Programmer Documentation

Software Engineering Team: Arianne Butler, Michael Graham,

Samuel Horovatin, Kristof Mercier, and Chris Mykota-Reid

Contents

[System Overview 3](#_Toc469081383)

[Model 3](#_Toc469081384)

[The Stats Logger 3](#_Toc469081385)

[View 3](#_Toc469081386)

[Controller 3](#_Toc469081387)

[Interpreter 4](#_Toc469081388)

[Architecture Overview 4](#_Toc469081389)

[System Architecture Diagram: 4](#_Toc469081390)

[JavaDocs 4](#_Toc469081391)

[Testing 5](#_Toc469081392)

[Delta Change Log 5](#_Toc469081393)

[Model 5](#_Toc469081394)

[View 6](#_Toc469081395)

[Controller 6](#_Toc469081396)

[Interpreter 6](#_Toc469081397)

[Implementation Details 6](#_Toc469081398)

[Running the Program 6](#_Toc469081399)

[Languages and Environment 6](#_Toc469081400)

[System libraries 6](#_Toc469081401)

# System Overview

## Model

### The Stats Logger

The Stats Logger contains functionality to record and update game statistics corresponding to each robot. It accomplishes this by using a hash-map of Robot ID’s mapped to an inner class of Robot Stats, and creating a JSON from them. The Robot Stats class contains fields for all statistics recorded throughout the game. The Stats Logger is called throughout the game whenever statistics need to be recorded or updated. For example, in the GameMaster class, the function gameOver() calls the Stats Logger to update wins, losses, and death and survival statistics. At the end of the game, the Stats Logger produces a JSON containing robot stats for all robots in the game. This JSON is then uploaded to the Robot Librarian for record keeping in the online database.

***Insert Mike’s paragraph’s about the Model classes***

* ***Decide which classes, if any, require their own paragraph, and which can be combined***
* ***For each paragraph:***
  + ***Include technical description of the class or component***
  + ***Include the major interfaces between the Model and other components of the system***

## View

The View is comprised of six main classes and their corresponding views: The Start Screen, the Set-Up Menu Screen, the Set-Up Menu Pop-Up, the Game Screen, the Pass the Controller Screen, and the End Screen. These classes handle the user input by the use of action listeners. The View classes’ handle their own input with the use of action listeners. The only parts of the View that interface with the controller component are the Set Up Menu Screen and the Set Up Menu Pop Up. The set up functionality is handled by the Set Up Menu Controller, which parses a JSON string of robots and statistics, populates both the Software Engineering Team and Robot combo boxes, and creates Gangs of Robots based on the user’s selections.

## Controller

The controller is comprised of the Set Up Menu Controller and the Game Master.

The Set Up Menu controller is responsible for handling the user input selection during game set up. It begins by parsing a JSON file of Robot statistics and using it to populate three combo boxes of Software Development Teams, one for each Robot type; Scout, Sniper, and Tank. Once the user selects from a team, the JSON will be parsed again to search for the Robots created by that development team. The found robots will populate the corresponding Scout, Sniper, and Tank combo boxes. Once the user has selected three robots for every team in play, they can click Start. The Start button calls a function in the controller to set up an array of gangs, each gang with its own array of three robots.

The Game Master is responsible for handling the turn order, for providing a list of Gangs in the game to other classes, and for providing the functionality to search for a Gang by its colour. The Game Master’s internal turn-handling functions are accessed by the Game class. When a turn is a complete, the Game Master updates the list of Robots in the turn queue, checks if the game is over, and repeats.

## Interpreter

The interpreter is made up of two classes: Interpreter and Interpreter Functions. These classes allow a piece of Forth code from a properly formatted JSON file to be run for the AI logic. The Interpreter class parses and handles the code through its execution, and the IntepreterFunctions class acts as an intermediary between the code and the rest of the system. The interpreter loads its translation dictionary upon construction from the file words.txt. To run a Robot’s Forth code on their turn, the play( ) function is called with the desired robot as its parameter. This will cause the Interpreter to parse and move through the Forth code’s Linked List, calling the corresponding function for each word. The parsing step is done in lookup( ), where all function calls are removed so that there are only variables, literals, and words from the Forth language.

# Architecture Overview

The architecture for the system is a Layered Architecture that uses a Model-View-Controller (MVC) design pattern and independent Aspect-Oriented Components. The MVC design pattern was chosen to force developers to separate the concerns of each individual class, resulting in a reduction of coupling in the system. It should also serve to reduce the amount of time spent on system changes. 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.

## System Architecture Diagram:

# JavaDocs

***Someone please figure out how to generate these and link them to this dank document***

# Testing

Our team planned extensive testing for the Robo Sport system (as outlined in our Testing Plan), including unit testing, integration testing, and database/network testing. During implementation, we came close to fully achieving our unit testing goals, while integration and database testing were neglected due to time constraints. Our system includes extensive unit testing for all functions in the Model, Controller, and Interpreter components, while the View was tested with simple user input protocols (as we ran out of time to implement code hooks as described in our Testing Plan).

Due to our limited time frame, the only integration testing done on our system was on the Interpreter. We decided to prioritize the Interpreter due to its complexity and its strongly interconnected components. If there had been more time, later builds of our system would include complete unit testing, full integration testing, and implementation of code hooks to test the user interface. Lastly, due to an inability to finish implementation of multiplayer functionality, we were unable to implement testing for those particular components.

Test coverage is as follows:

Comprehensive Testing:

* GameMaster
* Interpreter
* Gameboard
* Gang
* Robot
* StatsLogger
* StartScreen View
* SetUpMenu View
* SetUpMenuPopUp View
* PassTheController View

Partial Testing:

* InterpreterFunctions
* GameScreen
* EndScreen

# Delta Change Log

## Model

* Robot Class:
  + added setMovesRemaining( ): refreshes the robots moves
  + added setCanShoot( ): changes the robots canShoot variable to true
  + getMovement( ): returns the max distance a robot can move (if it has moves remaining)
  + - initSprites( ): loads the corresponding image for each robot
  + - getStartingCoord(int direction): gets the coordinate the current robot starts at
  + - getColour( ): gives the colour of the robot
  + - getDirection( ): gives the direction the robot is facing
  + - getRange( ):gets the range of the robot
  + - receiveMsgTest( ): a function for testing messaging
  + - mailFull( ): returns true if mailbox is full, false otherwise
  + - getInbox( ): returns the inbox of the robot
  + - getSprite( ): returns the sprite for the robot
  + - getCurrentRotation( ): gets the robots current rotation in radians
* Coordinate:
  + - add( ): adds two coordinates together and returns the resultant coordinate
  + - getHex( ): gets the hexagon that corresponds to the coordinate
* ***Insert Mike’s changes to the Model classes since Construction Experience Document (point form)***
* ***Insert Chris’s changes to the Stats Logger since Construction Experience Document (point form)***

## View

* Changed the way the view interfaces with the controller so that the view communicates with the Set Up Menu Controller only

## Controller

* Removed classes Input Controller, Start Screen Controller, and End Screen Controller
* ***Insert Chris’s changes to Game Master since Construction Experience Document (point form)***

## Interpreter

* Interpreter Class:
  + added getter and setter for currentElement
* InterpreterFunctions Class:
  + added leave( ): leaves the current loop
  + added doNothing( ): Does nothing. This is needed because each forth word calls a function, and the word “then” has no functionality, so it must call a function which does nothing
  + split loop( ) into guardedLoop( ) and countedLoop( ): to start each kind of loop
  + added iterator to get the value of the current iteration of the loop
  + added until( ): to hande the end of a guarded loop
  + added loopEnd( ): to handle the end of a counted loop
  + added the struct loopStruct: holds the necessary information for a loop declaration

# Implementation Details

## Running the Program

To run, navigate to the folder where the .jar is located inside your command line (or terminal) and type:

java -jar robo-sport.jar

## Languages and Environment

The system was developed in the IntelliJ Development Environment using GitLab version control. It was implemented in Java, Aspect J, and Forth.

## System libraries

GSON, java.util, ***any additional system libraries***