Programmer Documentation

Software Engineering Team: Arianne Butler, Michael Graham, Samuel Horovatin, Kristof Mercier, and Chris Mykota-Reid

Contents

System Overview

Model

The model is made up of five classes and two enumerations; all of them working together to store and manipulate data pertaining to the game. The classes in the model are the StatsLogger, Robot, Gang, GameBoard, and Coordinate. Nearly all of the programs logic is handled inside these classes. In using them, the programmer can easily create new Robots, Gangs, Coordinates (also known as hex tiles) and GameBoards. The three most significant classes in the Model component are the StatsLogger, Robot and GameBoard. Further detail for these classes is provided below.

The StatsLogger

The StatsLogger contains functionality to record and update game statistics corresponding to each robot. It accomplishes this by using a hash-map of RobotID's mapped to an inner class of RobotStats, and creating a JSON from them. The RobotStats class contains fields for all statistics recorded throughout the game. The StatsLogger is called throughout the game whenever statistics need to be recorded or updated. For example, in the GameMaster class, the function gameOver() calls the StatsLogger to update wins, losses, death and survival statistics. At the end of the game, the StatsLogger 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.

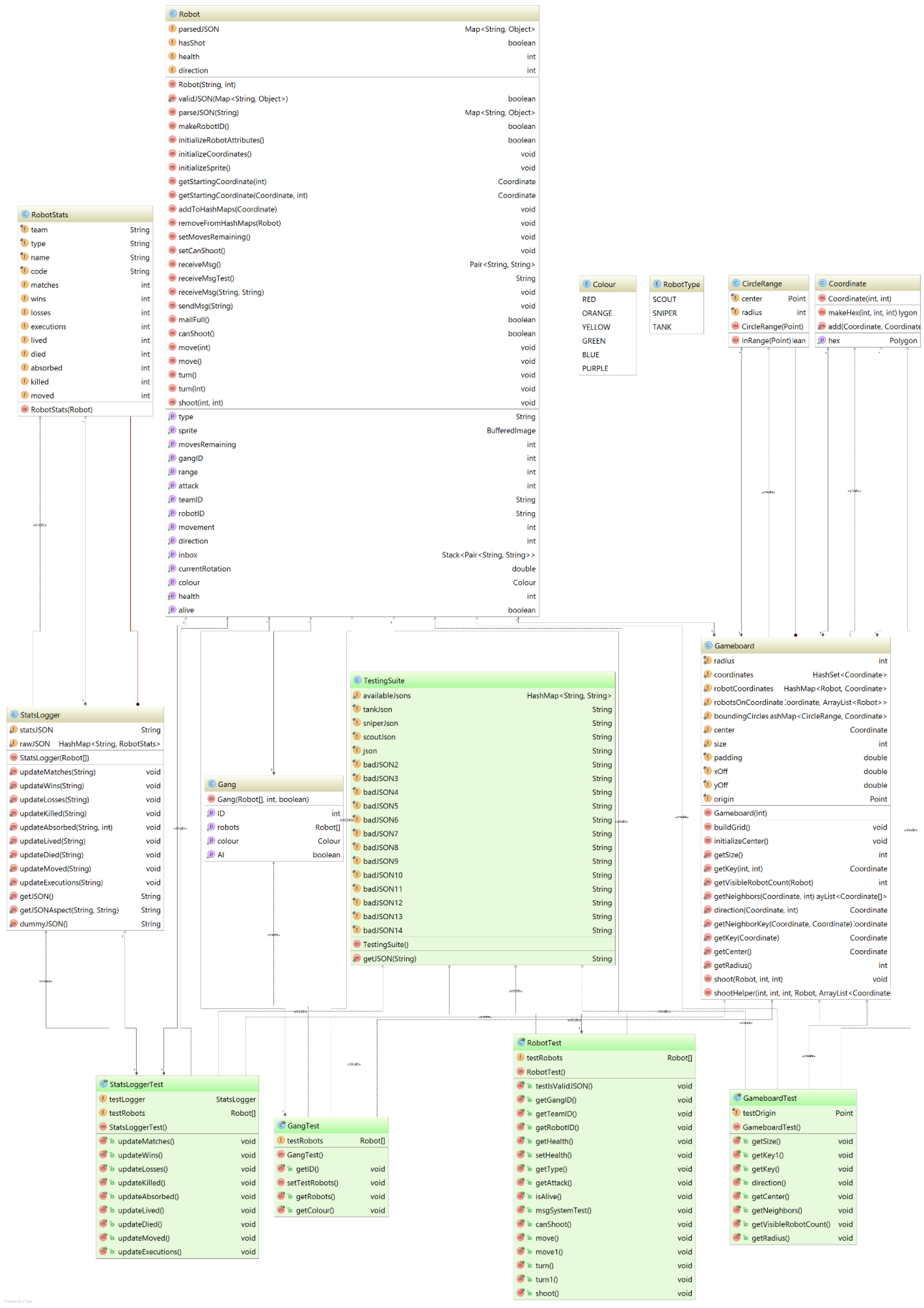
Robot

The Robot is responsible for storing all information pertaining to a given Robot. It also contains the methods that are called by the Game class to perform actions on individual Robots (such as moving and shooting). Upon its instantiation, the Robot JSON is parsed by GSON, and all of the required information is initialized. Based on the Robot's Gang's ID, it also positions the Robot on the GameBoard, and orients it in the correct direction. This makes creation of a new Game quick and simple, with most of the hard work done upon instantiation.

Gameboard

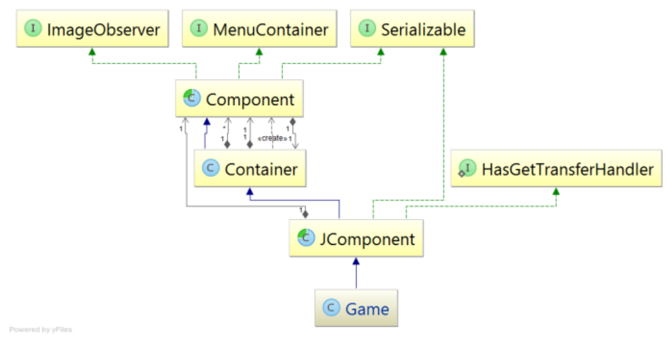
The GameBoard oversees all information pertaining to itself, and is also responsible for the creation of the hexagonal game map. It contains information regarding the different hex tiles, Robot positons, etc. The GameBoard can be queried for the neighbors of a certain Robot or Coordinate, and it will return those neighbors based on the Robot or Coordinate's relative position. This is beneficial because it allows the Robots to easily receive information pertaining to their state, without having to do any complex translations. The GameBoard also handles invalid queries efficiently. It knows that there is a specific range that a hex tile may cover, so if a programmer or user accidentally requests information on a given hex tile, but they are actually a couple of units off, the GameBoard will accommodate the error and give them the correct tile.

Model Component UML Diagram

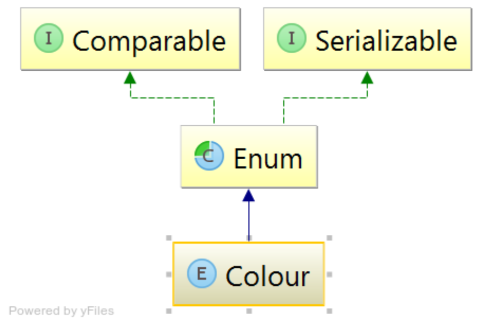


Model Class Dependency Diagrams

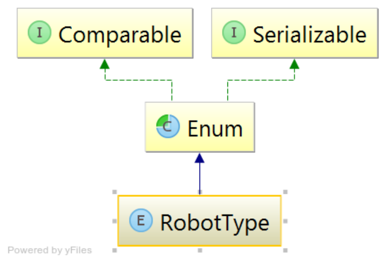
Game:



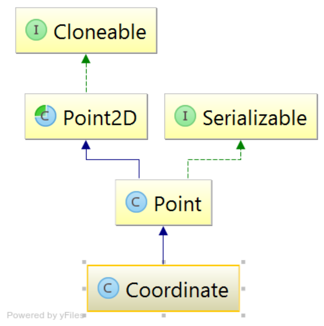
Colour:



Robot:



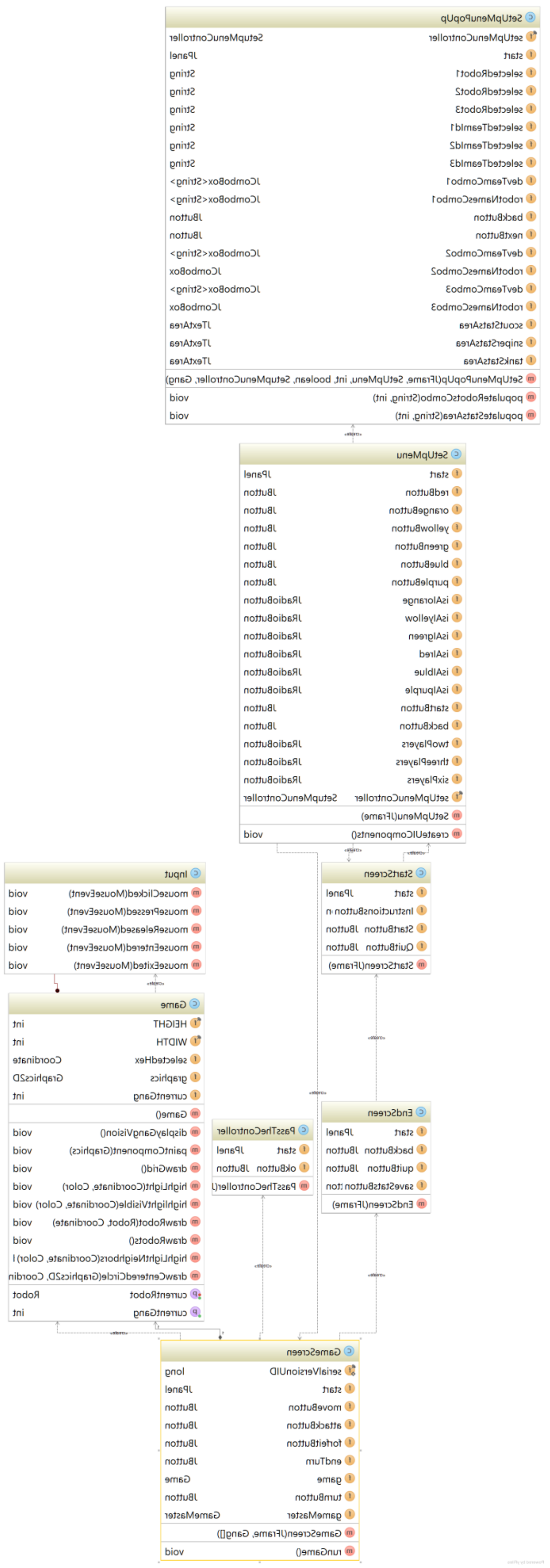
Coordinate:



View

The View is comprised of six main classes and their corresponding views: The StartScreen, the SetUpMenuScreen, the SetUpMenuPopUp, the GameScreen, the PasstheControllerScreen, and the EndScreen. These classes handle the user input by the use of action listeners. The only parts of the View that interface with the controller component are the SetUpMenuScreen and the SetUpMenuPopUp. The set up functionality is handled by the SetUpMenuController, which is launched as soon as the user selects one of the team colour buttons. It begins by parsing a JSON string of robots and statistics in order to populate the Software Engineering Team combo box. Once the user selects a Software Engineering Team, the SetUpMenuController parses the JSON again to populate the Robots combo box with all Robots created by the selected team. When the user selects a Robot, the JSON is parsed one last time to fill the stats area with the selected Robot's individual statistics. When the user has selected all of the teams that will participate in the game and clicks Start, the Set Up Menu Controller creates a global array of Gangs, each with its own array of Robots based on the user's selections.

View Component UML Diagram



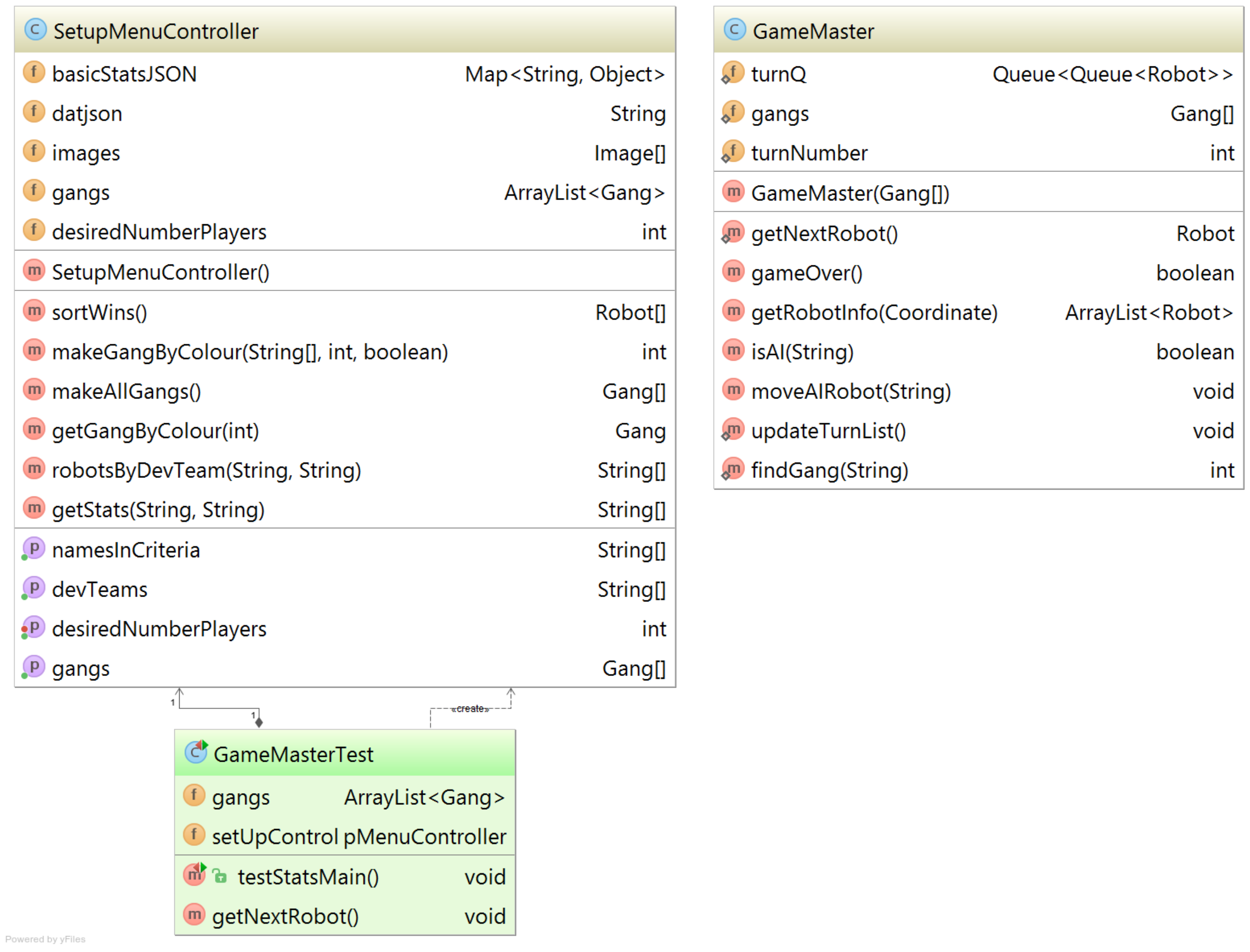
Controller

The controller is comprised of the SetUpMenuController and the GameMaster.

The SetUpMenuController is responsible for handling 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 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 GameMaster 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 GameMaster's internal turn-handling functions are accessed by the Game class. When a turn is a complete, the GameMaster updates the list of Robots in the turn queue, checks if the game is over, and repeats.

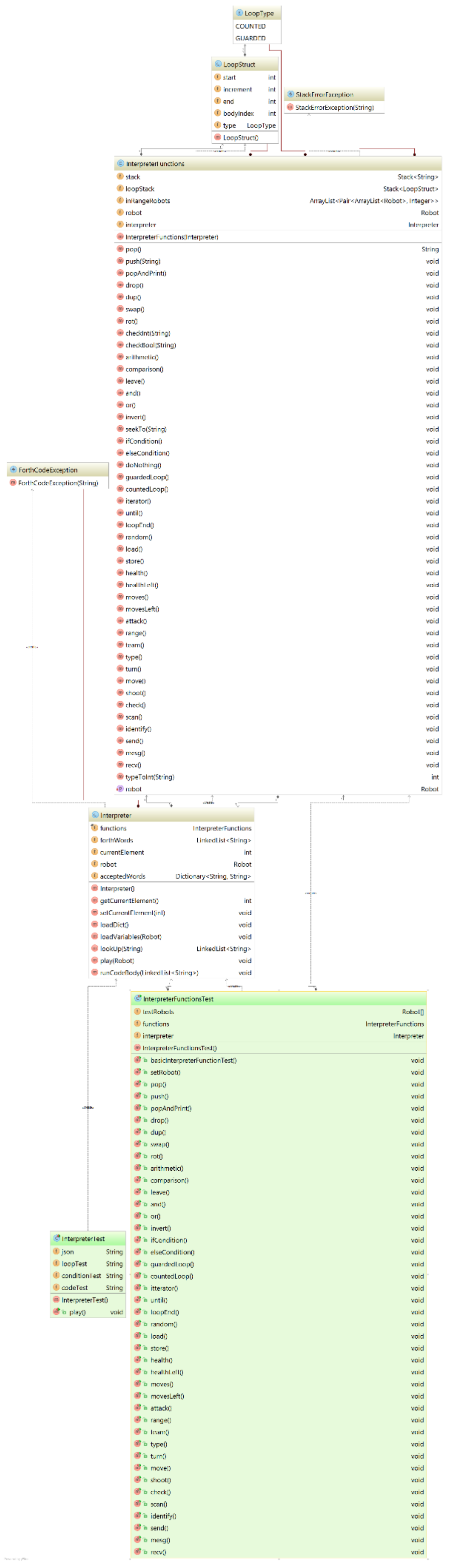
Controller Component UML Diagram



Interpreter

The Interpreter is made up of two classes: Interpreter and InterpreterFunctions. 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 codes 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 remaining.

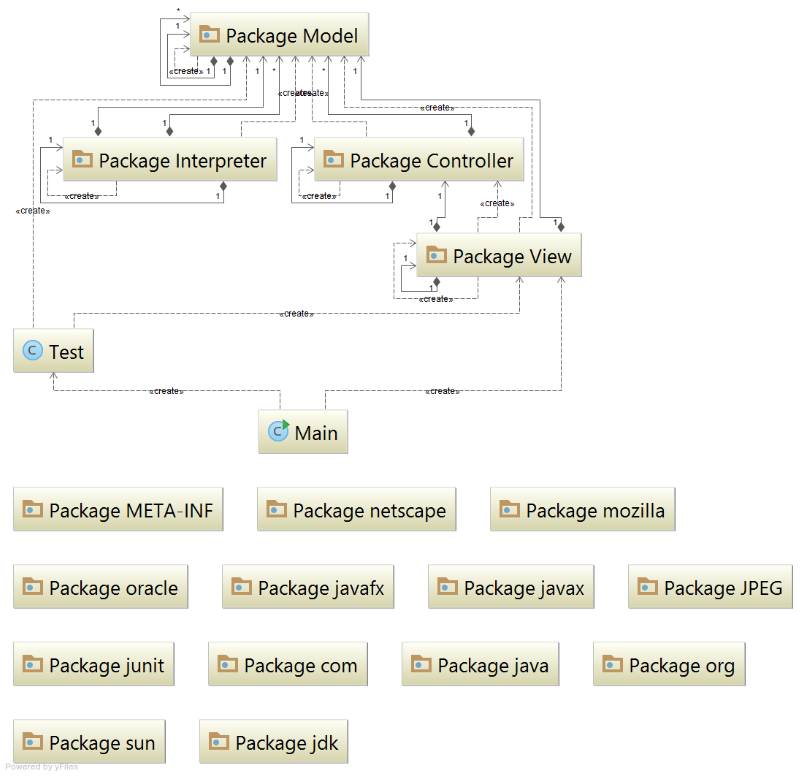
Interpreter UML Diagram:



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:



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:
- o GameMaster
- o Interpreter
- o Gameboard
- o Gang
- o Robot
- o StatsLogger
- o StartScreen View
- o SetUpMenu View
- o SetUpMenuPopUp View
- o PassTheController View

Partial Testing:
- o InterpreterFunctions
- o GameScreen
- o EndScreen

Delta Change Log

Model

- Robot Class:
  - o added setMovesRemaining( ): refreshes the robots moves
  - o added setCanShoot( ): changes the robots canShoot variable to true
  - o getMovement( ): returns the max distance a robot can move (if it has moves remaining)
  - o - initSprites( ): loads the corresponding image for each robot
  - o - getStartingCoord(int direction): gets the coordinate the current robot starts at
  - o - getColour( ): gives the colour of the robot
  - o - getDirection( ): gives the direction the robot is facing
  - o - getRange( ):gets the range of the robot
  - o - receiveMsgTest( ): a function for testing messaging
  - o - mailFull( ): returns true if mailbox is full, false otherwise
  - o - getInbox( ): returns the inbox of the robot
  - o - getSprite( ): returns the sprite for the robot
  - o - getCurrentRotation( ): gets the robots current rotation in radians
- Coordinate Class:
  - o - add( ): adds two coordinates together and returns the resultant coordinate
  - o - getHex( ): gets the hexagon that corresponds to the coordinate
- StatsLogger Class:
  - o - getJson( ): returns the JSON
  - o - getJSONAspect(String robotID, String aspect): returns a particular aspect for the robot, defined by robotID and the given aspect
- Gameboard
  - o Changed getKeyFromRange(Coordinate c) to getKey(Coordinate c)

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
- GameMaster:
  - Variables:
    - int turnNumber: the number of turns taken in the current round
    - int turnLength: the number of turns per round
  - Functions:
    - findGang(String robotID): finds the index for a gang containing the robot described by robotID

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

JavaDocs

The JavaDocs for RoboSport by Group D2 are available in the zip folder labeled "RoboSport-D2-Javadocs.zip". To view them, unzip the folder and open Index.html in a browser.

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
```

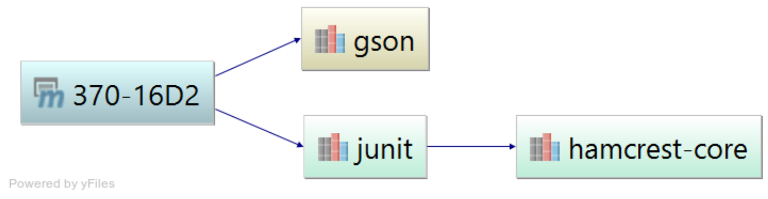
Languages and Environment

The system was developed in the IntelliJ Development Environment using GitLab version control. It was implemented in Java, and Forth. Dependency management tools like Maven were used to ensure that all developer's environments had the same versions of a given dependency. Junit was used extensively in unit testing for all of our tested classes.

System libraries

GSON, java.util, Junit, JavaFX, Javax.

System Dependency Diagram


370-16D2
gson
junit
hamcrest-core
Powered by yFiles