RoboChess

DESIGN DOCUMENT
VERSION 1.1
OCTOBER 23, 2016

Prepared by: Team B5
LIN, YUCHEN | NELSON, JORDAN | PARK, RYAN | WANG, XINGENG | VAN HEERDE, WILLIE

Table of Contents

1. Architecture

1.1 Our Architecture

Our team chose to use a client-server model architecture fused with a model-view-controller subsystem as the major software architecture in our game design. A Client-Server model architecture partitions tasks and workloads between the server and various clients. The client-server model is more often used in web programming which has a real physical server to run the backend program and has the web browser to execute the client program. In our design, we will not have the server backend program run on an actual server, but we will design the game to use the client-server model using various threads. We will treat each player as a client, and the game model as a server. The server will only communicate with one client at a time, depending on which player's turn it is. This means that all the clients will be blocked from sending a request to the server except the one client whose turn it is. After one client sends a request about what they want to do in the game (e.g. move, fire), the server will update the game information and send the updated information back to the current client. After every turn, the server will give the client a copy of the updated board to use, to keep each client up to date on the current model of the game.

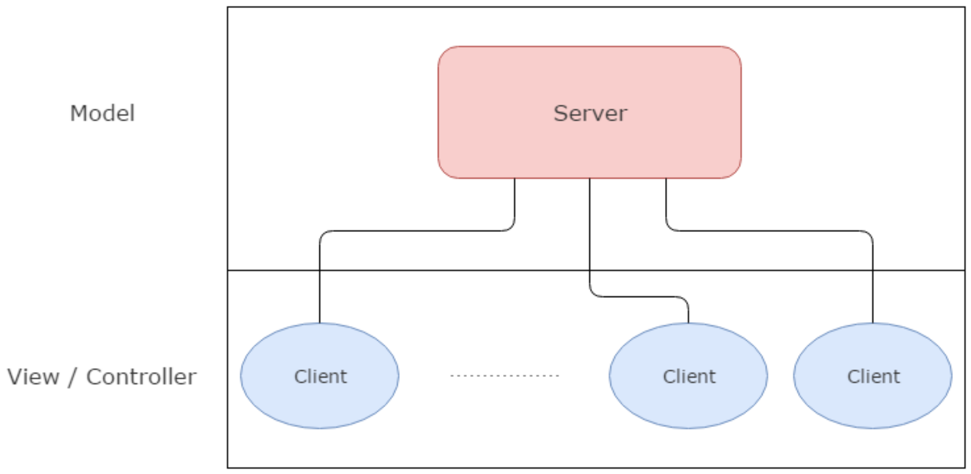

Figure 1. MVC/Client-Server Hybrid Architecture

Although the overlying architecture of our system is client-server based, we have fused it with a model-view-controller subsystem, where the client thread acts as the controller and the 'ControlPanel' class acts as the main view. The server thread acts as the model, storing and calculating various variables to model the game.

1.2 Justification

There are several reasons why we chose to use a client server fused with a MVC (Model–View–Controller) model as our software architecture rather than just using a MVC design which all of us are familiar with. The first reason is that we found this made sense logically based on the structure of the game in the abstract. Several different player instances communicating with a single board model is very similar to several clients communicating with a single server. By making the game model the server and each player a client, the remaining design of the game naturally structures itself around this central design, where no players communicate with each other and only communicate with the model itself. The second reason for choosing a client-server architecture is that it was a challenge and something that deviated strongly from the other groups, which we all found appealing as a simple way to challenge ourselves and do something a little bit more interesting than the standard. The third reason we chose our architecture was it gave us great flexibility if we ever wanted to modify our game to work over a network (i.e. Have multiplayer on multiple computers, with one computer acting as a host).

Although we chose a Client-Server / MVC fusion architecture as our main architecture, we've only implemented it after the actual game instance is running. For our main menu that is loaded before the game begins (the menu sub-system) we've used more of a view-controller architecture, where a single class runs and monitors each menu, and when a new menu is needed the controller class calls the class needed to display that menu. In this way, we can keep the code running the graphical user interface separate from the code running in the background, while keeping everything concise and easy to implement.

2. Class Design

2.1 UML Diagram

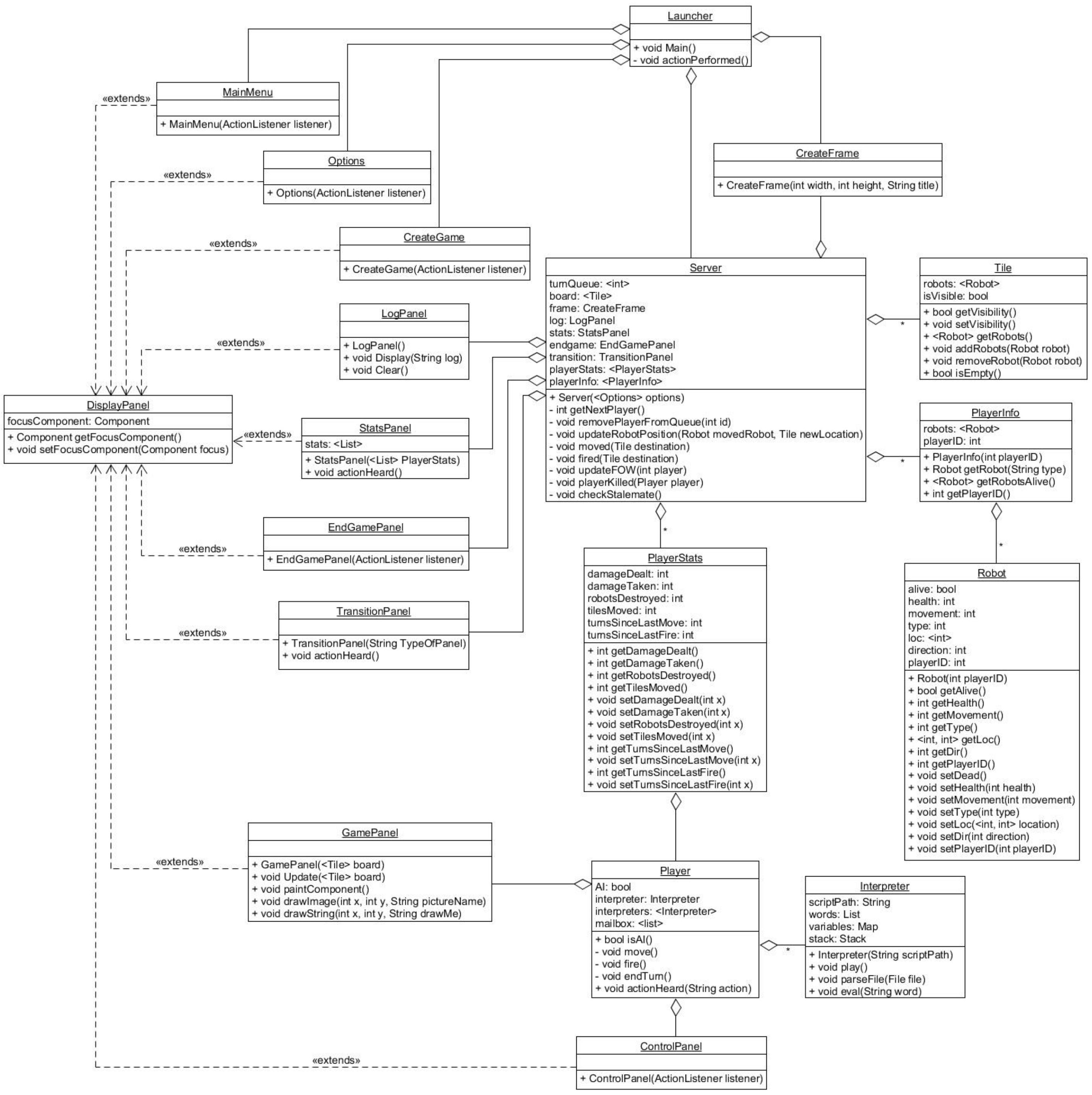
Launcher
+ void Main()
- void actionPerformed()
MainMenu
+ MainMenu(ActionListener listener)
Options
+ Options(ActionListener listener)
CreateGame
+ CreateGame(ActionListener listener)
CreateFrame
+ CreateFrame(int width, int height, String title)
LogPanel
+ LogPanel()
+ void Display(String log)
+ void Clear()
DisplayPanel
focusComponent: Component
+ Component getFocusComponent()
+ void setFocusComponent(Component focus)
StatsPanel
stats: <List>
+ StatsPanel(<List> PlayerStats)
+ void actionHeard()
EndGamePanel
+ EndGamePanel(ActionListener listener)
TransitionPanel
+ TransitionPanel(String TypeOfPanel)
+ void actionHeard()
Server
turnQueue: <int>
board: <Tile>
frame: CreateFrame
log: LogPanel
stats: StatsPanel
endgame: EndGamePanel
transition: TransitionPanel
playerStats: <PlayerStats>
playerInfo: <PlayerInfo>
+ Server(<Options> options)
- int getNextPlayer()
- void removePlayerFromQueue(int id)
- void updateRobotPosition(Robot movedRobot, Tile newLocation)
- void moved(Tile destination)
- void fired(Tile destination)
- void updateFOW(int player)
- void playerKilled(Player player)
- void checkStalemate()
Tile
robots: <Robot>
isVisible: bool
+ bool getVisibility()
+ void setVisibility()
+ <Robot> getRobots()
+ void addRobots(Robot robot)
+ void removeRobot(Robot robot)
+ bool isEmpty()
PlayerInfo
robots: <Robot>
playerID: int
+ PlayerInfo(int playerID)
+ Robot getRobot(String type)
+ <Robot> getRobotsAlive()
+ int getPlayerID()
PlayerStats
damageDealt: int
damageTaken: int
robotsDestroyed: int
tilesMoved: int
turnsSinceLastMove: int
turnsSinceLastFire: int
+ int getDamageDealt()
+ int getDamageTaken()
+ int getRobotsDestroyed()
+ int getTilesMoved()
+ void setDamageDealt(int x)
+ void setDamageTaken(int x)
+ void setRobotsDestroyed(int x)
+ void setTilesMoved(int x)
+ int getTurnsSinceLastMove()
+ void setTurnsSinceLastMove(int x)
+ int getTurnsSinceLastFire()
+ void setTurnsSinceLastFire(int x)
Robot
alive: bool
health: int
movement: int
type: int
loc: <int>
direction: int
playerID: int
+ Robot(int playerID)
+ bool getAlive()
+ int getHealth()
+ int getMovement()
+ int getType()
+ <int, int> getLoc()
+ int getDir()
+ int getPlayerID()
+ void setDead()
+ void setHealth(int health)
+ void setMovement(int movement)
+ void setType(int type)
+ void setLoc(<int, int> location)
+ void setDir(int direction)
+ void setPlayerID(int playerID)
GamePanel
+ GamePanel(<Tile> board)
+ void Update(<Tile> board)
+ void paintComponent()
+ void drawImage(int x, int y, String pictureName)
+ void drawString(int x, int y, String drawMe)
Player
AI: bool
interpreter: Interpreter
interpreters: <Interpreter>
mailbox: <list>
+ bool isAI()
- void move()
- void fire()
- void endTurn()
+ void actionHeard(String action)
Interpreter
scriptPath: String
words: List
variables: Map
stack: Stack
+ Interpreter(String scriptPath)
+ void play()
+ void parseFile(File file)
+ void eval(String word)
ControlPanel
+ ControlPanel(ActionListener listener)
«extends»
*

Figure 2. RoboChess UML Diagram

2.2 Class Description

2.2.1 Launcher

- Summary:
  - The Launcher class is the class first executed by the executable. It has two functions, main() and actionPerformed(action), as well as one object, an ActionListener al. Launcher's job is to create and run the main menu system and when prompted, create a server object which actually runs the game, passing in all options as arguments to Server.
- Methods:
  - + void Main()
    - main() initializes the menu frame and actionListener, then displays the main menu using the MainMenuPanel class.
    - Pre-condition: none
    - Post condition: none
  - **-** void actionPerformed()
    - actionPerformed(action) takes as an argument 'action', which is a string explaining which action was performed, or in this case which button the user pressed. Based on this argument, the method does a variety of things, including displaying the Options screen, Create Game screen, recalling the Main Menu screen, or creating a server.
    - Pre-condition: none
    - Post condition: the server is created.

2.2.2 Server

- Summary:
  - The server is responsible for all the interactions that the players have with the system. The server regulates the flow of the program by determining whose turn it is and what they can and cannot do. By utilizing the server the process of running and updating the game is streamlined and done solely in one class.
- Fields:
  - **-** <PlayerID> turnQueue
    - The designated order of how each round will start and end.
  - **-** <Tiles> board
    - The game board broken into individual tiles. We chose to use an array because unlike a list the board size will not add or remove tiles as the game continues.
  - - CreateFrame frame
    - The frame in which the game is displayed.
  - - LogPanel log
    - The log to be displayed on the screen.

    - - StatsPanel stats
        - The stats to be displayed after the match is over.
    - - EndGamePanel endgame
        - The end game menu displayed after the match is over.
    - - TransitionPanel transition
        - The transition panel used for loading and transitioning between turns.
    - - <PlayerStats> playerStats
        - Holds statistics for each player, human or AI every movement and fire action will have their corresponding statistics updated.
    - - <PlayerInfo> playerInfo
        - Holds the information regarding the player’s units and the player’s id number.
- Methods:
    - - Int getNextPlayer()
        - Summary: Gets the next player in the turnQueue.
        - Pre conditions: The game has not been resolved.
        - Post conditions: Return the next player in the queue.
    - - void removePlayerFromQueue(playerID ID)
        - Summary: Removes a player from turnQueue, this is called whenever a player is removed from the game.
        - Parameters:
            - ID: the player that will be removed from the turn queue.
        - Pre conditions: A player has lost all their robots.
        - Post conditions: The player has been removed from the turnQueue parameter.
    - - void updateRobotPosition(Robot movedRobot , Tile newLocation)
        - Summary: If a robot has moved successfully its position must be updated.
        - Parameters:
            - movedRobot: The robot that has moved.
            - newLocation: The new tile location the robot has moved to.
        - Pre conditions: A robot has moved to a different tile.
        - Post conditions: The location of the robot and the tile it now occupies has been updated.
    - - void moved(tile destination)
        - Summary: Handles a received move request from a player.
        - Parameters:
            - Destination: the tile that the player wants to move their tank to.
        - Pre conditions: Received a move request.

    - Post conditions: Approves or denies the request based on if it is a legal move and the player’s corresponding playerStats is updated & updateRobotPostion is invoked.
- - void fired(tile destination)
    - Summary: Handles a fire request from the current player. If the tile is within range and the player has not attacked this turn the destination tile is damaged.
    - Pre conditions: The player has not attacked this turn & the tile is within range.
    - Post conditions: All robots in the fired tile (if any) receive damage.
- - void updateFOW(playerID player)
    - Summary: After a robot has moved or been destroyed the fog of war must be recalculated for the effected player.
    - Parameters:
        - Player: the player that will have their fog of war updated.
    - Pre conditions: A robot has moved or been destroyed.
    - Post conditions: The fog of war for that player has been updated accordingly.
- - void playerKilled(player)
    - Summary: Removes a player from the game.
    - Pre conditions: All of the player’s robots have been destroyed.
    - Post conditions: The player will no longer appear in the turn queue and their playerInfo parameter is updated.
- - void checkStalemate()
    - Summary: Checks whether a stalemate has been reached which is when there are only AI players and there have not been any shots fired or movement after x turns.
    - Pre conditions: only AI players remain.
    - Post conditions: If the stalemate has been met terminate the game, else continue game.
- + void Server(<Options> options)
    - Summary: Server initializes the game board and creates the player threads.
    - Parameters:
        - Options: a list of options for the server
    - Pre-condition: none
    - Post condition: none

2.2.3 Tile

- Summary:
    - This class is used for each tile on the game board. A tile has to be able to store multiple robots that are currently occupying it and calculate if it is visible or not for fog of war calculations. Because of this there are multiple methods to get and set parameters for robots that leave and enter the list and get/set the tile visibility when the fog of war needs to be updated.
- Fields:
    - - <Robots> robots:
        - The robots that are currently occupying the tile.
    - - Bool isVisible
        - Determines if this tile is visible when it is a player's turn and the board needs to be rendered.
- Methods:
    - + bool getVisibility()
        - Summary: Returns true if the tile is within a player's view range, false otherwise.
        - Pre conditions: none.
        - Post conditions: returns true if the tile is visible to the current player, false otherwise.
    - + void setVisibility(bool toggle)
        - Summary: Sets the visibility of the tile. This will be used whenever the fog of war needs to be updated.
        - Parameters:
            - Toggle: the new visibility state of the tile.
        - Pre conditions: none.
        - Post conditions: The tile has been assigned a new visibility.
    - + <Robots> getRobots()
        - Summary: Returns all the current robots on the tile. This can include multiple tanks from different players or an empty list.
        - Pre conditions: none.
        - Post conditions: Return a copy of the Robots parameter.
    - + void addRobots(Robot robot)
        - Summary: Adds a robot to the tile, this occurs when a robot moves onto the tile.
        - Parameters:
            - Robot: the robot to add to the tile.
        - Pre conditions: none.

        - Post conditions: the robot has been added to the Robots list parameter.
    - + void removeRobot(Robot robot)
        - Summary: Removes a robot from the tile.
        - Parameters:
            - Robot: the robot on the tile to be removed.
        - Pre conditions: none.
        - Post conditions: The robot has been removed from the Robots list field.
    - + bool isEmpty()
        - Summary: Checks if the tile has any robots on it.
        - Pre conditions: none.
        - Post conditions: Returns true if Robots is empty if not then returns false.

2.2.4 Player Info

- Summary:
    - The playerInfo class is used to get and store the robots of a player. It also stores the player ID which is a thread ID for each player client in order to communicate with them.
- Fields:
    - - <Robots> robots
        - A list that store the robots of corresponding player.
    - - int playerID
        - An integer represents the player and also represents the player's client thread.
- Methods:
    - + void PlayerInfo (int playerID)
        - Summary: create and initialize the playerInfo class, and store the robots belonging to the player with 'playerID' to the <Robots>robots.
        - Parameters:
            - playerID: integer referring to a threadID.
        - Pre-condition: playerID is a valid thread ID.
        - Post condition: new player instantiated.
    - + Robot getRobot (String type)
        - Summary: return the robot from the <Robots> robots with correct type, return null if not found
        - Parameter:
            - type: string that specifies a robot type.
        - Pre-condition: type is one of (sniper, scout, tank).
        - Post condition: none.
    - + <Robot> getRobotsAlive ()

        - Summary: return the robots of corresponding player that are still alive, return null if no alive robot
        - Pre-condition: none.
        - Post condition: none.
    - int getPlayerID ()
        - Summary: return playerID.
        - Pre-condition: none.
        - Post condition: none.

2.2.5 Robots

- Summary:
    - The robots class stores all the current information related to the robots. This does not include basic stats, but rather the current up to date stats of each individual robot.
- Fields:
    - - Bool alive
        - A Boolean variable represents whether the robot is alive.
    - - Int health
        - An integer represents the robot's health.
    - - Int movement
        - An Integer represents the total number of tiles the robot moved.
    - - Int type
        - An integer represents the type of robot.
    - - <Int> loc
        - A two-dimensional array represents the current location of robots on the board.
    - - Int direction
        - An integer represents the direction robot is currently facing.
    - - Int playerID
        - An integer represents the player that own the robot.
- Methods:
    - + void Robots (int playerID)
        - Summary: Create a new robot class, initialize the player ID equals to parameter and initialize other attributes equal to zero for integer and null for Boolean and array.
        - Parameters:
            - playerID: the playerID that the robot belongs to.
        - Pre-condition: none.
        - Post condition: a new Robots is constructed.
    - + Bool getAlive ()
        - Summary: return the attribute alive that represents whether the robot is alive.

    - Pre-condition: none.
    - Post condition: Returns true if the robot has not been destroyed otherwise return false.
- + int getHealth ()
    - Summary: return the current health of robot.
    - Pre-condition: none.
    - Post condition: the current health of the robot is returned.
- + int getMovement ()
    - Summary: return the total number of movement points of the robot.
    - Pre-condition: none.
    - Post condition: return the current number of movement points left for the robot.
- + int getType ()
    - Summary: return the type of the robot.
    - Pre-condition: none.
    - Post condition: return an integer corresponding with the robot types.
- + <int, int> getLoc ()
    - Summary: return the current location of the robot.
    - Pre-condition: none.
    - Post condition: return the location of the robot.
- + int getDir ()
    - Summary: return the direction that the robot is currently facing.
    - Pre-condition: none.
    - Post condition: return an int representing the current direction of the robot.
- + int getPlayerID ()
    - Summary: return the player ID of the player that is in control of the robot.
    - Pre-condition: none.
    - Post condition: returns a playerID.
- + void setDead ()
    - Summary: set the attribute of the alive to false.
    - Pre-condition: none.
    - Post condition: the robot is marked as destroyed.
- + void setHealth (int health)
    - Summary: set the current health of the robot.
    - Parameters:
        - Health: the new health points of the robot.
    - Pre-condition: none.
    - Post condition: the number of health points has been changed.
- + void setMovement (int movement)
    - Summary: set the total remaining movement points of the robot equal to parameter.

        - Parameter:
            - Movement: the new number of remaining movement points that the robot will be assigned.
        - Pre-condition: none.
        - Post condition: The total remaining movement points of the robot has been updated.
    - + void setType (int type)
        - Summary: set the type of robot.
        - Parameter:
            - Type: designates the robot type.
        - Pre-condition: none.
        - Post condition: the robot has their type assigned.
    - + void setLoc (<int, int> location)
        - Summary: set the current location of robot.
        - Parameter:
            - Location: the new location the robot will be assigned.
        - Pre-condition: none.
        - Post condition: the robot's location has been updated.
    - + void setDir (int direction)
        - Summary: set the current direction the robot is facing.
        - Parameter:
            - Direction: the direction that the robot will face.
        - Pre-condition: none.
        - Post condition: the robot's facing direction has been changed.
    - + void setPlayerID (int playerID)
        - Summary: set the player ID.
        - Parameter:
            - playerID: the playerID that the robot will be assigned.
        - Pre-condition: none.
        - Post condition: the robot has a player ID assigned to them.

2.2.6 PlayerStats

- Summary:
    - The PlayerStats class is used for storing the stats of various actions that occur during the game. This includes various stats such as number of shots fired, total tiles moved, etc. At the end of the game these will be displayed. This class also serves as a mean to determine if a stalemate has occurred with AI players if neither of them have fired after 10 turns.
- Fields:
        - - int damageDealt
            - An integer that represents the total damage dealt to enemies.
        - - int damageTaken

    - An integer that represents the total damage points taken during the game.
  - - int robotsDestroyed
    - An integer represents the total number of robots destroyed.
  - - int tilesMoved
    - An integer represents that holds the total number of tiles that the player has moved.
  - - int turnsSinceLastMove
    - An integer that stores the number of turns since the player last moved.
  - - int turnsSinceLastFire
    - An integer that stores the number of turns since the player has fired.
- Methods:
  - + int getDamageDealt ()
    - Summary: returns the total damage dealt statistic.
    - Pre conditions: none.
    - Post conditions: the total damage inflicted is returned.
  - + int getDamageTaken ()
    - Summary: returns the current total damage taken stat.
    - Pre conditions: none.
    - Post conditions: returns the total damage taken statistic.
  - + int getRobotsDestroyed ()
    - Summary: returns the total number of robots destroyed.
    - Pre conditions: none
    - Post conditions: an integer value is returned that represents the total number of robots destroyed.
  - + int getTilesMoved ()
    - Summary: returns the current number of tiles moved statistic.
    - Pre conditions: none.
    - Post conditions: returns the total number of tiles traversed.
  - + void setDamageDealt (int x)
    - Summary: set the value of x as the new damage dealt to the enemy statistic.
    - Parameters:
      - X: the new damage dealt statistic.
    - Pre conditions: none.
    - Post conditions: the new damage dealt statistic has been set.
  - + void setDamageTaken (int x)
    - Summary: set the value of x as the new damage taken statistic.
    - Parameters:
      - X: the new damage taken statistic.
    - Pre conditions: none.

    - Post conditions: the new damage taken statistic has been set.
- + void setRobotsDestroyed (int x)
    - Summary: set the value of x as the new total number of robots destroyed statistic.
    - Parameters:
        - X: the new total tanks destroyed statistic.
    - Pre conditions: none.
    - Post conditions: the new total number of robots destroyed statistic has been set.
- + void setTilesMoved (int x)
    - Summary: set the value of x as the new total number of tiles traversed.
    - Parameters:
        - X: the new tiles traversed statistic.
    - Pre conditions: none.
    - Post conditions: the new tiles moved statistic has been set.
- + int getTurnsSinceLastMove ()
    - Summary: get the number of turns since the player last moved.
    - Parameters:
    - Pre conditions: none.
    - Post conditions: the number of turns without movement from a player has been returned.
- + void setTurnsSinceLastMove (int x);
    - Summary: set the value of x as the new number of turns since last movement.
    - Parameters:
        - X: the new turns since last movement statistic.
    - Pre conditions: none.
    - Post conditions: the new number of idle turns statistic has been set.
- + int getTurnsSinceLastFire ()
    - Summary: get the number of turns since the player last attacked.
    - Pre conditions: none.
    - Post conditions: the number of turns since last fire command has been returned.
- + void setTurnsSinceLastFire (int x);
    - Summary: set the value of x as the new number of turns since last attack statistic.
    - Parameters:
        - X: the taken statistic to replace the current statistic.
    - Pre conditions: none.
    - Post conditions: the new value for number of turns since last attack has been set.

2.2.7 Player

- Summary:
    - This is the class that contains all the possible actions a player can do when it is their turn. It incorporates a view and controller part when interacting with the server, this is because all the calculations & updates are done in the server class.
- Fields:
    - - Bool AI
        - Specifies if the player is AI controlled or not.
    - - Interpreter interpreter
        - Specifies the interpreter with unique scripts, if the player is not AI this parameter will be null.
    - <List> interpreters
        - A list of interpreters which communicate with scripts.
    - <List> mailbox
        - A list containing messages for each AI
- Methods:
    - + bool isAI()
        - Summary: Checks if player is human or AI.
        - Pre conditions: none.
        - Post conditions: Returns true if the player is AI controlled and false otherwise.
    - - void move()
        - Summary: Sends a request to the server to move the current robot to another tile.
        - Pre conditions: The player has at least 1 robot that is alive and has enough movement points.
        - Post conditions: The tank has been moved to a new tile and has its remaining movement points updated and the player's corresponding playerStats is updated.
    - - void fire()
        - Summary: Sends a request to the server to attack a tile.
        - Pre conditions: The player has at least 1 robot that is alive & has not attacked this turn.
        - Post conditions: The robot fires onto the specified tile and the player is unable to fire for the rest of the turn. The attacker and attacked both have their corresponding playerStats is updated.
    - - void endTurn()
        - Summary: Signals the server that the player has ended their turn.
        - Pre conditions: none.

        - Post conditions: The server transitions to the next player in the turn queue.
    - + void actionHeard(String action)
        - Summary: takes as an argument 'action', which is a string explaining which action was performed, or in this case which button the user pressed. Based on this argument, the method does a variety of things, including displaying the Options screen, Create Game screen, recalling the Main Menu screen, or creating a server.
        - Parameters:
            - Action: the action that has been heard by the listener.
        - Pre conditions: an action is heard.
        - Post conditions: the corresponding action for the action is performed.

2.2.8 Interpreter

- Summary:
    - A representation of a Forth language interpreter. Each AI player thread should have 3 instances of this interpreter (one for each robot type). Upon creation, the given script is parsed into a list of forth words.
- Fields:
    - -String scriptPath
        - The directory path to the saved script.
    - - <String> words
        - A parsed list of words for a given script.
    - - <Map> variables
        - A map to store Forth declared variables.
    - - Stack stack
        - A stack used to interpret Forth scripting language
- Methods:
    - + void Interpreter(String scriptPath)
        - Summary: Constructs the interpreter class
        - Parameters:
            - scriptPath: absolute path of the script file location
        - Pre-conditions: none
        - Post-conditions: new interpreter instantiated
    - + play()
        - Summary: Iterates through the words list and calls eval() for each Forth word
        - Pre-conditions: Forth script has been parsed (at class construction time)
        - Post-conditions: Control is returned to the server thread
    - - parseFile(File file)
        - Summary: Reads in characters from the file and creates Forth words and adds them to the words list
        - Parameters:

        - file: file stream of script file
    - Pre-conditions: Valid Forth script file is passed in.
    - Post-conditions: words is populated with Forth words.
- - eval(String word)
    - Summary: Evaluates the given word, pushing/popping them onto/off the stack.
    - Parameters
        - word: a string representation of a Forth word
    - Pre-conditions: word is a valid Forth word
    - Post-conditions: none

2.2.9 CreateFrame

- Summary:
    - CreateFrame is a class which simply creates the frame on which either the menu or game is run, depending on the parameters.
- Methods:
    - + void CreateFrame(int width, int height, String title)
        - Summary: Creates a window frame to display the game.
        - Parameters:
            - width: the width of the frame
            - height: the height of the frame
            - title: the title to display above the frame
        - Pre conditions: a frame is requested from the system.
        - Post conditions: A frame has been build to the specifications of the system.

2.2.10 Main Menu

- Summary of Class
    - The MainMenu class is in charge of creating the panel for the main menu GUI in our system, and is a subclass of the DisplayPanel class. The main menu has three choices, 'Create Game', 'Options', and 'Quit'.
- Methods
    - + MainMenu(actionListener listener)
        - Summary: the constructor for the main menu class. Takes a action listener as a parameter and builds a panel passing it to the 3 possible choices.
        - Parameters:
            - Listener: An action listener that is used when any of the 3 buttons are pressed.
        - Pre conditions: none.
        - Post conditions: the main menu has been constructed and is displayed.

2.2.11 Options

- Summary:
  - Options is a class in charge of creating the view for the options screen within the menu system. It displays an option for mute volume, spectator mode, and debug mode, as well as an option to go back to the main menu. Options is a subclass of DisplayPanel.
- Methods:
  - + void Options(ActionListener listener)
    - Summary: Options only has a constructor method, which takes as a parameter an actionListener. The constructor then creates the panel, passing the ActionListener along to each one of the option objects mentioned above.
    - Parameters:
      - Listener: gets passed to the constructer every time a button is clicked.
    - Pre condition: none
    - Post condition: if any options have been changed the new configurations are saved.

2.2.12 CreateGame

- Summary:
  - The CreateGame class creates the Create Game view in the menu system. It displays options for number of players, amount of human players, which CPU players to play with, and which map size, where applicable. It also has a back button to go back to the main menu, and a confirm button to create the game. CreateGame is a subclass of DisplayPanel.
- Methods
  - + void CreateGame (ActionListener listener)
    - Summary: CreateGame only has one method a constructor, the constructor is given an ActionListener parameter, which it then uses when creating each of the options displayed on screen as listed above.
    - Parameter:
      - Listener: gets passed to the constructer every time a button is clicked.
    - Pre condition: none.
    - Post condition: the create game panel closes and transitions to the game view.

2.2.13 LogPanel

- Summary:
  - LogPanel is a class which creates the panel showing the 'log' section of the screen in the main game view. The class is an extension of DisplayPanel. It displays a message to the current player whenever the player interacts with the board.
- Methods

- + void LogPanel ()
  - Summary: The first is the constructor method, which constructs the panel itself.
  - Pre condition: the game should be already running instead just in set up panel and the game is not over yet.
  - Post condition: the log panel has been built.
- +void Display(String log)
  - Summary: The display method, which takes a message as a parameter which is displayed on the panel. An example would be "While firing, your robot killed Player 3's scout!"
  - Parameter:
    - Log: the message to display in the log panel.
  - Pre condition: log panel is already built already.
  - Post condition: the log panel has the message displayed.
- + void Clear()
  - Summary: Wipes the log clear of any previous messages, this is used every time a turn transition occurs.
  - Pre condition: a player's turn has ended.
  - Post condition: the log panel has been cleared.

2.2.14 StatsPanel

- Summary:
  - The StatsPanel class is a class that displays player stats, which are displayed optionally when the game is won. The panel displays only one player's stats at a time, with the ability to cycle through different StatsPanel panels to see other player's stats. The StatsPanel class is an extension of DisplayPanel.
- Fields:
  - - <List> Stats
    - A list of statistics for each player passed as a parameter when the object was created.
- Methods
  - +void StatsPanel(<List> Playerstats)
    - Summary: the StatsPanel constructor will create a new StatsPanel.
    - Parameters:
      - list: the list of statistics from the resolved game.
    - Pre-condition: the game has been resolved.
    - Post condition: a StatsPanel class is constructed and is displaying the stats for the first player.
  - +void actionHeard ()
    - Summary: The actionHeard method is an implementation for actionListener, which enables the ability to cycle between different players to compare stats.
    - Pre-condition: a stats panel must be displayed on the screen.

        - Post condition: A action is received, and a new stats panel is constructed and displayed.

2.2.15 EndGamePanel

- Summary
    - The EndGamePanel class is a class which displays a menu when the game is over. It is a subclass of DisplayPanel. The menu has three choices: play again, statistics, or main menu. Above the options is displayed the winner of the game (such as "Player 1 wins!").
- Methods
    - +void EndGamePanel (Actionlistener listener)
        - Summary: The EndGamePanel constructor will create a new EndGamePanel class
        - Parameters:
            - listener: an Actionlistener object, which is passed along to options which include play again, statistics and main menu, to allow user interaction with each option
        - Pre-condition: Game must be running
        - Post condition: A new EndGamePanel class is created, and a new End Game panel displays.

2.2.16 TransitionPanel

- Summary
    - TransitionPanel is a class which creates a panel used during transitional phases of the game. This includes a loading panel, a ready next player panel, and a A.I. playing panel. These are grouped into a single class because the screens are identical except for the words displayed on the panel, and a button used in the ready next player panel. This class is a subclass of DisplayPanel.
- Methods
    - +void TransitionPanel (String TypeOfPanel)
        - Summary: The TransitionPanel constructor class will create a new TransitionPanel class
        - Parameters:
            - TypeOfPanel: a string that represents the type of panel needed.
        - Pre-condition: A player has ended their turn.
        - Post condition: A new TransitionPanel class is created is displayed.
    - +void actionHeard ()
        - Summary: The method actionHeard () is a button listener for the player transition panel.
        - Pre-condition: the button has been clicked.
        - Post condition: The transition screen is removed and the game board is shown.

2.2.17 ControlPanel

- Summary:
  - The ControlPanel class is a subclass of DisplayPanel, and creates a panel used in the bottom of the main game screen, which contains all of the buttons used to interact with the game. These include move, fire, end turn, confirm, and back.
- Methods
  - + void ControlPanel(actionlistener listener)
    - Summary: the constructor of ControlPanel.
    - Parameters:
      - listener: the listener that is used by all the buttons in the control panel.
    - Pre condition: The game has started.
    - Post condition: The control panel frame has been drawn and displayed.

2.2.18 GamePanel

- Summary of Class
  - GamePanel is a subclass of DisplayPanel that creates the main panel used in the game screen. This is the panel that shows the board and current player's view perspective.
- Methods
  - + void GamePanel(<tiles> board)
    - Summary: The constructor of the GamePanel class.
    - Parameters:
      - Board: the game board's tiles stored in a 2D array.
    - Pre conditions: The game has started.
    - Post condition: The game panel has been drawn and displayed.
  - + void Update(<Tile> board)
    - Summary: redraws the board in the event something has changed.
    - Parameters:
      - Board: the game board stored in a 2D array.
    - Pre condition: The game has started.
    - Post: The game panel has been updated.
  - + void paintComponent(<Tile> board)
    - Summary: paints the game scene, including backround and board
    - Parameters:
      - Board: the game board to be painted
    - Pre-condition: The game has started
    - Post condition: The game scene is painted
  - + void drawImage(int x, int y, String pictureName)
    - Summary: takes an image and screen coordinates and draws the image to the screen.
    - Parameters:
      - X: the x coordinate where the image will be drawn.

        - Y: the y coordinate where the image will be drawn.
        - pictureName: the image name that will be written at the x & y coordinates.
    - Pre conditions: none
    - Post conditions: The image was printed at the specified coordinates.
- + drawstring(int x, int y, String drawMe)
    - Summary: Takes a string and displays it to the screen.
    - Parameters:
        - X: the x coordinate where the string will be printed.
        - Y: the y coordinate where the string will be printed.
        - drawMe: the string that will be written at the x & y coordinates.
    - Pre conditions: none.
    - Post conditions: the drawMe will be written at the x & y coordinates.

2.2.19 DisplayPanel

- Summary:
    - DisplayPanel is a superclass of most GUI classes in our game. Its purpose is to obtain the focus when a frame is created.
- Fields
    - - Component focusComponent
        - DisplayPanel has as an attribute a Component focusComponent, which is the component that should be in focus whenever the panel is.
- Methods
    - + Component getFocusComponent()
        - gets the focusComponent
            - Pre-condition: none.
            - Post condition: The component to be focused is returned.
    - + void setFocusComponent(Component focus)
        - sets the focusComponent field
        - Parameters:
            - focus: the component to be set as the focusComponent
        - Pre-condition: none.
        - Post condition: the focusComponent is set to the parameter.

3. Changes Made

3.1 Spectator Mode

In the require document, we display the spectator mode option in the create game screen, however, we have decided to move it the options screen, as reflected in this design document.

3.2 Debug Mode

We have decided to add a debug mode which can be enabled in the options screen. This option will generate additional messages in the log panel displaying various debug information such as when functions are reached. This mode will also disable the clear() method in LogPanel, so the log is not cleared after every turn.