Design Document for the Robo Sport Software System

Software Engineering Team: D2

Arianne Butler

Kristof Mercier

Michael Graham

Samuel Horvatin

Christopher Mykota-Reid

Contents

Glossary

Robot: a moveable game object that plays on a team of three

Gang: a group of three robots, all of which are controlled by either a human player or a computer AI

Computer AI: Controls the robots that are not controlled by human players.

Human player: a human player that is controlling one of the gangs of robots in the game

Player: a human player or a computer player

Scout: the robot player with the highest range of movement

Sniper: the robot player with the highest visual/firing range

Tank: the robot player with the highest attack strength and health

Team: the team of software engineers creating the system

Tile: a hexagon shaped unit that makes up the game map

Forth: a language that will be used as a standard for sharing Robots with other systems

View: anything the player can see on the screen is a view

Model: an overarching term for anything that holds data that is regularly modified

Control: anything that has a large logical component and manages other components of the system

Project Scope

The Robo Sport board game will serve a variety of purposes. First and foremost, Robo Sport is being developed to satisfy the requirements of CMPT 370. It is being used to demonstrate the practicality as well as the importance of proper Software Engineering practices including (but not limited to): planning requirements and designing and implementing software. Furthermore, it should serve to entertain and challenge the human players who use it. The game will consist of computer AI's developed by a collection of student software teams. The computer AI's will be obtainable from a server and the user will be able to view the AI's statistics and choose to play with or against any of 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.

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 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. Gangs of robots must work together to "see" beyond each of their own visual ranges, allowing them to strategize together against the other robot gangs. Another major advantage that is offered by a software system is the ability to track and record game statistics. Some statistics will be displayed to the user during gameplay and all statistics will be uploaded to a server post game, 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. The wide range of computer AI's will challenge the user in a different way based on their implementation. This serves to improve strategic thinking in varying scenarios. A user can also improve their strategy against other human players. There may be two, three, or six players in a game, where any number of them may be controlled by computer AI's.

The game is designed for use by a variety of demographics. First, 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 entertainment.

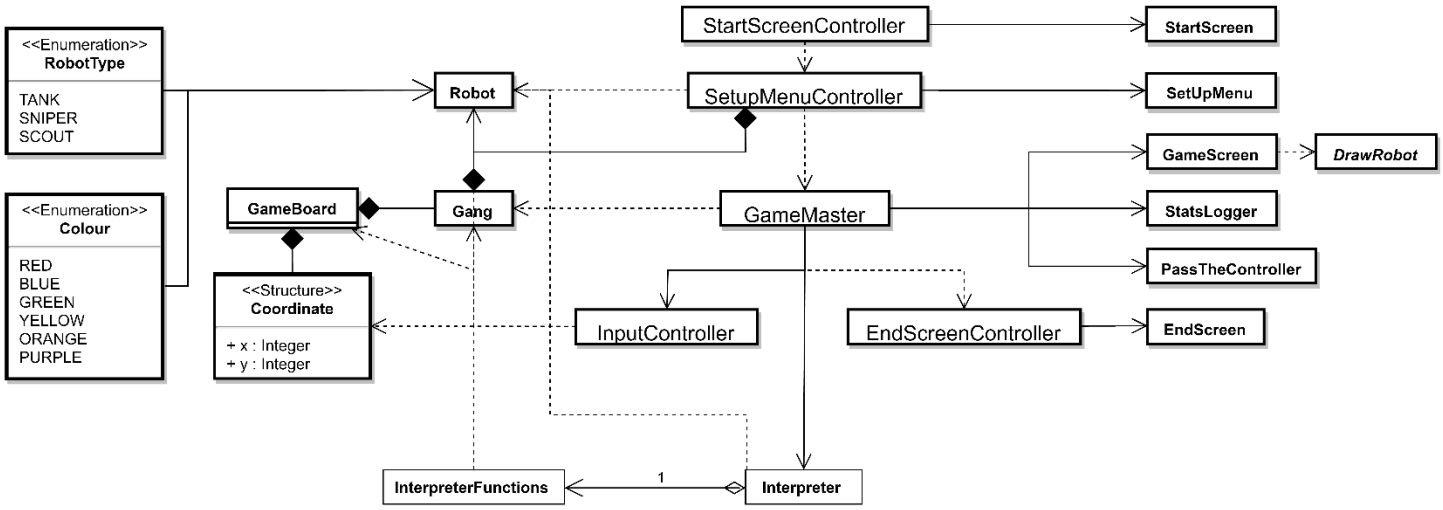
Architecture

The architecture for this system is a Layered Architecture that uses a Model-View-Controller (MVC) design pattern and independent Aspect-Oriented Components. Not only will this allow for new developers to easily understand the system without getting lost in the implementation, but it will increase the system's extensibility, reliability and reduce the amount of maintenance required. This is because the use of the MVC design pattern forces developers to separate the concerns of each individual class. In turn, this results in a reduction of the total amount of coupling that takes place in the system. A bonus side effect of this is that the amount of time spent on system upkeep when changes are made to a component is greatly reduced, as developers can locate the affected areas quickly. The MVC design pattern is a good choice for Robo Sport because it allows for the game's data, logic and display to be separate entities, resulting in a clean and extensible system. The use of additional Aspect-Oriented Components allows for system diversity and extensibility, as users are able to define their own AI as well as their AI's functions using Forth.

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 makes it beneficial 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 for the regulation of the computer AI's. The use of various robots, gangs of robots, and the hexagonal game board also call for a Model component, which will hold the basic structure of

each of these objects. Due to the systems ability to share computer AI's, it is beneficial to include additional components to the overall architecture. The system requires a Forth interpreter for the computer AI's, as well as the ability to access robot JSON files from a server. Model-View-Controller fails to esthetically and efficiently encapsulate the needs of these unique mechanisms. Adding extra components to accommodate 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.

System Architecture Diagram using Model-View-Controller/Component Based Design:

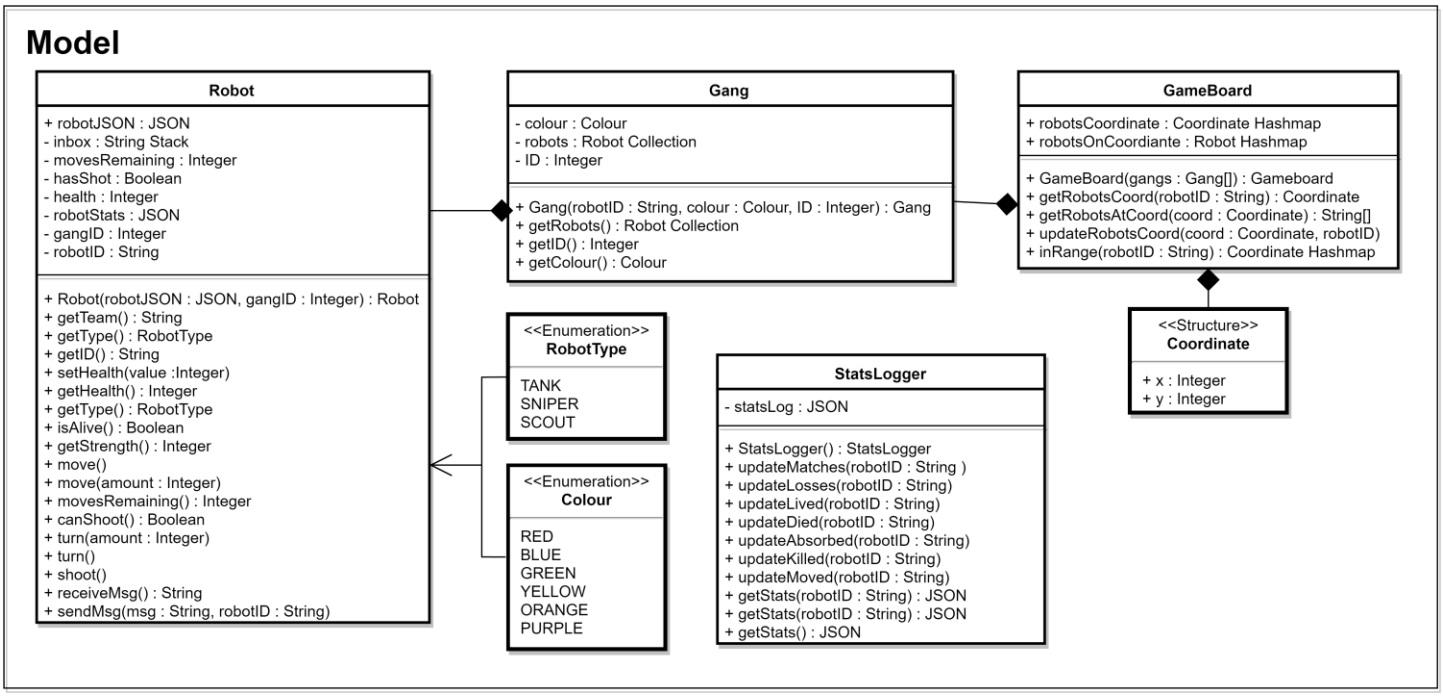


The above diagram is an overall view of all components in the system. Further encapsulation of Model, View, Controller, and Interpreter will appear later in the document (on pages 6, 10, 18, and 21 respectively). The View component encompasses all components that display information to the user. The View Classes are as follows: StartScreen, SetUpMenu, GameScreen, PassController, EndScreen, DrawRobot (with DrawScout, DrawSniper, and DrawTank all extending the DrawRobot class). The Model component of the system is responsible for data storage. The Model classes are as follows: Robot, Gang, StatsLogger, and Gameboard. Both the Model and the View interact with the Controller, which acts as an intermediary between them. The View sends the information it receives from the user to the Controller and the Controller processes the data in some way. The Controller then sends the processed information to the Model which updates itself accordingly. The Controller Classes are: StartScreenController, SetUpMenuController, GameMaster, InputController, and EndScreenController. Lastly, the Interpreter component, which consists of the Interpreter class and the Interpreter Functions class are responsible for interpreting the Forth words provided by the computer AI's and converting them into meaningful code.

Architectural Components

Model:

Model UML Diagram:



The above image is a detailed UML diagram for the Model component of our system. It contains four classes (Robot, Gang, StatsLogger, and Gameboard) as well as two enumerations (RobotType and Colour). The RobotType enumeration is used to store what class of tank the robot belongs to and the Colour enumeration is used to store the colour that is designated to each Gang. Finally, the Coordinate structure is used to encapsulate the (x,y) position of each tile or hexagon.

Model Classes:

Robot Class:

The Robot class is responsible for enclosing all 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 constructed during game set up and it is instantiated by the SetUpMenuController. User input pertaining to the Robot actions is handled by the view and sent to the InputController.

Interactions:

The GameMaster class interacts with the Robot to update its position on the GameBoard. The Gang Class stores a list of three Robots that are members of its Gang. The GameView stores a hash-map keyed by a robot's ID and is used to query the specific coordinate that the Robot is on. It also stores a second hash-map keyed by a coordinate to query which Robots are on that specific hex tile. Player input is handled by the GameScreen view and is sent to the InputController, which makes the calls to the specific Robot functions in the Robot class.

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 unique combination of the robot's name, teamID and gangID
- gangID: an integer value assigned to each robot on a given team upon robot creation
- inbox: a stack of strings used to hold messages sent from other robots
- movesRemaining: an integer tracking how many moves that 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(amount): 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(amount): Moves the robot forward by an amount
  - move( ): Moves 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(amount): Turns the robot a certain amount of times
  - 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(msg, robotID): Pushes a message to the receiver's inbox stack
- Miscellaneous:
  - getTeam( ): Gets the team of that robot and returns a string with the team name
  - getGang( ): Returns the gangID
  - getID( ): Returns that robot's unique ID

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. It simplifies the limiting of a gang of Robots visual range. This

implementation creates a “fog of war” effect which will force players to think strategically when moving units. A Gang consists of a grouping of three robots (one scout, one sniper, and one tank) and each Gang is assigned its own unique Gang colour.

Interactions:

The Gang class interacts with the GameMaster class (which contains the list of all Robot Gangs currently in play), the StatsLogger class (that keeps track of stats on all Robots), and with the SetUpMenuController which creates the various robot gangs at the start of the game.

Variables:

- robots[ ]: a collection of the three robots that are in the Gang (scout, sniper, tank)
- ID: the ID of the Gang, represented by an integer
- Colour: the colour of the Gang

Functions:

- Gang(robots[ ], colour, ID): returns a newly created and initialized Gang object
- getRobots( ): returns robots[ ], a collection of robots associated with the Gang
- getID( ): returns the ID of the Gang as represented by an integer
- getColour( ): returns a enumeration called Colour which represents 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. This will be known as the StatsLogger. It will create one JSON file for each instance of the game; this file will be known as the statsJSON file. This file will record statistics on the game and inside will be one entry in for each unique robot in the game. NOTE: Two different gangs can both be using the same robot in one game. Because of this, a robot’s stats will be recorded under their name+teamID instead of their robotID. This will allow for stats to be recorded for each unique robot; not each unique gang.

Interactions:

The StatsLogger is called by the InputController and GameMaster so that it may record all of the information from each robot’s turn. At the end of the game this information is used by the EndScreenController to update the robot’s statistics online.//

Variables:

- statsJSON: a JSON file containing the key to the stats for that game of each individual robot

Functions:

- StatsLogger( ): creates and initializes a statsJSON with a set of keys, each initialized to an individual robot in the game
- updateMatches(robotID): increments the number of times the robot has played
- updateWins(robotID): increments the number of times the robot has been on the winning team
- updateLosses(robotID): increments the number of times the robot has been on the losing team
- updateKilled(robotID): increments the number of times the robot has killed another team’s robot
- updateAbsorbed(damage, robotID): increases the amount of damage that robot has taken
- updateLived(robotID): increments the number of times the robot has remained alive for the entire game
- updateDied(robotID): increments the number of times the robot has been killed
- updateMoved(robotID): increments the number of times the robot has moved from one space to another

- updateExecutions(robotID): increments 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 and a hashmap to look up the robots that are on a coordinate. This will promote fast look-up time, which is beneficial due to the frequent need to query robot's positions.

Interactions:

The GameBoard class interacts with the Robot class (more specifically a collection of robots), the positions of which it will be storing. The GameBoard class is only instantiated once (by the SetupMenuClass) and it is passed into the GameMaster upon its construction. As the Gameboard is only responsible for storing the robot's position the GameMaster must update each robot's coordinates by calling the corresponding functions of this class when necessary.

Variables:

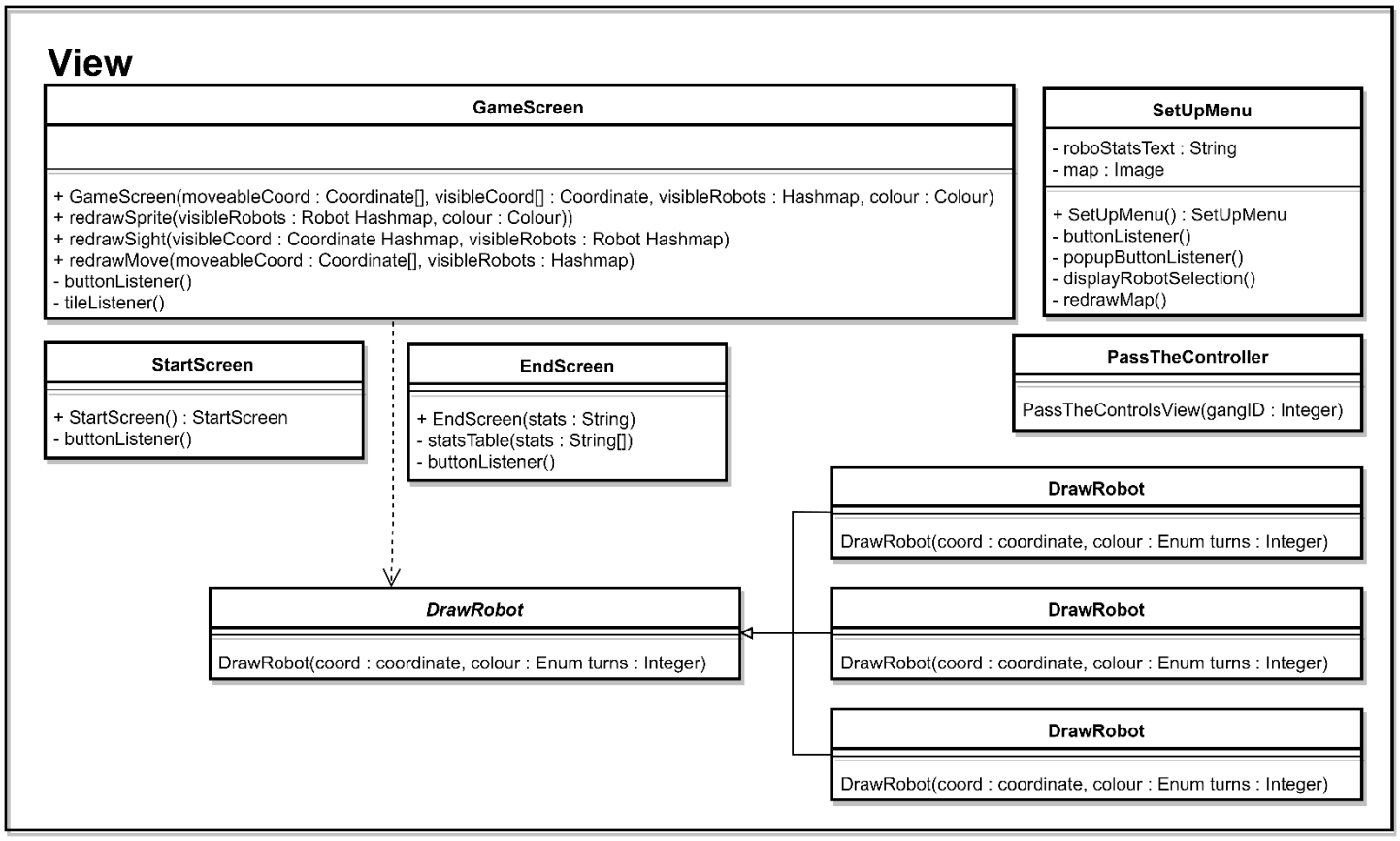
- robotCoord<robotID, coord>: a hashmap keyed by robotID which returns the coordinate of the current robot's location
- robotOnCoord<coord, robots[ ]>: a hashmap that is keyed by a coordinate and returns a list of robots that are currently occupying that coordinate, if any

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:

View UML Diagram:



The above image is a detailed UML diagram of the View component of our system; it contains all possible screens that may be opened or viewed by a human player. This includes: the StartScreen, the SetUpMenu, the GameScreen (which uses the DrawRobot and its subclasses to draw and redraw the Robot sprites), the PassController screen, and the EndScreen. The View is responsible for handling initial user input and passing this data to the Controller component.

View Classes:

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 Start button opens the SetUpMenu screen, the Instructions button instantiates a small pop-up containing game rules and instructions, and the End button closes the game window. The StartScreen requires its own class because it has 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 buttonListener will call the appropriate functions. If the Start button is pressed the StartScren will alert the StartScreenController which will then handle the event.

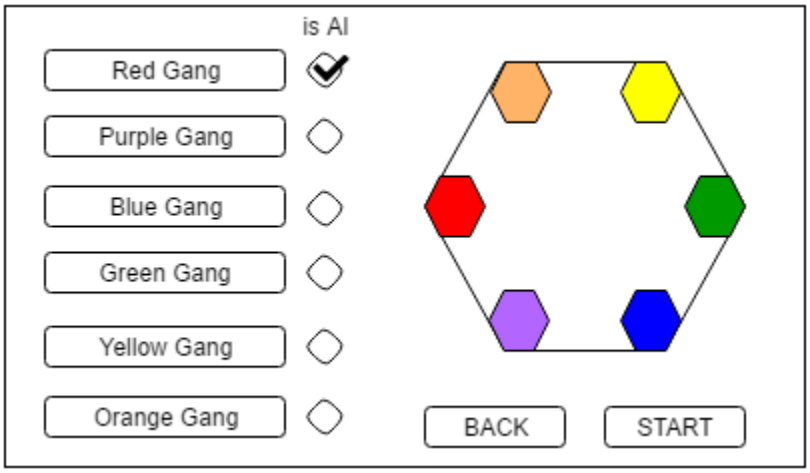
Functions:

- 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:

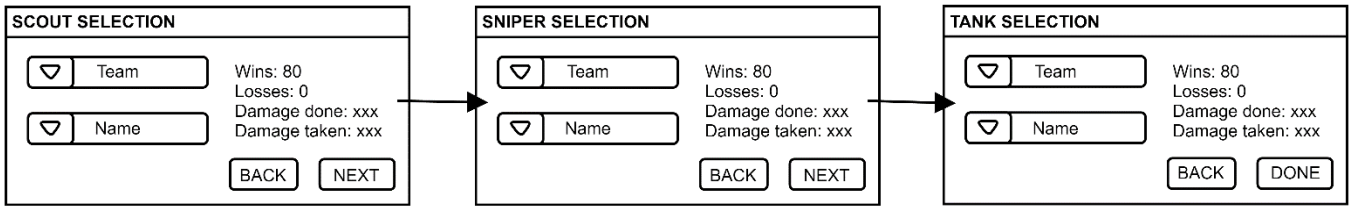
The SetUpMenu class is the portion of the View that displays information and user input options for seting up a match prior to the start of a new game. It displays a game-map graphic, a list of Gang buttons by colour, and a Start and Back button. All of the event handling is done by the use of two buttons: buttonListener and popupButtonListener(). When buttons are clicked that require the controller to update the model the SetUpMenuController is informed, and it calls the appropriate functions.

SetUpMenu View:



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, the SetUpMenuController will instantiate a SetUpMenuPopUp.

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. Upon the robot's selection, a list of statistics for that robot will appear on the right-hand side of the SetUpMenuPopUp 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 the Done button 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 constructs the SetUpMenuView. Once the user has selected their robot gangs, the user may press the Start button. Upon pressing Start, the View informs the controller that the

GameMaster must be constructed. If the user selects the Back button, the SetUpMenuController passes back control to the StartScreenController which re-instantiates the StartScreen view.

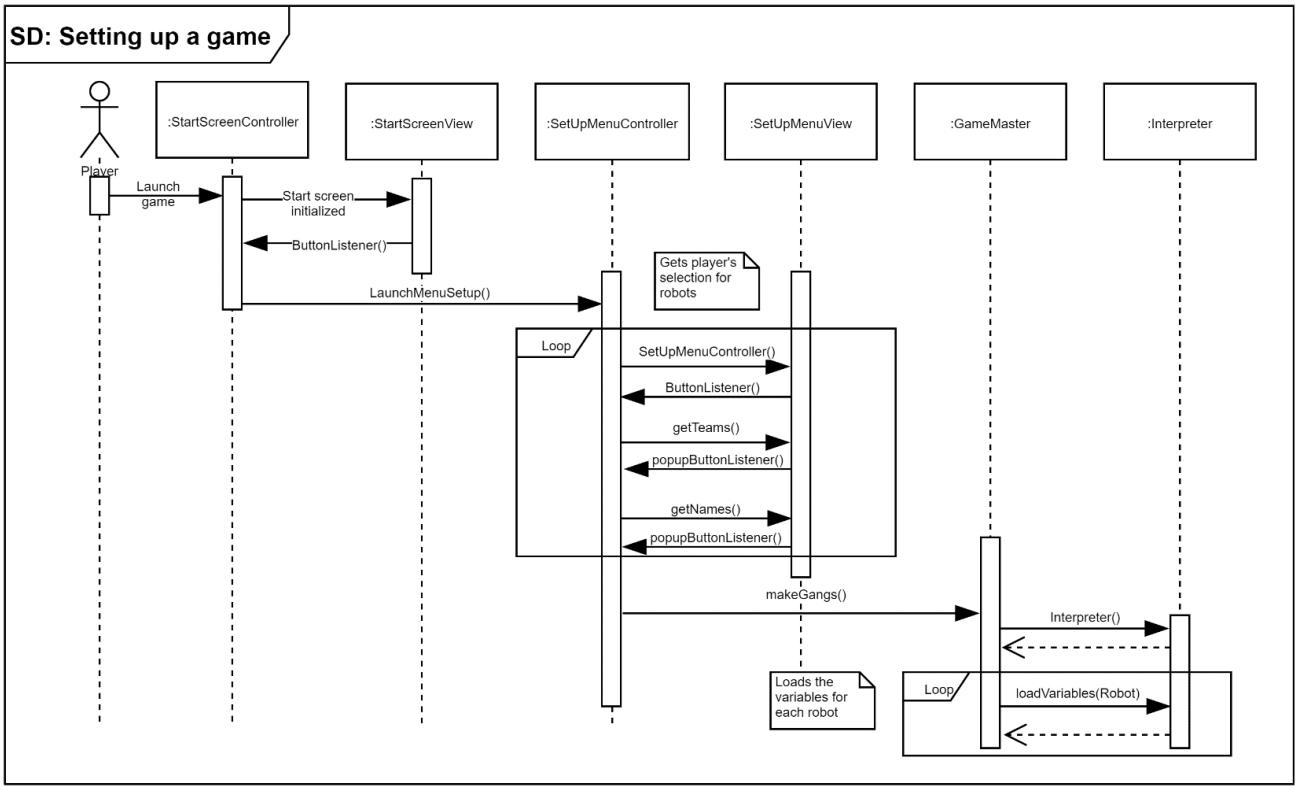
Variables:

- roboStatsText: a string which gets updated by the SetUpMenuController to dynamically display the stats of a given robot
- 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 SetUpMenuSequence Diagram for Game Setup

Sequence Diagram for Game Setup:



The above diagram shows the sequence of events for setting up the game. The user begins at the StartScreen, where they can select from the Instructions button, the Start button, and the Exit button. If the Start button is selected, the SetUpMenuController will be instantiated which launches the Setup View. In this view, the player can select which colour they want to assign robots to. Upon clicking on the Gang colour, the player will be

prompted by a pop-up window to select a Robot (specifically a scout, tank or sniper). When they select the Robot they want, they can click the Next button and will be prompted to select their next Robot (until a scout, tank and sniper have been chosen).

GameScreen Class:

The most important feature of the game from the perspective of a user 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 having a background image of a greyed-out game map. Tiles that are within the current player's range of vision will then be drawn in on top. 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 three buttons to the side of the game-board: Move, Attack, and Forfeit. The GameScreen class uses several different Controller components to dictate its behaviour.

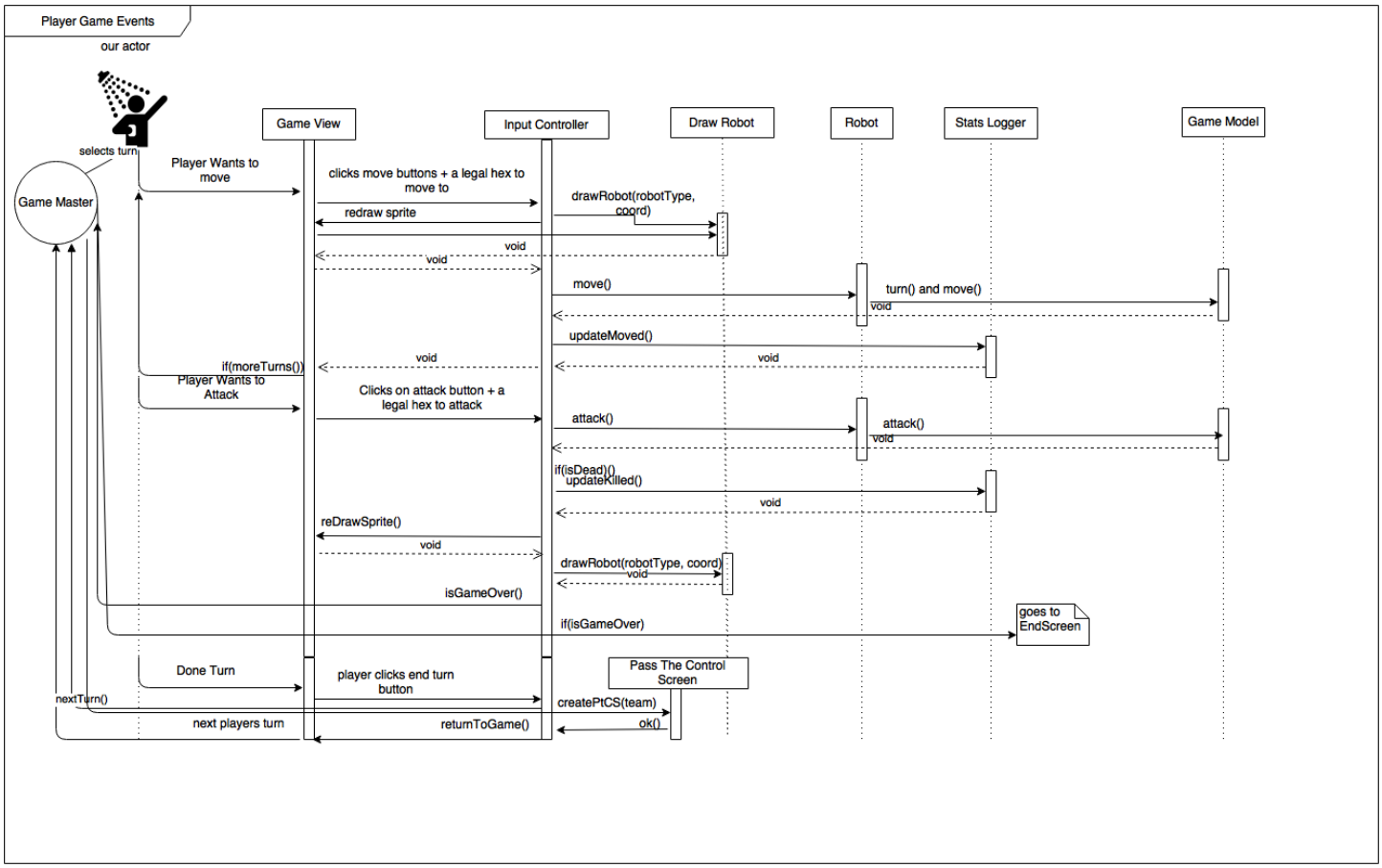
Interactions:

The GameScreen class interacts with the GameMaster, who receives coordinates of the tanks displayed on the board. It will also interact with the DrawRobot class for drawing all visible robots.

Functions:

- buttonListener( ): listens for button down events and calls the respective controller functions in the InputController
- tileListener( ): listens for tile selection and tile hover events, 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

Sequence Diagram For Player Game Events



The above diagram illustrates the game view which begins by awaiting input from a player. If a player chooses to move, they click on the move button and then they will click tile they wish to move to. The Input Controller prompts the view to redraw to the robot sprite, moving it to the new location, and calls the robot to move the robot in the model. After the move the Stats Logger records the move and the player can make another move (if they have moves remaining) or attack (if they have attacks remaining).

If the player chooses to attack they click on the attack button, then on a legal hex within range of their robot to send their attack to. The input controller is called to handle the event then the Input Controller calls the robot to make the attack on the Game Model. The system then calls up, back to the Input Controller, which checks if the robot attacked is dead and if so tells the stat logger to update killed robots. Still under this conditional the system redraws the robots and then checks to see if the game is over. The Game Master is called to check if the game is over and if it is the End Game Screen is created by the Game Master.

After the player has completed their moves or no longer wishes to continue their turn they click the End Turn button and the Input Controller calls back to the Game Master which creates the Pass the Controller View passed with the teamName. Once OK has been pressed on that screen the Input Controller then calls back to the Game Master which starts the next turn.

Note: There is no mention of the hover over tile function since it is not a part of the events that take place that bring about the completion of turns or the game. It is simply a function of the system that could be taken advantage of by the user if they chose to.

PassControls Class:

Since our game can only exist on one screen, a screen between turns is necessary to prevent the accidental viewing of a player's screen by an opponent. This screen is only necessary when there is more than one human player in the game at a time. The PassControls 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 PassControls 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:

PassControlsView(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.

EndScreen Class:

The EndScreen class displays statistics from the current game upon its conclusion. Additionally, it provides options for how a user can proceed. The EndScreen contains a stats table that displays individual statistics for robots, a single column showing the wins and losses of the participating teams, and three buttons: A Back button, a Save-Stats button, and a Quit button.

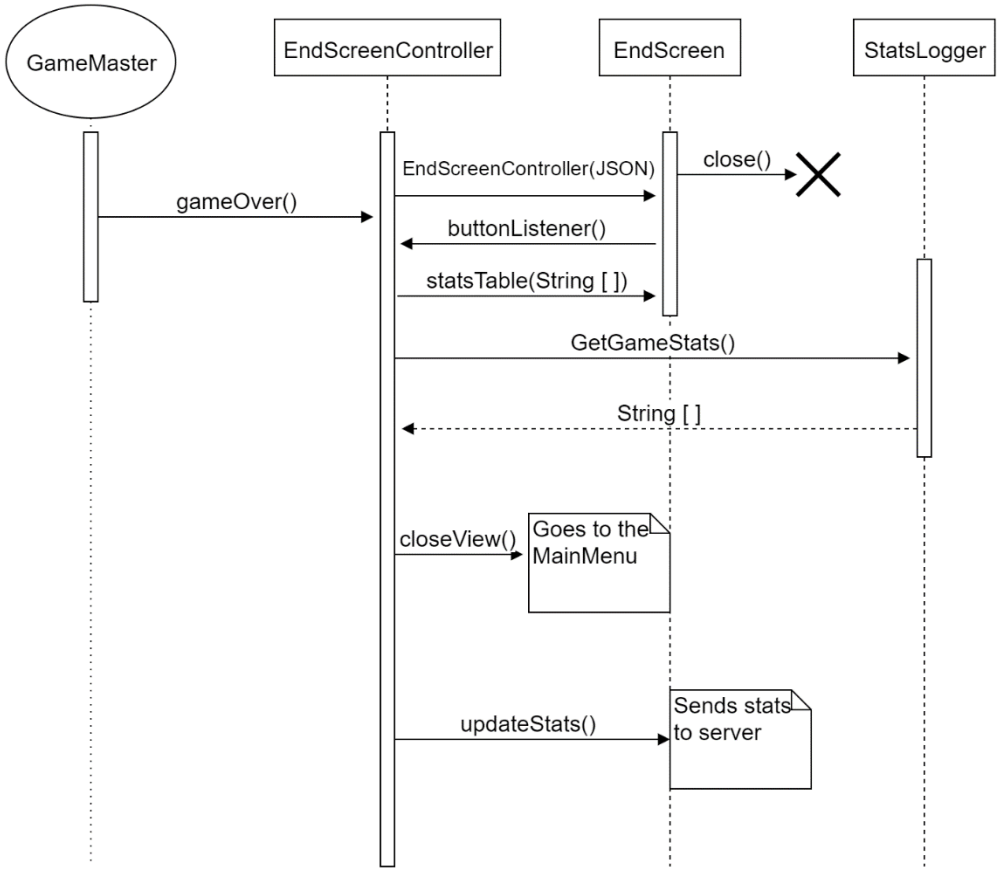
Interactions:

EndScreen interacts with the StatsLogger to display information in it's tables. All of the EndScreen's input is handled by the EndScreen's buttonListener.

Functions:

- EndScreen(string array):
  - Initializes the EndScreen with buttons and stats tables. Passes the string array to the statsTable( ) function which parses the strings and displays them in the stats tables.
- statsTable(string array): parses the strings array and populates the stats tables
- buttonListener( ): listens for any button down evens and signifies the appropriate controller function in the EndGameController

EndScreen Sequence Diagram:



DrawRobot Class:

The DrawRobot class is an abstract class that facilitates the redrawing of robots upon turning or movement. DrawRobot is extended by the three Robot draw classes (DrawScout, DrawTank, and DrawSniper) that provide sprite specific redrawing functions. The direction that the Robot is facing is determined by the Gang's colour. Because the colour's position is static, we can use the Gang's colour to determine what direction the Robots were originally facing. From there, only the number of turns need be recorded, and modulus division can be used to see where the robot should face.

Interactions:

The DrawRobot class is extended by DrawTank, DrawSniper, and DrawScout. It interacts with the GameScreen view to draw the Robot sprites on the GameBoard.

Functions:

- DrawRobot(coord, COLOUR, turns): abstract function for implementation in subclasses

DrawScout Class:

The DrawScout class is a subclass of DrawRobot that redraws the scout Robot sprite upon turning or movement. DrawScout draws the scout Robot sprite facing in the specified direction (in the event of a turn) or redraws the scout Robot sprite onto a new hex tile (in the event of a movement).

Interactions:

The DrawScout class interacts with the DrawRobot class, which in turn interacts with the GameScreen view. The GameScreen view calls the DrawScout functions when the redrawing of a scout Robot is required.

Functions:

- DrawRobot(coord, COLOUR, turns): draws a scout Robot sprite at the given coordinate, facing in the calculated direction. The direction the scout Robot sprite will face is determined by its position relative to the center at the beginning of the game, as well as the edge number currently pointed to by the Robot.

DrawSniper Class:

The DrawSniper class is a subclass of DrawRobot that redraws the sniper Robot sprite upon turning or movement. DrawSniper draws the sniper Robot sprite facing in the specified direction (in the event of a turn) or redraws the sniper Robot sprite onto a new hex tile (in the event of a movement).

Interactions:

The DrawSniper class interacts with the DrawRobot class, which in turn interacts with the GameScreen view. The GameScreen view calls the DrawSniper functions when the redrawing of a sniper Robot is required.

Functions:

- DrawRobot(coord, COLOUR, turns): draws a sniper Robot sprite at the given coordinate, facing in the calculated direction. The direction the sniper Robot sprite will face is determined by its position relative to the center at the beginning of the game, as well as the edge number currently pointed to by the Robot.

DrawTank Class:

The DrawTank class is a subclass of DrawRobot that redraws the tank Robot sprite upon turning or movement. DrawTank draws the tank Robot sprite facing in the specified direction (in the event of a turn) or redraws the tank Robot sprite onto a new hex tile (in the event of a movement).

Interactions:

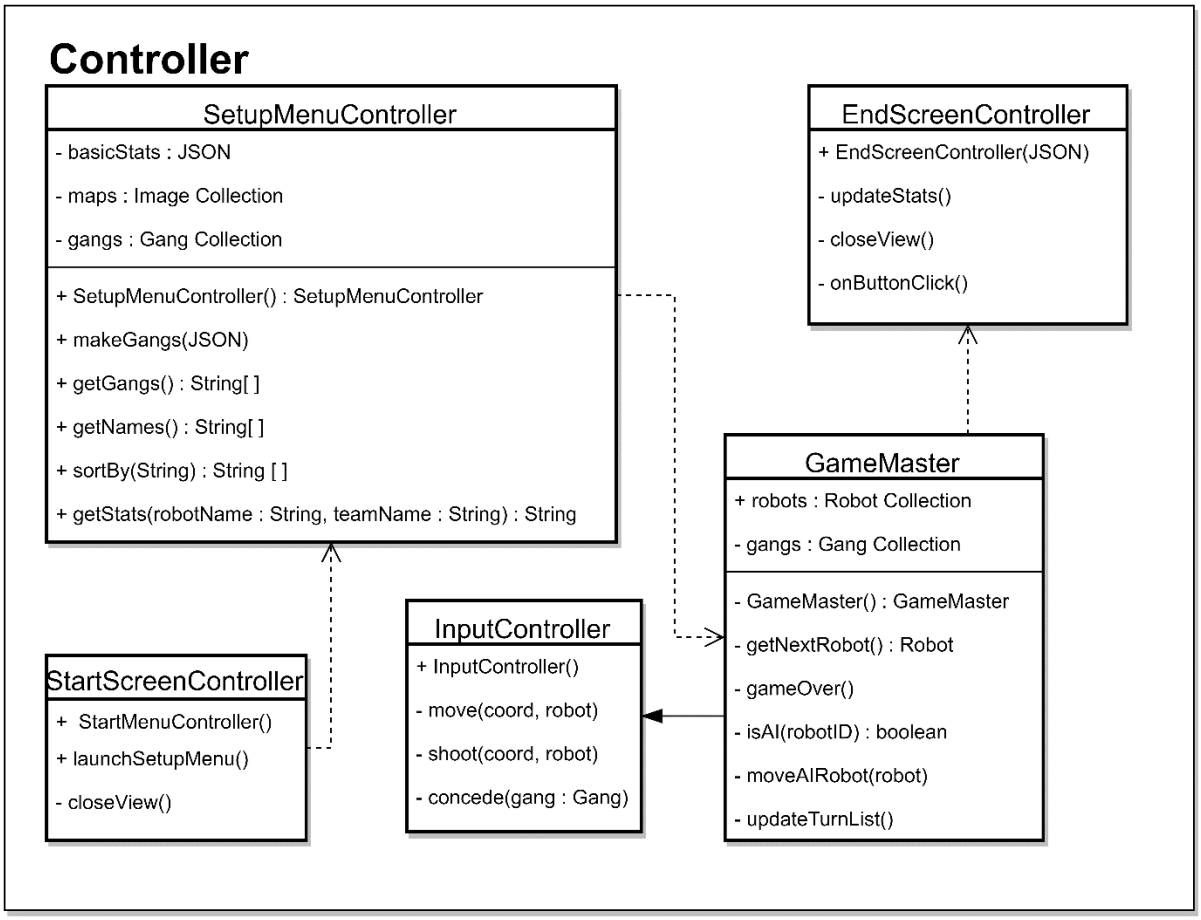
The DrawTank class interacts with the DrawRobot class, which in turn interacts with the GameScreen view. The GameScreen view calls the DrawTank functions when the redrawing of a tank Robot is required.

Functions:

- DrawRobot(coord, COLOUR, turns): draws a tank Robot sprite at the given coordinate, facing in the calculated direction. The direction the tank Robot sprite will face is determined by its position relative to the center at the beginning of the game, as well as the edge number currently pointed to by the Robot.

Controller:

Controller UML Diagram:



The above image is a detailed UML diagram of the Controller component of our system. It is responsible for operating on the data provided to it by the user and updating the Model accordingly. The Controller acts as an intermediary between the View component and the Model component. The Controller component of our system contains five classes: The StartScreenController, the SetUpMenuController, the InputController, the GameMaster controller, and the EndScreenController.

Controller Classes:

StartScreenController Class:

The StartScreenController is responsible for the control of the StartScreen view component. Each view component has its own controller component to reduce the size of the main controller (the GameMaster) and to reduce coupling.

Interactions:

Interacts with the StartScreen view by handling Button down events.

Functions:

- StartMenuController( ): constructs the StartScreenController and initializes the StartScreen view

- onButtonPress(Start button): closes the StartScreen view and instantiates the SetUpMenuController
- closeView( ): closes the StartScreen view

SetUpMenuController Class:

The SetUpMenuController is responsible for all changes made within the SetUpMenu view. Its primary purpose is to ensure that the gangs of robots are set up and instantiated correctly. From the SetUpMenu view, the user is able to select from the group of computer AI's designed by the various student software teams, each team denoted by teamID.

Interactions:

The SetUpMenuController interacts with the SetUpMenu view, with the Gang model to instantiate the Gangs of robots, with the Robot Librarian to select the robots for a given Gang, and with the GameMaster, which is instantiated by the SetUpMenuController. The SetUpMenuController itself is instantiated by the StartMenuController.

Variables:

- basicStatsJSON: holds the basic stats of all the robots in the Robot Librarian
- maps[ ]: the set of images for various styles of the game-board (depending on number of Gangs)
- gangs[ ]: hold the set of robots in a given Gang

Functions:

- SetUpMenuController( ): initializes the SetUpMenuController and downloads the basicStatsJSON file from the Robot Librarian
- getNamesInCriteria( ): returns an array of the robot names in the basicStatsJSON file that match the current "Created-by" criteria selected in the SetUpMenu view
- sortWins( ): returns an array of robots sorted by wins
- makeGangs(gangJSON) This function is passed a JSON object of all robots that were selected in the SetupMenuView. From there, it downloads all of the corresponding robots from the RobotLibrarian and assigns them to the appropriate gangs.
- getGangs( ): This function returns the list of gangs that were created in make Gangs. It is used to pass the gangs to the constructor of the GameMaster
- getStats(robotName, teamID): returns a string of stats pertaining to a given robot

GameMaster Class:

The GameMaster is the main controller for the system. It is responsible for managing and calling the necessary views required by the system. Those views, in turn, call their own controller classes. The GameMaster is also responsible for keeping track of whose turn it is and for detecting when the game is over. Although the GameMaster is the main controller and coordinator of the other controller components, our system splits the controller into smaller components to reduce coupling, increase cohesion, and to keep related functionality in a compact package.

Interactions:

The GameMaster interacts with the StatsLogger model by passing it to various controller components, with the GameBoard model by passing its data to the view components, and with the Gang class by acquiring a list of

Gangs which it uses to handle turn-order and pass Gang info to the view components. The GameMaster is also responsible for calling the Forth Interpreter when a Gang is being controlled by computer AI's. Lastly, it handles the instantiation of the PassControls view, the GameScreen view, and the EndScreen view.

Variables:

- gangs[ ]: a circular collection of robot gangs, used to track turn order amongst robot gangs
- robots[ ]: a collection of robots, used to track turn order amongst individual robots

Functions:

- GameMaster(gangs[ ]): takes in the Gangs created in the SetUpMenuController and constructs a GameMaster object
- getNextRobot( ): determines which robot and which team is up next in the turn-order
- gameOver( ): determines when the game has ended
- getRobotInfo (coord): returns all information pertaining to that coordinate
- isAI(robotID): determines if a robot is being controlled by a computer AI
- moveAIRobot(robotID): calls computer AI move function
- updateTurnList( ): removes dead robots from turn-order list

InputController Class:

The InputController class is another controller component, specifically responsible for handling input from the user. User generated events are listened for in the GameScreen view class and are translated into function calls in the InputController.

Interactions:

The InputController interacts with the Robot class by calling its move, shoot, and concede functions. It also interacts with the GameScreen class which listens for events. Interacts with the GameMaster to track whose turn it is.

Functions:

- InputController( ): instantiates a new InputController object
- move(coord, robotID): calls the robot's move function with a coordinate parameter
- shoot(coord, robotID): calls the robot's shoot function with a coordinate parameter
- concede(gangID): kills all of the player's robots and removes them from the game
- endTurn( ): ends the current players turn whether they have moves left or not

EndScreen Controller:

The EndScreenController is responsible for the control of the EndScreen view which is displayed at the end of a game. It is instantiated by the GameMaster when only one gang of robots has one or more robots remaining. It can instantiate the StartScreen view if the Back button is pressed, or it can close the game window if the Quit button is pressed.

Interactions:

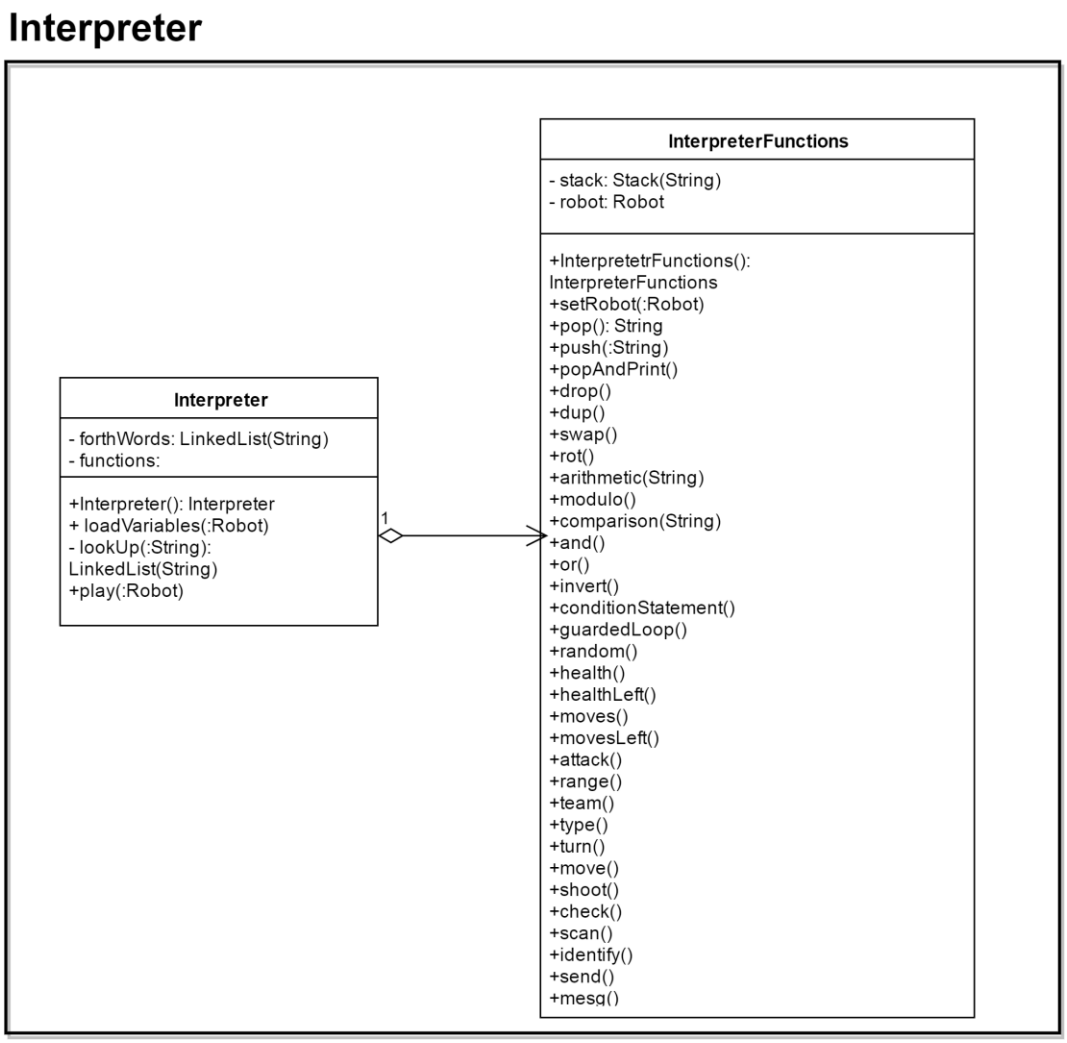
The EndScreenController is instantiated by the GameMaster. It accesses the StatsLogger and can both close the game window and re-instantiate the StartScreen controller.

Functions:

- EndScreenController(JSON object): The constructor of the EndScreenController. Instantiates the EndScreen and passes it the game statistics of the current game
- udapteStats( ): Passes the statistics to the RobotLibrarian
- closeView( ): closes the game window
- onButtonClick(Back/Close button): Begins by updating all game statistics and then closes game window or instantiates StartScreenController depending on button pressed

Interpreter Component:

Interpreter UML Diagram:



The above UML diagram describes the interpreter component of our system. It is responsible for interpreting the Robot's Forth code and altering the system accordingly. The interpreter acts as an intermediary between the GameMaster and the Model component. The Interpreter class handles the Forth code, and the InterpreterFunctions class acts as an interface to the model.

Interpreter Classes:

Interpreter Class:

The Interpreter class allows the program to run functionality based on the logic within a body of Forth code. These Forth functions are responsible for controlling the computer AI Robots on their turn. The interpreter works by traversing the list of forth code and running equivalent functions from the InterpreterFunctions class based on what is found. The purpose of the interpreter is to allow the system to run code as specified by the RoboSport370 Language document. This provides the functionality which is necessary when sharing a code standard among many different systems, meaning that any robots that are created by this program can be used by all other RoboSport370 applications.

Interactions:

The interpreter is meant to act as an intermediary between the controller and computer AI controlled Robots. It will be Instantiated in the GameMaster class before the game begins. The GameMaster will then call on it to initialize the local variable storage within each Robot. Once the game begins, the Interpreter will be called by GameMaster each time it is a computer AI's turn. The Interpreter will then call a sequence of InterpreterFunction functions to act out robot's turn. There are also cases where the Interpreter will have to access the Robot's JSON object to access its stored variables and functions.

Variables:

- forthWords: holds a LinkedList of Strings which represent individual words of Forth code.
- functions: holds an instance of the InterpreterFunctions class.

Functions

- Interpreter(): Initializes the above variables.
- loadVariables(Robot): Stores the needed local functions and variables within the Robot's JSON object.
- LinkedList<string> lookUp(String): Looks up the corresponding function or variable in the Robots JSON object. If it is setting the variable, pop from the InterpreterFunctions stack and set the value in the JSON object. If it's getting the variable or function definition, return a LinkedList where each element is one word of the found definition.
- void play(Robot): calls functions.setRobot(Robot) to let it know which Robot we are currently working on. It will call lookUp and adds its output to the forthWords list, at which point it will begin traversing the list.

Traversing the list Example:

"2 dup 1 shoot! "
will be translated to the following sequence of IntepreterFunctions function calls:
push(2), dup(), push(1), shoot()
These functions all are done using postfix evaluation on a stack.

Further information on the forth code can be found in the RoboSport370 Language document.

If a word is encountered that is not recognized, lookUp() must be called with that word given as an arguments. The output List of parse should be added to the forthWords list in place of the variable.

InterpreterFunctions:

This class acts as an interface to the system for the Forth interpreter. It provides a list of functions that directly translate from the Forth language functions, allowing the Interpreter to simply call whichever one matches each word in a body of Forth text. These functions will act on a stack, and will call the internal functions of the Robot class.

Interactions:

When the Interpreter traverses a list of Forth code, it will call these InterpreterFunctions
These functions will then call the robot class to act out the desired functionalty.
Because the Robot.move() function requires coordinates to be passed in as an argument, it is necessary to call on the GameBoard's getRobotCoord function.

Variables:

- stack: a stack which holds all variables for the functions to use.
- Robot: a robot whose functions will be called

Functions:

- InterpreterFunctions() Initialize the above variables.
- setRobot(Robot): sets the robot whose functions will be called.

The following functions mirror those in the RoboSport370 Language document unless additional specification is given:

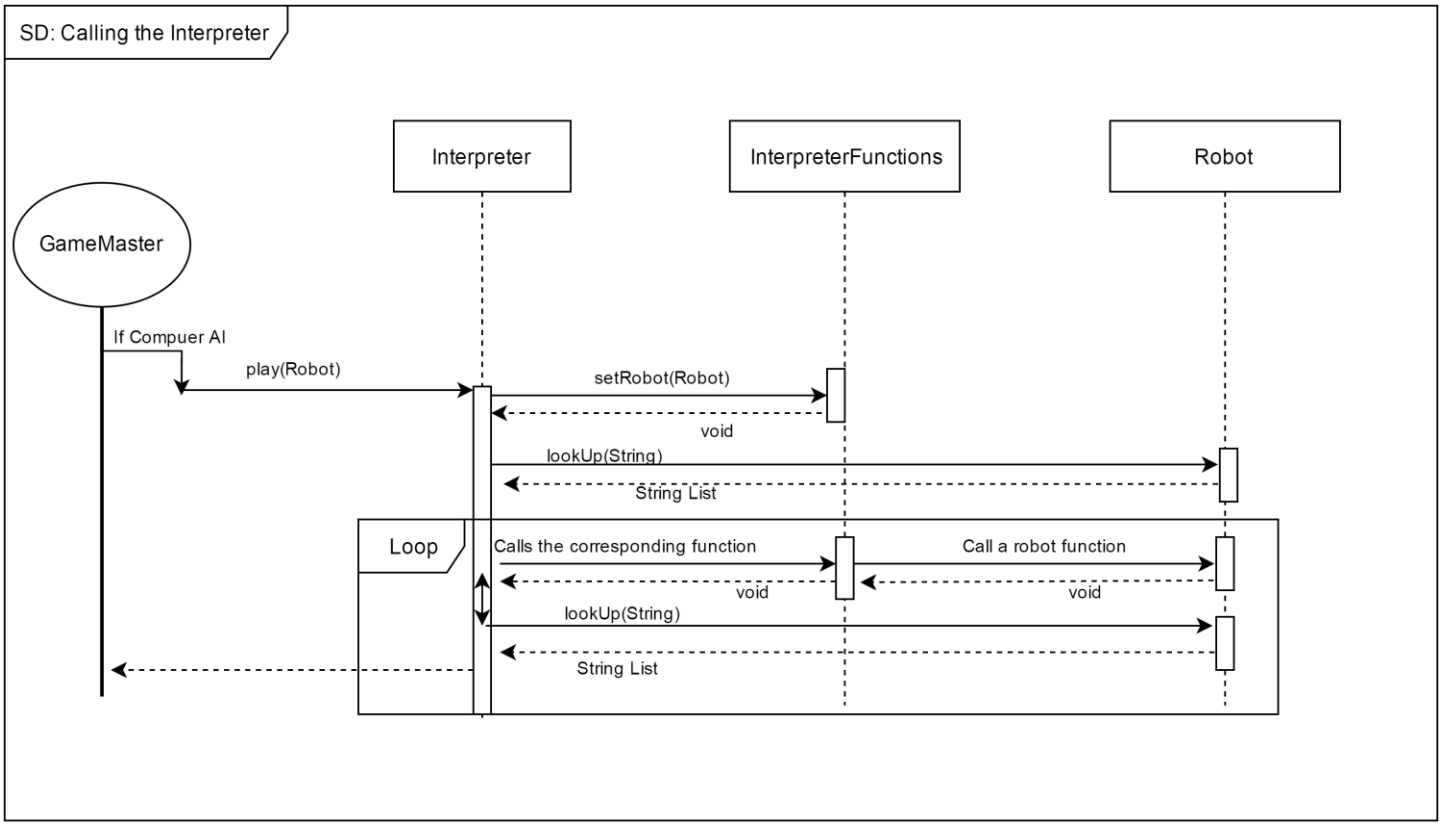
- String pop(): returns the value off of the top of the stack.
- push(String): pushes a Word onto the stack
- popAndPrint(): pops the stack and prints the result.
- drop()
- dup()
- swap()
- rot()
- arithmetic()
- modulo()
- comparison()
- and()
- or()
- invert()
- conditionStatement()
- guardedLoop()
- countedLoop()
- random()

The following functions will call the Robot functions using variables popped from the stack, and pushing their return values to the stack.

- health()
- healthLeft()
- moves()
- movesLeft()
- attack()

- range()
- team()
- type()
- turn()
- move(): This function will need to call getRobotCoord from the GameModel with the robotID.
- shoot()
- check()
- scan ()
- identify()
- send ()
- mesg()
- recv()

Interpreter Sequence Diagram:



The above sequence diagram begins when the GameMaster class calls the Interpreter's play(Robot) function. The Interpreter will call setRobot() to tell the InterpreterFunctions which Robot should be modified. It will then call the lookup function to load the initial Forth code for the Robot's play function. The Interpreter will then perform a sequence of actions in a loop. It will move through the body of code and call the corresponding robot functions. If the current word is not defined as an InterpreterFunctions function, it will use lookUp() to find the Robot's definition for that word. That definition will be returned to the Interpreter so that it can process it before returning to the function that called for the lookup. The Interpreter's play() function will return once all of the Forth code has been interpreted and the GameMaster will regain control of the program.

Changes made from Requirements Document:

The changes that have occurred between the Requirements Document and the Design Document are as follows:

- The StartScreen was described in the Requirements Document as displaying previously obtained high scores. This feature has been altered and moved to the EndGame screen so that the user may view the updated game statistics post game.
- The transition from the StartScreen to the SetUpMenu has been altered to exclude the pop up window that queried the number of players. Instead, our system will now include this functionality in the SetUpMenu. Our new design simply requires that the user select the Gang colours that they would like to play with. Any colours not selected are simply treated as players that will not be included in the game. This affects the control flow of the Start button on the StartScreen.
- As a result of the above change, the SetUpMenu has been altered. The map of the game is now included in the SetUpMenu view, and will show the formation of the board as more players are added to a game. The map will update itself any time the user adds or removes a player from the SetUpMenu selection screen.
- In the GameScreen, functionality to display game statistics on the screen at all times has been removed. Now, statistics will only be displayed if the user hovers over a hex tile that is in their range of vision. On the mouse hover event, the statistics of all Robots currently residing on the tile will be displayed.
- A Concede button was added to the GameScreen view so that a user has the option to forfeit a game.
- An EndScreen view was added where the end-game statistics will be displayed. This has changed from the Requirements Document, where the end-game statistics were laid over the GameScreen. The EndScreen will contain a Back button which returns the user to the StartScreen, and a Quit button, which will close the game window.
- The actors originally identified for the system were the Robot Librarian, the Computer Player, and the Human Player. We have since identified a fourth actor to our system; the GameMaster. The GameMaster is responsible for setting up the game environment and instantiating the game models and the views. It is also responsible for communicating with the server. The roles of the GameMaster are distinct enough that it requires its own position as an actor of the system.