Design

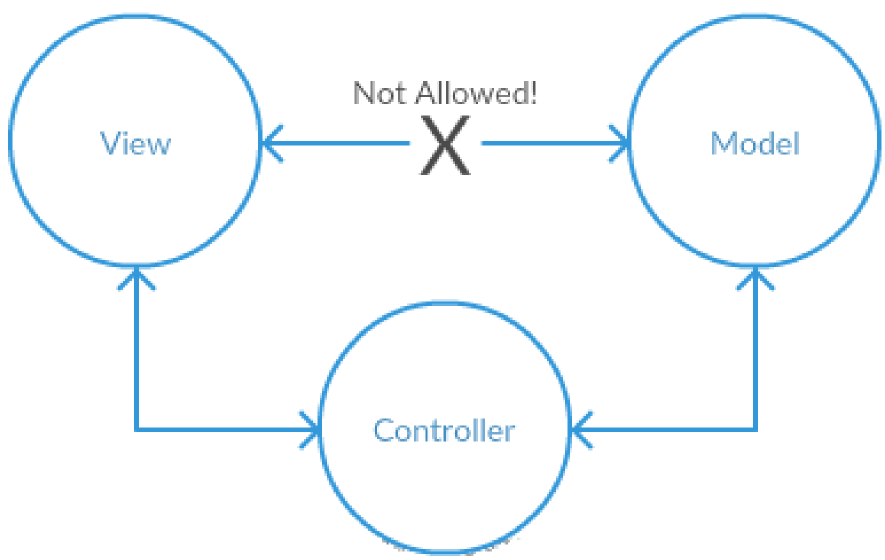
Preliminary Design

Need:

- ~~Data Flows~~
- ~~Large modules/packages~~
- ~~Architectural Design~~
  - ~~The modules, and~~
  - ~~Their interactions~~

Architecture

We will be utilizing the Model-View-Controller architecture. Since this architecture splits the project into 3 parts, it will allow us to easily collaborate on code. Each module will have it's own API, or public interface, allowing someone working on one part of the project to easily interact with another part without worrying about the underlying implementation. Further, the person working on the one module will be able to make changes without fear of breaking it for the other modules, since they know how the other modules are interacting with it.



Modules

The architecture for this project will be broken into 3 parts, as per the Model-View-Controller architecture:

1. Model

The Model contains all the data for the application. This includes things like the robot objects, and the game board. The controller uses the data in this module to simulate the game. The controller will also use the public methods in this module to control the robots and game board.

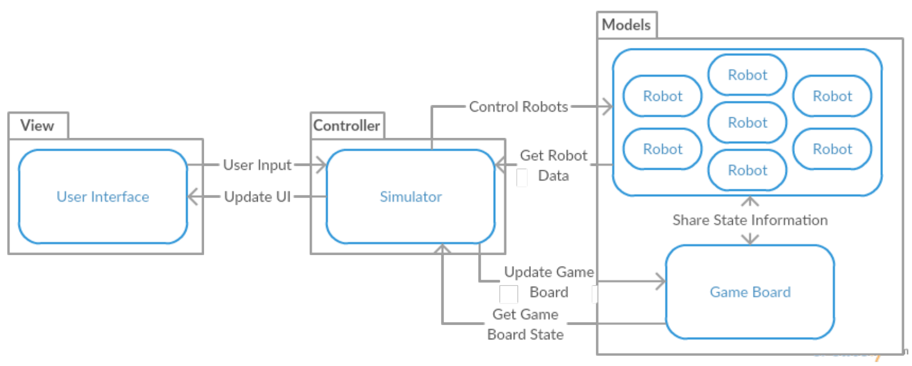
2. View

The View module of the MVC architecture will be responsible for creating the graphical user interface. It will have public methods to allow the controller to output information to the user.

3. Controller

The controller is where all of the game logic will be performed. This is where the simulator will be implemented. The controller is the brains of the application, and will be able to retrieve data from the model, and output data to the view.

Data Flows



With the Model-View-Controller architecture, The controller acts as the communication hub for the three modules. Thus, the only communication paths that are allowed are:

- Models <—> Controller
- Views <—> Controller

- Models <—> Models
- Views <—> Views
- Controller <—> Controller

The controller must communicate with the view to output data to the user, and process user input. The controller has to talk to the model to control the robots, update the game board, and retrieve information on the robots and board to run the simulation.

Detailed Design

Need:

- Data Formats/Table Layouts
- Code interfaces (i.e. public class interfaces)
  - Method names
  - Post-, pre-, and error conditions
- Optional pseudocode for complex operations

Code Interfaces

Model

Robot Object
int teamNumber
int robotNumber
int pointsLeft
int maxMoves
int movesLeft
int power
int health
int range
bool hasFired
void: move(coords)
void: fire(coords)
Robot: scan(coords)
Stats: getStats()

Controller

Match Controller Object
Robot [ ] robots *Cell* [ ] cells *int* turnNumber *int* maxTurns *int* currentTeam *int* executionSpeed *void*: stepForward() *void*: stepBack() *Robot*: getContentsOfCell(coords) *void*: damageRobot(robot)

Views

Main Menu View
void loadWatchMatch() *void* loadInstantResults() *void* loadTestBench()

Team Select View
Team [ ] teams *void* loadTeam(int slot) *void* confirm()

Watch Match View
MatchController controller *void* step() *void* play() *void* stop() *void* setRate(int rate) *void* updateLog(string log) *void* (string log)

Test Bench View (inherits from Watch Match View)
void runCommand()

Match Results View
score [ ] scores *void* returnToMenu

ADTs

Cell

Game Controller

Sets:

GC: set of Game Controller Objects
Robot: set of Robots
Cell: set of Cells
Coord: set of cell coordinates
Team: set of robot teams
int: set of integers (Z)

Signatures:

newGameController(teams) Team[] –> GC
GC.getCells() –> Cell[][]
GC.getCell(coord) Coord –> Cell
GC.stepForward() –> GC
GC.stepBack() –> GC
GC.pause() –> GC
GC.play() –> GC
GC.setSpeed(speed) int –> GC
GC.viewStats(robot) Robot –> Stats
GC.selectRobot(robot) Robot –> GC

Semantics:

newGameController(teams): Returns a new game controller initialized with the robots in the teams given.

GC.getCells(): Returns an matrix such that matrix[x][y] contains the cell at coords(x, y)

GC.getCell(coord): Returns the cell object located at coord.

GC.stepForward(): After execution the program counter for the currently executing robot will be increased by one, switching to the next robot if the program completes. Execution of the instruction will change robots and cells according to the semantics of the instruction itself, within the limitations of the language.

GC.stepBack(): Resets the GC to the state before the previous stepForward();

GC.play(): Begins to stepForward the GC at every *time* interval.

GC.pause(): Causes GC to hold at the current state until stepBack, stepForward, or play are called.

GC.setSpeed(speed): Sets the interval between steps of the GC to some inverse of speed.

GC.viewStats(robot): Returns the relavent statistics about the robot for viewing.

GC.selectRobot(robot): This function selects a given robot to the be the target of further actions such as viewStats.

UML Class Diagram:

GameController
Properties:
-speed: int -cells: Cell[ ][ ] -teams: Team[ ] -paused: bool -selected: Robot -currentlyExecuting: Robot
Functions:
+constructor(teams: Team[ ]): GameController +getCells(): Cell[ ][ ] +getCell(coord: Coordinate) Coord: Cell +stepForward(): void +stepBack(): void +pause(): void +play(): void +setSpeed(speed: int): void +viewStats(robot: Robot): Stats +selectRobot(robot: Robot): void

Test-Bench Controller

Test-Bench Controller extends the specifications from the Game Controller ADT.

Sets:

TB: set of Test Bench Controllers String: Set of strings Stat: An object containing statistics pertaining to an individual robot.

Signatures:

TB.executeCommand(command) String -> TB

Semantics:

TB.executeCommand(command): This function takes the string "command" (presumably from a user input box) and executes the command against the selected robot, according to the robot language specification and limitations.

UML Class Diagram:

TestBenchController (inherits from GameController)
Functions:
+executeCommand(command: String): void

Robot Object

Sets:

CO : Set of all game board coordinates
R : Set of all Robots
I : Set of all Integers

Signatures:

newRobot(i, i, i, i, i, i) –> R
R.move(co) CO –> R
R.fire(co) CO –> R

R.scan(*co*) CO –> R
R.getStats() –> S

Semantics:

newRobot(*i, i, i, i, i, i*): Takes the following values as Integers:

- teamNumber
- maxMoves
- power
- health
- range

And returns a Robot initialized with those values. You don't need to supply robotNumber because that will be computed by the program. Also, pointsLeft and movesLeft are default values.

r.move(*co*): Moves the robot to the given coordinate on the game board

r.fire(*co*): Fires at the robot at the given coordinate on the game board

r.scan(*co*): If there is a Robot at the given coordinate on the game board, return that Robot. Otherwise, return nothing.

r.getStats(): Returns a Stats object for the Robot

UML Class Diagram:

Robot
Properties:
-teamNumber: int -robotNumber: int -pointsLeft: int -maxMoves: int -movesLeft: int -power: int -health: int -range: int -hasFired: bool
Functions:
+constructor(int teamNumber, int maxMoves, int power, int health, int range): Robot +move(coord: Coordinate): void

Robot
+fire(coord: Coordinate): void +scan(coord: Coorginate): void +getStats(): stats: Statistics