Design Document

Project Scope:

The Robo Sport board game will serve a variety of purposes. First and foremost, Robo Sport is a game, and as such, it should serve to entertain and challenge the human players who use it. The game will consist of clever computer AI's who can beat a human player, but who are not so clever that a skilled human player could not win against them. The game will also provide a platform for multiple human players to face-off against each other. The emphasis on strategy will cater to a user who is interested in both entertainment and mental stimulation. Beyond these general purposes, Robo Sport will solve several problems that a real life board game version would face.

Implementing the Robo Sport board game as a software system will allow for advanced additional functionality that would not be possible on a real life game board. One major advantage the computer game will have is "the fog of war", which hides aspects of the game from the current player's field of view. This creates the additional challenge of locating the other team's robots on the game board. Teams of robots must work together to "see" beyond each of their own visual ranges, allowing them to strategize together against the other team(s). Another major advantage that is offered by a software system is the ability to track and record game statistics. Without effort, the human player is kept informed of everything currently occurring in the game. Once a game is finished, statistics will be uploaded to a server, allowing players to track the progress of different robots.

The use of computer AI's and "the fog of war" are two aspects which will make this game challenging. Users can select from a number of different computer AI's, each of them challenging the user in a different way based on their implementation. This will allow the user to strategize against different kinds of game play, enabling them to improve their level of strategical thinking in many different scenarios. A user can also improve their strategy with or against other human players. Two to three users can play on a team against two to three teams controlled by computer AI's. Two, three, or six users can also join a game and play solely against one another. The variety of ways in which the teams or players can be organized will provide colourful gameplay that does not become boring or repetitive for those who wish to use it.

The game is designed for use by a variety of demographics. Firstly, the game will be limited to people who are somewhat familiar with computers. Secondly, it will target users who wish to improve their skills in strategic thinking. Thirdly, it will aim to engage people who play computer games for the purpose of entertainment. People looking to challenge themselves can play against a variety of computer AI's, or against other people. As a whole, the game is designed for a user who is looking to challenge themselves in a multitude of different ways, as well as to sit down and have fun.

Architecture:

The Architecture for the Robo Sport system is a combination of Model-View-Controller and Component-based Architecture.
In course of creating a complex software system, it is essential to find a common abstraction of the system on which all of its elements can be constructed. The architecture of our system will allow developers to understand the over-arching concepts and major functionality without the complexity of implementation.

Careful consideration of the different high-level components in the system led to the decision to combine two different architectures. The level of interaction between the system and the user make it necessary to have both a View component (for displaying game content to the user) and a Controller component (for gathering and reacting to user input). The Controller component is also responsible, to a degree, for the regulation of the computer AI's. The use of various robots, teams of robots, and the hexagonal game board also calls for a Model component, which will hold the basic structure of each of these objects. Due to the systems ability to access data via a network, the necessity to include additional components to the overall architecture becomes apparent. The system requires a Forth environment for the

computer AI's, as well as the ability to access JSON files from a server. Model-View-Controller fails to esthetically and efficiently encapsulate the needs of these unique components. Adding extra components to accommodate network access and the AI functionality will serve to create an architecture which is both practical and system specific. The two architectures in combination will accommodate each of the systems distinct, high-level components, including the Forth environment for the AI's and abstract server communication.
Model-View-Controller plus Component based architecture allows us, the developers, to easily divide the pieces of the system, as well as the work required to construct them.

Class Descriptions:

Model:

Robot Class:

The Robot class is responsible for enclosing all of the functions and attributes pertaining to each individual robot.

- Interactions:
  - The GameMaster Class: uses the robot to update the robots position
  - Interpreter Class: is passed a robotID and determines how to handle the robot's "words"
  - Team Class: Stores a list of robots for it's team
  - Gameboard:
    - Stores a Hashmap of robots that are on a given coordinate
    - Stores a Hashmap of corrdinates that a robot is on
- Variables:
  - words: a JSON collection of functions that are defined in Forth inside of the completeRobot JSON file
  - variables: a JSON collection of variables defined in Forth inside of the completeRobot JSON file
  - inbox: a stack used to hold messages sent from other robots
  - movesRemaining: an integer tracking how many moves a robot has left in it's turn
  - hasShot: a boolean variable to keep track of whether or not the robot can shoot
  - health: an integer to store the health of the robot during gameplay
  - robotStats: a JSON object provided by the RobotLibrarian that is sent during construction and is not edited during gameplay. Used for quickly accessing robots in the GameStats file.
- Functions:
  - Construction:
    - (JSON object): The robot constructor - takes a JSON file from the Robot Librarian and creates a robotID by combining the robots team + name
    - getJSON( ): Gets the Robot-Stats JSON object file from the RobotLibrarian
  - Forth:
    - getTeam( ): Gets the team of robots from JSON (scout, sniper, tank) and returns a string with the team name
    - getID( ): returns an ID for a single robot on a team
  - Health:
    - setHealth(integer): Used to update the robot's health when it takes damage
    - getHealth( ): Returns an integer value holding the robot's health attribute
    - getType( ): Returns the type of a robot as an enum (scout, sniper, or tank)
    - isAlive( ): check if robots health is above 0, return true if yes
    - getStrength( ): returns the integer value holding the robots strength attribute

    - Actions:
        - move(integer): Used to move the robot forward according to the integer parameter
        - move( ): Used to move the robot forward one space
        - movesRemaining( ): Returns the amount of moves a robot has remaining
        - canShoot( ): Returns true if the robot can still shoot, false otherwise
        - turn(integer): Turns the robot according to the integer parameter
        - turn( ): Turns the robot once clockwise
        - shoot( ): Shoots at a tile in a straight line from the robots current position
        - play( ): If the robot is not a human, this function is called to play the computer AI
    - Messaging:
        - receiveMsg( ): Grabs a message from the inbox stack and returns a string
        - sendMsg(string, robotID): Pushes a message to the receiver's inbox stack

Team Class:

The Team class provides a way to represent the teams of robots for both the View component and the GameMaster class inside of the Controller component. The Team class makes it much easier to provide each team of robots with a limited visual scope as per their visual range. This implementation enables "the fog of war", which gives rise to new strategic possibilities.

- Interactions …
- Variables:
    - robots[ ]: collection of three robots associated with the team (scout, sniper, tank)
    - name: the name of the team
    - colour: the colour of the team
- Functions:
    - Team(robots[ ], colour, name): returns a newly created and initialized Team object
    - getRobots( ): returns robots[ ], a collection of robots associated with the team
    - getName( ): returns the name of the team
    - getColour( ): returns and integer representing the colour of the team

StatsLogger Class:

In order for the system to store game statistics, it requires a class which will log game events. Our system will download one JSON file for every robot entering the game; these files will be known as the JSON Robot-Stats files. These files contain a record of robot statistics from previous games. Our system will create one JSON file for an instance of the game; this file will be known as the JSON Game-Stats file. The JSON Game-Stats file will record and upload all of the robot statistics for the game to the individual JSON Robot-Stats files at the end of each game. The decision to store these statistics in a JSON file is a consequence of JSON's fast look-up time and the notion that it is advantageous to store the data in one place.

- Used by the Input Controller.
- Variables:
    - StatsLog: JSON file containing a list of robots

- Functions:

  Update functions update individual statistics in a single instance of a JSON Robot-Stats file (or does it update our file?)

  robotID stores a unique identification tag for each robot

  - StatsLogger( ): creates and initializes a JSON list of all robots in the game
  - updateMatches(robotID): updates the number of times the robot has played
  - updateWins(robotID): updates the number of times the robot has been on the winning team
  - updateLosses(robotID): updates the number of times the robot has been on the losing team
  - updateKilled(robotID): updates the number of times the robot has killed another team's robot
  - updateAbsorbed(robotID): updates the number of times the robot has been hit by another team's robot
  - updateLived(robotID): updates the number of times the robot has remained alive for the entire game
  - updateDied(robotID): updates the number of times the robot has been killed
  - updateMoved(robotID): updates the number of times the robot has moved from one space to another
  - updateExecutions(robotID): updates the number of times a computer AI has been used

GameBoard Class:

The GameBoard class provides a system level game-board which will store the locations of all of the robots in play. Having a centralized location for the storage of robot positions aids in the simplification of the design. The system will make use of a hashmap to look up the coordinates of a given robot quickly. The use of a hashmap will promote fast look-up time, which is beneficial due to the frequent need to query a robot's position.

- Interactions:
  - The GameBoard class interacts with the Robot class, specifically the collection of robots, the positions of which it will be storing. It also interacts with the GameMaster, which will contain a copy of the game-board for the purpose of updating the view between plays.
- Variables:
  - robotCoord<robotID, coord>: a hashmap keyed by robotID which returns the coordinate of the current robot's location
  - robotOnCoord<coord, robots[ ]>: a hashmap that is keyed by a coordinate and returns a list of robots that are currently occupying that coordinate, if any
- Functions:
  - GameBoard(teams[ ]): Creates, initializes, and returns a new GameBoard object
  - getRobotsAtCoord(coord): returns a list of robots at a given coordinate
  - getRobotCoord(robotID): returns the current coordinate location of a given robot
  - updateCoord(coord, robotID): update the coordinates of a given robot
  - inRange(robotID): returns a list of robots in range of a given robot

- Private Variables:
    - - robots <name, robot>: hashmap
    - - matches: the number of times the robot has played
    - - wins: the number of times the robot has been on the winning team
    - - losses: the number of times the robot has been on the losing team
    - - killed: the number of times the robot has killed another team's robot
    - - absorbed: the number of times the robot has been hit by another team's robot
    - - lived: the number of times the robot has remained alive for the entire game
    - - died: the number of times the robot has been killed
    - - moved: the number of times the robot has moved from one space to another
    - - executions: the number of times a computer AI has been used