RoboSport 370

Design Document

https://drive.google.com/file/d/0BxYBewaR90TALUotcVZjLU5wVG8/view?usp=sharing

Group C3

Corey Hickson - crh208

Daniel Sanche - drs801

Landon Tetreault - lzt639

Levi Paradis - ldp343

Matthew Wind - mdw466

October 30th, 2015

I. Introduction

Within this document we have described in detail how we will implement the RoboSport370 project. First, we discussed the different types of architectures to design our program around. Following the discussion of the architectures we used a matrix diagram to decide which architectures were the most applicable to the project.

II. Requirement Revisions

For this section, we will discuss the various changes made to our requirements document since the design phase. These changes were made to better suit our design.

A. Scope:

We went back to the scope of the requirements documents and made a couple of changes. We started by changing the must have category. We changed the line “be able to run simulations with robots designed by players. This involves writing an interpreter to interface between the robots and the simulation.” to “be able to run simulations with robots designed by players and stored in a repository using JSON. This involves writing an interpreter to interface between the robots and the simulation.” This was done to set a more specific task that the interpreter would have to perform.

We changed the should have category as well. The line “and have a debug/testing simulation in which we can view a match at different speeds, including rewind, fast forward, and viewing stats about each robot.” included some secondary actions that should be moved down in the scope, as they are less important than others. We changed the line to “and have a debug/testing simulation in which we can view a match at different speeds, fast forward, and viewing stats about each robot.” The rewind implementation was moved down to the if time permits category of the scope.

B. Actors:

The actors category for our requirements document did not need changing. It already identified and included the points of external interaction on our system, and did not require changes to their descriptions.

C. Use Cases:

The use cases did not need changing either. It thoroughly defines every use case that our actors want to perform on our system. The robot is fully covered by the use cases load into simulator, execute turn, be destroyed. The spectator is fully covered by the use cases set up game and its sub cases, watch match through GUI, run multi-match simulation, and view results of match. The tester is covered through the debug mode use case, and all of its subcases.

D. Activity Diagrams:

E. Storyboards:
F. explain why we didn't make changes and why it was right to begin with JSON & JSON Library addition (standards meeting).
G.

III. Architecture Details

For this section we will discuss our architecture decision process which we used to reach a conclusion on which architectures will apply to our design.

A. Decision Matrix:

Our group used a decision matrix (seen below) to determine which architectures would apply to our project. For the decision matrix we chose a number of attributes as seen below. The primary attributes included applicability and collaborability; these attributes were chosen because of the nature of the project (ie. a group project to develop a game with a GUI). We also looked at modularity, ease of implementation, adaptability, usability, and testability as they were all important attributes for our architectures to properly apply good design principles. Lastly, security was considered, although lowly, due to the non-sensitive nature of the project.

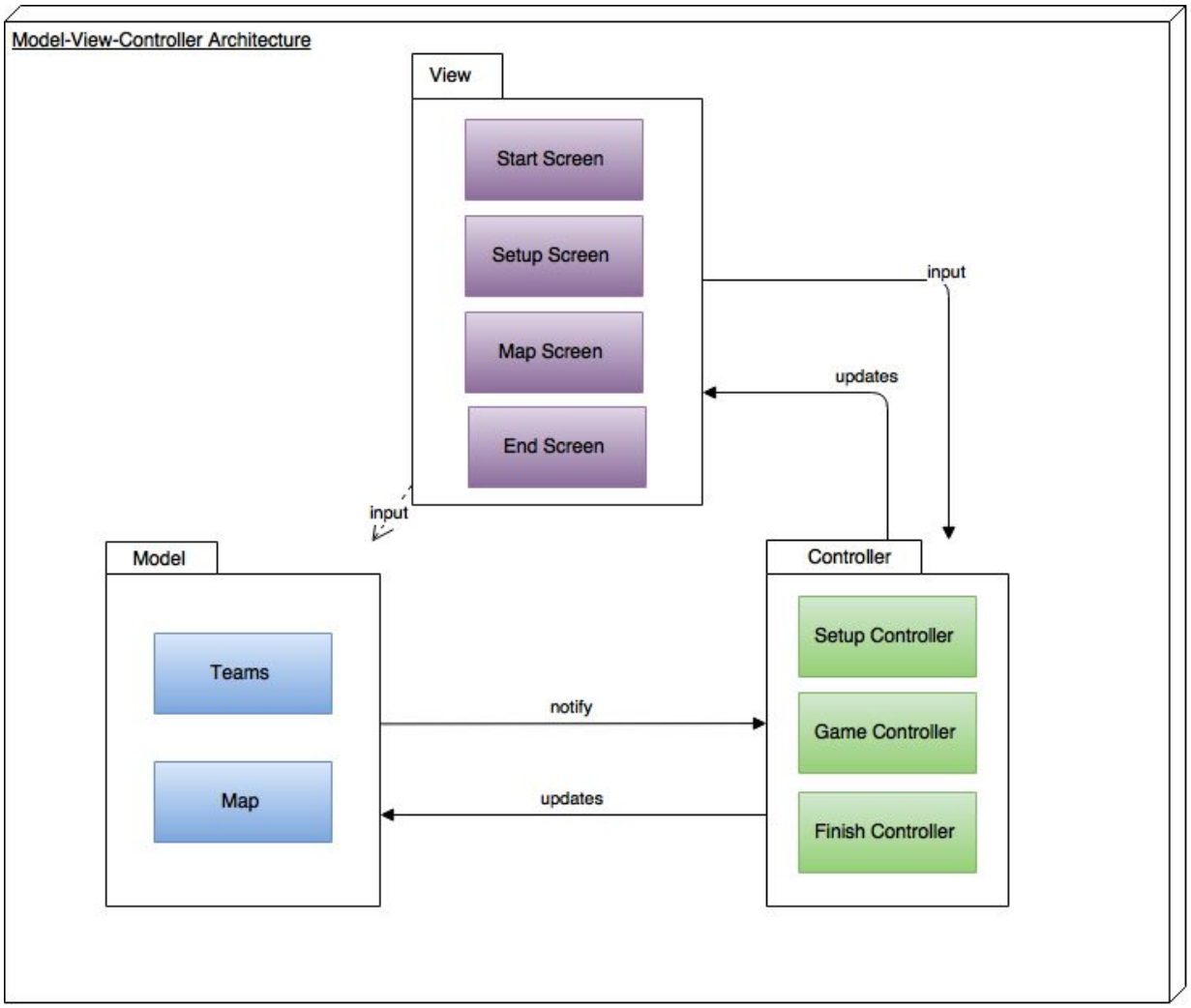
From the decision matrix, it follows the pipes and filters architecture, interpreter architecture and model-view-controller architecture came forth as appropriate possible architectures. During brainstorming, we decided to apply a model-view-controller architecture primarily and interpreter architecture as a portion for the robots.

				Rankings					
	Applicability	**Modularity**	**Ease of Implementation**	**Adaptability**	**Collaborability**	**Useability**	**Security**	**Testability**	**Total**
Weight (1->5)	4	3	3	3	4	3	2	3	
Architectures (1->10)									
Monolithic	2	2	5	2	2	5	3	4	76
Pipes and Filters	7	4	6	6	5	5	4	7	140
Objects & Components	2	7	5	5	5	6	3	5	118
Events	3	4	5	4	4	4	2	5	98

Repositories	2	2	3	4	4	7	7	5	101
Layered	5	2	3	5	3	6	4	5	103
Interpreters	7	4	6	4	4	7	4	6	133
Model-View-Controller	8	7	6	7	7	7	4	7	170

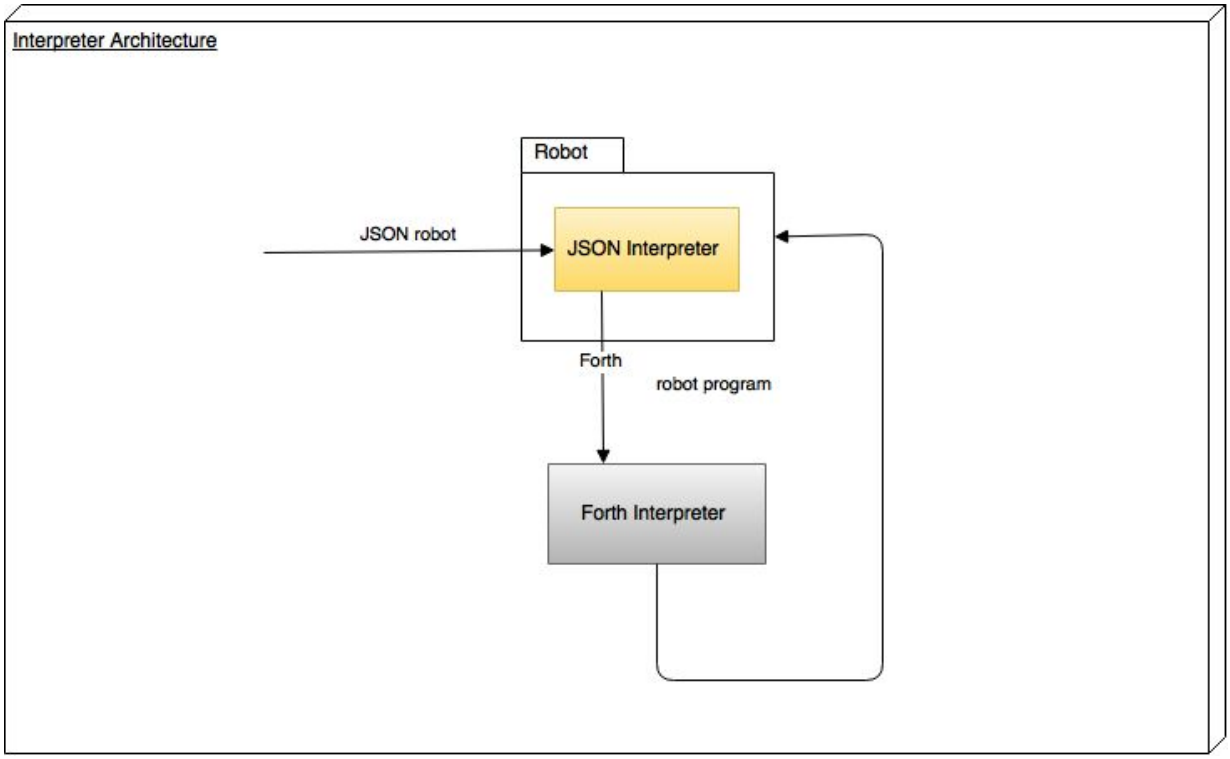
B. Architectural Diagrams

Following the model-view-controller and interpreter architectures we decided, we outlined them as below.



The model-view-controller focused on building an interface between the data storage (model) and the controller to maintain an accurate representation of the system, while the interface between the view and the controller focussed on handling and representing input to whichever player is using the program.

The interpreter was chosen to describe the flow from JSON robots to a functioning robot within the model of the model view controller. This implies that robots will be a part of teams in the previous structure.



IV. Classes

For this section, our group created and described classes to fit our design. These classes are described below. Additionally, we have included the UML class diagrams and briefly touched on the libraries relevant to the classes and their design.

A. Class Descriptions

1. Robot:

   The class Robot will contain all of the aspects of one robot. This class is meant to contain the health, strength, identifying information, which team it belongs to, and the actions that the robot can perform. The Robot objects will be created at the beginning of the simulation, and the classes fields will be updated whenever the particular instance of robot is interacted with.

2. Team:

   This is the class in our program that will contain all of the robots that are on one team, and the information about that team. Some of the information included will be the number of wins and losses, the team

name, the number of currently alive robots, and a field that determines which robot goes next.

3. GameController:

   Game controller is the class that controls all of the game play. Once the game is started by SetupController this controller takes over and runs all of the simulations. It coordinates which team goes next, and it runs actions from the ForthInterpreter, such as moving robots around, and firing shots. This Controller also pauses, resumes, and ends the game, however the game will only be ended when the simulation is complete, and there is a winner. MapView will be used to display the controller's data.

4. SetupController:

   The setup controller is the class that will be used to set up the simulation, this class will set the map size, the teams, and then begin the match. This class also decides which type of match will be run, a GUI match, a headless match, or a debug match. The controller displays all of its data using SetupView.

5. EndController:

   This controller takes over after the simulation has ended. EndController takes data from the GameLog and calculates what needs to be displayed, which team won the game as well as stats about each team and robot such as the number of shots fired. The data calculated is then displayed using the EndView.

6. DebugGameController:

   Our DebugGameController is what will allow us to move through the robot simulation in either direction. This will allow us to determine what actions were taken and when to help us find errors. We will also be able to change various attributes of the simulation such as robot health, and robot strength. This controller will provided the data that DebugMapView will display.

7. GameLog:

   This class is used to store all of the data from events during the simulation. It will be used to run debugging view and provide information to the EndController. The logger will track events such as shots fired, deaths, and movement.

8. JSONInterpreter:

   The JSON Interpreter is what we will be using to translate between Java and JSON. We will use this interpreter to read data into each robot object in the simulation. The data will be read in as JSON and then translated into Java. We will also be able to take the robot data and translate it back into JSON, so that we can provide an updated object with the new stats after each match.

9. FourthInterpreter:

This interpreter will be used interpret the forth logic associated with each robot. The interpreter will run at two points: to initialize each robot's forth logic at the start of a match, and to execute each robot's turn. It will work by reading the forth functions and variables in each method, and then passing along actions to execute to the GameController

10. RobotView:

    This view is used to display a robot sprite that can be moved around the map. Depending on what the robot is doing a different image will be shown.

11. EndView:

    The end view will be used at the end of a simulation to show the final results that were calculated by the end controller.

12. DebugMapView:

    This view is very similar to MapView but will also provide interface elements to allow reversing through the game and changing the various robots attributes.

13. MapView:

    Map View is what will provide the graphics of the simulation, it will animate all of the movements of the robots and the actions that they perform. This view will be represented using 2D graphics.

B. Libraries

Along with the classes listed above, we are also using a number of 3rd party libraries and frameworks. These include:

1. Java Libraries:

   We will be using the java language and the standard java libraries to build this project. This will provide us with standard functions and objects in order to speed up the development process

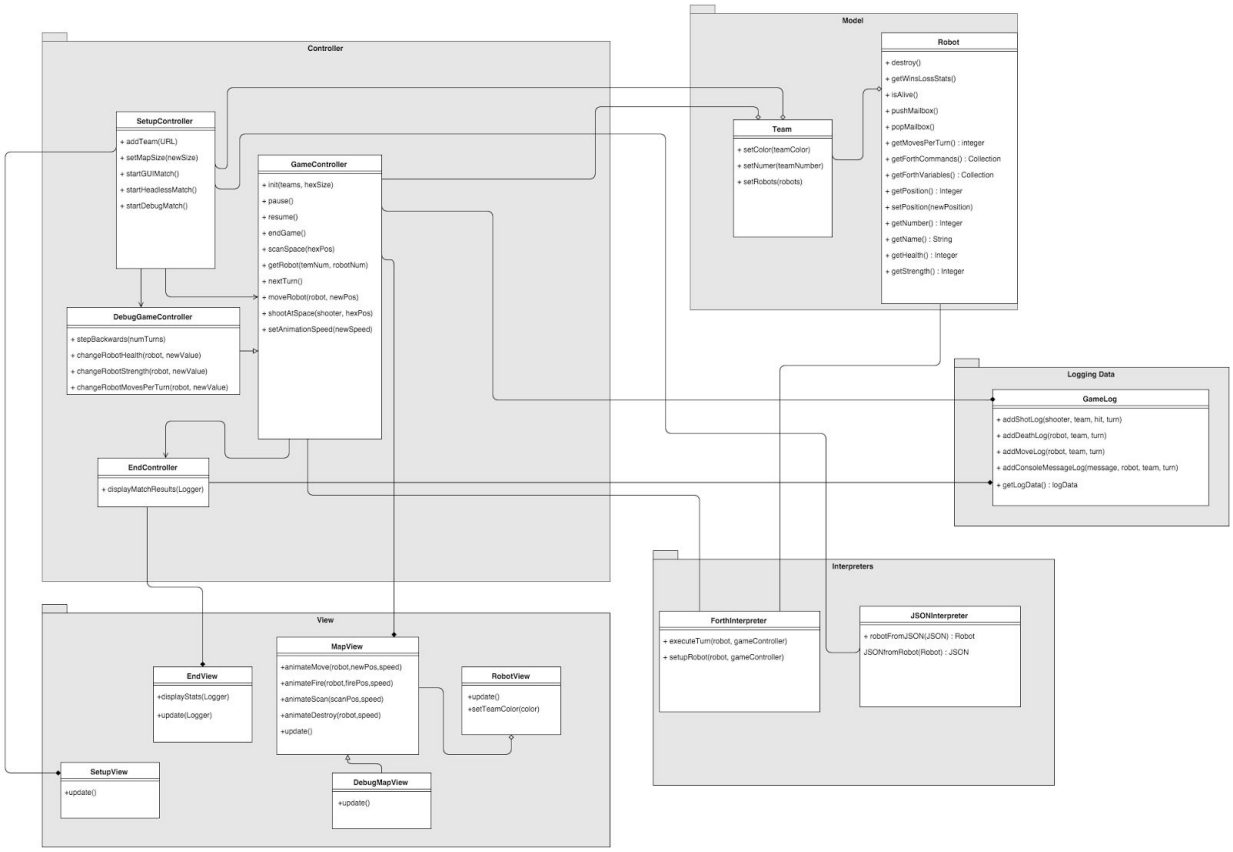
2. A GUI/Game development library

   We will be using a 3rd party library to provide us with graphical user interface elements, and to assist us with animations. This will allow us to make a much better looking interface in less time. The specific library is currently to be determined

3. JSON

   We will use the JSOn standard to read robot data into our system. This standard will allow us to easily exchange robots with other groups in a standard way. Our system will both read JSON robot data, and write win/loss data in JSON after a match has completed.

C. UML Diagram



Full size diagram will be attached to assignment submission

V. Time Estimates

We are estimating that the entire implementation will take around 3 weeks. Implementing the controllers and views should be one of the shorter jobs, while it is expected that implementing the interpreters will take a large chunk of time. This is due to the experience of the group with the different pieces. Final details such as advanced graphics and sounds will not be implemented unless time permits.

VI. Conclusion

In conclusion, we have decided to use a Model-View-Controller type architecture for the system. The system will contain a sub architecture, an interpreter, for the Forth/Java interpreter. The pipes and filters architecture was a viable option but we decided that it didn't quite meet the requirements. The classes that were chosen are still subject to change but represent what the detailed design of the project will look like. We identified a number of different views that would be needed to display the proper information, as well as many different controllers that will

modify each view as it needs to be. Following this we will begin implementing the design we have created and review.