CMPT 370 Software Design Document for BattleBots

Mackenzie Power

Haotian Ma

Ryan Tetland

Will Revell

Mitchel Kovacs

Purpose

The purpose of this design document is to present our architecture's description, plan the classes that will later be implemented, along with interactions between them, and state our changes from our requirements document that must be made. The \_\_\_ section will state why we chose to implement our architecture, the pros and cons of it, and why we did not chose other architectures that we considered. In the \_\_\_ section, We will describe what each class should do, what information they will contain, and what information they will send to other classes. It will not contain any code specifications, just how the different classes are connected. This will be shown through class UML diagrams, as well as detailed descriptions. After designing the architecture, we expect to come across newly discovered difficulties that we did not know when writing our requirements, and as a result will have to update our promises made in the requirements document. We will state the changes from our requirements and also state why these changes must be made. This document will describe the classes that will later be implemented,and show their interactions with each other.

System Overview

Our architecture we will implement is Model-View-Controller. The view will receive messages from the controller of what it should be displaying to the user. Itwill also send messages to the controller when the user interacts with it by clicking buttons. It will not have any direct interactions with the model of the system. The model will take messages from the controller and perform the required task then send information back to the controller. These parts of this architecture along with their interactions will be described more in the \_\_\_ section.

Pros and Cons Of MVC

A benefit of this architecture is the separation of the interface and the model. This will make the code easier to work on because each class will have a specific purpose. This architecture also will result in less coupling. This is a benefit because changes in a class will not affect many other classes. Cohesive classes also result in much easier testing because each class can be tested individually. One disadvantage with MVC is it may be difficult to divide the work other than between the model, the view and the controller. This is because these parts of the architecture have many interactions within themselves. Another disadvantage is that it may take more time planning the interactions properly to interactwithout having to many unnecessary coupling. We feel this planning will benefit us when we are at the code implementation stage because each class will have a high level of cohesion.

Definitions and Acronyms

Unified Modeling Language(UML): a diagram to show the function signatures, and entities of classes. It also shows relations between classes.

Model-View-Controller(MVC): design architecture that is divided into three

separated parts that communicate with each other. The model manages the data and behind the scenes workings of the program, The view is the interface for the user, and the controller communicates between the other two.

Coupling: the inter-dependencies between classes.

Cohesion: a measure of specific functionality within a class.

JavaScript Object Notation (JSON): a data file that will be used to transmit

robot scripts, and robot statistics into Java.

Robot Librarian: a system for access of a collection of scripts that will control the CPU player's robots. It also updates the robots statistics after the match is completed.

Scan: this is when a robot checks it's surroundings, that it is allowed to see,

for enemies. The AI code will use this information to determine it's actions for it's turn.

Constraints

In our system, we will be using a given robot librarian system to access AI scripts. This will force us to implement our game board's, and robot's, coordinate system in a way that complies the AI scripts. The coordinate system is relative to each robot. When a robot is facing a direction, straight ahead of it is always coordinate zero, and the hexagons one move away from it have a direction of zero though five in a clockwise direction. Spaces two moves away are labeled zero through eleven, and spaces three moves away are labeled zero through seventeen.

Each robot has a given set of resources. This includes attack, health, movement,and range. These variables must be kept track of to determine when the robot is out of moves, when it is dead, how many spaces it can see, and how much damageis dealt out. The following is each robot with it's resources: Scout has 1 attack, 1 health, 3 movement, and 2 range; Sniper has 2 attack, 2 health, 2 movement, and 3 range; Tank has 3 attack, 3 health, 1 movement, and 1 range.

Changes From Requirements Document

Robot Librarian

After receiving feedback from our requirements document, we now realize that the robot librarian is not a part of our system because we are not implementing it ourselves. The robot librarian should have just been shown as an actor instead, because it will be given to us as a separate system that we will be using.

What happens when robot tries to move out of bounds?? Turns to invalid direction?? shoots when not allowed??

If a human or AI robot attempts to move to a location that is out of bounds, the robot will just stay in the hexagon that they are currently in.

End turn for human

During the design process, we realized that we did not include an end turn option for a human player. This is important to have because a player may want to end their turn even if they still have moves or shots left. We will add a button in the game screen interface labeled "End Turn", that will end the player's turn and allow the next players turn to start.

In our requirements document we stated some functionality that the game "should have", now during the design process we have decided to include a section on the side of the game play screen that displays each robots current statistics. We will also display the final statistics upon completion of each game.

We also had a section in our requirement document with functionality that the game "could have". We have decided to not include any of these so we can focus on the more important functionality of the game. We decided that it would be more important to have the game functioning properly than have a game with fancy attributes but does not work.

Architecture Design

The chosen architecture for designing and implementing the BattleBots program outlined in the requirements document will be a Model-View-Controller.

Model-View-Controller consists of three different parts: Model, View and Controller. The Model does not rely on either View or Controller (it does not care how it is being controlled or viewed) it just controls the behavior of the application following by the rules and manage the data of the application domain. Model can update and display the state to View so the user can see what changes has he/she made by using the Controller. View needs to register into the which Model it wants to view, once it has been registered it will display whatever the Model allows it to the user. If View is the output tool in this relationship, Controller will be the input tool in the relationship, it interacts with the user by taking user’s input from either keyboard or mouse and informing it to the Model.

There are a number of reasons why this architecture has been chosen over other types. The primary reason for this design choice is that all members of the A5 software team already have a familiarity with how this architecture should work. All members of the group have taken computer science courses which involved creating programs based on a Model-View-Controller architecture, so experience with this method is present. We acknowledge that generally a Model-View-Controller architecture is a more efficient architecture for projects that are much larger than this. However, we feel as though our experience with this architecture relative to other architectures outweighs any difficulties we may have in implementing such a small program with a Model-View-Controller. Another reason why this architecture was chosen is because of the heavy use of visual displays driven by the user (relative to other types of programs). This program is driven by the users' input. Everything from viewing the rules; selecting team attributes; choosing game properties; and executing individual moves require input from the user in order to be performed. Every action done by the user will require a change in display either as a new interface window or as a visual change in board state. There will be very few actions done without the user's input.

A benefit of this architecture is the separation of the interface and the model. This will make the code easier to work on because each class will have a specific purpose. This will result in less coupling because changes made to a class will not affect many other classes. Cohesive classes also result in much easier testing because each class can be tested individually.

Despite all these advantages, there are some disadvantages that we must acknowledge. One disadvantage with Model-View-Controller is it may be difficult to divide the work other than between the model, the view and the controller. This is because these parts of the architecture have many interactions within themselves. Another disadvantage is that it may take more time planning the interactions properly to interact without having too much unnecessary coupling. We feel as though the benefits of a high level of cohesion achieved through this architecture will outweigh the time costs we may endure during designing and implementing the program. The aspects of Model-View-Controller architecture are……

MVC vs Pipe-and-Filter

Pipe-and-Filter contains individual programs transforming input data to output data, it is different than MVC because it only simply consist with the three main parts Model, View and Controller. Base on what we have learned, we see Pipelines as an unidirectional tool, it may going to cause us some unnecessary troubles for the later designing stages. Additionally, we know that Pipe-and-Filter has very high reuse potential but for us to look back in the individual parts will be a huge amount of time consuming, MVC seems pretty friendly for editing because the Model is basically the only portion we need to edit if it is needed.

MVC vs Layered System

Layered System is effective for separating concerns by creating multiple layers, also reduce the impact of change when changes don’t affect layer interfaces. But because they are too many layers, program’s performance will be degraded, especially it is a multi players game that is going to challenge players patient which we do not want to see that.

MVC vs Blackboard Style

Since we are not designing this game for networking reason, blackboard is not appropriate for this program.

MVC vs Independent Component

To have all Components individually allows decoupling and autonomy of components it also makes enhances reuse and evolution much easier due introducing new components without affecting existing ones. However, the connections between components are not guaranteed for example: components announcing events cannot be sure getting a response nor have control over the order of responses. MVC on the other hand, View will always watching Model once it gets registered and Controller has the control about the output through Model.

CMPT 370 Class outlines

GameController(controller) implements ActionListener, KeyListener

Attributes

GAMEWIDTH

GAMEHEIGHT

numhumans

numplayers

boardsize

Functions

handleEvent(keyPress)

handleEvent(HexClick)

handleEvent(buttonPress)

startGame()

The GameController class allows user and AI input on the games models and updates the display(view) accordingly. The GAMEWIDTH and GAMEHEIGHT attributes are global variables representing the desired size of the game screen. While numhumans, numplayers and boardsize will be used to create and populate the board. There are three separate handleEvent functions. The first is utilized only during the game screen and allows the user to move the robot using the AWEDXZ keys on the keyboard. The second is also only utilized during the game screen and allows the human player to shoot other robots by clicking on them with the

mouse. The third and final handleEvent function is used to navigate throughout all the game screens and the game itself using the mouse to click on the navigation buttons such as continue and quit. The startGame function creates a game screen of a set size and takes the user to the MainMenu panel, the games beginning.

View(view) extends JFrame

Attributes

Functions

createView(width, height)

createPanel(panelClass)

showMainMenu(actionListener)

showHelpPanel(actionListener)

showGamePropertiesPanel(actionListener )

showTeamSelectionPanel(actionListener, keyListener )

showWinnerPanel(actionListener)

showStatsPanel(actionListener, StatsTracker)

showGamePanel(actionListner, keyListener, )

The View class is the "view" in MCV architecture and is a frame that displays the various views of the game. The createView function initiates the view frame at the desired size while the createPanel function creates a basic panel which contains no content. Then next 7 functions display the various Panels of the game including the MainMenu, HelpPanel, GameProperties,

TeamSelectionPanel, WinnerPanel, StatsPanel, and GamePanel with an actionListener and a keyListener as required.

gangClass(model)

Attributes

gangColor

numRobots

Scout

Sniper

Tank

Functions

CreateGang(gangColor)

getgangColour()

getnumRobots()

setnumRobots()

The gangClass will represent each team color on the game board and contains the attributes gangColor representing its team color and numRobots showing the number of remaining robots on the team. The CreateGang function takes in the teams color as a parameter and creates the corresponding scout, sniper and tank. There is basic get functions for the gangColor and numRobots allowing for easy access and a set function for numRobots allowing editing to show when robots are killed.

gangAIClass(model)

Attributes

gangColour

numRobots

ScoutAI

SniperAI

TankAI

Functions

CreateGang(gangColour)

getgangColour()

getnumRobots()

setnumRobots()

The gangAIClass is identical to the gangClass save for the ScoutAI, SniperAI and TankAI attributes which have been changed from Scout, Sniper and Tank. The create function will now create the AI classes instead of the human controlled robots.

GameBoard(model)

Attributes

HexSpace spaces[x][y]

Functions

getRobots()

CreateGameBoard(boardsize)

The GameBoard class represents the hex grid on which the game is played, it will utilize a 2-dimensional coordinate system with the center of the board being the origin(0,0). The Hexspace attribute is an array that contains every hex on the game board with its corresponding coordinate value. There are two functions, the getRobots function creates a list of every robot on the correct

board size picked in game properties.

HexSpace(model)

Attributes

robotArray

isEmpty

Functions

getisEmpty()

setisEmpty()

The HexSpace class will be the individual hexagon spaces that make up the game board. Each space must be able to store a robot of any type (scout, sniper, tank) in an array. The amount of HexSpaces will be initialized by the GameBoard Class. It includes the attributes isEmpty which will be true if no robots are on the hex, and robotArray which is an array that contains every robot on the hex.

Robot(model)

Attributes

ShotsLeft

Movement

Range

Damage

movementLeft

health

healthLeft

relativeDirection

directionDimension

isTurn

robotType

gang

horizontalLocation

verticalLocation

Functions

Create\_Robot()

getMovement()

getShotsLeft()

getRange()

getDamage()

getmovementLeft()

gethealth()

getrelativeDirection()

getdirectionDimension()

gethealthLeft()

getrobotType()

getgang()

getisTurn()

getverticalLocations()

gethorizontlLocation()

setverticalLocation()

sethorizontalLocation()

setisTurn()

setrobotType()

setgang()

sethealthleft()

setmovementLeft()

setreleativeDirection()

setdirectionDimension()

setShotsLeft()

setIsTurn()

getIsTurn()

The robot class will represent the basic robot type and will contain attributes used for every robot subtype, including; the robots movement range, attack range, shot damage, remaining movement left, the direction relative to the game board, if its the robots turn or not, the current directions dimension it is facing (whether it is 1,2 or 3 squares away depending on the range) its type(scout, sniper or tank), the horizontal location coordinate, the vertical location coordinate, its team or gang and its current health value. It also contains several important functions including; a basic create\_robot function that creates a robot object with attributes set to default values, it also contains a getvalue function for each of its attributes allowing access to each attributes value and a setvalue function for its health, direction, direction dimension, isTurn, robot type, gang/team, horizontal direction, vertical direction, shots left and movement allowing the values to be incremented.

Each of the Scout, Sniper and Tank classes extend the basic robot class and will be the robot objects controlled by the games human players. They all have the four attributes movement, range, damage and health from the basic robot class, but each is now set to the robot types default values. An updated create function now makes a new specific type robot with the appropriate attribute values.

The ScoutAI, SniperAI and TankAI classes extend the Scout, Sniper and Tank classes and add five new functions. Move which takes in a target location and changes the robots current location to those coordinates. The create function will take in a string of AI instructions and upload it to the new robot object. Scan which takes in the robots current location and checks all hexes within the robots range for enemy and allied robots. Shoot which takes in a target location and deals the robots attack damage to all robots on that hex. And Turn which takes in the desired direction as

a parameter and turns the robot relative to the game board.

Scout(model) extends the Robot class

Attributes

Movement = 3

Range = 2

Damage = 1

Health = 1

Functions

Create\_Scout() - changes the Create\_Robot() function to create robot

with specific stats

Sniper(model) extends the Robot class

Attributes

Movement = 2

Range = 3

Damage = 2

Health = 2

Functions

Create\_Sniper() - changes the Create\_Robot() function to create robot

with specific stats

Tank(model) extends the Robot class

Attributes

Movement = 1

Range = 1

Damage = 3

Health = 3

Functions

Create\_Tank() - changes the Create\_Robot() function to create robot

with specific stats

ScoutAI(model) extends Scout class

Attributes

AIinstructions

Functions

Create\_ScoutAI()

move(target)

shoot(target)

scan(horizontalLocation, verticalLocation)

turn(direction)

SniperAI(model) extends Sniper class

Attributes

AIinstructions

Functions

Create\_SniperAI()

move(target)

shoot(target)

scan(horizontalLocation, verticalLocation)

turn(direction)

TankAI(model) extends Tank class

Attributes

AIinstructions

Functions

Create\_TankAI()

move(target)

shoot(target)

scan(horizontalLocation, verticalLocation)

turn(direction)

RobotStats(Model)

Attributes

kills

deaths

plays

endedAlive

wins

loses

damageInflicted

damageAbsorbed

distanceTravelled

Functions

CreateRobotStats()

getkills()

getdeaths()

getplays()

getendedAlive()

getwins()

getloses()

getdamageInflicted()

getdamageAbsorbed()

getdistanceTravelled()

setkills()

setdeaths()

setplays()

setendedAlive()

setwins()

setloses()

setdamageInflicted()

setdamageAbsorbed()

setdistanceTravelled()

The RobotStats class tracks the stats of each robot during the game it keeps track of the attributes kills, deaths, games played, games ended alive, wins, loses, damage inflicted, damage absorbed, and distance travelled. The create function initiates a RobotStats object with all attributes set to zero. For each attribute there is a set and get function to allow access and manipulation of

the attribute values.

StatsTracker(Model)

Attributes

Robots[]

Functions

CreateStatsTracker()

The StatsTracker class is a collection of all the robot stats for each robot in the current game. Its sole attribute is an array that contains the RobotStats for each robot in the game. Its create function makes an object that takes in every robot in the games and creates a RobotStats for each one in the array.

MainMenu extends JPanel

Attributes

Functions

mainMenuCreate()

MainMenu class has only one function, its create function which initiates a MainMenu object with three JButtons, Begin, Help and Quit.

HelpPanel extends JPanel

Attributes

Functions

helpPanelCreate()

The HelpPanel class only has a create function that makes an object with the two Jbuttons, Back and Quit also well as a block of text describing the rules of the game.

GamePropertiesPanel extends JPanel

Attributes

Functions

gamePropertiesCreate()

The GameProperties has no attributes, and its create function generates 3 sets of radio buttons for the nmber of players, number of humans, and the board length.

?????????????????????????

TeamSelectionPanel extends JPanel

Attributes

Function

teamSelectionCreate()

updatedisplayTeams()

??????????????????????????

WinnerPanel extends JPanel

Attributes

Functions

winnerCreate()

The WinnerPanel has only a create function which constructs an objects with three Jbuttons called Play Again, Stats, and Quit.

StatsPanel extends JPanel

Attributes

Functions

statsCreate()

The StatsPanel has no attributes and its only function is its create function, which builds an object with two Jbuttons, Play Again and Quit, and the updated stats of the game

GamePanel extends JPanel

Attributes

Functions

GameCreate(keyListener, ????????)

drawImage()

drawString()

updateGame()

The GamePanel is the actual game board panel on which the game is played. It has no Attributes and four functions.????????????? The GameCreate functio????????????????????The drawImage function draws an image of a specific type at a specific location. The drawString function draws a string of characters at a specified location. The updateGame function updates the view by checking if the positions of the game objects have changed and calling the drawImage and drawString functions to redraw the game panel with the correct new positions.