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. This includes things like the robot's health, as well as its functions that are defined in Forth. The robot is only constructed during setup of the game and it is instantiated by the SetUpController.

- 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:
  - robotJSON : all of the information pertaining to a particular robot in a JSON object which comes from the Robot Librarian upon construction
  - robotID: a combination of the robot's name + teamID + gangID
  - gangID: an integer value assigned to each robot on a given team upon robot creation
  - 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:
    - Robot(robotJSON, gangID): The robot constructor, takes a JSON file from the Robot Librarian and an integer value for the gangID and creates a robotID by combining the robots name + teamID + gangID
  - Status:
    - 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
- 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
- Miscellaneous:
    - getTeam( ): Gets the team of robots from JSON (scout, sniper, tank) and returns a string with the team name
    - getGang( ): Returns the gangID
    - getID( ): returns an ID for a single robot on a team
    - getJSON( ): Returns the robotJSON object

Gang Class:

The Gang 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 Gang class makes it easy to provide each Gang 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:
    - The Gang class interacts with the GameMaster class which contains the list of all robot gangs currently in play, with the StatsLogger class which holds the robotJSON file, and with the SetUpMenuController which creates the various robot gangs at the start of the game.
- Variables:
    - robots[ ]: collection of three robots associated with the Gang (scout, sniper, tank)
    - name: the name of the Gang, represented by an integer
    - colour: the colour of the Gang
- Functions:
    - Gang(robots[ ], colour, name): returns a newly created and initialized Gang object
    - getRobots( ): returns robots[ ], a collection of robots associated with the Gang
    - getName( ): returns the name of the Gang as represented by an integer
    - getColour( ): returns and integer representing the colour of the Gang

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 robotJSON files. These files contain a record of robot statistics from previous games as well as functions for the robot behaviour. Our system will create one JSON file for an instance of the game; this file will be known as the statsJSON file. The statsJSON file will record robot statistics. 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.

- Interactions:
    - The StatsLogger interacts with the SetUpMenuController which creates the various robot gangs at the start of the game, with the InputController which records data from the different user moves, and with the Interpreter class which records individual robot moves. It also interacts with the GameMaster, which passes the statsJSON file around to the various views.
- Variables:
    - statsJSON: a JSON file containing the key to the robotJSON of each individual robot

- Functions:

    All update functions will update individual statistics in a single instance of a statsJSON file which contains a key for each of the robotJSON files

    robotID stores a unique identification tag for each robot

    - StatsLogger( ): creates and initializes a statsJSON with a set of keys, each initialized to an individual robot 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 reduces coupling in the system. It will make use of a hashmap to look up the coordinates of a given robot. This 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, and the robot coordinates as it updates their positions on the board. It also interacts with the GameMaster, which will contain a copy of the game-board for the purpose of updating the view between plays. The Gameboard also interacts with the SetUpMenuController when it creates the initial game board.
- 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

View:

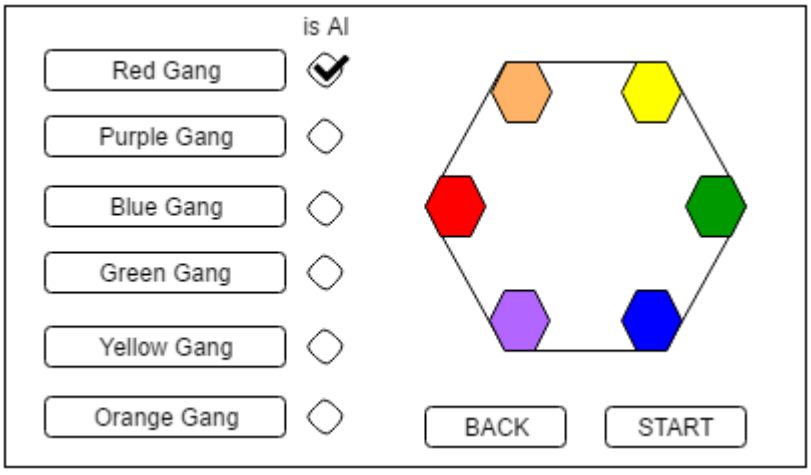
StartScreen Class:

The StartScreen class is the portion of the View that displays information and user input options on the initial start-up of the program. It displays the Game Title, a background image, a Start button, an Instructions button, and an End button. The StartScreen requires its own class because it requires a unique set of event handlers.

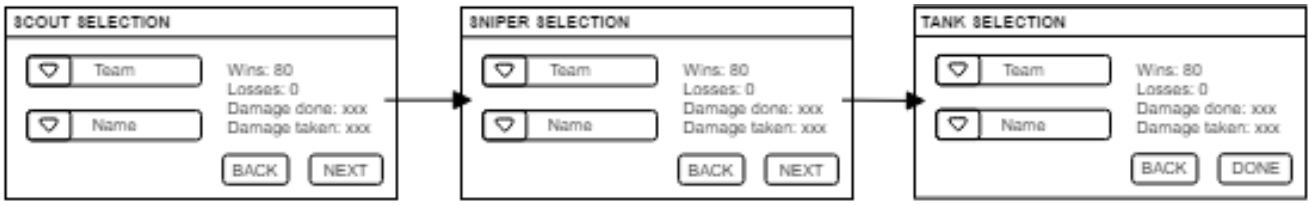
- Interactions:
  - The StartScreen class will interact with the StartScreenController class, which calls the StartScreen class upon opening the game. Any time the user pushes any of the StartScreen buttons, the event handling is done by the StartScreenController Class.
- Functions:
  - constructor( ): Constructs the Game Title, image, and buttons upon initial opening of the StartScreen
  - buttonListener( ): listens for button selection from the StartScreen class and calls the respective controller function from StartScreenController

SetUpMenu Class:

The SetUpMenu class is the portion of the View that displays information and user input options for match setup prior to the start of a new game. It displays a game-map graphic, a list of Gang buttons by colour, and a Start and Back button. All of the event handling for the buttons in the SetUpMenu view are handled by the SetUpMenuController. Due to the large number of unique inputs in the SetUpMenu, it requires its own class to handle this input.



Each button corresponds to a different Gang colour. The purpose of the Gang class is to give a grouping of 3 robots a unique number to differentiate them from other groupings of 3 robots. The player will use the toggle button to indicate whether the Gang will be controlled by a human or a computer AI. Once the player has selected a Gang Number, the SetUpMenuController will instantiate a SetUpMenuPopUp.



The SetUpMenuPopUp opens once a player has selected a Gang Number. For each type of robot in the Gang (Scout, Sniper, Tank), the user can select from various robots created by a given Software-Team. If the Back button is pressed, the Gang Number is deselected and the popup closes. Once the user selects a Software-Team, it can select a robot by name. Once a robot is selected by name, a list of statistics for that robot will appear on the right hand side of the SetUpPopUp screen. The Next button will take the user through the selection of all three robots, and once this selection is complete, the user can click done to close the SetUpMenuPopUp and return to the SetUpMenu.

- Interactions:
    - The StartScreenController instantiates the SetUpMenuController when the Start button is pressed in the StartScreen view. The SetUpMenuController draws the view for the SetUpMenu. Once the user has selected their robot gangs, the user may press the Start button. Pressing Start is handled by the SetUpMenuController, which calls the GameMaster to draw the Game View. If the user decides to select the Back button, the SetUpMenuController passes back control to the StartScreenController which re-instantiates the StartScreen view.
- Variables:
    - roboStatsText: a string representing what is visible in the robot selection screen
    - map: an image of the hexagonal game grid
- Functions:
    - Listeners:
        - buttonListener( ): listens for button selection from the SetUpMenu view and calls the respective controller function from SetUpMenuController
        - popUpButtonListener( ): listens for button selection from the SetUpMenuPopUp view and calls the respective controller function from SetUpMenuController
    - Display:
        - SetUpMenu( ): Constructs and initializes the view of the SetUpMenu, called by the SetUpMenuController
        - displayRobotSelection( ): Displays the SetUpMenuPopUp for individual robot selection
        - redrawMap( ): Redraws the map on the SetUpMenu

PassTheControls Class:

Due to the fact that our game can only exist on one screen, a screen between turns is necessary to prevent the accidental viewing of one players screen by an opponent. This screen is only necessary when there is more than one human player in the game at a time. The PassTheControls screen will consist of a simple graphic, an instruction of whose turn it is, and an OK button. The user whose turn it is should click the OK button to confirm that the controls have been successfully passed over from the previous player.

- Interactions:
    - The PassTheControls view interacts with the InputController, as well as the GameMaster when the user clicks the OK button. The InputController handles the input and the GameMaster re-opens the GameScreen.

- Functions:
    - PassTheControlsView(Gang Number): Creates a window containing a graphic, a message telling the user to pass the controller to the next player, and an OK button.

GameScreen Class:

The most important feature of the game is the GameScreen itself. The GameScreen displays the hexagonal game board with a restricted view according to whose turn it is. The “fog of war” will be made possible by overlaying an image that displays only the hex tiles that are within the current player’s range of vision. Hovering over a game tile that is within the current players range of vision will reveal a display of all of the robots currently residing on that tile. The GameScreen also contains thee buttons to the side of the game-board: Move, Attack, and Forfeit. The GameScreen class uses several different Controller components to dictate its behaviour.

- Interactions:
    - The GameScreen class interacts with the GameMaster, who receives coordinates of the tanks displayed on the board.
- Functions:
    - buttonListener( ): listens for button down events and calls the respective controller functions in the GameMaster
    - tileListener( ): listens for tile selection and calls the appropriate controller functions in the GameMaster
    - GameScreen(moveableCoord[ ], visibleCoord[ ], visibleRobots(<robotType, coord>, COLOUR)):
        - Creates a view with clickable hex tiles from moveableCoords[ ] (the set of all coordinates that a robot may move to), visible hex tiles from visibleCoords[ ] (the set of coordinates that are within the range of the current gang’s vision, and all of the visibleRobots (queried from a dictionary containing the robots in the current game).
    - redrawSprite(visibleRobots<robotType, coord>, COLOUR):
        - Redraws the sprites onto a map after each turn or movement.
        - Is called whenever a new tank comes into view or the turn changes from one player to another.
        - Uses DrawRobot subclass to draw the specific robot type of a given robot
    - redrawSight(visibleCoord[ ], visibleRobots<robotType, coord>, COLOUR):
        - Redraws the sight overlay for all robots in the current gang.
        - Called upon a robot’s movement or when the turn changes from one player to another
    - redrawMove(moveableCoord[ ], visibleRobots<robotType, coord>, COLOUR):
        - Redraws the possible move overlay for all robots in the current player’s gang.
        - Called upon robot movement or when the turn changes from one player to another.

- 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