server console alongside the client web-page view of this process running.

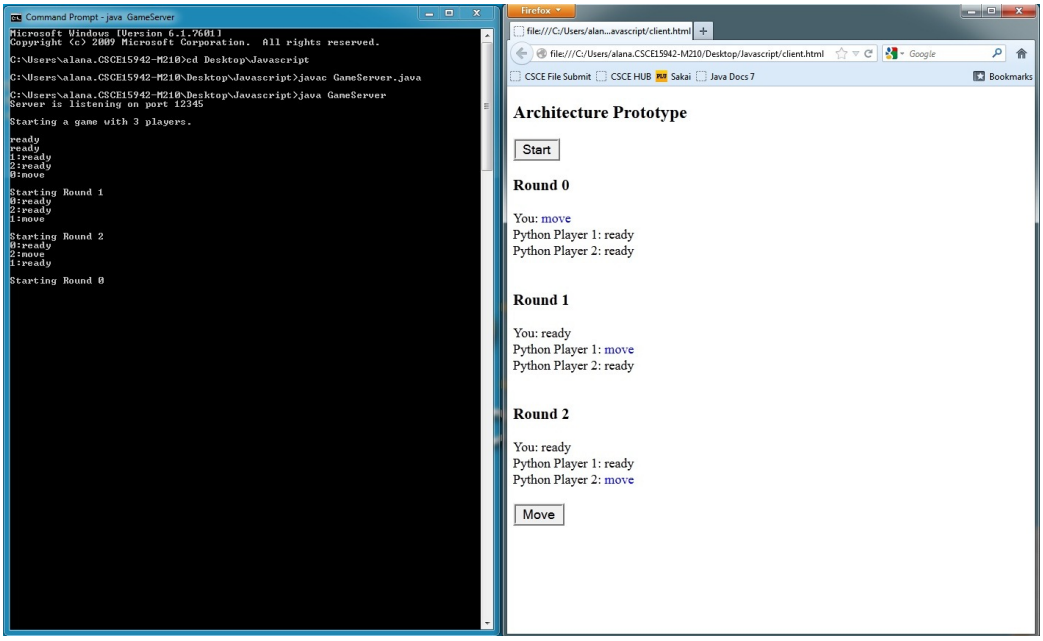


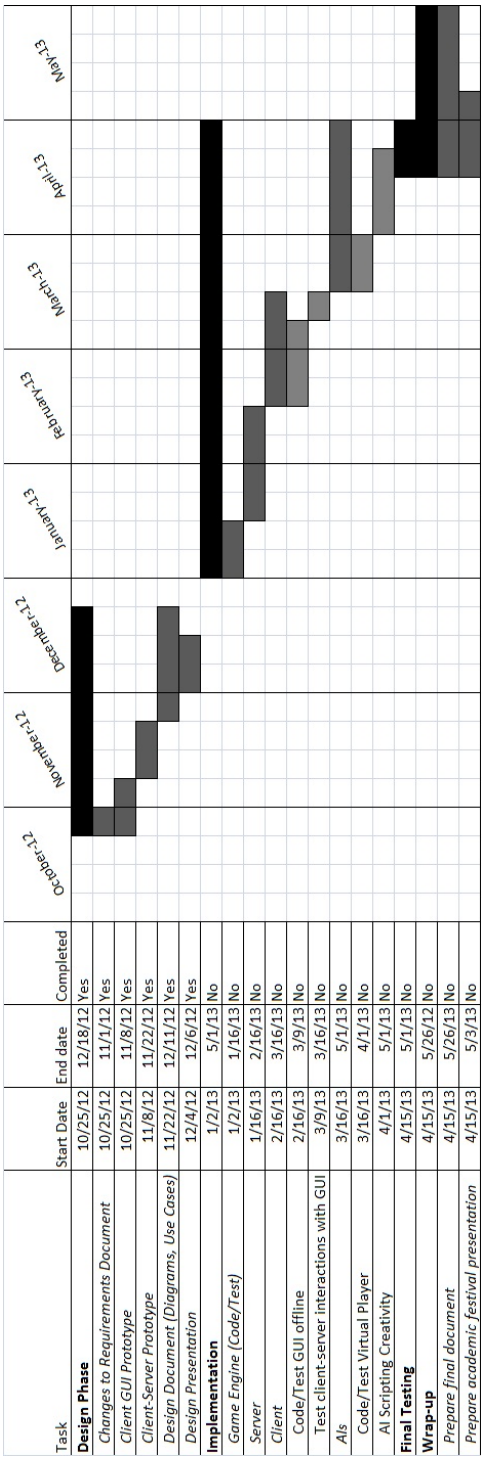
Figure 9: Server Prototype Screenshot

5 Future Work

1. Game engine
   (a) Code Planet, Region, and Player; test with jUnit
   (b) Code Move and Gamestate; test with jUnit
   (c) Code gameplay portions of Engine and test with jUnit
      i. Processing moves
      ii. Checking for winners
      iii. Producing game changes and game states (XML)
2. Server
   (a) Implement basic server functionality, including receipt of requests and creation of handlers
   (b) Code server interactivity portions of Engine
   (c) Write handlers for all contexts
   (d) Test and debug
3. Client
   (a) Create GUI
   (b) Process input from HTML/JS GUI
   (c) Write JS client-server interactions
   (d) Process XML
4. Player interactions through server
   (a) Code AI client
   (b) Code easy (random) AI to generate moves
   (c) Test interactions among GUI client, server, engine, and AI clients
5. AIs - Implement AI techniques as described in this document
6. Prepare Academic Festival Presentation

6 Updated Gantt Chart

See the following figure for the updated timeline of the project in the form of a Gantt chart.



Task	Start Date	End date	Completed
Design Phase	10/25/12	12/18/12	Yes
Changes to Requirements Document	10/25/12	11/1/12	Yes
Client GUI Prototype	10/25/12	11/8/12	Yes
Client-Server Prototype	11/8/12	11/22/12	Yes
Design Document (Diagrams, Use Cases)	11/22/12	12/11/12	Yes
Design Presentation	12/4/12	12/6/12	Yes
Implementation	1/2/13	5/1/13	No
Game Engine (Code/Test)	1/2/13	1/16/13	No
Server	1/16/13	2/16/13	No
Client	2/16/13	3/16/13	No
Code/Test GUI offline	2/16/13	3/9/13	No
Test client-server interactions with GUI	3/9/13	3/16/13	No
AIs	3/16/13	5/1/13	No
Code/Test Virtual Player	3/16/13	4/1/13	No
AI Scripting Creativity	4/1/13	5/1/13	No
Final Testing	4/15/13	5/1/13	No
Wrap-up	4/15/13	5/26/12	No
Prepare final document	4/15/13	5/26/13	No
Prepare academic festival presentation	4/15/13	5/3/13	No

Figure 10: Updated Gantt Chart

References

[1] J. Polkinghorn and A. Weber, "Requirements: Web-based strategy game with complex ais." Requirements Document for this project.

[2] S. Rabin, *Introduction to Game Development*, ch. 5. Course Technology PTR, 2 ed., 2009. Small sections with practical information.

[3] Levitin, *Introduction to the Design and Analysis of Algorithms*. Addison-Wesley, 3 ed., 2012. Useful algorithms for graph manipulation.

[4] I. Millington, *Artificial Intelligence for Games*. Morgan Kaufman Publishers, 1 ed., 2006. In-depth study of artificial intelligence in gaming.

[5] "W3schools." Webpage, 2012. HTML, Javascript, CSS, XML tutorials.

7 Glossary

AI: Artificial Intelligence, a general term for the computer players, encompassing both the AI client and the AI scripts.

AI Client: The Python program that handles communication between the server and an AI script. Functionally equivalent to the human client.

AI Script: An AI program responsible for decision-making. An AI Client will pass it a Gamestate, and from that it will follow some set of procedures to create a Move based on that.

Attack: An optional part of a player's move where the player sends fleets from an owned planet to an enemy-controlled planet. The player's fleets will battle the defending fleets until the defenders are defeated or the attackers retreat. Attacks are the only way to gain more territory. A player can make as many attacks per turn as desired.

Deployment: A required part of a player's move where the player places fleets on any planet he/she/it owns. More fleets are granted for deployment if the player controls one or more regions.

Gamestate: A full description of the game, including game-related information such as player numbers and fleet counts and graphics-related information such as planet colors. The player clients update the Gamestate

with Gamechanges over the course of the game. This is transmitted as XML, but stored as an object within the server and clients.

Gamechange: A description of the change between one turn and the next containing information on fleets per planet and new owners of planets. The server sends a Gamechange to each player as XML after each player's turn. The clients use this Gamechange to update their Gamestates.

Reinforcement: An optional part of a player's move where the player moves fleets from one owned planet to another owned planet. A player can make as many reinforcements per turn as desired.

Region: A grouping of planets pre-defined by the given map. Each region is worth a certain number of fleets, and if a player controls all the planets in a region, that player gets a set number of bonus fleets each turn.

8 Appendix: Class Diagrams

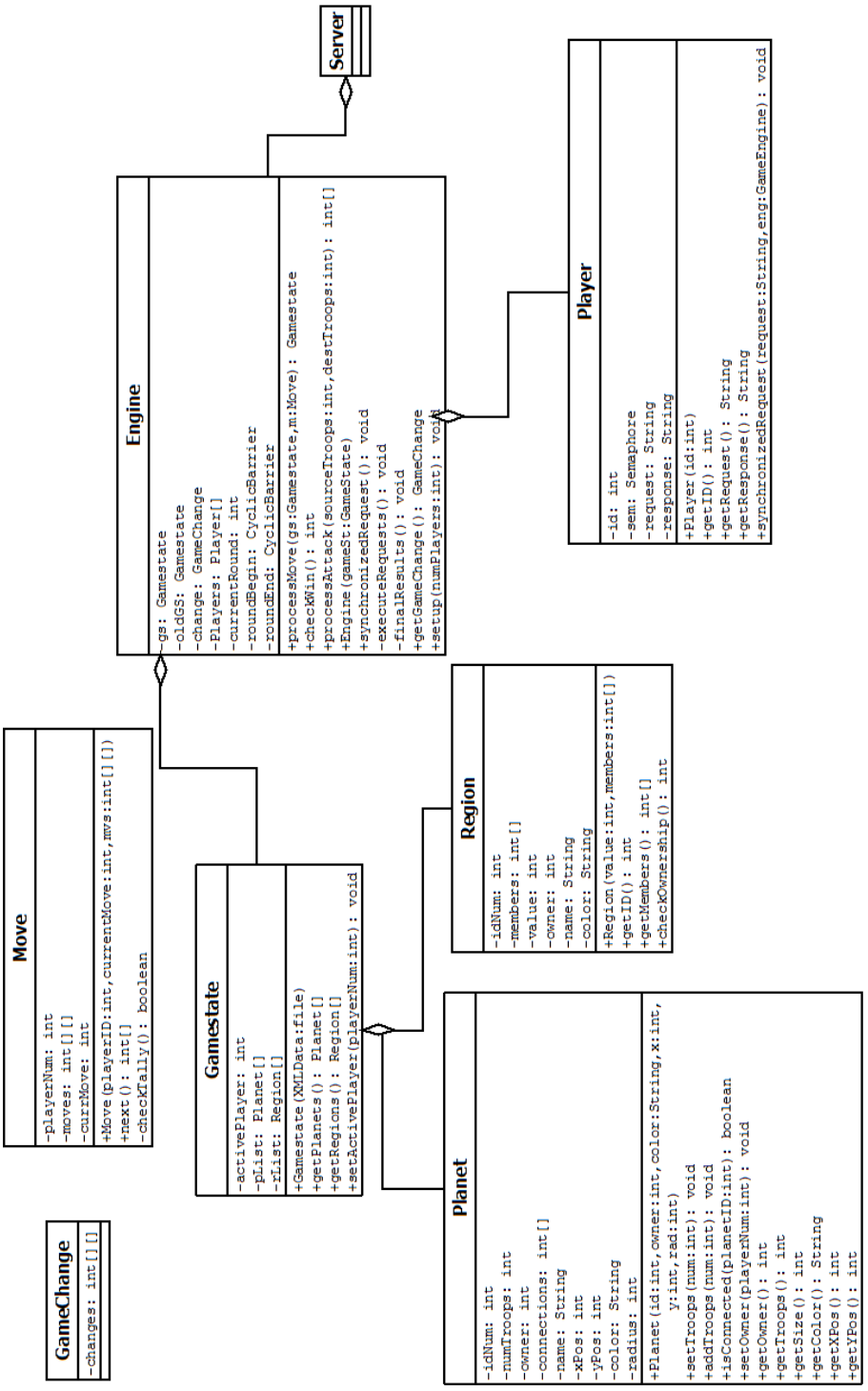


Figure 11: Game Engine Class Diagram

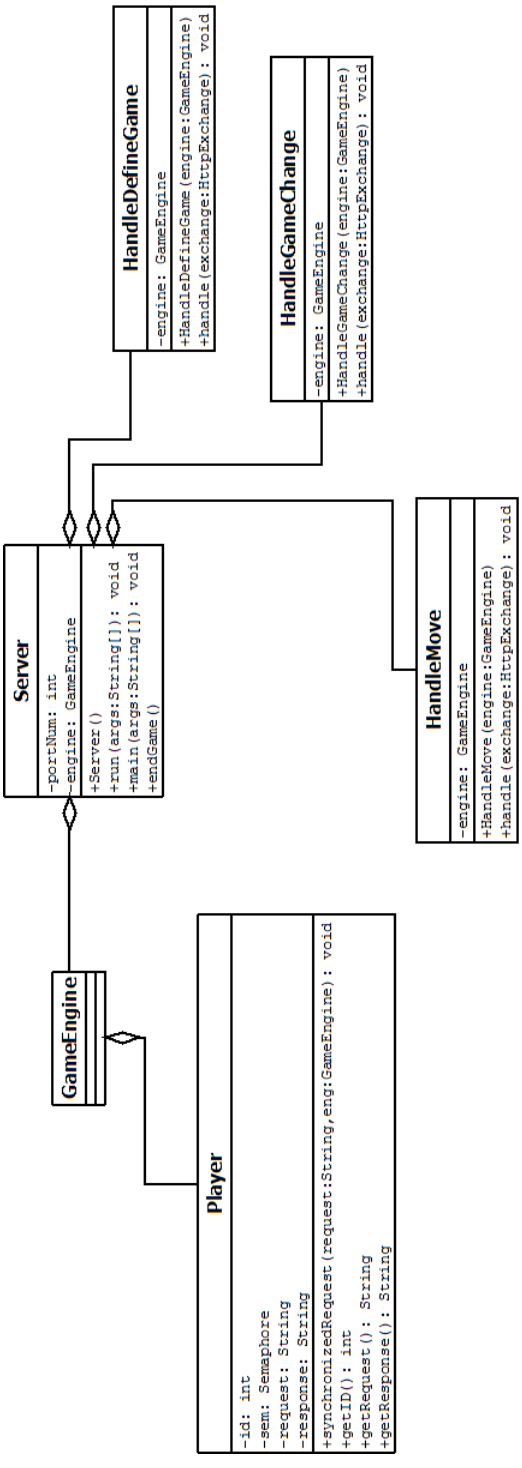


Figure 12: Game Server Class Diagram